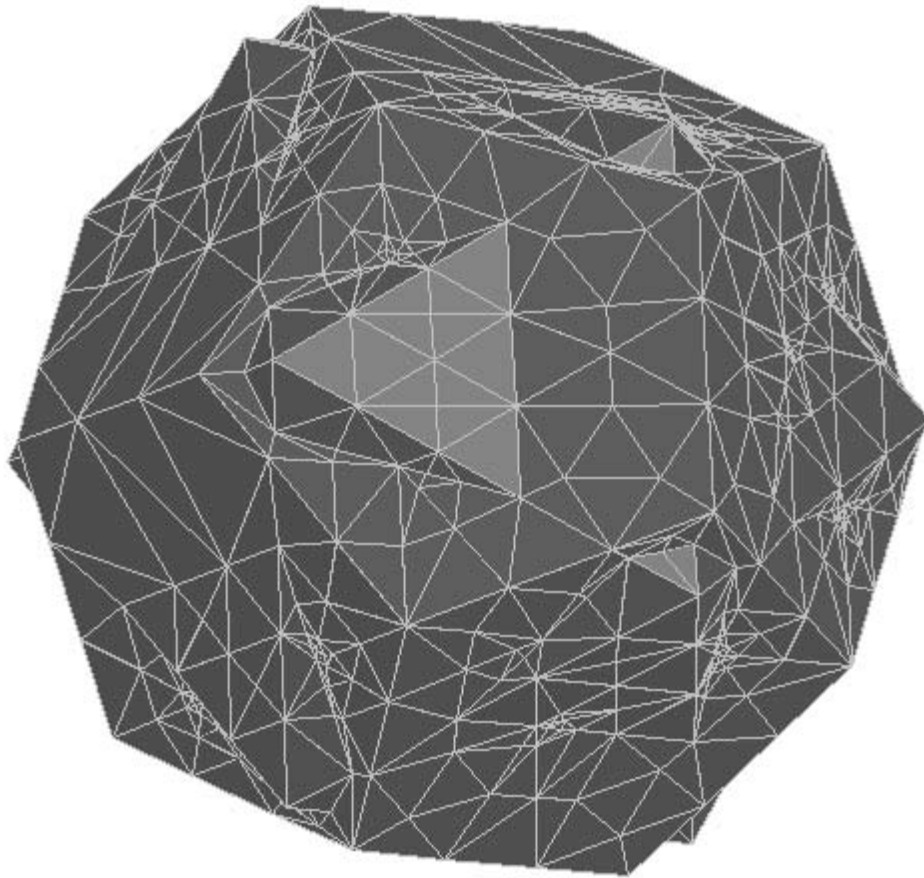


An algorithm for overlaying 3D features using a tetrahedral network



Master of Science Thesis
Arno van der Most

Delft, September 2004

An algorithm for overlaying 3D features using a tetrahedral network

Professor: Prof.dr.ir.P.J.M.van Oosterom
Supervisors: Ir. E.Verbree
Drs. W.C. Quak

Master of Science Thesis
Arno van der Most

Delft, September 2004

Delft University of Technology
Faculty of Civil Engineering and Geosciences
Department of Geodesy
Section GIS technology



Preface

This thesis was written as part of my graduation assignment at Delft University of Technology. The research associated with this thesis was performed at the section GIS-technology of the faculty of Civil Engineering and Geosciences of Delft University of Technology. The graduation assignment is the final hurdle I need to take to complete my Master of Science program and conclude my life as a full-time student at the university in Delft. With the completion of this thesis I have almost passed that hurdle. I've jumped and hope to land safely on the other side to cross the finish line victoriously.

The road to completing the graduation assignment was not a smooth one. Due to personal circumstances, performing the research and writing this thesis was a long and laborious process. I could not have produced this thesis and the results laid out therein without the patience and support of my supervisors ir. E. Verbree and drs. W. Quak. Although we didn't see much of each other, my graduation professor, prof. dr. ir. P.J.M. van Oosterom, provided usefull comments and sugestions and gave me enough spirit and confidence to complete the assignment. Although there was only enough time to process some of his comments, they helped a lot in improving this thesis.

With a subject at the frontline in both GIS and meshing research and an increasing number of possible applications, it was interesting to work on the subject of three-dimensional feature overlay algorithms. Although the research didn't result in a working algorithm, I think the idea presented here has a future and I hope that the results of the research I did are of some help to those who want to design or implement such an algorithm.

Delft, September 2004
Arno van der Most

Summary

The development of 3D GIS has been progressing slowly but steadily over the last ten or so years. The focus in the development of 3D GIS has mainly been on the visualisation aspects of 3D data from a database and less on data storage of three-dimensional data. Only recently the attention is slowly shifting towards user interaction with the 3D GIS through manipulation and analysis of the 3D data within a 3D GIS. There already are some buffering and height analysis tools available, but not much research has been done on one of the most often used tools in two dimensional GIS, the map overlay. Overlay algorithms are among the more complex algorithms in GIS that one can think off. Complex data structures containing both geometrical and topological data of huge numbers of features are subject to complex queries. In this thesis an attempt is made to unravel this complexity and present an algorithm that can overlay three-dimensional data using a tetrahedral network.

The algorithm described in this thesis doesn't overlay maps, or layers, as traditional map overlay algorithms do. The algorithm is object oriented and therefore overlays individual features instead. Although this wasn't one of the constraints set up before the research started, it is important to distinguish between these approaches to overlaying spatial data. To avoid confusing the approach taken by the algorithm presented here with the approach genuine map overlay algorithms have, the term feature overlay algorithm is used.

The fact that overlay algorithms are among the more complex algorithms in GIS that one can think off together with the higher complexity of three-dimensional algorithms compared to two-dimensional algorithms made it necessary to explore the algorithm in two dimensions first. The algorithm was designed in the following three steps:

1. Formulate a general 'container' feature overlay algorithm
2. Fill the container algorithm with two-dimensional sub-algorithms
3. Translate the two-dimensional sub-algorithms to three-dimensional sub-algorithms

Overlay strategy

The strategy that the feature overlay algorithm uses to overlay the features is the same for both the two-dimensional and the three-dimensional feature overlay algorithm. The algorithm processes features iteratively. One at a time features are overlayed with the result of the overlay upto the moment that the feature is presented to the algorithm. The algorithm breaks down the features into smaller and smaller parts until the parts no longer intersect with features that were already overlayed. It then iteratively adds these parts, called sections, to the result, i.e. it overlays the new feature with the features already overlayed and that are present the result, called the overlay triangulation/tetrahedralization.

The complete algorithm is shown in figure 0.1. Features are broken down into facets, facets are broken down into edges and edges are finally broken down into sections. So for three-dimensional features this results in the four nested iterations shown in figure 0.1. Because of the lower dimension, the two-dimensional algorithm doesn't need to break down the features into facets, so for the two-dimensional algorithm the insert facets iteration step can be omitted leaving three nested iterations.

As input the algorithm only requires a list of directed edges per feature for two-dimensional features and a list of facets per feature for three-dimensional features, where facets are triangular elements of the boundary of the feature obtained by triangulating the boundary of the feature. The output of the algorithm will be a constrained triangulation or tetrahedralization with the boundaries of the overlayed features represented as the constraints.

Two-dimensional algorithm

After the two-dimensional algorithm has selected an edge from the feature data, it adds the start and end vertices of the edge to the overlay triangulation and then needs to find the sections. It does this by starting at the inserted start vertex of the new edge in the overlay triangulation and walking from one face to the next along the line connecting the start and the end vertex of the edge. Each time a constraint is found (i.e. an edge or vertex that belongs to a feature that was already inserted into the overlay triangulation) it inserts a vertex at the intersection of the line and the constraint effectively breaking the new edge and the constraint into sections.

When the triangle walk algorithm reaches the end vertex of the new edge, it has found all the sections

that make up the edge and it is ready to insert these sections into the overlay triangulation. This is done by flipping all the edges in the overlay triangulation that still intersect with the section out of the way. After all sections have been processed, the entire boundary of the feature is present in the overlay triangulation. To complete the overlay the feature identifier of the new feature is assigned to all faces in the overlay triangulation that belong to the new feature.

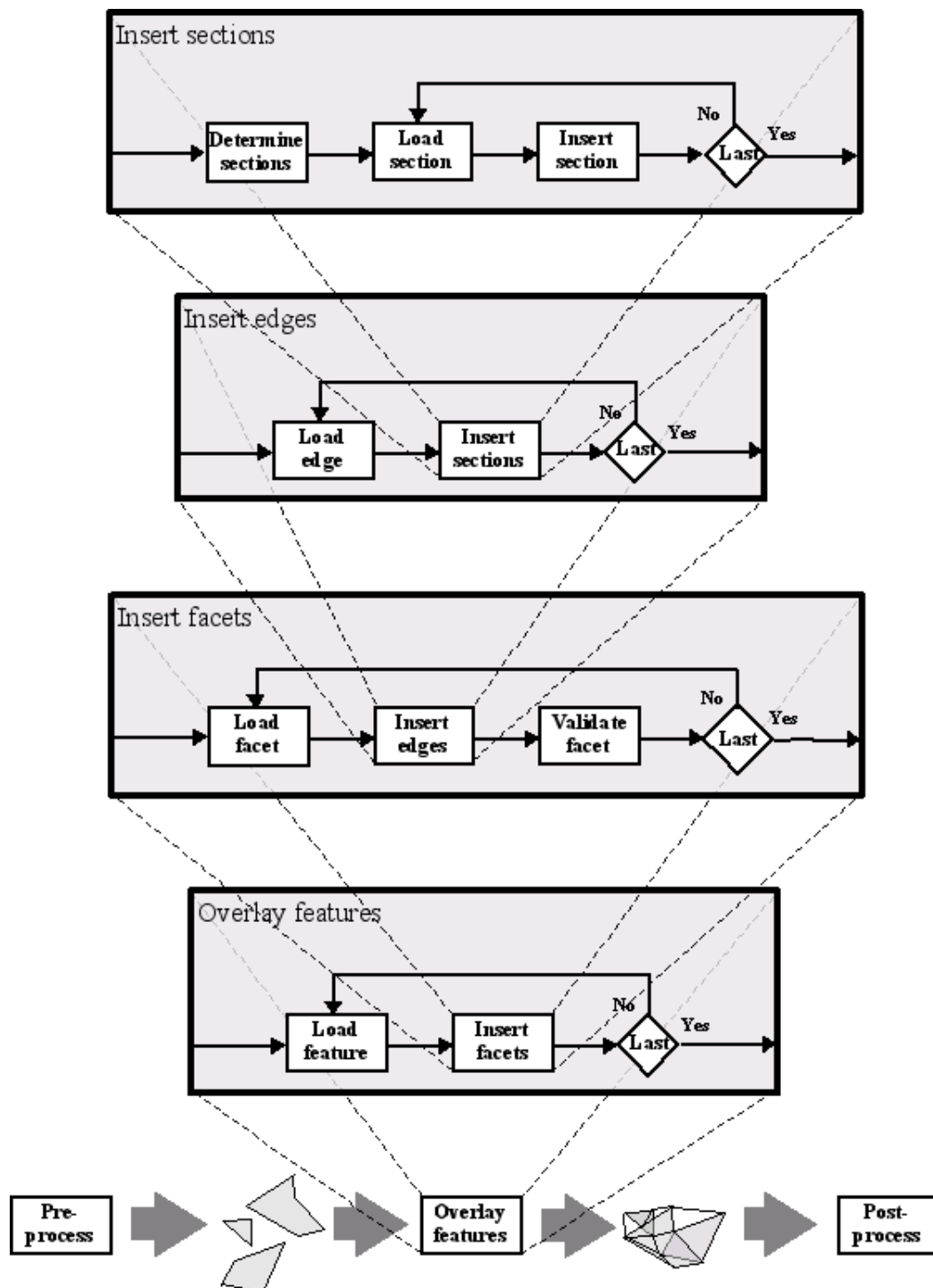


figure 0.1: the container algorithm

For the two-dimensional algorithm the strategy chosen not only makes it possible to overlay features using a triangulation. Walking along the faces of a triangulation, calculating intersections and flipping edges also gives it some nice properties:

- The algorithm only adds vertices at intersections of constrained edges
- Very little information about the features is needed to be able to overlay them
- With one exception, non-convex features, such as features with intrusions and holes are no problem for the algorithm
- the algorithm can easily be modified to behave as a map overlay algorithm in stead of a feature overlay algorithm

Three-dimensional algorithm

The three-dimensional feature overlay algorithm uses the same strategy to overlay features as the two-dimensional feature overlay algorithm. It also uses the triangle walk algorithm to find the sections (it just walks through tetrahedrons in stead of triangles). It also uses the flip algorithm to make sure that non-constrained facets in the overlay tetrahedralization don't intersect with the sections anymore. After it has finished doing that it is not ready to assign the feature identifiers yet though.

After inserting all of the edges of a facet of a feature, only the boundary of the facet is present in the overlay tetrahedralization. The facet itself is not necessarily present, because there might be other facets sticking through the new facet without intersecting with any of the edges of the new facet. If the facet doesn't exist in the overlay triangulation, the intersecting facets are flipped out of the way in a same way that they were flipped out of the way when an intersection with a section was found.

When the missing facets have been reconstructed by the flipping algorithm the feature identifiers can be assigned to all the tetrahedra that are part of the new feature.

The three-dimensional feature overlay algorithm would have the some properties as the two-dimensional version. The three-dimensional has one problem though. It is very difficult to prove that the process of flipping will be able to create the constraints needed in the final tetrahedralization. Although the proof hasn't been found yet, there are indications that it does work in most situations.

Samenvatting

De ontwikkeling van 3D GIS heeft de laatste jaren langzaam maar zeker vooruitgang geboekt. Het zwaartepunt bij het onderzoek en de ontwikkeling van 3D GIS lag vooral bij de visualisatie aspecten van driedimensionale gegevens uit databases en in mindere mate bij de opslag van driedimensionale gegevens. Pas sinds kort verschuift de aandacht geleidelijk naar gebruikersinteractie en analyse binnen een 3D GIS. Er zijn al wat buffer en hoogteanalyse tools beschikbaar, maar er is nog bijna geen onderzoek gedaan naar een van de meest gebruikte tools in 2D GIS, de map overlay. Overlay algoritmes behoren tot de meer complexe algoritmes binnen GIS. Complexe datastructuren die zeer grote hoeveelheden geometrische en topologische gegevens beschrijven zijn onderhevig aan complexe queries. In deze thesis wordt een poging gedaan om orde te scheppen in deze complexiteit en een algoritme te beschrijven dat een overlay kan uitvoeren op driedimensionale gegevens met behulp van een tetrahedron netwerk.

Het algoritme dat beschreven wordt in deze thesis voert geen overlay uit op maps, of kaartlagen zoals map overlay algoritmes dat doen. Het algoritme is object georiënteerd en voert derhalve een overlay uit op individuele features in plaats van maps. Hoewel het geen voorwaarde was voor het onderzoek is het wel van belang onderscheid te maken tussen deze twee verschillende benaderingen van de gegevens voor het doen van een overlay. Om verwarring te voorkomen wordt het algoritme dat hier wordt behandeld aangeduid met de term feature overlay algoritme.

Het feit dat overlay algoritmes behoren tot de meer complexe algoritmes binnen GIS samen met de hogere complexiteit van driedimensionale algoritmes ten opzichte van tweedimensionale algoritmes maakt het noodzakelijk eerst het algoritme in twee dimensies te verkennen. Het algoritme was ontworpen in de volgende drie stappen:

1. Formuleer een algemeen 'container' feature overlay algoritme
2. Vul het container algoritme met tweedimensionale sub-algoritmes
3. Vertaal de tweedimensionale sub-algoritmes naar driedimensionale sub-algoritmes

Overlay strategie

De strategie die gebruikt wordt door het overlay algoritme is hetzelfde voor tweedimensionale en driedimensionale features. Het algoritme verwerkt features op een iteratieve wijze. Een voor een worden de features geoverlayed met het resultaat van de overlay tot dan toe. Het algoritme knipt de features op in steeds kleinere delen net zolang tot die delen niet meer snijden met features die al behandeld zijn door het algoritme. Daarna voegt het op een iteratieve manier die kleinste delen (sections) toe aan het resultaat (de overlay triangulatie). Het algoritme voert dus eigenlijk een overlay uit tussen het nieuwe feature en de overlay triangulatie.

Het complete algoritme is te zien in figuur 0.1. Features worden opgeknipt in facets, facets worden op hun beurt weer opgeknipt in edges en deze edges worden uiteindelijk weer opgeknipt in sections. Voor driedimensionale features zijn er dus in totaal vier geneste iteraties. Daar de lagere dimensie heeft het tweedimensionale algoritme een iteratie minder nodig. Het is niet nodig om tweedimensionale features op te knippen in facets. Het feature zelf is immers al tweedimensionaal.

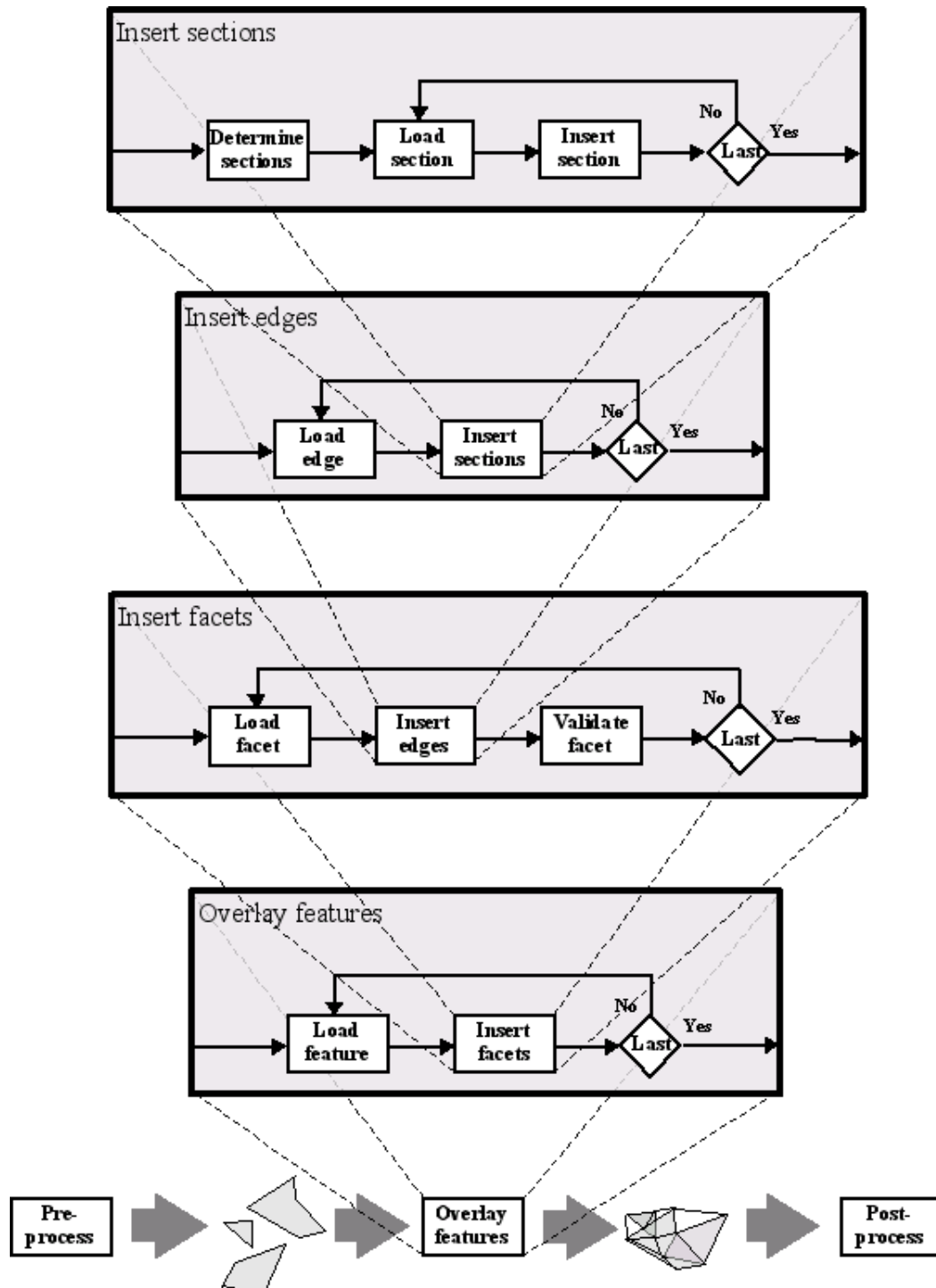
Als invoer heeft het algoritme alleen een lijst van gerichte edges per feature nodig in het geval van tweedimensionale features. Voor driedimensionale features is een lijst met gerichte facets nodig. Facets zijn faces die verkregen zijn door het oppervlak van het feature te trianguleren. De uitvoer (het resultaat) van het algoritme is een constrained triangulation of constrained tetraëder netwerk. De constraints in deze datastructuren representeren de randen van de features.

Tweedimensionaal algoritme

Nadat het tweedimensionale algoritme een edge heeft gekregen uit de feature data voegt het algoritme de start en eind vertex toe aan de overlay triangulatie. Daarna zal het de sections moeten vinden. Dit wordt gedaan door langs de lijn tussen de start en eind vertex van face naar face te lopen en iedere keer als er een constraint (een edge of een vertex van een feature dat al is behandeld door het algoritme) in de triangulatie tegengekomen wordt daar een vertex in de overlay triangulatie in te voegen. Deze nieuwe vertex knipt de nieuwe edge en de edge in de triangulatie op en creëert op deze manier nieuwe sections.

Als het triangle walk algoritme de eind vertex van de nieuwe edge heeft bereikt, dan heeft het

algoritme alle sections gevonden en kan het deze sections gaan toevoegen aan de overlay triangulatie. Dit toevoegen gebeurt door alle edges van de overlay triangulatie die de section nog doorsnijden uit de weg te flippen. Als alle sections van alle edges van een feature op deze manier zijn verwerkt is de rand van het feature aanwezig in de overlay triangulatie. Om de overlay af te ronden kent het algoritme aan alle faces van de overlay triangulatie die binnen de rand van het feature liggen de identifier van het feature toe.



Figuur 0.1: het container algoritme

De strategie maakt het voor het tweedimensionale algoritme niet alleen mogelijk om een overlay uit te voeren met behulp van een triangulatie. Het lopen langs faces in een triangulatie, het berekenen van intersecties en vervolgens alle nog snijdende edges wegflippen zorgt ook voor een aantal goede eigenschappen:

- het algoritme voegt alleen vertices toe waar features elkaar snijden
- er is maar heel weinig informatie nodig om features te kunnen overlayen
- met een uitzondering vormen niet-convexe features zoals features met deuken en gaten geen probleem
- het algoritme kan makkelijk aangepast worden zodat het zich als map overlay algoritme gedraagt in plaats van een feature overlay algoritme

Driedimensionaal algoritme

Het driedimensionale feature overlay algoritme gebruikt dezelfde strategie als het tweedimensionale algoritme om features te overlayen. Net als het tweedimensionale algoritme gebruikt het een triangle walk algoritme om de sections te vinden (al loopt het dan wel door tetrahedra in plaats van door faces). Het gebruikt ook een flip algoritme om nog doorsnijdende facets uit de weg te flippen van de sections. Het verschil met het tweedimensionale algoritme is dat na deze stap het driedimensionale algoritme nog niet klaar is.

Nadat alle edges van een facet van een feature zijn opgenomen in het overlay tetraëder netwerk is alleen de rand van het facet nog maar aanwezig. Het facet zelf hoeft nog niet te bestaan. Het is namelijk mogelijk dat er nog tetraëders het facet doorsnijden zonder dat ze een van de randen van het facet doorsnijden. Als de facet nog niet bestaat in het overlay tetraëder netwerk zullen de nog doorsnijdende facets uit de weg moeten worden geflipt op dezelfde manier als bij het invoegen van de sectoïns.

Als de nog missende facets gereconstrueerd zijn door het flip algoritme kan de feature identifier worden toegekend aan alle tetraëders in het overlay tetraëder netwerk die binnen de rand van het nieuwe feature liggen.

Het driedimensionale feature overlay algoritme zou dezelfde eigenschappen hebben als de tweedimensionale versie. Het is alleen echter nog zo, dat er nog geen bewijs is dat het flip algoritme altijd werkt in drie dimensies. Hoewel het bewijs nog niet gevonden is zijn er wel aanwijzingen dat het in de meeste situaties wel werkt.

Contents

Preface	iii
Summary	iv
Samenvatting	vii
1. Introduction	3
2. Project information	6
2.1 Introduction	6
2.2 Project strategy	7
2.3 Project resources	8
3. 2D feature overlay algorithm	10
3.1 Introduction	10
3.2 Algorithm overview	12
3.3 Phase 1: pre-processing	14
3.3.1 <i>Different starting situations</i>	14
3.3.2 <i>The feature data format</i>	16
3.4 Phase 2: inserting features	17
3.4.1 <i>Flipping</i>	18
3.4.2 <i>Inserting edges</i>	19
3.4.3 <i>Sections explained</i>	20
3.4.4 <i>Processing sections</i>	22
3.4.5 <i>Special cases and their implications</i>	24
3.5 Phase 3: re-assembling features	27
4. 3D feature overlay algorithm	30
4.1 Introduction	30
4.2 The 3D container algorithm	31
4.3 From two to three dimensions	33
4.3.1 <i>Three-dimensional feature data format</i>	34
4.3.2 <i>Translation of sub-algorithms</i>	35
4.3.3 <i>Special cases</i>	38
5. Implementation of 2D feature overlay algorithm	40
5.1 Introduction	40

5.2 Phase 1: pre-processing	44
5.3 Phase 2: inserting features	47
5.3.1 <i>Processing edges</i>	48
5.3.2 <i>Processing sections</i>	51
5.4 Phase 3: re-assembling features	59
6. Conclusions and recommendations	62
6.1 Conclusions	62
6.2 Recommendations	63
Glossary	65
Refferences	67
Appendix A: source code for 2D feature overlay algorithm	70

1. Introduction

Although the killer application for three-dimensional geographical information systems hasn't been found yet, the development of such 3D GIS has been progressing slowly but steadily over the last ten or so years. The focus in the development of 3D GIS has mainly been on the visualisation aspects of 3D data from a database and less on data storage of three-dimensional data. Only recently the attention is slowly shifting towards user interaction with the 3D GIS through manipulation and analysis of the 3D data within a 3D GIS. There already are some buffering and height analysis tools available. Not much research is done on one of the most often used tools in two dimensional GIS, the map overlay. This thesis describes the research project that attempts to design such an algorithm for three-dimensional data.

Overlay algorithms are among the more complex algorithms in GIS that one can think off. Complex data structures containing both geometrical and topological data of huge numbers of features are subject to complex queries. When performing an overlay, the overlay algorithm needs to take all these aspects into account. Designing overlay algorithms which are robust and give reliable results while still keeping operating time within the limits of human patience has proven to be a complicated task for two dimensional GIS. This added to the increased complexity of algorithms that deal with three dimensional data as opposed to the relatively familiar terrain of algorithms that deal with flat, two dimensional data, makes it necessary to set clear-cut goals for the research project.

Goal

The main goal of this research project was to design an overlay algorithm. This algorithm should overlay three dimensional data using a tetrahedral network as a data structure in which to perform the overlay. This goal was formulated in the following research question:

How can three dimensional data be overlayed using a tetrahedral network?

Answering this question directly is complicated. To make the challenge of answering this question easier, the main problem was broken down into the following five sub-problems:

- What strategy should be used for overlaying data?
- What data would be required as input?
- How would that strategy work when applied to two dimensional data?
- Can the algorithm be implemented robustly in a finite precision system?
- Can the two dimensional overlay algorithm be translated to work in three dimensions using the same strategy?

Working with three dimensional data and predicting the effects of actions taken in a three dimensional data structure can become complicated very fast. Take the simple shape of the cube for example. In how many tetrahedra can a cube be divided? If you haven't figured this out before, you would probably have some difficulty answering this question immediately. Because of this, the problem of designing an algorithm to work with three dimensional data is tackled by trying it out in an easier to understand two dimensional situation first.

Conditions

In order to be able to produce meaningful results within the time set for the project, the goal has been constrained by defining certain characteristics of the algorithm beforehand as well as making some assumptions on the ability of the operator to provide clean data. The main question itself already narrows down the search area for solutions drastically. It states that a tetrahedral network should be used to perform the overlay. This restriction was made for two reasons. First, the tetrahedron is one of the simplest three dimensional shapes. By using this shape, calculations can be kept relatively simple. Second, the tetrahedral network has a strong (well defined) topology which is advantageous to the algorithm.

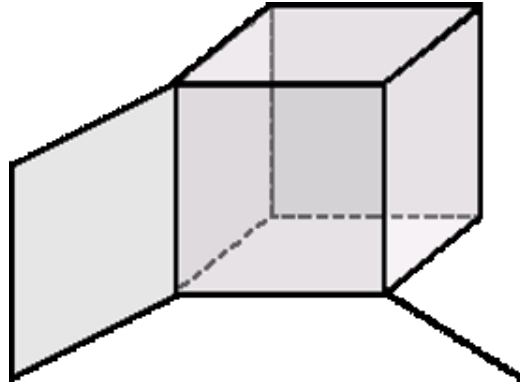


Figure 1.1: example of a non-regular feature

Appart from the condition for the data structure, the features that need to be embedded in that data structure also need to meet some conditions. To prevent spending too much time trying to solve exceptions features presented to the algorithm need to meet the following four conditions:

- Features should be regular
- It should be possible to tetrahedralize features without adding extra nodes
- The features should have valid and clean polyhedral representations
- Features are not allowed to have holes

Regular features are features that have volume. This excludes zero, one, and two-dimensional features as well as three-dimensional features that have extrusions or intrusions of two dimensions or lower [Bronsvoort, 2001]. An example of such an invalid feature is shown in figure 1.1. Not all features that have volume can be triangulated. A well known example of a feature that can only be triangulated when extra nodes are added is Schonhart's 6-vertex polyhedron shown in figure 1.2 [Aichholzer, 2002]. The algorithm presented in this thesis expects that features with either of these properties won't be presented to it. It also asumes that the feature data presented to it is clean [Van Oosterom, 2004]. For example, the sides of a polyhedron should connect seamlessly.

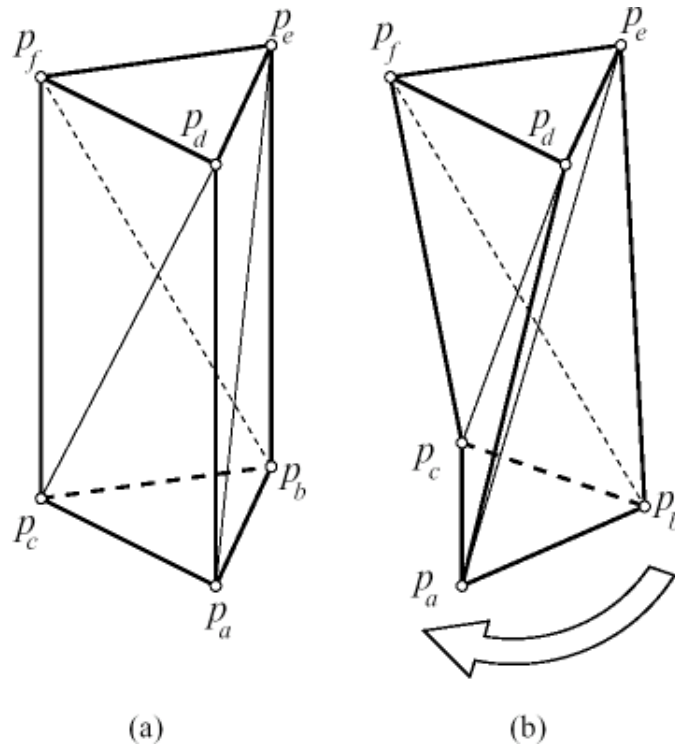


Figure 1.2: a) triangulizable polyhedron b) Schonhart's polyhedron

The algorithm described in this thesis doesn't overlay maps, or layers, as traditional map overlay algorithms do. The algorithm is object oriented and therefore overlays individual features instead. Although this wasn't one of the constraints set up before the research started, it is important to distinguish between these approaches to overlaying spatial data. To avoid confusing the approach taken by the algorithm presented here with the approach genuine map overlay algorithms have, the term feature overlay algorithm will be used for this algorithm throughout the rest of this thesis.

Structure of this thesis

Because of the high complexity of a feature overlay algorithm for three-dimensional data, the development of the algorithm did not start out with the 3D case straight away. In stead, a two-dimensional version of the algorithm was designed before diving into the complexity of the third dimension. A detailed description of the entire strategy chosen to tackle the problem together with more background information on the research project in general can be found in chapter 2. After that, the rest of the thesis follows along the same lines as the research strategy. So first, in chapter 3, the two-dimensional case of the feature overlay algorithm is described. Starting with a general overview of the entire algorithm and further into the chapter filling in the details. Chapter 4 describes the upgrade from the two-dimensional algorithm from chapter 3 to the final three-dimensional algorithm. Since the general structure of the algorithm is already described this chapter mainly focusses on the changes that need to be made in order to make it possible for the algorithm to process three-dimensional data. The results of the implementation of the two-dimensional feature overlay algorithm can be found in chapter 5. The structure of this chapter is analogue to chapter 3. The different steps of the algorithm as described in chapter 3 are accompanied by code snippets and images produced during the process of testing the code. Finally chapter 6 contains some thoughts on the algorithm and suggestions for further research.

The entire source code used for the implementation of the two dimensional feature overlay algorithm can be found in the appendices. There are not many comments added to the source code to keep it manageable. It is there mainly for the sake of completeness and should only be used as copy paste material. A description of all the important code snippets can be found in chapter 5. The source code is discussed within the thesis because of the importance of the implementation of the algorithm for the design process and because implementing the algorithm is not as trivial as it might seem. It illustrates both the dependency of the algorithm on specific characteristics of the triangulation data structure and the problems that finite precision poses for the implementation of an algorithm that assumes infinite precision.

Throughout the thesis a number of terms will be used which are specific to the subject of this research project. Most terms are explained when they are first used. If at any time the meaning of a term is not clear either because it's description was not in the thesis text or because you're not reading the entire thesis and just want to know what's going on quickly, the glossary contains short descriptions of most of the field specific terms used in this thesis.

2. Project information

All projects, large and small, need some form of preparation. The project needs to be defined, a background study needs to be done, a schedule needs to be set up, experiments need to be performed, and everything needs to be documented. In that respect, this project is not different from any other project. In this chapter, an overview is given of the problem specific parts of the strategy chosen to solve the problem as described in the introduction. First some more background information is given here in this section. Section 2.2 lists the different steps taken during the project. The structure of this thesis is analogue to the structure of the research project described here. The tools used during the development and implementation of the algorithm are discussed in section 2.3.

2.1 Introduction

Before discussing an overlay algorithm it is good to know what an overlay actually is. An elaborate search through the available literature showed that not many people have bothered to ask themselves this question before developing or using an overlay algorithm. When they did, the definitions were usually focussed around the specific implementation or application of the algorithm. A definition which can be applied to a reasonable wide variety of overlays is the map overlay definition of [Clarke, 1999]: Placing multiple thematic maps in precise registration, with the same scale, projections, and extent, so that a compound view is possible. A more formal definition, which focuses more on the computational geometry aspects of the overlay, was formulated earlier by [Kriegel, 1992]. The explanation of this definition as given by Kriegel is too long to produce here. The final definition is shown in figure 2.1 without the explanation.

$$\begin{aligned} \text{overlay: } M^* \times M^* \times \{f \mid f: (M_1 \cup \{r_1\}) \times (M_2 \cup \{r_2\}) \rightarrow (T_3 \cup \{r\})\} &\rightarrow M^* \\ (M_1, M_2, f) &\rightarrow M_3, \\ \text{where } M_3 = \{t = (t.P, t.A) \mid t.P \in \text{sep}(t_1.P \cap t_2.P), t.P \neq \emptyset, \\ &t.A = f(t_1, t_2), t.A \neq r, t_1 \in M_1 \cup \{r_1\}, t_2 \in M_2 \cup \{r_2\}\} \end{aligned}$$

Figure 2.1: definition of an overlay by Kriegel

These definitions use maps as a source of the overlay algorithm. Although the algorithm presented here overlays features in stead of thematic maps, the same principle still applies. Clarke's definition adapted to this algorithm would become the following: Placing multiple features from different thematic sources in precise registration, with the same scale, projections, and extent, so that a compound view is possible.

Besides knowing what it is, it is also good to know what it can be used for. Being able to imagine the uses of a theoretic computational algorithm such as this overlay algorithm can make it easier to understand how it works or at least make it more interesting to read about it. This algorithm is useful when the application requires a combined view of three dimensional datasets with different thematic data.

In more and more research and application fields, three-dimensional data is used as an aid to solving complex spatial problems. Three of such fields have been using three dimensional data for a longer period of time. Applications in these fields were also the testing ground for the development of the three dimensional GIS in the last decade. The first field is urban planning [Dodge, 1997]. Due to space becoming more scarce in urban areas and the increasing demand of the population, urban planning problems become more and more complex. Building on top of roads, building higher, building underground, multi-purpose buildings, new forms of transportation. These are just a few of the options used nowadays to combat the high demand on space. With all these buildings and other structures on top or over each other, planning becomes so complex that a sketch on a piece of paper or even a CAD model won't suffice anymore. The overlay algorithm can be used to support the planning process for structures such as a subway line, or the placement of communication antennae.

The increased complexity in urban development and the increasing claims on land also ask for a legal registration system that grows with it. 3D cadastres with three-dimensional features, 3D property units and parcels will become as complicated (or even more complicated) than the urban property situation they describe [Stoter, 2004]. A three-dimensional overlay algorithm can be a useful tool for extracting information out of a complex GIS of a 3D cadastre.

The second field is geology [Masumoto, 2002]. Three dimensional GIS is already widely used by

geologists to visualise the geological data such as sedimentation and fossil layers [Nigro, 2002]. An current research focuses on the implementation of spatial temporal three dimensional data in a GIS environment. In such an environment, the overlay algorithm could for example be used as a tool to analyse differences between different time stamps of the same region. The geological data could also be overlayed with totally different data, such as oil contingents of oil companies. In this way the overlay algorithm can support analysis of oil reserves for an oil company and answer questions concerning resource locations and ownership or mining rights.

The third and last field that will be addressed here is the field of environmental and climate studies. The algorithm could be used in the same way as explained for the geological data, i.e. analysing spatio-temporal data by comparing time slices of data of the same region. For example analysis of air and sea currents in combination with landmasses or the sea-floor. Other applications of the algorithm could be the analysis of sound and air pollution near airports, roads and factories. Data from models, measurement stations in the field and datasets of the area could be overlayed to gain a better understanding of the impact of a polluting object on its surrounding area.

2.2 Project strategy

As mentioned in the introduction of the thesis, the research field of GIS analysis tools for three dimensional data is relatively young. As a result, there isn't much literature on tools and algorithms for analysis of three dimensional data in GIS. There are however a number of related research fields that can supply some valuable information for this research project. Analysis in two-dimensional GIS is well documented as well as manipulation of three dimensional datasets for the purpose of visualisation and collision detection in the field of 3D computer graphics.

The lack of any reference material on subjects in the same research field also has consequences for design process of the 3D feature overlay algorithm. Feature overlay algorithms are among the more complex spatial algorithms used in GIS. [Oosterom, 1994] This, together with the higher complexity of algorithms that deal with three dimensional data compared to those that deal with two dimensional data, has some consequences for the project strategy. Roughly, the design process, can be divided in three stages:

1. Formulate a general 'container' feature overlay algorithm
2. Fill the container algorithm with two-dimensional sub-algorithms
3. Translate the two-dimensional sub-algorithms to three-dimensional sub-algorithms

By going through these stages, the highly complex problem of designing a feature overlay algorithm for three-dimensional data was made manageable in two ways. First by splitting the algorithm into sub-algorithms, second by designing an algorithm for two-dimensional data first and translating that to a three-dimensional algorithm.

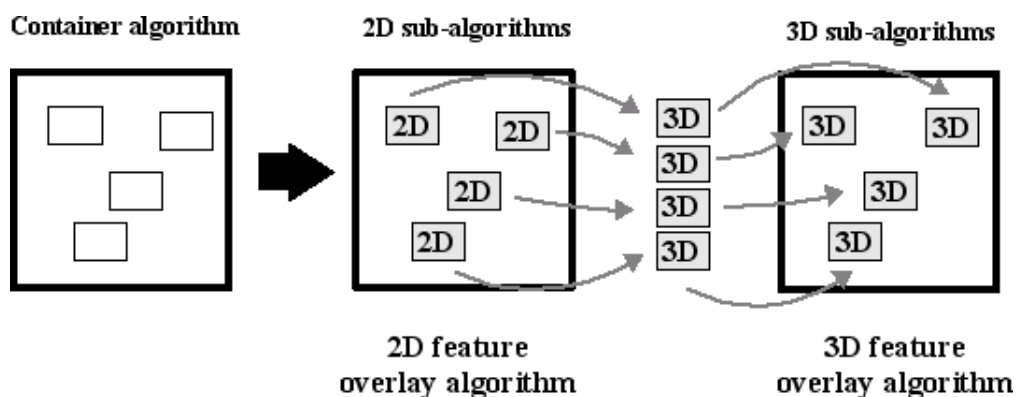


Figure 2.2: the stages of the research project

In the first stage a container algorithm is defined. This is a general description of the steps that need to be taken when features are being overlayed. Later on in the design process, the container is filled with sub-algorithms that, when put together, make up the feature overlay algorithm. By doing this, the complex problem of designing a feature overlay algorithm is broken down into simpler sub-problems.

In the second stage the container algorithm is filled with two-dimensional sub-algorithms. These two-dimensional algorithms can be designed and implemented more easily than their three dimensional versions. In this way algorithm concepts can be developed and tested faster. Any problems that arise during the design or implementation of the two dimensional algorithms can also be solved more easily. The knowledge gained by doing this can then later be used to solve similar problems in the more complex three-dimensional counterparts of these two-dimensional algorithms.

The design of the two dimensional feature overlay algorithm has to be done with the migration to its three dimensional version in step three in mind. In order to make this possible there are two aspects of the design process that need attention. First, the container algorithm (i.e. the general description of feature overlay algorithm) has to be thought through before any other design work is done for either the two dimensional or the three dimensional algorithm. When the container algorithm needs to be changed drastically when migrating later on in the design process the profits gained by following the modular approach are largely nullified. Secondly, the sub-algorithms that are chosen for the two dimensional algorithm need to be chosen in such a way that the translation of the sub-algorithm to the three dimensional case is possible. Preferably, this translation should be easy and insightful as well. This because of the limited timespan of the project.

Implementing the two dimensional algorithm serves three goals. First, by implementing it the parts of the algorithm that are sensitive to floating point errors could be tested. Secondly, seeing things happen or trying to realise them stimulates the process of finding solutions to design problems. Finally, the results of the implementation can be used to visually check the results of the algorithm. Originally the implementation of the three dimensional algorithm was also one of the goals of the research project. The implementation of the two-dimensional algorithms would then also be used to become familiar with the development environment. During the implementation of the two dimensional algorithm it became clear that the implementation of the three dimensional algorithm was not feasible within the time available to the project.

In the third stage, the two-dimensional sub-algorithms are translated to three dimensional sub-algorithms that perform the same task. These three-dimensional sub-algorithms will then be inserted into the container algorithm to form the three-dimensional feature overlay algorithm. During the translation of the sub-algorithms the two-dimensional feature overlay algorithm is implemented using the tools described in section 2.3.

2.3 Project resources

The feature overlay algorithm itself has been designed without the aid of computer programs. A couple of brainstorm sessions and a lot of sketches and try-outs on paper were used to come to the theoretical design of the feature overlay algorithm based on a tetrahedron data structure that is described in this thesis. Software was needed for the implementation of the algorithm, because building an entire application from scratch would have been too time consuming. To aid the implementation the following four tools were considered:

- Computational Geometry Algorithms Library (CGAL)
- Java3D programming libraries
- DirectX programming libraries
- VRML builders

All four of the tools can handle three-dimensional data in some way. A closer examination of the tools showed that the latter three focussed mainly on the visualisation of three dimensional data and less on the manipulation of it. The data structures that were available with these tools also supported the visualisation aspect better than the manipulation aspect. With the first one, CGAL, this was just the other way around. CGAL contained a vast amount of data structures and functions for creating and manipulating those data structures. In order to shorten the implementation time, as much functionality of CGAL as possible was to be used for the implementation of the feature overlay algorithm.

CGAL is a collaborative effort of several computational geometry research sites in Europe and Israel. The goal is to make the most important of the solutions and methods developed in computational geometry available to users in industry and academia in a C++ library. [www.cgal.org]

At the time the feature overlay algorithm was implemented, CGAL itself had no functionality for

visualising the results of the feature overlay algorithm during testing and after its complete implementation. It did however support a few different means of communicating with other libraries which could visualise the data that CGAL produces. A full list of these libraries can be found in the CGAL documentation. The library chosen for this project was the Qt library. This library was chosen because it was relatively easy to use and it was the first library that produced on-screen results. During the implementation of the feature overlay algorithm it became clear that two other libraries were also needed. The entire implementation was finally created using the following libraries:

- CGAL
- Qt
- Library of Efficient Data types and Algorithms (LEDA)
- Standard Template Library (STL)

Qt (not an abbreviation) is a complete C++ application development framework, which includes a class library and tools for multi-platform development and internationalisation. Its main use is to design graphical user interfaces for applications. This is also what it is used for during the development of the feature overlay algorithm. The CGAL library already contains some classes for mesh visualization with Qt, so not much extra programming is needed to present visual information during the implementation of the algorithm.

The LEDA library provides an exact number type and functions for performing exact computations. At the time of the implementation CGAL did not provide any way to do exact calculations itself, so this library is used to prevent problems with floating point numbers. A more detailed description of the LEDA library can be found in [Mehlhorn, 1999].

The STL library, or Standard Template Library, is a general-purpose library of generic algorithms and data structures. CGAL uses some of the principles applied in this library and some of the algorithms provided by STL were also useful for the implementation of the feature overlay algorithm. Iterators (and circulators) are used to process lists of features or parts of features. CGAL also uses iterators for performing various operations on triangulations. The vector sequence container presented in the STL library is used to store preliminary results during the overlay. More information on the STL can be found in the STL reference guide [Musser, 2001].

3. 2D feature overlay algorithm

Visualising and describing three-dimensional processes in such a way that it's easily comprehensible and insightful for the reader is a difficult task. In order to make the explanation of the three-dimensional feature overlay algorithm easier to understand this chapter contains a detailed description of the same algorithm for two-dimensional features. Before continuing with the algorithm itself. Some important terms used during the discussion of the algorithm are elucidated in this section. After this, a general overview of the entire algorithm is given in section 3.2. This is done by globally describing the steps that are taken in the three main phases of the algorithm. Each of these phases, pre-processing, feature insertion, and post-processing, will then be described in more detail in the three following sections.

3.1 Introduction

The feature overlay algorithm described in this chapter overlays features by inserting them into a triangulation. When discussing the process of overlaying features in this manner it is important to distinguish between the properties of the features before insertion, the properties of the features after insertion, and the properties of the triangulation. An elaborate description of each of these properties is given in the rest of this section. As reference, a shorter description of the introduced concepts can also be found in the glossary.

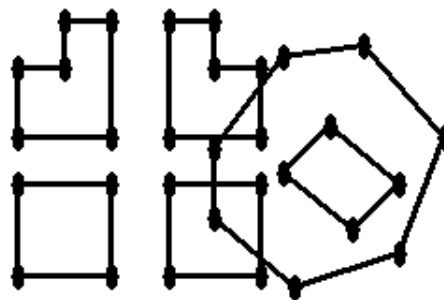


Figure 3.1: a feature set

A feature set is a collection of features which are all available at the moment the feature overlay algorithm starts. Features in these feature sets can come from different sources such as different tables in the same database, different databases, text-based files and map-layers, as long as all the features can be accessed by the feature overlay algorithm at any time after the algorithm has started.

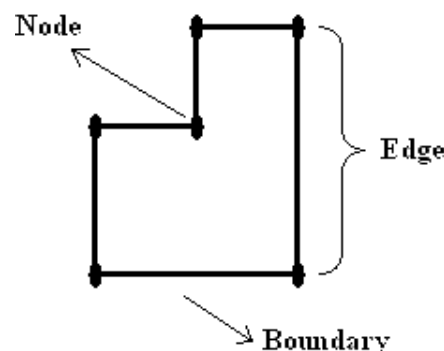


Figure 3.2: feature properties

Features in a feature set have three types of properties: geometrical, topological, and informative properties. The informative properties are defined by the system designer and only the unique identification of the feature is used by the feature overlay algorithm. The shape of a feature is described by its boundary. The boundary of a feature consists of a number of nodes connected by straight line segments called edges. This is visualised in figure 3.2. When the algorithm is going to overlay a feature, the edges that make up the boundary will be split into sections. The exact definition

of sections and a description of the way they are used during the overlay can be found in section 3.4. The topology, which describes the relation of the feature to its surroundings, is used by the algorithm to distinguish between which side of the boundary lies inside the feature and which side lies outside of the feature. All the other topological properties are ignored by the algorithm.

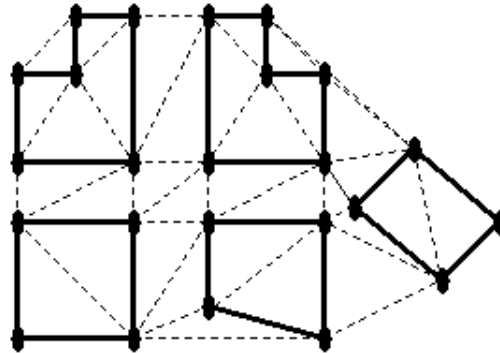


Figure 3.3: an overlay triangulation

The overlay triangulation is the triangulation which is used by the algorithm to overlay the features from the feature set. An example of an overlay triangulation can be seen in figure 3.3. The triangulation consists of a collection of vertices connected by edges. These edges form a network of triangles where each triangular shaped area is denoted by the term face. In literature the intersections of edges in a triangulation are usually referred to as nodes. Here the term vertex (pl. vertices) is used for two reasons. First, this makes it easy to distinguish between the nodes as part of the boundary and the nodes as part of the triangulation. Second, the term vertex is used by the geometry library used for the implementation of the two-dimensional feature overlay algorithm. In order to maintain a consistent use of terms throughout the thesis and avoid confusion the nodes of the target triangulation are denoted as vertices.

When features are inserted into this overlay triangulation a lot of the topological and informative information is ignored. Features are no longer defined as separate entities, but as parts of the overlay triangulation. When discussing features as part of the overlay triangulation terms such as boundary and section don't apply anymore. In stead the terms displayed in figure 3.4 are used. The edges and sections that make up the boundary of the original feature are addressed as constrained edges; depicted as solid lines in figure 3.4. All other edges in the overlay triangulation are called unconstrained edges; depicted as dotted lines in figure 3.4. During the insertion of the features into the target triangulation the constrained edges are the only entities that contain information (informative as well as topological) about the features they belong to. Furthermore, the original input edges are not subdivided due to the triangulation. That is, the triangulation is a constrained triangulation and not a conformal triangulation where vertices are allowed at other locations than intersections of constrained edges. An overlay of features using a conformal triangulation is described by [Cavalcanti, 1999].

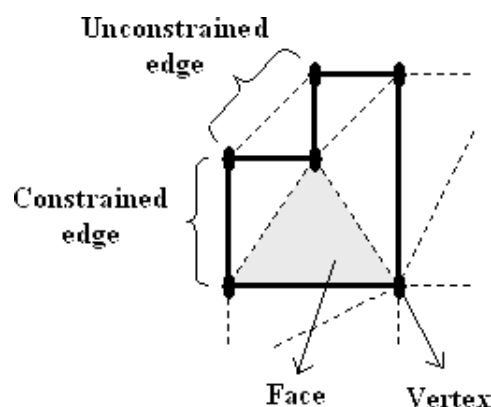


Figure 3.4: overlay triangulation properties

When referring to the geometry of the elements that make up the triangulation the following terms are used: for the geometry of the face the term triangle is used, for the geometry of the edge the term segment is used, and for the geometry of the vertex the term point is used. This is visualized in figure 3.5. The relationships between the geometry and the feature properties is shown in figure 3.6.

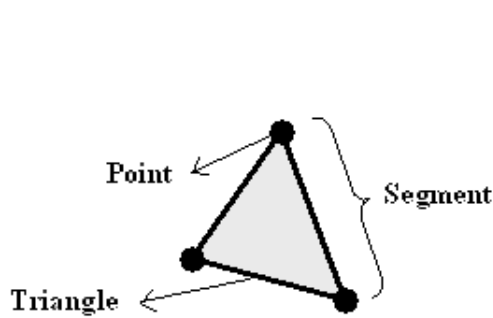


Figure 3.5: geometrical properties

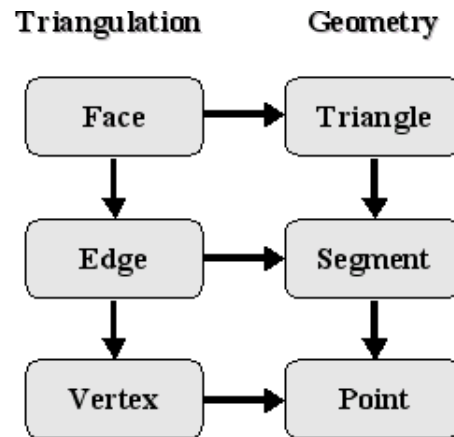


Figure 3.6: relationship between terms

3.2 Algorithm overview

In the project strategy the first step in the process of designing the actual overlay algorithm was to devise a container algorithm, i.e. describe the general steps that the overlay algorithm would have to take. This container algorithm would then be filled with other algorithms to perform the tasks described in every step of the container algorithm. Analogue to the design strategy the container algorithm will be discussed here first. This will make it easier to place the details discussed later on in this chapter in the context of the entire overlay algorithm.

Before any steps can be defined, it must first be clear what the algorithm needs to process and which results the algorithm should produce. In the introduction of this thesis conditions on both the input and the output of the final algorithm have been laid down. The input must consist of features which have volume, don't have holes in them and can be triangulated without having to change the features themselves or the tetrahedral network they are inserted into, e.g. adding Steiner or boundary points. The output has to be a tetrahedral network containing all overlayed features that were offered to the overlay algorithm as input. Figure 3.7 shows this starting point in a schema.

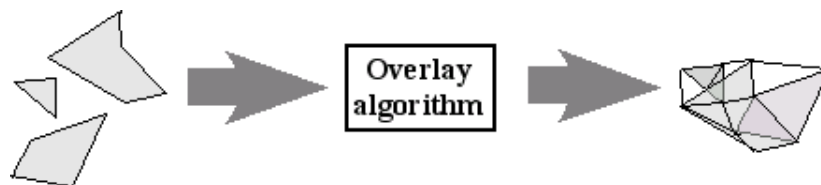


Figure 3.7: algorithm input and output

The features presented to the algorithm need to satisfy certain conditions. The exact nature of these conditions is presented in the next section. For now it is sufficient to know that in most cases the features which need to be overlayed. So before the algorithm can start overlaying features the operator has to pre-process the features that need to be overlayed. When the algorithm is finished overlaying the pre-processed features, the direct result of the overlay will most likely not be sufficient in most cases. So depending on the exact wishes of the operator, the result of the feature will have to be post-processed. Although the pre and post-processing phase aren't strictly part of the overlay algorithm itself due to the variant nature of these phases, they will be discussed here because of the influence the overlay algorithm has on them.

In the pre-processing phase the feature data is selected and prepared for the overlay. Before any features can be overlayed the operator has to select the features that need to be overlayed. The algorithm overlays features iteratively and doesn't distinguish between features by shape, relative or absolute position, the data source they originated from, or the order in which they are presented to the algorithm. This gives the operator the possibility to select the source features very accurately and also makes it possible to modify the set of source features very easily, even after the algorithm has been started.

After the overlay algorithm has overlayed all the features, only the entities which contain feature information are the constrained edges of the overlay triangulation. In the post-processing phase the topology which is required by the operator will have to be restored or created. In the sections where the post-processing phase is discussed, an example is worked out where identifiers are added to all the faces of the overlay triangulation.



Figure 3.8: phases of the algorithm

After the source data has been pre-processed and a feature set is created, the overlay algorithm can start processing the features. This is an iterative process, i.e. one feature at a time. After it has completely finished processing the feature it will start with the next one. Figure 3.9 shows a close-up of the 'Overlay features' phase with the iteration captured within the schema. On the left side, features are retrieved from the feature set. On the right side they are inserted into the overlay triangulation.



Figure 3.9: iterative process of the 'Overlay features' phase

The insertion of the features into the overlay triangulation is also an iterative process. All the edges of a feature are processed iteratively. The algorithm could just iterate over the edges of the features in the feature set, i.e. edges from different features could be processed indifferently. This is not a good idea however. Why this is not a good idea will be explained in section 3.3. So in the schema in figure 3.10 all features entering from the left are features from the same feature. In the right side the edges have been inserted into the overlay triangulation.



Figure 3.10: iterative process of inserting features

Inserting edges into the overlay triangulation finally is also an iterative process. Each edge is divided into sections. The algorithm then processes each of these sections. Figure 3.11 shows how this fits into a schema. The sections are inserted into the overlay triangulation by adding vertices to the triangulation and flipping non-constrained edges of the triangulation. This process also results in the division of some constrained edges that are already present in the triangulation into more sections. How this is done exactly is described in section 3.4. After all sections of all edges of all features are inserted into the overlay triangulation the entire feature set is overlayed and the post-processing phase can start.

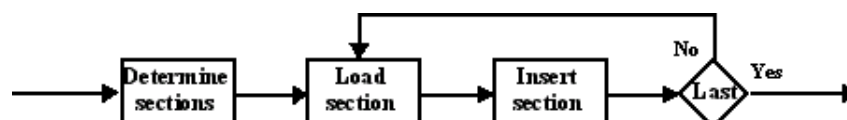


Figure 3.11: iterative process of inserting edges

When putting all the previous schema together the schema of figure 3.12 is produced. This schema shows the entire algorithm from input data to final result.

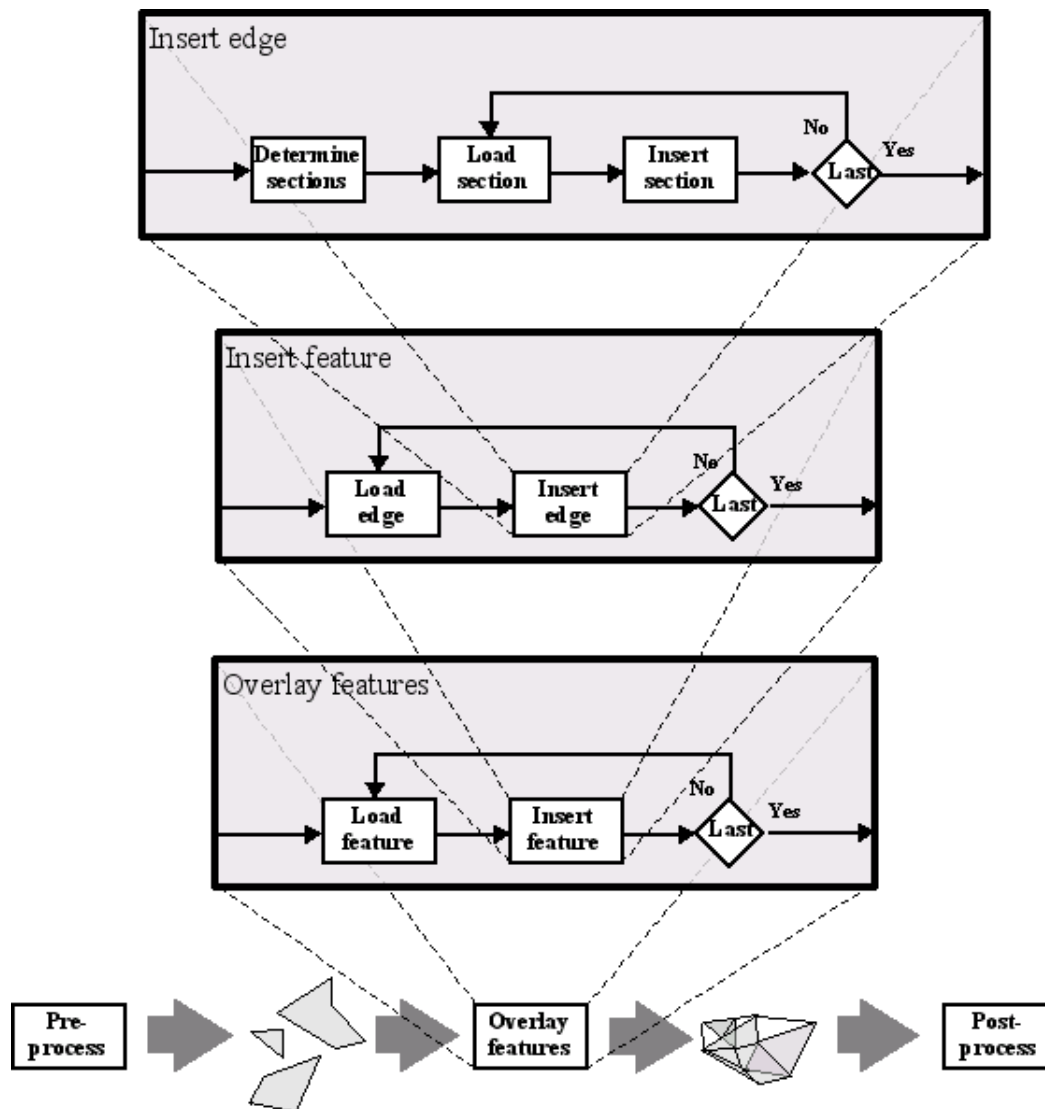


Figure 3.12: entire overlay algorithm

3.3 Phase 1: pre-processing

The pre-processing phase isn't strictly part of the feature overlay algorithm itself for one simple reason: it is impossible to design an algorithm that could perform the necessary operations on every data source to make the feature overlay algorithm applicable to it. It is necessary to throw some light upon a few aspects of the pre-processing phase in order to make it possible for the operator to design his own pre-processing algorithms. The operator can benefit from the pre-processing scenarios which are described in sub-section 3.3.1. By knowing when to choose for a certain pre-processing scenario the algorithm can process the feature data more efficiently. When the operator has chosen one of the scenarios he needs to know the exact requirements that the algorithm has for the feature data in order to be able to process it. These will be described in sub-section 3.3.2.

3.3.1 Different starting situations

The feature overlay algorithm requires that the feature data presented to it meets certain conditions. There are many different ways in which to store spatial data, ranging from complicated topological databases to simple text files. It is unlikely that the feature data of many of these sources meets all the conditions that the feature overlay algorithm has for the feature data. So before the algorithm can start

with overlaying features some form of pre-processing is needed in order to make the feature data useable for the algorithm. With so many different sources of feature data it would be impossible to describe the pre-processing steps that need to be taken for all these sources. It is possible to group all the sources of the feature data into two distinct categories. Both of these categories have their own pre-processing scenario.

In both scenarios the same sequence of steps is followed. First the operator selects features from different data sources. These data sources are for example different tables in a database, tables from different databases, text files, or files produced by other applications and algorithms. After the selection the data of each of the selected features is converted into a format which is useable by the feature overlay algorithm and the converted feature data is added to the feature set that needs to be processed. The features in the feature set are processed and the algorithm produces a result; a triangulation containing an overlay of all the features from the feature set. The scenarios differ in the kind of data conversion which is needed and the order in which the feature data is handed to the algorithm.

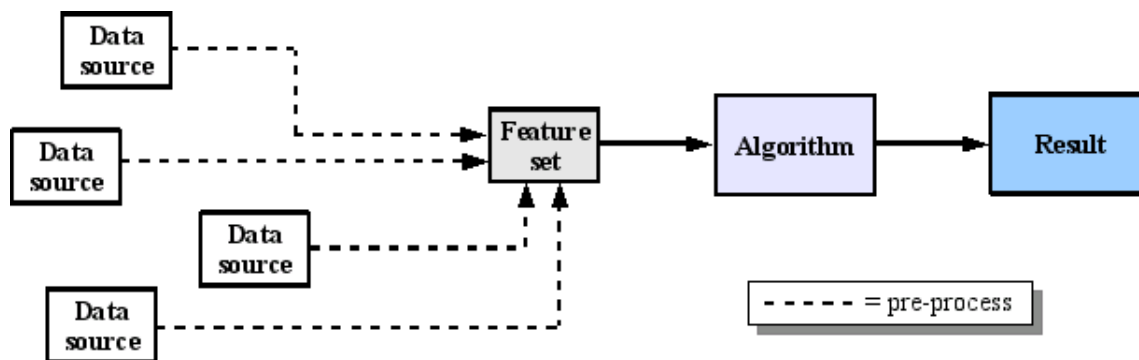


Figure 3.13: scenario with no knowledge about overlapping features

The first scenario is the standard scenario. A flow diagram of this scenario can be seen in figure 3.13. In this scenario the operator performs some pre-processing on each of the selections of features from each data source. As far as the feature overlay algorithm is concerned, there are no restrictions on the number of data sources or type of data structure used for each of these data sources. It is the task of the operator to ensure that each of these data sources gets processed to fulfill the conditions of the feature overlay algorithm. After the features from a source have been processed by the operator, they are added to the feature set which in turn will be processed by the feature overlay algorithm. The order in which the features are added to the feature set has no effect on the final result of the overlay. Although the overlay triangulation might be slightly different when the same features are added in a different order, the constrained edges of the triangulation will be the same.

This scenario is best applied to situations where the operator has no prior knowledge about the relative position of the features within the different data sources; i.e. it is unknown if features from the same data source overlap or touch. It is also the scenario to use when it is certain that each of the data sources contain overlapping features. Very often there is information available on the relative position of features within a data source however. In these cases it is much more efficient to follow the second scenario.

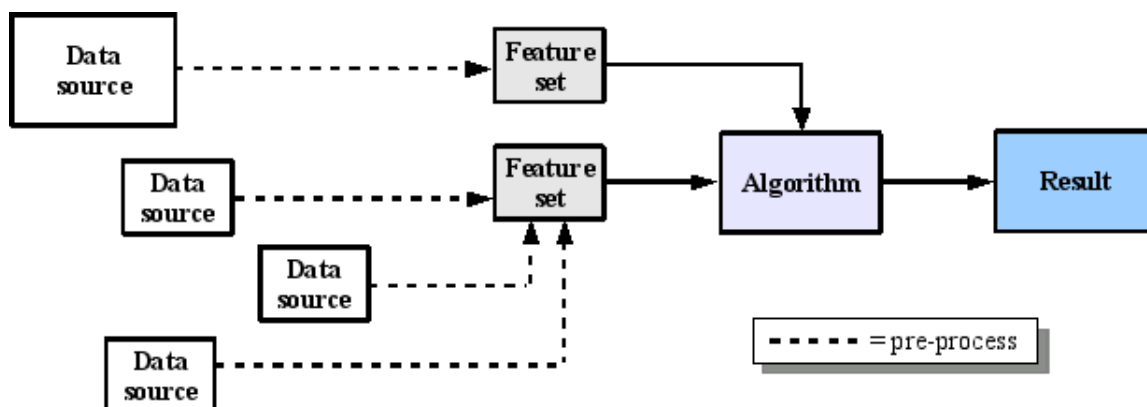


Figure 3.14: scenario with at least one data source without overlapping features

In the second scenario the information about the relative position of features is used to make the algorithm more efficient. A flow diagram of this scenario can be seen in figure 3.14. As can be seen in the flow diagram, the number of dashed lines hasn't decreased. The same amount of pre-processing has to be done by the operator as in scenario one. The knowledge about the position of the features doesn't make the pre-processing more efficient. By presenting the pre-processed feature data to the feature overlay algorithm in a particular order the overlay algorithm itself can be made more efficient however.

First the operator should pre-process the data source which contains the non-overlapping, non-touching features. The features that result from this process should then be passed to the feature overlay algorithm with the indication that the presented features do not overlap. The feature overlay algorithm will then start to add these features to the overlay triangulation and while inserting each feature it will skip the checks which are needed to detect and process overlapping features. After all the features of the first feature set have been processed the algorithm can continue with the features from the second feature set in the same way as it processed the features in the first scenario.

The entire algorithm is made up of three nested iterations. After each successfully inserted feature, the preliminary result is a valid triangulation that describes the overlay of all features processed upto that point. Therefore adding features to the result at a later time doesn't require any pre-processing of the previous result.

3.3.2 The feature data format

All features which need to be overlayed by the feature overlay algorithm need to be passed to the algorithm in a specific format. In the general overview of the algorithm the iterative nature of the algorithm was explained. The conditions for the data format of the features that need to be overlayed all relate to the elements of the iterations: the feature itself and its edges. For overlaying features according to the first scenario described in the previous sub-section there are three conditions that need to be met:

1. the data has to be presented as a list of edges or list of sequential nodes per feature
2. the list of edges or the list of sequential nodes must have a reference to a specific feature
3. the edges or the lists of nodes need to be directed counter clockwise

The algorithm processes the features by iterating over the edges of the features. Because of this, the features need to be presented in a data format that allows the algorithm to identify the separate edges without omission and without doublings. The first condition ensures that the algorithm can access each edge of a feature independently. An example of both the list of edges and the list of sequential nodes can be seen in figure 3.15. It's of no consequence for the algorithm which method is used. The choice for either method will depend on the ease with which the source data can be converted to either of these data structures.

The list of edges consists of pairs of start and end nodes for each edge in the feature. Each node appears twice in the list, once as start node and once as end node. In the example there are six edges so there are also six pairs of nodes. The list of sequential nodes is created by starting at an arbitrary node, which is added to the list. Next, the boundary of the feature is followed in one direction and each time a node is passed along the way, that node is added to the list. This continues until the node at which the process started is added to the list again. All nodes of the feature appear only once in the list except for the start node. An edge is constructed by taking a node $n(i)$, the start node, and the next node in the list $n(i+1)$, the end node. By adding the first node twice there is an $n(i+1)$ for all edges when loading the nodes. In this way the algorithm doesn't need to treat the last edge as a special case. The second entry of the first node can also be used as an indicator for the end of the feature data of that feature.

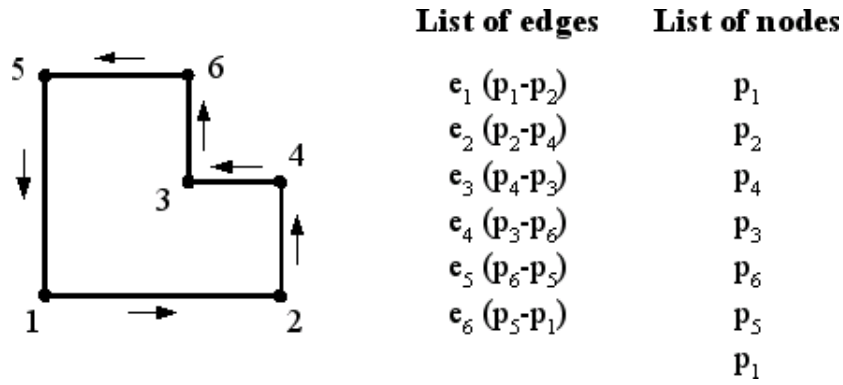


Figure 3.15: feature defined by a list of directed edges and a directed list of nodes

The second condition states that a unique identifier should be available with each feature. The overlay algorithm uses this identifier to keep track of which edges belong to which feature during the overlay. This identifier is also needed during the post-processing phase. The overlay algorithm ignores all the properties of the features which it doesn't need for the overlay. It just adds the unique identifier to each of the constrained edges and faces of the overlay triangulation. Post-processing algorithms can then use these identifiers to reconnect the ignored properties or add new properties to the overlaid features.

The third condition needs to be met to make it possible for the overlay algorithm to reconstruct the feature boundaries after the features have been inserted into the overlay triangulation. The sub-algorithm that reconstructs the features in the overlay triangulation assumes that the interior of a feature is on the left hand side of a constrained edge. Normal edges don't have a left-hand side or a right hand side. The left hand side and the right hand side depend completely on the direction along the edge you are looking at. Directed edges have a start node and an end node. When standing on the start node and looking towards the end node there is a left hand side and a right hand side.

For the list of edges this means that the first of the two nodes is the start node and the second of the two nodes is the end node. These should be chosen such that the interior of the feature is on the left-hand side of the edge. For the list of sequential nodes this means that the boundary of the feature should always be followed in a counter clockwise direction. This way, the interior of the feature is always on the left-hand side when traversing the nodes in the list. Both the edges in the list of edges and the lists of nodes in figure 3.15 are directed with the interior of the feature on the left-hand side.

The fourth and last condition is also concerned with the reconstruction of the features. The reconstruction algorithm can only reconstruct the features within the overlay triangulation if all the sections of that feature are present in the overlay triangulation as constrained edges. Strictly speaking the order in which the edges are processed doesn't have any consequences for the result after insertion of all the edges. Edges of different features can be processed in any order as long as all edges of all processed features are present in the overlay triangulation before the post-processing phase starts. For the stability and robustness of the algorithm it is wise however to make sure all edges are present before a feature is added to the feature set which is processed by the overlay algorithm.

3.4 Phase 2: inserting features

When the feature data meets conditions described in the previous section, the features from the feature set can be inserted in the overlay triangulation. As with the complete algorithm, this also is an iterative process. When inserting a feature, all its edges are processed one at a time until all edges have been inserted into the overlay triangulation. The edges in turn are also processed iteratively. When an edge is inserted, it is split up into several sections. These sections are added to the overlay triangulation as constrained edges. The insertion of these sections also splits edges that were already inserted into the overlay triangulation into more sections. After all the sections of one edge have been added to the overlay triangulation all the new constrained edges combined form the inserted edge in the target triangulation. The constrained edges are created through the use of flips.

3.4.1 Flipping

Flipping edges is an essential part of the process of inserting features into the target triangulation, i.e. by flipping non-constrained edges sections of feature edges are created in the overlay triangulation. To be able to understand the insertion of a feature edge in the overlay triangulation it is essential to understand what flipping is. An example of a flip is shown in figure 3.16. It shows two triangulations of the same set of points. The triangulation on the right was created by flipping edge e_1 in the triangulation on the left.

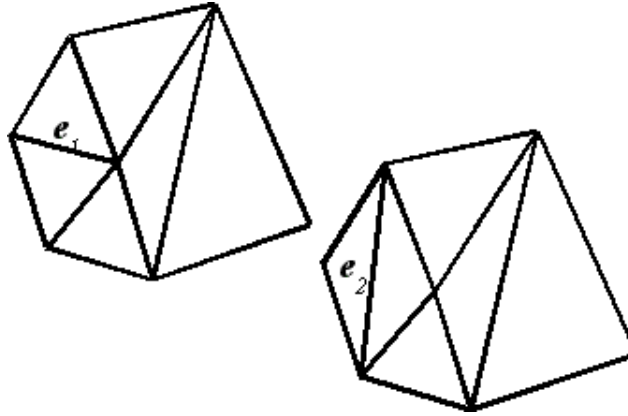


Figure 3.16: triangulation before and after flipping edge e_1

The flip of this edge is explained in Figure 3.17. It shows a close-up of the edge e_1 and its immediate surroundings from the triangulation of Figure 3.16. The edge e_1 that needs to be flipped can be seen as the diagonal of the quadrangle C in which it is contained. $C = \delta(t_l \cup t_r)$ is the boundary of t_l and t_r , the triangles to the left and to the right of e_1 respectively. The flip operation is nothing more than the removal of edge e_1 and replacing it with edge e_2 , the other diagonal of C .

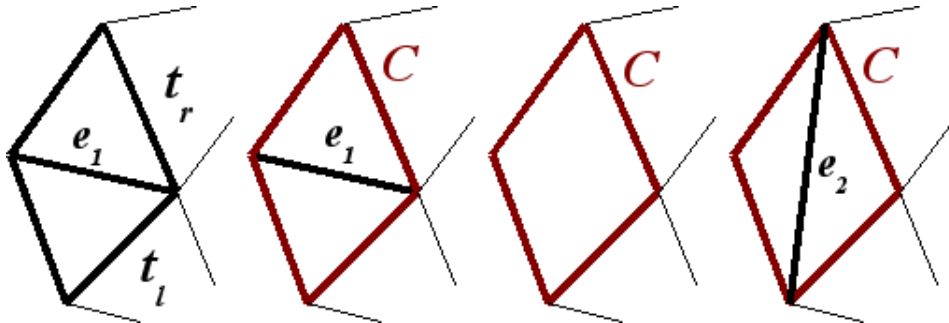


Figure 3.17: flipping edge e_1

An edge can only be flipped when the triangulation is still valid after the flip is made, i.e. edges of the triangulation may only intersect each other at the vertices. This is not always the case however. Figure 3.18 illustrates a situation where the triangulation is no longer valid after edge e has been flipped. These situations occur when the second diagonal of the quadrangle C that encloses the flipped edge e lies outside of C . It is only possible to retain a valid triangulation while flipping an edge when C is convex [Hurtado, 1996].

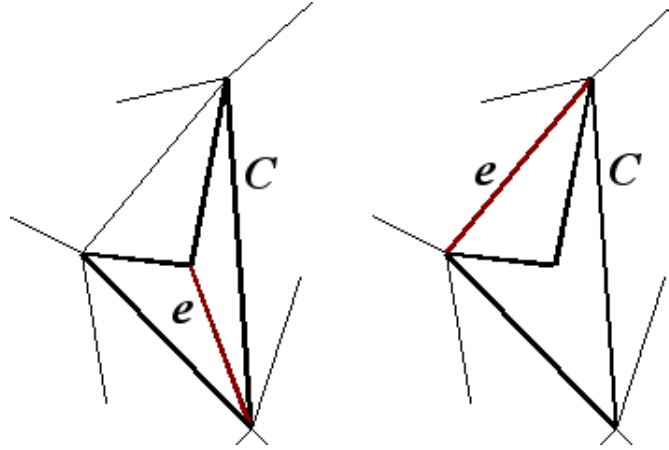


Figure 3.18: illegal flip

Some extend the definition of flipping to also include insertion and deletion of vertices [Shewchuk, 2003]. These flips are not used by the 2D version of the algorithm. All the flips made when overlaying 2D features are done by changing the diagonal of a quadrangle C .

3.4.2 Inserting edges

The overlay algorithm inserts edges into the overlay triangulation in three steps. First it prepares the overlay triangulation so it can determine the sections of the edge. After that it determines the sections. And finally constrained edges that represent the sections will be created in the overlay triangulation through the means of flipping non-constrained edges. Sections are determined by projecting the edge onto the overlay triangulation and searching for intersections. In order for this to work efficiently the algorithm needs to know where to project the edge onto the overlay triangulation and which part of the triangulation needs to be tested for intersections. In the first step these two problems will be solved.

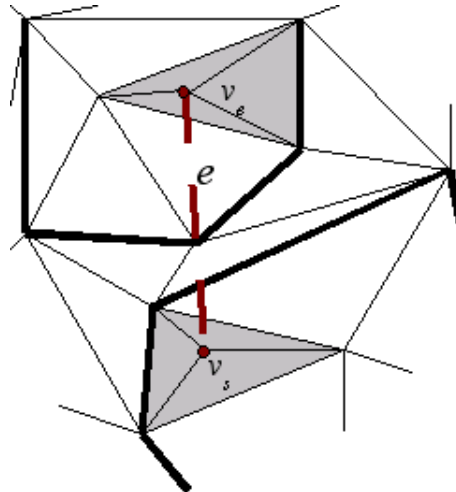


Figure 3.19: inserting the edges start and end node

The first problem, locating the projection of the edge within the overlay triangulation, is solved by inserting the start and end node of the edge into the overlay triangulation. This situation is shown in figure 3.19. The figure shows part of an overlay triangulation with an edge e projected onto it. The start and end node have been inserted into the triangulation as vertices v_s and v_e . The algorithm used to insert the start and end node of the edge into the overlay triangulation depends heavily on the data structure that is used for the overlay triangulation. The insertion of a node into the triangulation is done in two steps. First a point location procedure is performed to locate the face of the triangulation

in which the node will be inserted. After the face has been located, a new vertex is created at the co-ordinates of the node and the face in which the vertex was created is split into three new faces. In figure 3.19 these faces are shown in grey. The overlay algorithm can then use the newly created vertex and its surrounding faces to locate the projection of the edge within the overlay triangulation. Although the scenario described here is the most likely one to occur, new vertices aren't always inserted in the interior of faces, but could also be inserted into edges or onto existing vertices. These exceptions are discussed in sub-section 3.4.5.

There are many different methods of locating points within a data structure ranging from simple methods using ordered lists of co-ordinates to more elaborate methods such as planar subdivision methods and methods that utilise binary search trees [Snoeyink,1997][Mucke,1996]. The choice for the algorithm that best suits the task of inserting points into the overlay triangulation depends on the exact implementation of the overlay triangulation [Goodrich,1997]. In the implementation of the algorithm with CGAL, a point location algorithm from the family of triangle walk point location algorithms was used. A short explanation of the principle of the algorithm used can be found in subsection 3.4.4.

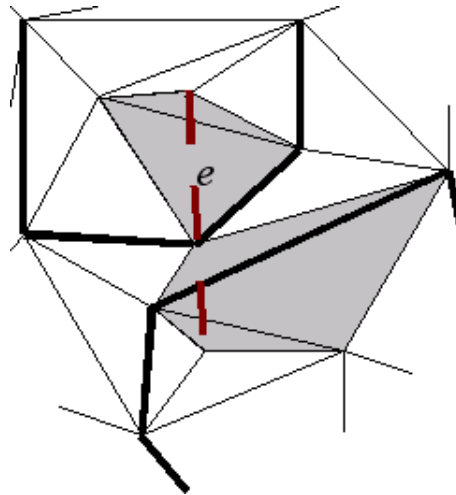


Figure 3.20: locating the faces that intersect with the projection of the edge

When the vertices of the projected start and end nodes of the edge are created the overlay algorithm knows where to start determining sections and where to stop. It doesn't yet know which faces, edges and vertices of the overlay triangulation lay between these two vertices and intersect with the projection of the edge. An algorithm similar to the point location algorithm used to insert new vertices into the triangulation can be used to determine the (non)constrained edges and vertices that intersect with the projection of the edge, i.e. traverse the triangulation by following the direction of the new edge.

When determining the sections and reconstructing them, extra vertices will be added to the target triangulation. As a result existing faces will be changed and new faces will be created. Since it is not known beforehand which of the faces will be modified or which will be created, it is of no use to actually determine all the elements that intersect with the projection of the edge at this point. Therefore the algorithm that can locate the elements will determine the intersecting elements on the fly and only needs to be initialised at this point. The algorithm that locates intersecting elements will be discussed in more detail in sub-section 3.4.4 when the step of inserting sections into the overlay triangulation is explained. The elements that would be found by this algorithm in the example of figure 3.19 can be seen in figure 3.20 in grey.

3.4.3 Sections explained

When the start and end point of the new edge have been inserted into the overlay triangulation and all the faces of the overlay triangulation that intersect with the new edge are identified, the sections of the new edge can be determined. Each edge e consists of i sections e_i so that $\bigcup e_i = e$ and sections of the same edge do not overlap. The amount of sections per edge is determined by the location of the edge in the target triangulation.

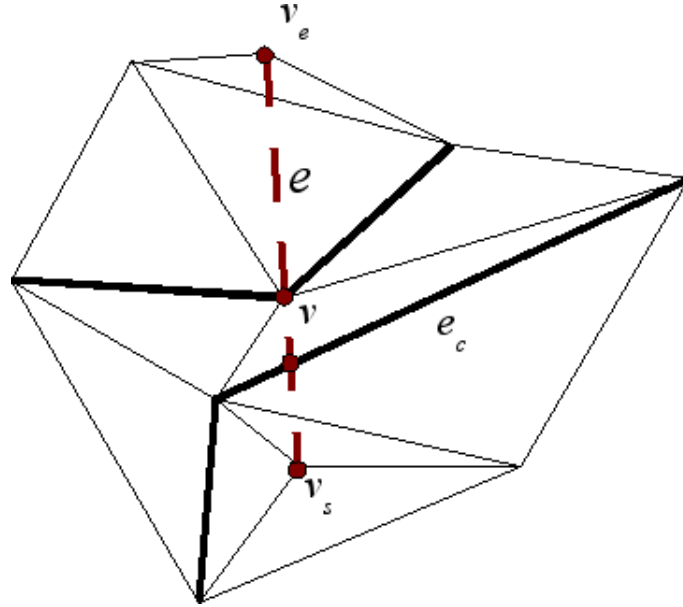


Figure 3.21: dividing an edge into sections

Each section starts and ends at an intersection between the edge and a constraint in the overlay triangulation. Such a constraint is either a vertex, or a constrained edge. Since the start and end nodes of each edge have been inserted into the target triangulation as vertices, all edges consist of at least one section, i.e. the start and end nodes of the edge are present in the overlay triangulation as constraints. Figure 3.21 shows part of a target triangulation and an edge that needs to be inserted into the triangulation. The start and end points have already been inserted. The new edge intersects with one constrained edge, two non-constrained edges and one vertex. The two constraints divide this edge into three sections. The first section starts at the vertex v_s representing the start node of the edge and ends where the edge intersects with the constrained edge e_c . The second section starts where the edge intersects with the constrained edge e_c and ends where the edge goes through vertex v . The third section starts at vertex v and ends at the vertex v_e representing the end node of the edge.

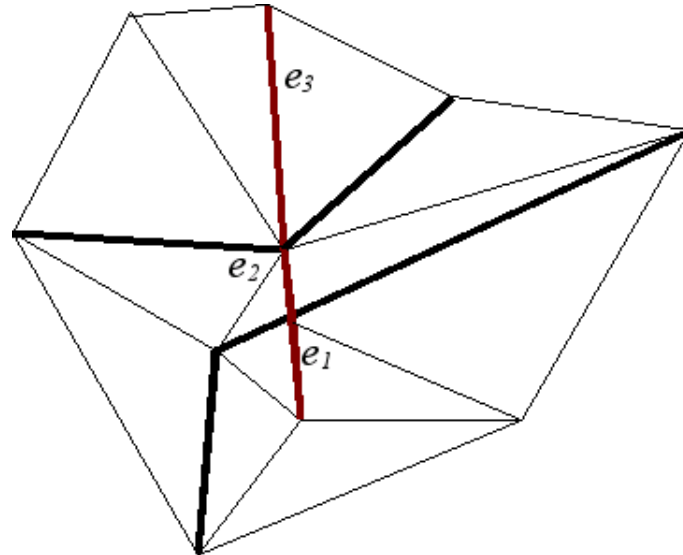


Figure 3.22: sections as constrained edges

After a section has been processed by the algorithm the overlay triangulation will have gained one new constrained edge. This edge is the representation of the section which was processed. When all sections of an edge are processed the edge is effectively inserted into the overlay triangulation. The edge can be reconstructed by combining the constrained edges that represent the sections of the edge. Figure 3.22 shows the overlay triangulation after all three sections from figure 3.21 has been inserted. The three highlighted constrained edges e_1 , e_2 and e_3 correspond with the three sections of the edge as shown in figure 3.21. After an edge has been inserted into the overlay triangulation, the number of

sections that make up the edge is not fixed. It can increase as a result of adding new intersecting edges afterwards. In the example this happens with edge e_c . The visible section is split into two new sections when the new edge is inserted. The increase of the number of sections doesn't have any influence on the shape or length of the edge. When a section is split into two new sections, the union of those two new sections is the same as the section before it was split. (This is not the case in the implementation however, computers don't have infinite precision.)

3.4.4 Processing sections

Sections are processed one at a time starting with the section that contains the starting vertex of the projected edge. Each time a new section needs to be inserted into the overlay triangulation the overlay algorithm takes the following three steps:

1. determine the projection of the section onto the overlay triangulation
2. add the end vertex of the section that was found to the overlay triangulation
3. flip non-constrained edges of the overlay triangulation that intersect with the projection

In the first step the algorithm gathers the information it needs for the second and third step: the end point of the section and the part of the overlay triangulation that needs to be modified. It does this with the triangle walk algorithm mentioned earlier. The line that will be walked along was determined earlier during the preparations for inserting the edge. It starts in the vertex that represents the start node of the edge that is being processed and it goes through the vertex that represents the end node of the edge that is being processed.

When the line is followed into a face (i.e. the algorithm stepped/walked into the face) there are three possible faces it can continue in. It can continue in the face that shares the edge of the current face in clockwise direction, it can continue in the face that shares the edge of the current face in counter clockwise direction or it can continue through the vertex opposite of the edge it entered the face through. The third possibility is considered an exception and will be discussed in the next sub-section. The other two are determined by comparing the direction of the line that the algorithm is walking along with a second line that starts at the same vertex as the walking line and goes through the vertex opposite of the edge through which the algorithm stepped into the face. How this works will become clear in the following example.

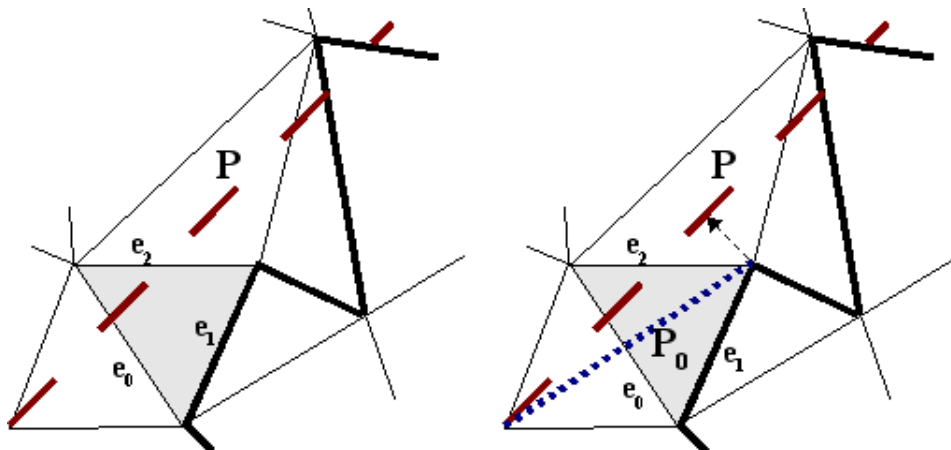


Figure 3.23: determining sections using a triangle walk algorithm

Figure 3.23 shows part of an overlay triangulation. The triangle walk algorithm is processing the projection P of an edge represented by the red dashed line and has walked into the grey face. The edge that it has just passed (edge e_0) is a non-constrained edge, so the end of the section that it's looking for hasn't been found yet. It now needs to find out which face will be the next face under P . The next face is either the face sharing edge e_1 with the grey face, or the face sharing edge e_2 with the grey face.

The algorithm does this by constructing a second line from the start vertex of P to the vertex opposite of edge e_0 . This second line P_0 is shown in figure 3.23 in blue. It then determines if the

projection of P_0 onto P is positive or negative. If the projection was positive, the next face will be the face sharing edge e_2 with the grey edge. If the projection was negative, the next face will be the face sharing edge e_1 with the grey face. In the example the projection is positive, so the next face will be the one sharing edge e_2 . Note that only the sign of the projection is important, not the exact projection itself. For this two dimensional case the sign of the projection can be obtained by determining the determinant of a two by two matrix [O'Rourke, 1998].

After walking along a certain number of faces, the triangle walk algorithm will come across a constraint. If the constraint was a vertex, the algorithm will skip the second step (the vertex is already inserted). In most cases however the constraint will be a constrained edge. In those cases the exact intersection of the projected edge and the constrained edge has to be calculated. This intersection point will then be inserted into the overlay triangulation, creating a vertex inside the constrained edge. The effect for the overlay triangulation will be that the each of the two faces on both sides of the constrained edge will be split into two new faces. Figure 3.24 shows this for the example started in figure 3.23.

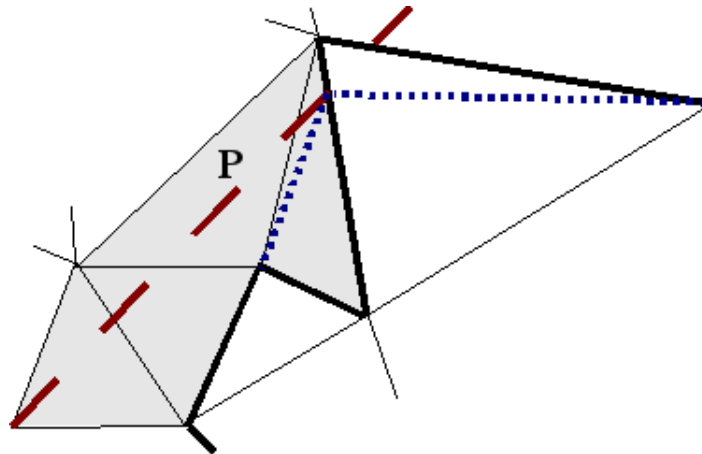


Figure 3.24: inserting the end vertex of a section

All the information which is needed to insert the section is now available. The start vertex and the end vertex have been inserted into the overlay triangulation and all the faces and non-constrained edges that need to be modified are known. For the example these faces are colored grey in figure 3.24. The section is constructed by flipping all the non-constrained edges that intersect with the projection of the section until none of them intersect anymore. In the example three flips are needed to get the desired result. Figure 3.25 shows the results of each of these flips. The blue edge in each step is the edge that is flipped in the next step. After the last flip, the edge that was flipped last is the exact projection of the section onto the overlay triangulation. This edge will then become a constrained edge belonging to the feature of which the edge that is inserted is part of.

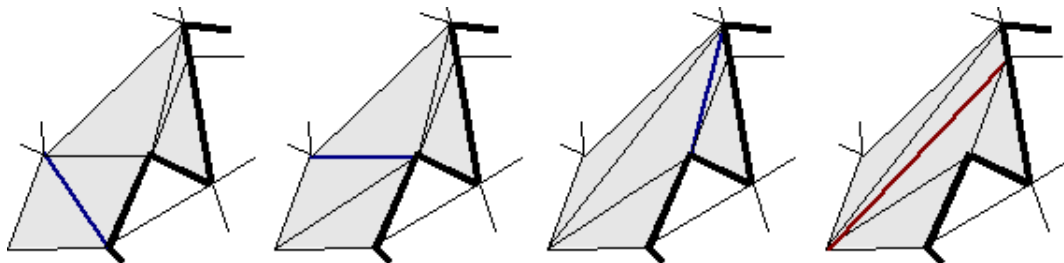


Figure 3.25: flipping non-constrained edges to construct the section

Proof

In the article 'Flipping edges on triangulations' Hurtado proves that if a polygon Q_n has k reflex vertices, then any triangulation of Q_n can be transformed to another triangulation of Q_n with at most $O(n + k^2)$ flips. This proof can be applied to the insertion of sections when:

- a. there is a triangulated polygon Q_n within the overlay triangulation that contains both the start and end vertex of the section that needs to be inserted
- b. that polygon has a triangulation T' that contains the edge that represents the section that needs to be inserted

A triangulation of a polygon Q that contains edge between two vertices v_1 and v_2 only exists if v_2 is visible from v_1 in Q , i.e. a straight line can be drawn inside the boundary of the polygon between v_1 and v_2 without crossing the boundary of the polygon. Figure 3.26 shows both situations. On the left vertices v_2 is visible from v_1 and on the right it is not.

A triangulated polygon that contains the start and end vertex of a section is created by selecting adjacent faces from the target triangulation after the start and end vertex have been inserted. The polygon that is created by taking the union of the faces that were found by the triangle walk algorithm fulfills both conditions that are needed to apply the proof presented by Hurtado. The triangle walk algorithm starts at a face containing the start vertex of the section and stops at a face containing the end vertex of the section. All edges that are crossed when walking from start to end vertex are part of the interior of the polygon created by the union of the found faces, so the end vertex is visible from the start vertex.

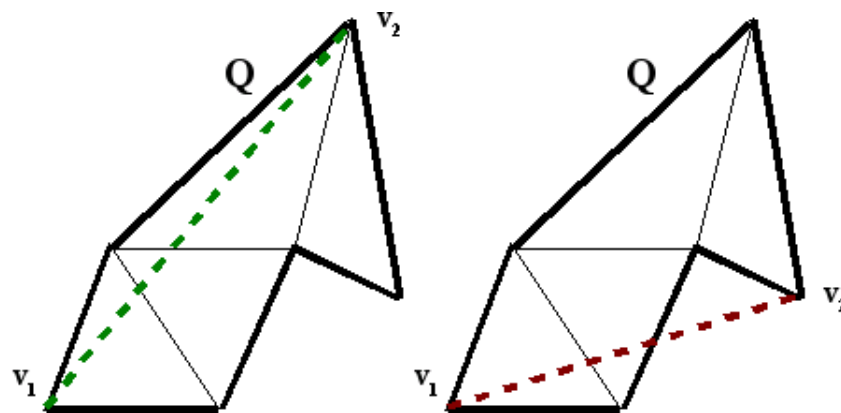


Figure 3.26: visibility of vertices

The time complexity of most triangle walk algorithms is $O(\sqrt{n})$ where n equals the number of vertices in the triangulation that is walked through [Sundareswara, 2003], [Mucke, 1996]. This time complexity is only valid for random point location, i.e. the triangle walk algorithm doesn't have a known triangle to start at. If the triangulation is not indexed in some way, this will be the time complexity of inserting the start and end vertices of edges into the triangulation. The insertion of vertices in edges will more efficient in most cases. The triangulation has a known start location, namely the previously inserted vertex. In these cases n is equal to the number of vertices in the polygon that is used for the construction of a section by flipping edges. [Hurtado, 1999] proves that the time complexity of section reconstruction is $O(n + k^2)$ where n is the number of vertices in the polygon and k is the number of reflex vertices in the polygon. When these two are combined the time complexity of the insertion of a section becomes $O(\sqrt{n}(n + k^2))$.

3.4.5 Special cases and their implications

The feature overlay described here doesn't always go as simple and as smooth as described. As with most algorithms manipulate well defined data structures such as a triangulation with constraints, this algorithm also has to deal with special situations that would normally cause the algorithm to crash or become trapped in an infinite loop. The exceptions described here are all of a logical nature. Implementing the algorithm will be a hazardous undertaking for a second reason as well: the so dreaded computer science problem of (infinite) precision. Questions such as 'When is a vertex located on a line?' and 'When do two faces touch each other?' need to be answered before a stable implementation of any of the exceptions discussed here can be implemented. Answers to these questions will depend on the environment the algorithm is implemented in, the nature of the data, the expected results and so on [Shewchuk, 1996]. When inserting features there are three actions that can raise exceptions:

- inserting vertices into the overlay triangulation
- determining sections
- flipping edges

Inserting vertices

When a new vertex is added to the overlay triangulation it can be located inside a face, on an edge, or on a vertex. These three different situations require different behaviour of the algorithms that update the triangulation. In case of the first situation, a face needs to be split into three new faces. The second situation requires the algorithm to split the two faces adjacent to the edge in which the vertex needs to be placed into four faces. The third situation doesn't require adding a new vertex, but some of the properties of the existing vertex need to be updated.

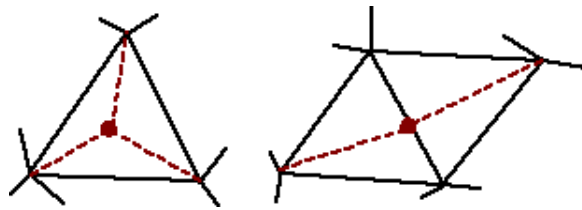


Figure 3.27: inserting vertices in a face (left) and in an edge (right)

There are two occasions in the algorithm where new vertices need to be added to the overlay triangulation. The first occasion is when the algorithm starts with the insertion of a new edge, the start and end node of the edge have to be inserted into the overlay triangulation. The second occasion is during the insertion of sections into the overlay triangulation. After a section has been determined its end point needs to be added to the overlay triangulation. When inserting the start and end vertex of a new edge, the situation where a vertex is inserted inside of an existing face is the most likely situation. Inserting a vertex on an edge or on an existing vertex are exceptions. When inserting end vertices of sections they will never be inserted inside faces. In these cases the most likely situation is the insertion of a vertex in an edge and the insertion of a vertex on an existing vertex is an exception.

Besides for the vertex insertion itself these exceptions also have consequences for the triangle walk algorithm that is used when inserting sections. Before the algorithm can start with calculating determinant to find the next face to walk through, it needs to locate the first face to start by circulating around all the faces that surround that start vertex of the section. If the start vertex was inserted inside of a face, the triangle walk algorithm has three faces to test. If the start vertex was inserted inside an edge, it has four faces to test (two on each side of the edge). If the vertex already exists in the triangulation, it has to test an unknown amount of faces.

Determining sections

Determining sections can cause exceptions as a result of less common topological relationships between edges and between edges and faces. [Clementini, 1993]. When the triangle walk algorithm determines the sections of an edge, it calculates the determinant of a two by two matrix. The sign of the determinant then determines the course of action of the algorithm as described in the previous sub-section. The determinant can however become 0. This is the first of two exceptions that can occur when determining sections. When the determinant is 0, the projected edge lies on top of the edge that was used to determine the next face that overlaps the projected edge. For the triangle walk algorithm this means that it has found a constraint, i.e. the projected edge goes through an existing vertex in the overlay triangulation.

The second exception occurs at the start of the ray shooting algorithm. The algorithm needs to determine the first face that overlaps the projected edge. It can't do this when the projected edge lies on top of one of the edges that come together in the start vertex of the projected edge. This situation is shown in figure 3.28.

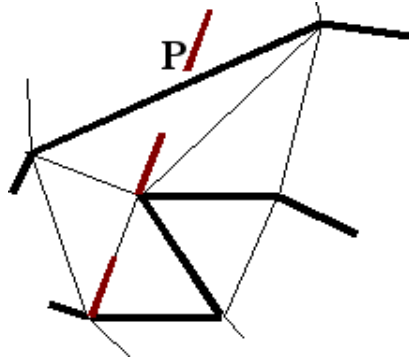


Figure 3.28: projected edge on top of existing edge

Flipping edges

When flipping edges to construct a section of an edge, there are two situations in which flipping an edge won't yield the correct result straight away. Both situations can be seen in figure 3.29 on the left. The figure depicts part of the overlay triangulation. The thick black lines are constrained edges, and are part of an object that was already inserted into the target triangulation. The red dashed line is the section that needs to be constructed by flipping the three edges it intersects with. Both the start and end vertex of the section have already been inserted into the overlay triangulation, and the faces overlapping the section have been determined (the grey faces).

The first situation that poses a problem is a result of the criterion for flipping as described in the first section of this chapter. The quadrangle surrounding the edge that needs to be flipped has to be convex in order to maintain a valid triangulation after the edge is flipped. Both quadrangles, depicted in light and dark grey, around edges e_1 and e_2 don't meet this criterion.

The second situation that poses a problem, arises when the quadrangle around an edge that intersects the section has two vertices on each side of the section. Edge e_3 in the figure is such an edge. After the edge has been flipped, it will still intersect the section that needs to be inserted. Since edges are only allowed to intersect at vertices, this edge still makes construction of the section impossible.

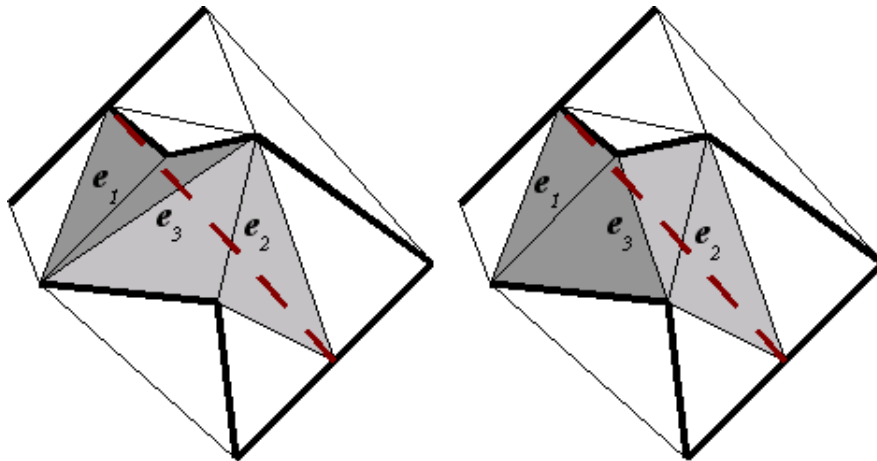


Figure 3.29: two cases in which flipping won't work immediately

The first situation can be resolved by flipping an edge of the quadrangle that intersects the section first. By doing this, the edge that couldn't be flipped will no longer be contained within a non-convex quadrangle, and as a result it can be flipped. Figure 3.29 on the right shows the same triangulation as on the left with one difference. Edge e_3 , which was an edge of the dark grey quadrangle as well as the light grey quadrangle has been flipped. As a result, both non-convex quadrangles around edges e_1 and e_2 have been transformed into convex quadrangles. Both of these edges can be flipped now and after they have been flipped, they no longer intersect with the projected edge.

Edge e_3 still intersects with the projected edge. This problem can be solved in the same way as

the previous one: by flipping one of the edges of the surrounding quadrangle that intersect with the projected edge. This will change the shape of the quadrangle so that it has three vertices on one side of the section and one on the other side of the section. In triangulation on the right in figure 3.29, flipping either edge e_1 or edge e_2 yield the required result.

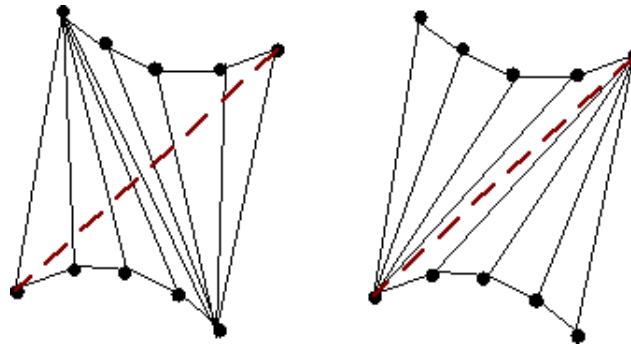


Figure 3.30: worst case non flippable edges

Both situations rely on flipping a 'neighbouring' edge to resolve the problem. This means that at least one of the edges that intersect the section need to be flippable, i.e. the surrounding quadrangle needs to be convex. Because all sections start and end at a vertex, the situation with only non-convex quadrangles can never occur. There is however a configuration of the faces for which an excessive amount of flips is needed to recreate the section. This configuration with a maximum number of reflex vertices, as described by [Hurtado, 1999], is shown in figure 3.30. In real data sets these situations occur when they contain polylines that approximate circular boundaries. For example buildings with curved walls. The result of the insertion of the dashed edge is shown on the right.

3.5 Phase 3: re-assembling features

After the algorithm has processed all the features in the feature set, the geometrical part of the feature overlay is finished. The result is a triangulation that contains the boundaries of all the features from the feature set as vertices and constrained edges with feature identifiers. This result can be used to restore the topology that was lost during the pre-processing phase as well as create a new topology. The exact (re)construction of the topology depends highly on the preferences of the operator, the environment the algorithm is used in and the question that needs to be answered by the overlay.

In order to be able to answer any questions the operator might have, the application that invoked the algorithm will need to know which areas of the result from the feature insertion phase contain which features. In terms of the definition given in chapter 2 this means that the compound view needs to be created. This labelling of the faces is done with the aid of a flood fill algorithm.

One of the conditions for the data format of the features in the feature set was that the edges, or the list of sequential nodes needed to be directed. The overlay algorithm has passed this direction property on to the constrained edges in the overlay triangulation. So for each face neighbouring a constrained edge it is known whether it lies outside or inside the boundary of the feature to which the constrained edge belongs. If a feature has holes in it and the edges of the holes have been passed to the algorithm with a clockwise direction, this is also true with for the holes, i.e. the floodfill algorithm won't enter the holes. This fact is used by the flood fill algorithm to assign a feature identifier to each face for every feature that the face belongs to.

The flood fill algorithm is initialised by providing it with a face that lies in the interior of a certain feature and with the feature identifier of that feature. After that the flood fill algorithm will start searching for other faces that lie in the interior of that feature until all the faces that make up the feature are found. It searches for faces by following this simple recursive procedure:

For each of the three neighbouring faces of the provided face do:

- If it has the provided feature identifier of the feature then stop with this neighbour face
- If the shared edge is a constrained edge with the provided feature identifier then stop with this neighbour face
- Assign the feature identifier to the face

Figure 3.31 shows part of an overlay triangulation. The red solid edges are constrained edges of the feature that needs to be processed by the flood fill algorithm. The black solid edges are other constrained edges. The dotted lines are non-constrained edges. A directed edge has been used to identify a face that lies inside of the feature. This face is shown in grey. The neighbouring faces that will be tested are shown in blue.

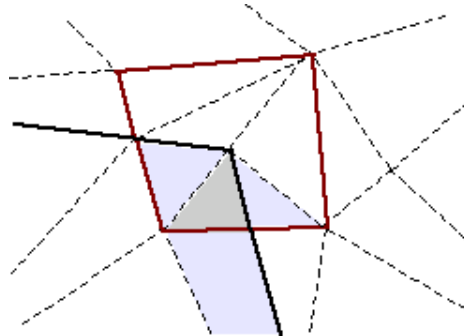


Figure 3.31: initialized flood fill algorithm

The edge between the grey face and the face below it is constrained and it has the feature identifier of the feature which is being processed. This means that nothing will be done with that face. The shared edge with the face to the right also is constrained, but this edge doesn't have the feature identifier of the feature which is being processed. The face also doesn't have the feature identifier, so this the feature identifier will be attached to that face. The face to the left will also get the feature identifier. Both the face on the left and on the right will be processed in the same way. Figure 3.32 shows the next three steps with the face which being processed coloured grey and the faces that are tested coloured blue.

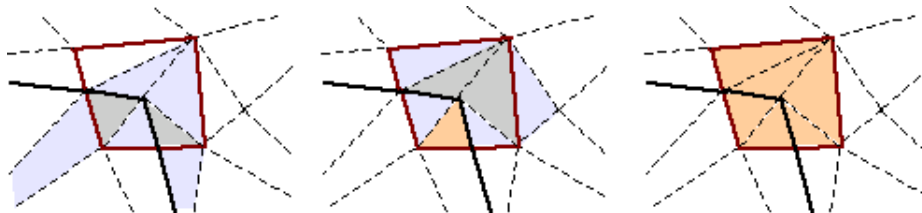


Figure 3.32: iteration steps of the flood fill of a feature

When features with holes are allowed there is one situation that poses a problem for this type of floodfill algorithm however. It is possible that holes within a feature create 'islands' as shown in figure 3.33. In these situations the 'islands' won't be labelled by the floodfill algorithm because it stops checking neighbours when it reaches the holes.

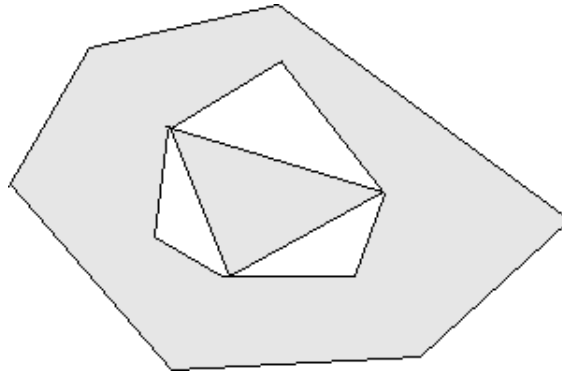


Figure 3.33: feature with an 'island' created by holes

4. 3D feature overlay algorithm

With a complete description of the container algorithm and the two dimensional sub-algorithms it is now possible to translate these to three dimensional algorithms. This chapter will describe the changes that need to be made to each of the algorithms in order for them to be able to process three dimensional data.

With the transition to the third dimension, some new entities and terms to address these entities arise. A description of these entities will be given here in the introduction analogue to the description of the terms used for the two dimensional feature overlay algorithm in the previous chapter. As reference, a short description of each of the terms explained here can be found in the glossary.

In the project strategy it was stated that the container algorithm should not need to change when the transition from the two dimensional to the three dimensional feature overlay algorithm is made. The container algorithm does change slightly however. Why and how the algorithm changes is described in section 4.2. After that the transition of the separate sub-algorithms is discussed in section 4.3.

4.1 Introduction

In triangulations the primitives with the highest dimension are the faces. In a three dimensional triangulation, i.e. a tetrahedral network, the primitives with the highest dimension are cells. Each cell consists of four facets arranged in a tetrahedral shape. When a single cell is considered facets are the same as faces in a triangulation. The other facets of the cell are the neighbours of the facet. Figure 4.1 shows two cells on top of each other. The hatched triangular area is a facet. Each facet is built up out of three vertices or nodes connected by edges.

The boundary of a three dimensional feature is not described by its edges and its nodes as with two dimensional features. In stead, the boundary is described by its facets. If the two cells in figure 4.1 would represent a feature, all the grey facets would describe the boundary of the feature. When the feature is inserted into a tetrahedral network, the facets describing the boundary are constrained facets and all other facets are non-constrained facets. In figure 4.1 the red facet is a non-constrained facet when these cells are part of a tetrahedral network.

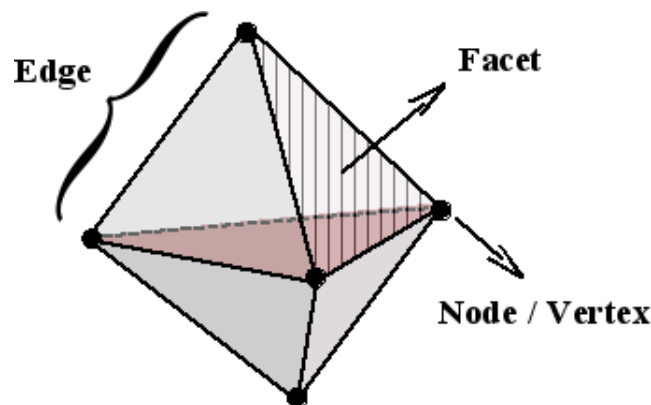


Figure 4.1: elements of a cell

The geometric representation of the faces in the two dimensional case is also valid for the three dimensional case. Points, segments and triangles are used in both triangulations and tetrahedral networks. In addition to these three geometric entities the tetrahedral network has one more geometric entity: the tetrahedron. Figure 4.2 shows one tetrahedron and with an indication of the other geometric entities within it. The relationships between the geometry and the feature properties is shown in figure 4.3.

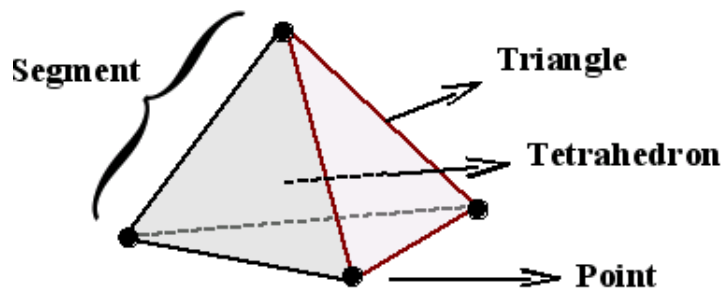


Figure 4.2: geometric properties of a cell

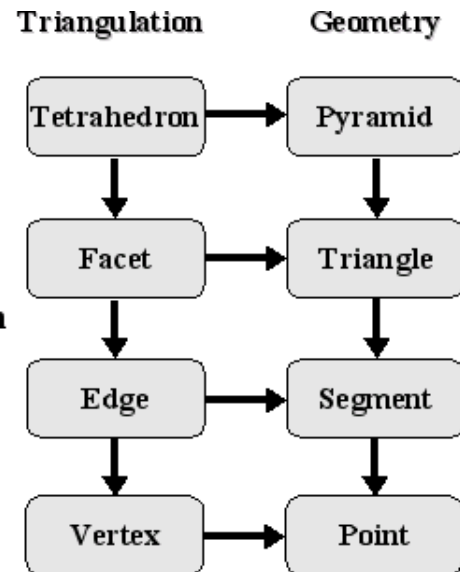


Figure 4.3: relationships between terms

4.2 The 3D container algorithm

Now that the container algorithm has been filled with two dimensional sub-algorithms the transition to the three dimensional situation can be made. As described in chapter 2, this is done by translating the individual two dimensional sub-algorithms to their three dimensional versions. These three dimensional sub-algorithms are then placed into the container algorithm to form the three dimensional feature overlay algorithm. The schema of the container algorithm as described in section 3.2 is shown in figure 3.12.

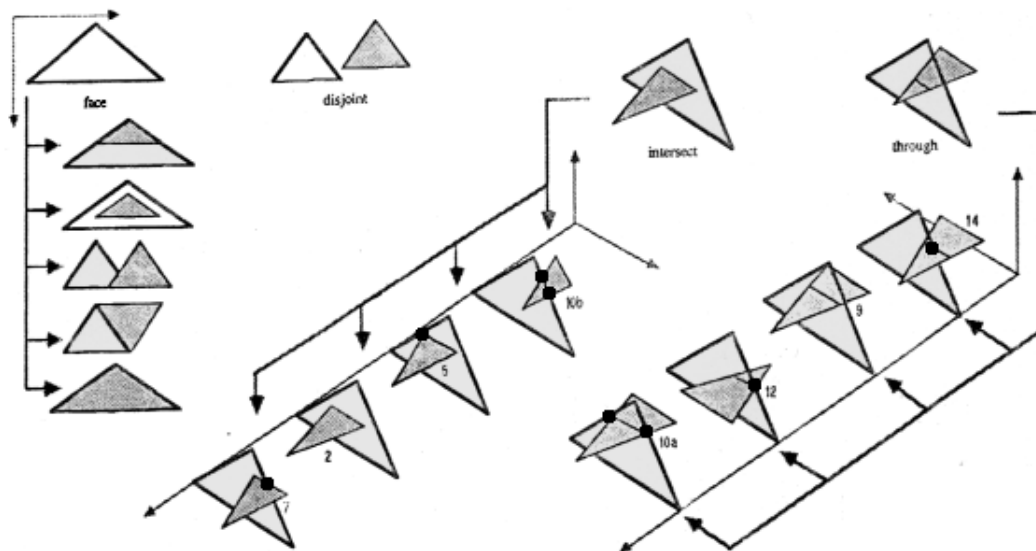


Figure 4.4: hierarchy of topological relationships between faces in R3 [Pigot, 1991]

The container algorithm wasn't supposed to change when the two dimensional feature overlay algorithm was translated to its three dimensional version. During the translation of the sub-algorithms it became clear that one small addition needed to be made to make the transition go smoothly.

If the container algorithm of figure 3.12 is used for overlaying three dimensional features, all of the edges of the feature would be present in the overlay tetrahedral network. The fact that the edges are present in the overlay network is not enough. The presence of these edges doesn't guarantee that the facets of the feature will also be present in the overlay network. Figure 4.4 shows the topological

relationship between faces in three dimensions [Pigot, 1991]. Only the intersections in 5, 7, 10a/b, 12 and 14 which are excentuated with a dot are found when the edges are inserted into the tetrahedral network. In some cases the correct constrained edges might not be present (5, 7, 12, 14) or, if no intersection has been detected at all, the constrained edges will definately not be present (2, 9). To ensure that the facets will also be present, the container algorithm needs one more iteration step. This step is shown in figure 4.5.

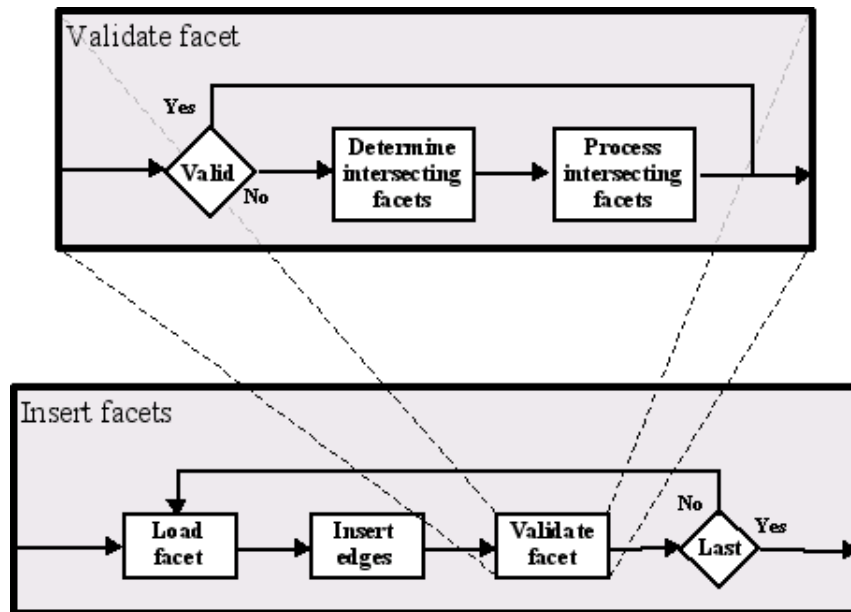


Figure 4.5: extra iteration step in the container algorithm

When all of the edges have been inserted in much the same way as the edges of a two dimensional feature, it is still possible the facets of the overlay network still intersect with the interior of the facet which is being inserted. These intersecting facets need to be found and processed. How they are processed depends on the type of facet. If it is a non-constrained facet it needs to be flipped out off the way. If it is a constrained facet, the intersection of the two facets needs to be determined. The exact process of testing, flipping and calculating intersections is described in the next section of this chapter.

The algorithm processes the building blocks of the boundaries of features from large to small. The largest building blocks of two dimensional feature boundaries were the edges, so within the iteration over the features, the algorithm would iterate over the edges. The largest building blocks of three dimensional features are facets and not edges, so within the iteration over the three dimensional features, the algorithm should iterate over the facets. The extra iteration step of figure 4.5 effectively comes between the iteration over the features and the iteration over the edges. The new container algorithm which is obtained when this iteration step is added can be seen in figure 4.6.

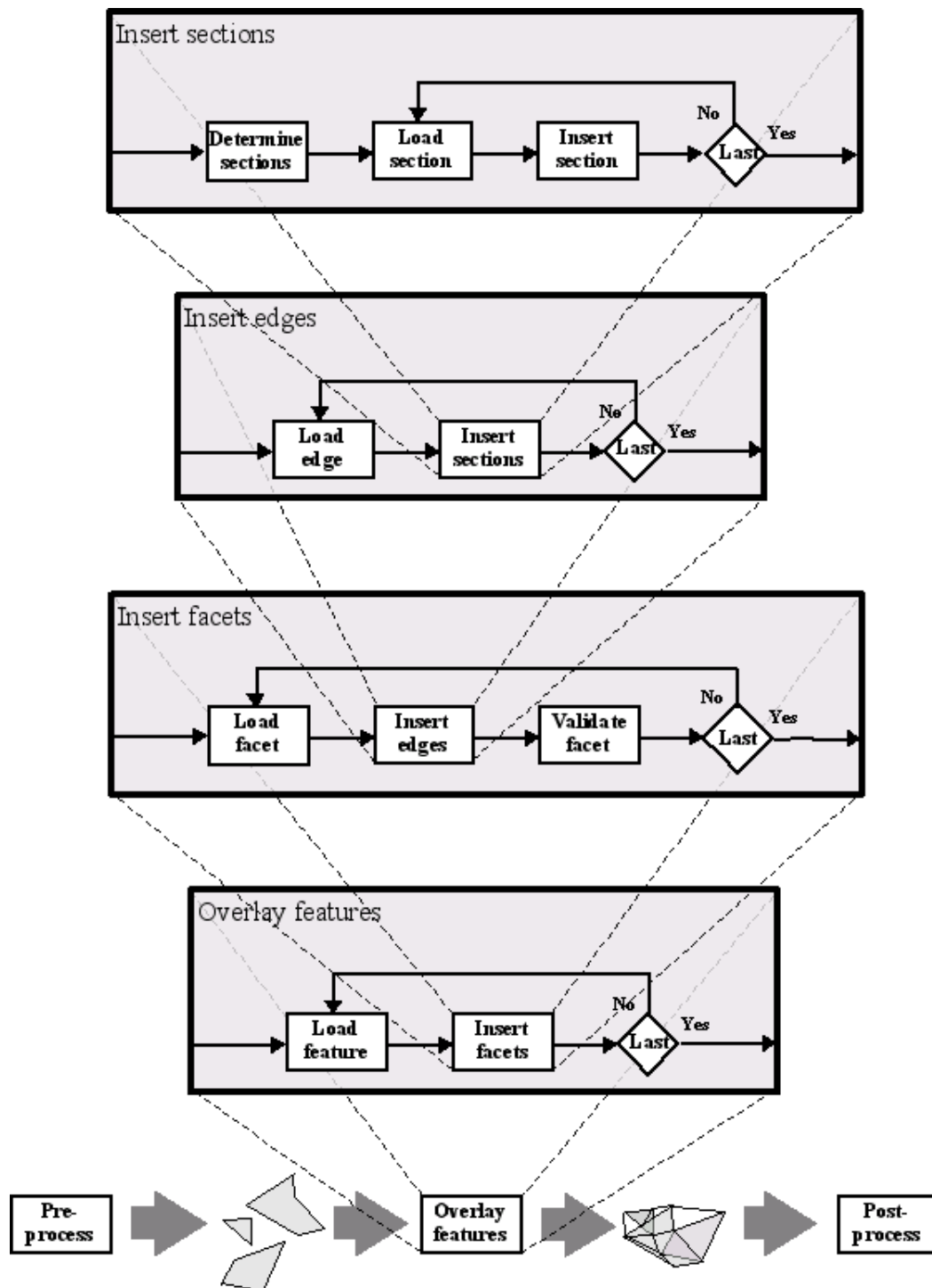


Figure 4.6: extended container algorithm

4.3 From two to three dimensions

In the previous chapter the container algorithm was filled with sub-algorithms for inserting edges, determining sections, flipping edges in a triangulation, and inserting vertices. These sub-algorithms are modified so that they can perform the same tasks for three dimensional data. Before the translation of these algorithms is discussed it has to be known which data will be available to the algorithms and which data is expected after they have performed their tasks. So before diving into the translation the new feature data format and the new iteration step will be explained.

4.3.1 Three-dimensional feature data format

All three of the conditions for the data format of the features in the feature set which were set in sub-section 3.2.2 are still valid for three dimensional features. In addition to those three conditions three dimensional features will also have to meet the following two conditions:

1. the boundary of the feature has to be triangulated with the preservation of the edges of the feature faces, i.e. a constrained triangulation, no internal tetrahedralisation needed.
2. all three points of all of the facets on the boundary of the feature are added to the list of points sequentially.

The first extra condition needs to be met to make it possible for the algorithm to reconstruct the faces of the features after they have been inserted into the overlay tetrahedral network. As explained in the previous section, inserting edges into the overlay tetrahedral network doesn't guarantee that the faces that these edges form are also present in the tetrahedral network. In order to solve this problem the algorithm needs to locate the facets in the tetrahedral network that intersect with the face of the feature that is being inserted. Finding intersecting triangles (facets) is much easier than finding intersections of arbitrary polygons [O'Rourke, 1998]. The algorithm uses this simpler intersection detection, so the faces of the features need to be triangulated.

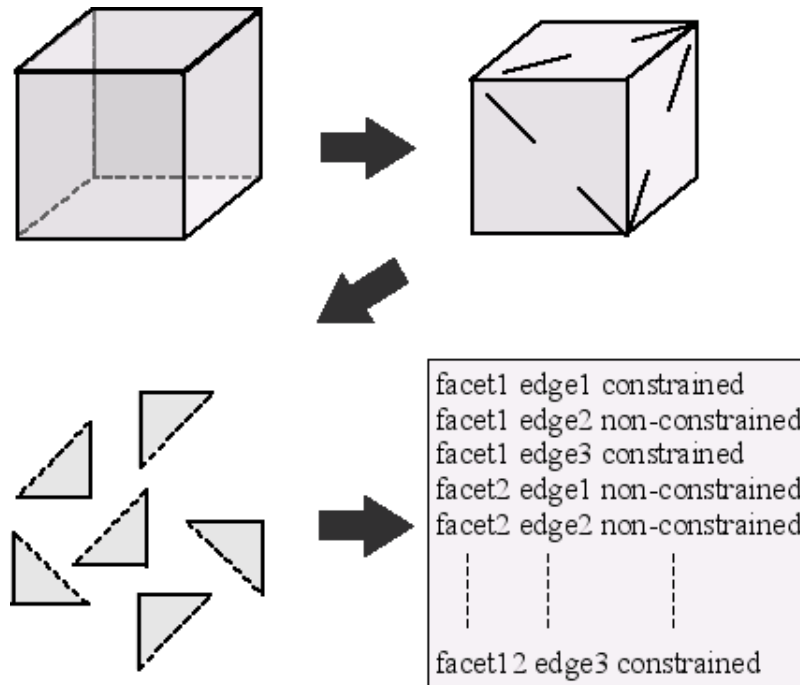


Figure 4.7: data format of three-dimensional features

The second extra condition has to be met to make it possible for the algorithm to process the facets of each boundary face individually. In the insert feature step in the container algorithm the feature overlay algorithm iterates over all the facets of the feature. It can only do that if the facets are individually identifiable in the feature set. Both the first and the second extra condition are illustrated in figure 4.7.

Instead of making these conditions, the triangulation of the boundary of the features could also have been made part of the overlay algorithm itself. The choice was made to exclude it from the algorithm itself because triangular irregular networks and triangular meshes are a popular data format for three-dimensional data [Baker, 1998]. In the cases where the feature boundaries aren't available in a triangular data format, there are already many efficient algorithms available.

4.3.2 Translation of sub-algorithms

Flipping

The first two-dimensional sub-algorithm that will be translated is the flip algorithm. At first glance, it seems this doesn't pose a problem. The translation would simply be flipping facets in stead of edges. Unfortunately flipping facets similar to the way that edges are flipped in two-dimensional triangulations is only possible if the tetrahedra are situated in a specific way. A more or less complete picture of flipping in a tetrahedral network is given in [Shewchuk, 2003]. Only the most relevant aspects of flipping in a tetrahedral network are discussed here.

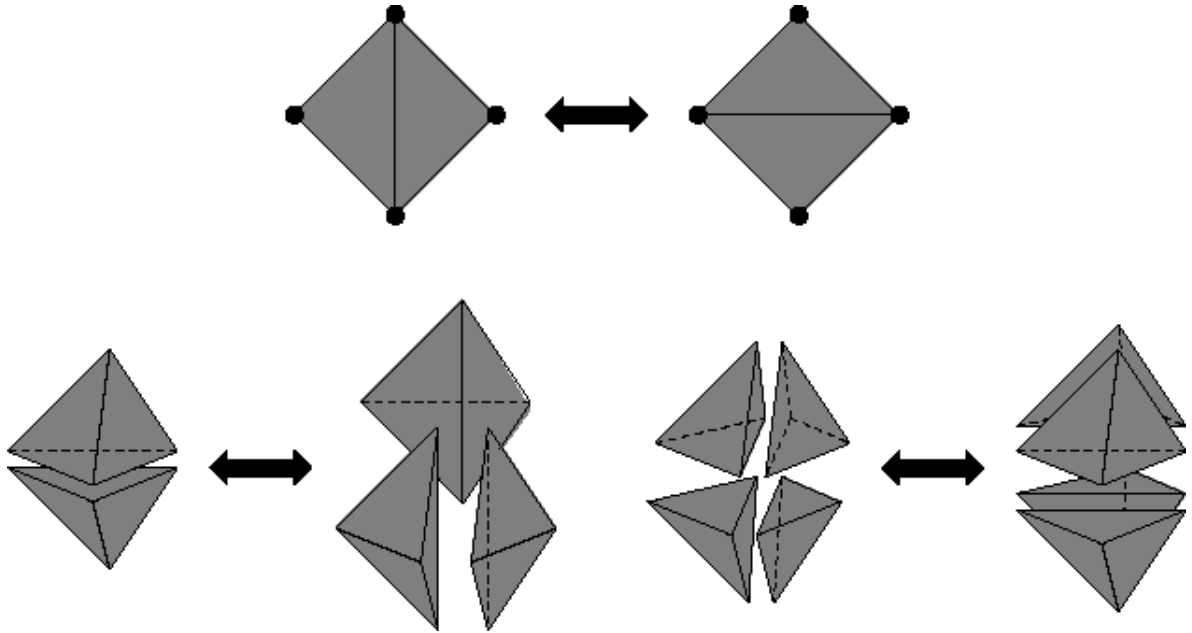


Figure 4.8: above: flip, below: 2-3 bistellar flip and 4-4 bistellar flip

When the flip algorithm is generalized to n -dimensional space, flips are called bistellar flips. The definition of bistellar flips is much wider than the definition used for the flipping in sub-section 3.4.1. Adding vertices to or removing them from a triangular network are also considered bistellar flips [Anwei, 2000] [Shewchuk, 2003]. In the context of the feature overlay algorithm the term flipping will be reserved for the bistellar flips that don't add or remove vertices from the tetrahedral network. Bistellar flips that add vertices to the tetrahedral network will be addressed by the term vertex insertion. This is done to maintain a consistent use of terms throughout the thesis and to satisfy the intuitive idea of flipping more closely. Figure 4.8 shows the bistellar flips that don't add or remove vertices for two-dimensional and three-dimensional triangulations.

The first three-dimensional flip is the 2-3 flip (shown in the bottom-left of figure 4.8). This flip destroys 2 tetrahedra and creates 3 new tetrahedra in their place, or vice versa. The second three-dimensional flip is the 4-4 flip (shown in the bottom-right of figure 4.8). The 4-4 flip is a degenerate case of the 2-3 flip. It destroys four tetrahedra and creates four new ones in their place. This is only possible when two facets in adjacent tetrahedra lie in the same plane.

Intuitively a three-dimensional flip would be a 2-2 flip where the facet that is shared by two tetrahedra is removed and replaced by another one, creating two new tetrahedra. This flip exists, but it is very rare. It is in fact a special case of the 4-4 flip. As with the 4-4 flip, the 2-2 flip also requires that two facets in adjacent tetrahedra lie in the same plane. Furthermore, both of the tetrahedra need to be situated on the same side of that plane and there aren't any tetrahedra allowed on the other side of that plane, i.e. the facets of the two tetrahedra that lie in the same plane also need to be part of the boundary of the tetrahedral network. The 2-2 flip is shown in figure 4.9.

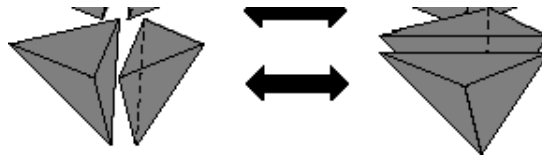


Figure 4.9: the 2-2 bistellar flip

Triangle walking

The second two-dimensional sub-algorithm that needs to be translated is the triangle walk algorithm. The principle of walking along a line until a constraint is found and recording all the non-constrained elements that are passed remains valid in the three-dimensional situation. The same technique of projecting a line onto the walking path can also be applied to find out which the next tetrahedron will be along the path [Watt, 2000]. There are however more efficient ways of locating the next tetrahedron on the line which is walked along [Amanatides, 1996] [Silva, 1997]. As with the two-dimensional triangle walk algorithm, the choice of the exact type of three dimensional triangle walk (or tetrahedron walk) algorithm that best suites the feature overlay algorithm depends on the exact implementation of the tetrahedral network which is used by the feature overlay algorithm.

Determining sections

The third and final two-dimensional sub-algorithm that has to be translated is the algorithm used to determine the sections. The definition for a section is the same for two-dimensional and three-dimensional sections. In both cases a section is a part of an edge with its start and end point at a constraint in the triangulation. In the tetrahedral network the start and end points of sections are located at vertices, constrained edges and constrained facets.

Just as with the two-dimensional algorithm a line is followed from the start vertex of a section through the intersecting tetrahedra. The algorithm walks along the line recording all the tetrahedra it comes across until it finds a constraint. When it finds a vertex, it doesn't have to do anything more than adding an identifier of the feature that is being processed to that vertex. When the constraint found was an edge, it inserts a vertex into the edge. This splits all tetrahedra sharing that edge into two new tetrahedra. See figure 4.10.

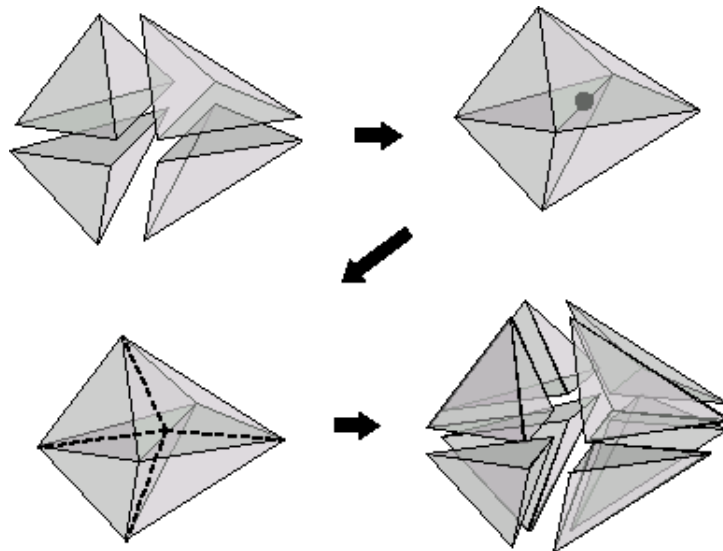


Figure 4.10: inserting a vertex into an edge

When a constrained facet is encountered while walking along the ray of the ray shooting algorithm, the algorithm inserts a vertex in a constaraint facet. This splits the two tetrahedra sharing the facet into three new tetrahedra. The split for one of the tetrahedra is shown in figure 4.11.

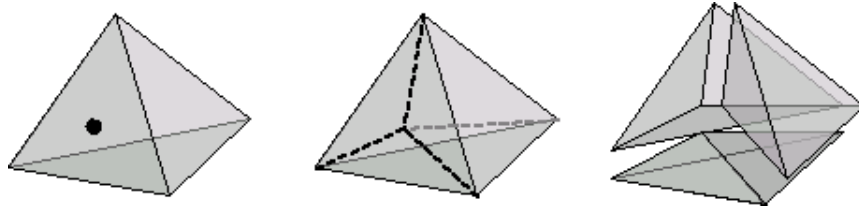


Figure 4.11: inserting a vertex into a facet

When the start and end vertex of the section are present in the tetrahedral network, all of the facets that intersect with the section need to be flipped out of the way. This is done by using the two bistellar flips shown in figure 4.8. As with flipping in the two-dimensional triangulation, the last flip creates an edge in the tetrahedral network which represents the section. Figure 4.12 shows a section which needs to be inserted with one last intersecting facet on the left. After flipping this facet (with a 2-3 flip) the situation as shown on the right in figure 4.12 is created. The section is now present in the tetrahedral network.

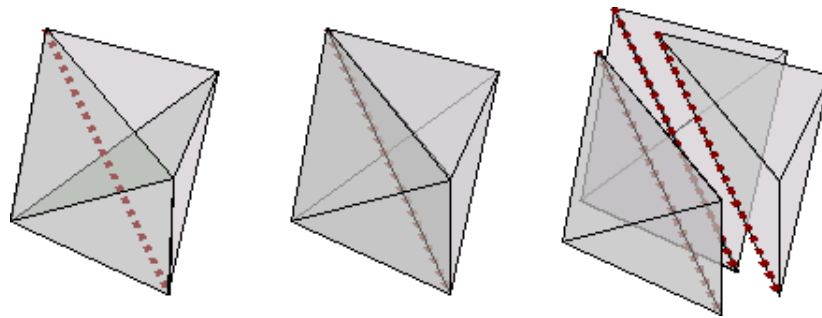


Figure 4.12: last flip produces section as an edge

At least that is how it should work. For this to be true, the conjecture: Given two different triangulations T_1 and T_2 of the same n three-dimensional vertices, T_2 can be obtained from T_1 by a finite sequence of local transformation procedures (flipping) [Joe, 1992]. This conjecture is still open however. A 97% recovery of constrained edges has been achieved by [Anwei, 2000], but the flip sequence presented there is based more on results of experiments than on actual proof.

Recently proof of this conjecture has been given for conformal triangulations [Shewchuk, 2003]. When constructing a conformal triangulation it is allowed to insert extra vertices in the constrained edges in order to preserve or add the constraints to the triangulation. In 1999 a similar approach was implemented without the proof by Cavalcanti [Cavalcanti, 1999]. Since the proof doesn't extend to arbitrary triangulations (yet) this can't be used for the algorithm presented here. Without enough time to find the proof that is needed the results presented in [Anwei, 2000] together with the knowledge that polyhedrons such as the Schonhart polyhedron cause the flipping algorithms designed up to now to falter [Anwei, 2000], [Aichholzer, 2002] will have to make it plausible that the algorithm works when these special polyhedrons are avoided.

Validating facets

In section 4.2 it was already mentioned that inserting edges isn't enough to make sure that the facets of a feature are present in the overlay tetrahedral network. Tetrahedra in the tetrahedral network might still intersect with the boundary facets of which the edges are inserted. A full list of configurations of facets that have intersections but won't be detected while inserting edges is as shown in figure 4.4. This problem is solved in two steps.

First the intersecting tetrahedra are located. There are several different ways in which these tetrahedra can be found. Popular algorithms for finding intersections between volumes and planes are depth ordering [Watt, 2000] and line sweeping [Silva, 1997]. These algorithms are widely used in computer graphics as part of the image rendering pipe-line. These algorithms don't use the topology of the tetrahedral network and they also neglect the tetrahedra that were found when the edges of the facet were inserted. Because they don't need much information to work, they can be used to find intersecting facets easily. By using the information embedded in the tetrahedral network the intersecting facets can be found much faster. An oil-spill algorithm similar to the one used in the

post-processing phase of the two-dimensional algorithm can be applied to this problem as well. In short the oil-spill would start at a tetrahedron that intersects with the facet and then move to neighbouring tetrahedra. It keeps moving to neighbouring tetrahedra if they intersect and stops with a tetrahedron if it doesn't intersect.

Second the intersecting tetrahedra need to be altered in such a way that they don't intersect with the inserted facet anymore. These tetrahedra are processed in the same way as the tetrahedra that intersected with the sections when they were inserted. If they are not constrained they are flipped out of the way. If all intersecting facets are non-constrained facets, the last flipped facet will create (part of) the facet that was being processed. As with the insertion of the constrained edges, there is only partial proof provided in [Shewchuk, 2002] [Shewchuk, 2003] and experiments showing that it works most of the time [Anwei, 2000]. If a constrained intersecting facet is encountered, the intersections of the edges of that facet with the facet that is being inserted are calculated and inserted into the tetrahedral network.

Reconstructing features

After all sections of all facets of a boundary face are inserted and all tetrahedra that intersect with the facets are processed the entire boundary face is present in the tetrahedral network. After all boundary faces have been inserted, an oil-spill algorithm similar to the one used for the two-dimensional feature overlay algorithm can be used to assign a feature identifier to all of the tetrahedra in the overlay tetrahedral network that are inside the boundary of the feature.

4.3.3 Special cases

In sub-section 3.4.5 the important degenerate cases for inserting vertices, ray shooting and flipping were discussed. These degenerate cases still apply to the three-dimensional sub-algorithms and on top of that several extra degenerate cases appear due to the extra dimension.

Inserting vertices where a vertex already exists and inserting a vertex into an edge are degenerate in both two-dimensional and three-dimensional triangulations. In three-dimensional triangulations inserting a vertex into a facet is also considered degenerate.

For the triangle walk algorithm there were two degenerate cases: walking on an existing edge and walking over an existing vertex. For the tetrahedron walk algorithm both of these exceptions can also occur and besides these two it could also happen that the algorithm walks on an existing facet.

In the three-dimensional algorithm, after the tetrahedron walk algorithm is finished and the edges of a new facet have been inserted, the algorithm needs to check for intersections between facets. There are 16 different configurations possible of two facets in three-dimensional space of which 15 can occur while searching for intersecting facets [Pigot, 1991] (see figure 4.4). In the two-dimensional situation there were only two. Although these degeneracies don't make it impossible for the algorithm to process all triangulatable features, it does make implementation of the algorithm more complex.

When flipping edges in a two-dimensional triangulation, there was one situation where the edge couldn't be flipped without destroying the topology of the triangulation. It was illustrated in figure 3.18. This situation can also occur in a tetrahedral network. Figure 4.13 shows this situation. Both images in the figure show the same four tetrahedra. When performing a 2-3 bistellar flip in the left most two tetrahedra, the red dotted line in the image on the right needs to be created. This edge lies outside of the hull of the hexahedron that is formed by the two tetrahedra which are involved in the flip. In this case the right most two tetrahedra will even be enclosed by the newly formed tetrahedra after the flip, leaving the topology of the tetrahedral network in shambles.

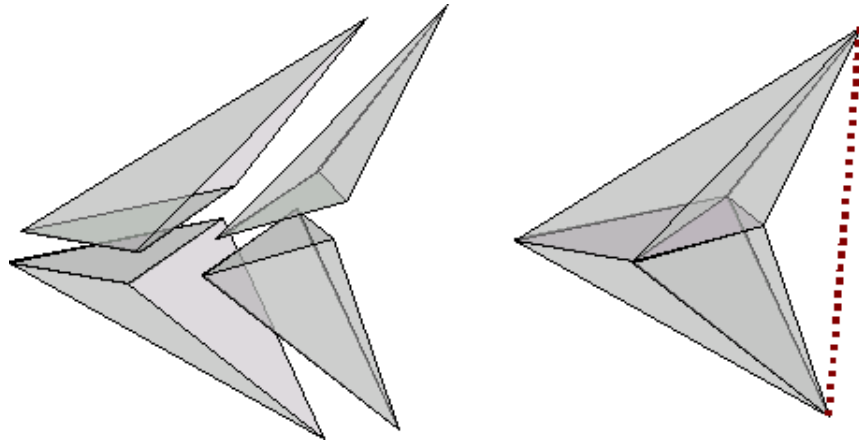


Figure 4.13: flipping is not possible in this tetrahedra pair

The second two-dimensional flipping degeneracy, where edges could be flipped but would still intersect with the sections that is being inserted as shown in figure 3.29, might also arise in the three-dimensional case. If this can happen could not be derived from the available literature and there was not enough time in the research project to implement the three-dimensional algorithm. Since it is always possible to reconstruct facets [Shewchuk, 2003] it is likely that this degeneracy can be solved in a similar way as the two-dimensional case should it occur.

5. Implementation of 2D feature overlay algorithm

This chapter contains a detailed description of the implementation of the two-dimensional version of the feature overlay algorithm. The Computational Geometry Algorithms Library (CGAL) was used to code the overlay algorithm itself. The results were presented using the cross-platform C++ GUI application framework Qt. More details on both CGAL and Qt can be found in chapter 2. In the rest of this section the implementation of the datastructure will be explained. The following sections of the chapter will discuss the implementation of the three phases analogue to chapter 3.

The code reached its final form long before the terminology of the thesis was thought through. As a result some of the terms used in the rest of the thesis won't appear to be present as variables, properties or functions in the code in this chapter. In the code in the code listings in this chapter, the term object is used in stead of feature. Object can denote the feature before it has been inserted into the overlay triangulation as well as the feature as part of the overlay triangulation. A explicit note will be made in the comments on the code when it is important to distinguish between the two. The overlay triangulation will appear in the code and code comments as target triangulation or tar_tr.

5.1 Introduction

The two-dimensional version of the feature overlay algorithm uses a triangulation to perform the overlay. The CGAL provides different triangulation data structures, but none of these data structures can contain information on constrained edges and the vertices in these triangulations can't contain information on the features of which they are part of. This meant that the two standard classes for vertex and triangulation had to be extended. The code shown in listing 5.1 adds two unique identifiers to the vertex class, creating a 'constrained vertex' class.

Listing 5.1: constrained vertex class definition

```
template < class Gt >
class Constrained_vertex_2

{
public:

    typedef typename Gt::Point_2    Point;

    // Constructor (empty): creates regular triangulation_vertex_base_2
    Constrained_vertex_2()
        : _p(Point()), _f(NULL), _id (0) { }

    // Constructor (point (face) ID): creates triangulation_vertex_base_2 at
    // the given point and sets the id to vertex_id
    Constrained_vertex_2(const Point & p, void *f = NULL, long vertex_id = 0)
        : _p(p), _f(f), _id(vertex_id) { }

    // constrained vertex methods
    void set_point(const Point & p) { _p = p; }
    void set_face(void* f) { _f = f; }
    void set_id(const long vertex_id) { _id = vertex_id; }
    void set_second_id(const long vertex_id) { _sid = vertex_id; }

    const Point& point() const { return _p; }
    void* face() const { return _f; }
    const long id() const { return _id; }
    const long sid() const { return _sid; }
    const bool has_id(const long vertex_id)
        { return ( (vertex_id == _id) || (vertex_id == _sid) ); }

    // the non const version of point() is undocumented
    // but it is needed to make the point iterator work
    // using Lutz projection scheme
    Point& point() { return _p; }

    bool is_valid(bool /* verbose */ = false, int /* level */ = 0) const
        {return true;}

private:
```

```

// the constrained vertex properties
Point _p;
void *_f;
long _id;
long _sid;
};

```

In the first two lines, the constrained vertex class is opened. It is immediately clear that this is no ordinary class. It is a template class. Templates are code constructs of the C++ language to make it easier for developers to develop cross-platform code. Apart from being a template, this class is special in a different way. The Gt in the template class inheritance definition stands for Geometric traits. Traits classes are another way used by CGAL to increase flexibility of the code which the library offers. These traits classes permit users of the library to adapt the code to their own needs and still be able to use the functions provided by the library. An explanation of the theory behind these concepts and the exact methods needed to use them is beyond the scope of this thesis. For a more extensive description of templates and traits classes see the CGAL documentation. [CGAL, 2003]

The class itself consists of two constructors, a number of methods and four properties. The first constructor

```

Constrained_vertex_2()
: _p(Point()), _f(NULL), _id (0) { }

```

creates an empty vertex. This vertex has got no geometric location, no position within a triangulation or an id. This constructor won't be used by the algorithm, but has to be added to fulfill the conditions of the vertex traits class. The second constructor is the constructor that will be used to create new vertices in the overlay triangulation.

```

Constrained_vertex_2(const Point & p, void *f = NULL, long vertex_id = 0)
: _p(p), _f(f), _id(vertex_id) { }

```

When this constructor is called, a vertex will be created with set values for three of the four properties. The co-ordinates of the vertex will be assigned to the point property `_p`. In the triangulation the vertex will belong to at least one face. A unique identifier of one of the faces that the vertex belongs to will be assigned to the face property `_f`. All vertices in the overlay triangulation belong to at least one feature. The unique identifier of the first feature that the vertex belongs to is assigned to the first identifier property `_id`. The second identifier property `_sid` remains undefined until it is set explicitly by another process, for example the insert section function.

In practice vertices could belong to more than two features. In stead of just adding two identifiers for features to the vertex class, an identification method which can link an unknown number of identifiers to the vertex should be used. These methods were not available in CGAL at the time. In order to keep the algorithm simple and implementation time within the bounds of the research period no more than two features may share the same vertex in this implementation of the overlay algorithm.

Most of the methods of the class just set or get the value of one of the properties of the constrained vertex. There are two methods that do something more than that:

```

const bool has_id(const long vertex_id) { return ( (vertex_id == _id) ||
(vertex_id == _sid) ); }
bool is_valid(bool /* verbose */ = false, int /* level */ = 0) const {return
true;}

```

The first method, `has_id()`, returns true if either the id or the second id of the vertex is equal to the requested id. i.e. the vertex belongs to the feature with `vertex_id` as unique identifier. The second method is used by CGAL to perform integrity checks on the triangulation when modifications of the triangulation are made.

With the adapted vertex class, it is possible to add nodes to the overlay while maintaining the

knowledge about the feature they belong to. The same adaptation has to be made to the edges of the triangulation class. The triangulation data structure in CGAL doesn't use edges explicitly so not only is it necessary to introduce feature identifiers, but also the edges themselves. The triangulations of CGAL use faces as building blocks. Listing 5.2 shows the heading of the face class to which the edges and feature identifiers were added.

Listing 5.2: constrained face class definition

```
template < class Gt >
class Constrained_face_2 : public Triangulation_face_base_2<Gt>
{
public:
    typedef Gt Geom_traits;
    typedef Triangulation_face_base_2<Gt> Fab;
    typedef Constrained_face_2<Gt> Constrained_face_base;
    typedef typename Gt::Point_2 Point;

    // Constructor (empty): creates regular triangulation_face_base_2
    Constrained_face_2()
        : Fab()
    {
        set_constraints(false,false,false);
        set_id(0);
    }

    // Constructor (vertices): creates triangulation_face_base_2 through the
    // given vertices and sets constraints to false
    Constrained_face_2(void* v0, void* v1, void* v2)
        : Fab(v0,v1,v2)
    {
        set_constraints(false,false,false);
        set_id(0);
    }

    // Constructor (vert. neighb.): creates triangulation_face_base_2 through
    // the given vertices and sets neighbours and
    // sets constraints to false
    Constrained_face_2(void* v0, void* v1, void* v2,
                      void* n0, void* n1, void* n2)
        : Fab(v0,v1,v2,n0,n1,n2)
    {
        set_constraints(false,false,false);
        set_id(0);
    }

    // Constructor (vert. neighb. constr.): creates triangulation_face_base_2
    // through the given vertices and
    // sets neighbours and sets constr.
    Constrained_face_2(void* v0, void* v1, void* v2,
                      void* n0, void* n1, void* n2,
                      bool c0, bool c1, bool c2,
                      long face_id = 0)
        : Fab(v0,v1,v2,n0,n1,n2)
    {
        set_constraints(c0,c1,c2);
        set_id(face_id);
    }

    // constrained face methods
    bool is_constrained(int i) const ;
    void set_constraints(bool c0, bool c1, bool c2) ;
    void set_constraint(int i, bool b);

    void set_id(long face_id);
    void set_second_id(long face_id);
    long id();
    long sid();
    bool has_id(const long face_id);

    void reorient();
    void ccw_permute();
    void cw_permute();
    bool is_valid(bool verbose = false, int level = 0) const;
```

```
protected:

    // constrained face properties
    bool C[3];
    long _id;
    long _sid;

};
```

As with the constrained vertex class, the constrained face class also is a template traits class and its class definition also consists of constructors, methods and properties. The face class is a more complicated class than the vertex class and depends on other classes defined in CGAL. The class definition in the second line is more elaborate, i.e. it states that the constrained face class will inherit all methods and properties of the in CGAL defined triangulation face base class. The four typedef statements are there to properly embed the new constrained face class in the CGAL template class structure.

The constrained face class has four constructors. The first one just creates a face without setting its properties. The last one creates a face and sets all its properties. The other two constructors are used when only values for specific properties are available when the face needs to be created. Because the last constructor sets all the properties it is sufficient to just discuss this constructor.

```
Constrained_face_2(void* v0, void* v1, void* v2,
                  void* n0, void* n1, void* n2,
                  bool c0, bool c1, bool c2,
                  long face_id = 0)
: Fab(v0,v1,v2,n0,n1,n2)
{
    set_constraints(c0,c1,c2);
    set_id(face_id);
}
```

v_0 , v_1 and v_2 are handles of the vertices that make up the face. Note that faces can only be constructed out of vertices and not out of points, i.e. points need to be inserted into the triangulation before they can be used to create a face. In case of the overlay triangulation v_0 , v_1 and v_2 will be handles of constrained vertices. n_0 , n_1 and n_2 are handles of the neighbouring faces of the face. These neighbouring faces are chosen in such a way that neighbour n_0 lies opposite to vertex v_0 . This is shown in figure 5.1 for all neighbouring faces of a face. c_0 , c_1 and c_2 define constraints on the edges of the face. These are chosen in the same way as the neighbouring faces. Finally, the $face_id$ is the identifier of the feature in which the face lies. This identifier isn't used in the feature overlay algorithm itself, but will be used in the post processing. As with the feature identifiers of the vertex class there are only two identifiers per face available while in practice the face could be part of more than two features. The same remarks that were made for the vertex feature identifiers apply here.

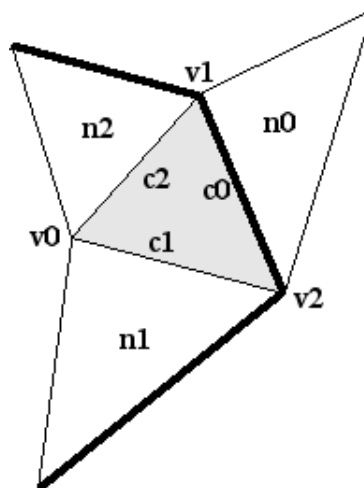


Figure 5.1: situation of neighbours and constraints in a face

The `has_id()` and `is_valid()` methods of the constrained face class have the same function as the equally named methods of the constrained vertex class. The `reorient()`, `ccw_permute()` and `cw_permute()` functions needed to be added to make the new constrained face class usable by the other classes in the CGAL. These methods aren't used by the overlay algorithm directly. The methods to manipulate the constraints are used by the overlay algorithm directly.

```
bool is_constrained(int i) const ;
void set_constraints(bool c0, bool c1, bool c2) ;
void set_constraint(int i, bool b);
```

The `is_constrained(int i)` function returns true if edge `ci` is constrained. `i` is either 0, 1 or 2. In figure 5.X `is_constrained(0)` would return true (edge `c0` is constrained) and `is_constrained(1)` would return false (edge `c1` is not constrained). The other two methods are used to set constraints. The `set_constraints()` method sets the constraints of all the edges at once. The `set_constraint()` method sets one constraint at a time. Note that setting a constraint is not the same as placing a constraint on an edge. Setting a constraint is setting the value of the property that contains the constraint. Placing a constraint on an edge is setting the value of the property that contains the constraint to true.

5.2 Phase 1: pre-processing

In the first phase features are loaded from a file and the nodes of the features are inserted into a feature triangulation. In this feature triangulation the convex hull of the feature will be determined. This is done to ensure that nodes in the list of nodes which will be presented to the algorithm are sequential and oriented. The file consists of a list of homogeneous points. Besides the `x` and `y` co-ordinate, homogeneous points also have a scale factor. The features in the file are separated by the point (1,1,1). This point was chosen as a separator because it made the preprocessing of the used dataset easier and because the standard input functions of CGAL could not handle non-numeric data. For the data used (1,1,1) was a safe coordinate to use. A file with two objects made up of 3 points will look something like this:

```
3  5  12
2  4   1
4  5  11
1  1   1
3  4   5
6  1   3
2  3   5
1  1   1
```

The function that loads features from such a file does this in two steps. First it reads the nodes and add them to the feature triangulation so that the convex hull can be determined later on. After the entire feature is loaded into the feature triangulation it determines a point which lies inside the boundary of the feature and saves it. This point will be used to initialize the flood fill algorithm in the feature reassembling phase. The code of the first step is shown in listing 5.3.

Listing 5.3: load a feature from a file into the feature triangulation

```
bool load_next_object(char* input_file, long object_id)
{
    // set-up the iterator for the input file if not already set-up
    static ifstream data(input_file);
    static std::istream_iterator < Point > it(data);
    static std::istream_iterator < Point > end;
    Vertex_handle new_vertex;

    // if EOF hasn't been reached load points, until (1,1,1) is read
    if (it != end) {
        Point dummy(1,1,1);
        while (*it != dummy) {
```



```

    new_vertex = obj_tr.insert (*it);
    new_vertex->set_id (object_id);
    ++it;
}

// move past the divider (1,1,1) in the file
++it;

```

If it is the first time that an object is loaded, an iterator is set up for the file that contains the feature data. The iterator is declared in the first three lines of the function as a static variable, i.e. it retains its values when the function is exited and can be used again when the function is called later on in the process.

```

static ifstream data(input_file);
static std::istream_iterator < Point > it(data);
static std::istream_iterator < Point > end;

```

Iterators are a code construct provided by the Standard Template Library (STL). Iterators provide a limited set of functions for managing and manipulating data stored in arrays, lists or files. As the name suggests, they are extremely useful for iterating through large data structures. The feature overlay algorithm is a highly iterative algorithm so iterators will be used very often in this implementation of the feature overlay algorithm.

The iterator retrieves the points from the file one at a time until it reaches a (1,1,1) point. These two lines insert the point into the feature triangulation and assign it the correct feature identifier.

```

new_vertex = obj_tr.insert (*it);
new_vertex->set_id (object_id);

```

When all the points of the feature are inserted into the feature triangulation, a point is calculated which will later be used by the floodfill algorithm. The code that takes care of this is shown in listing 5.4. First a face that lies inside the convex hull of the feature is selected with the `finite_faces_iterator`. Then the centre of mass is calculated. This is done, because the centre of mass of a triangle always lies inside the triangle and therefor inside the feature. Finally the point and the feature identifier are saved in a list for later use.

Listing 5.4: determine a point inside the boundary of a feature

```

// determine a point that lies inside the object and store it in the
// object_vector
Finite_faces_iterator ffi = obj_tr.finite_faces_begin();
Face_handle object_face = ffi;
Point p0, p1, p2;

p0 = object_face->vertex(0)->point();
p1 = object_face->vertex(1)->point();
p2 = object_face->vertex(2)->point();
// determine the centre of mass of the face ((x0+x1+x2)/3, (y0,y1,y2)/3)
Point object_point ((leda_real) (to_double(p0.x()) + to_double(p1.x()) +
                                to_double(p2.x())), (leda_real) (to_double(p0.y()) +
                                to_double(p1.y()) + to_double(p2.y())),
                    (leda_real) 3);

// set the values for the object locator
Object_locator_type object_locator;
object_locator.locate_point = object_point;
object_locator.object_id = object_id;
object_locator.locate_face = object_face;

// push the object locator onto the object_vector
object_vector.push_back(object_locator);
}
}

```

The algorithm needs an ordered list of vertices or a list of edges to be able to overlay the feature., but after the `load_object()` function has loaded a feature into the feature triangulation the feature data merely consists of some unsorted vertices. This shortcoming is overcome by determining the convex hull of the feature. Iterators and the CGAL concept of an infinite vertex in triangulations makes it possible to calculate the convex hull on the fly. For efficiency reasons the code that does this is placed the function that inserts edges into the overlay triangulation. When the first object is loaded however, the overlay triangulation doesn't exist yet and is created by the `det_hull()` function shown in listing 5.5.

Listing 5.5: determining the convex hull of a feature

```
void det_hull()
{
    // declare variables
    int infinite_index;
    Face i_face;
    int opposite_index;
    Face_handle opposite_face;
    Vertex_handle iv = obj_tr.infinite_vertex();
    Face_circulator fc = obj_tr.incident_faces(obj_tr.infinite_vertex());
    Face_circulator fc_end(fc);

    // remove the vertices that aren't on the convex hull
    Finite_vertices_iterator fvi = obj_tr.finite_vertices_begin();
    Finite_vertices_iterator fvi_end = obj_tr.finite_vertices_end();

    do {
        if (!obj_tr.is_edge(fvi,obj_tr.infinite_vertex())) obj_tr.remove(fvi);
    } while (++fvi != fvi_end);

    // set constraints for convex hull
    if (fc != 0) {
        do {
            i_face = *fc;
            infinite_index = i_face.index (iv);
            opposite_face = fc->neighbor (infinite_index);
            opposite_index = fc->mirror_index(infinite_index);
            opposite_face->set_constraint(opposite_index,true);
        } while(++fc != fc_end);
    }
}
```

One of the variable declared in this function is a circulator. Circulators are the same as iterators with one difference. Iterators stop iterating after reaching the last value in a list, while circulators go one step further by moving back to the first value in the list.

In a triangulation, all the vertices on the convex hull are connected to one virtual vertex, the infinite vertex. By checking if a vertex has the infinite vertex as a neighbour, the function can remove all vertices that lie inside the convex hull of the feature. This is done with the `is_edge()` method of the triangulation class. This method returns true if an edge defined by the two points that are passed as variables exists within the triangulation. If an edge between a feature vertex and the infinite vertex exists, the vertex is part of the convex hull.

The CGAL triangulation data structure doesn't contain edges explicitly, so the `det_hull()` function can't iterate over the edges of the convex hull. In stead it iterates over the faces and runs the following for lines of code to attach the feature identifier to the edges in the convex hull.

```
infinite_index = i_face.index (iv);
opposite_face = fc->neighbor (infinite_index);
opposite_index = fc->mirror_index(infinite_index);
opposite_face->set_constraint(opposite_index,true);
```

First it determines the index of the face that contains the edge on the convex hull. In the CGAL data structure this index is the same as the index of the vertex opposite of the face, i.e. the index of the infinite vertex. In the second line of code, it gets the face that contains the edge on the convex hull. That edge has the same index as the neighbouring face outside the convex hull. The third line of code

gets this index. Finally the fourth line of code sets the constraint for the edge on the convex hull. For an explanation of indexing within a face, check the introduction of this chapter.

5.3 Phase 2: inserting features

The second phase is the most complex phase of the algorithm. This is also expressed in the amount and complexity of the functions needed to perform all the steps in this phase. In order to keep the large amount of code manageable and the comments useful, this section will start with an overview of the used functions. Each of these functions will be discussed in detail after the relationships between the functions have been described.

There are seven custom functions and an even larger number of CGAL functions which are used to insert features into the feature overlay. For an explanation of the CGAL functions see the CGAL documentation [CGAL, 2003]. The custom functions can be divided into two categories. Three of these functions provide the framework of the second phase of the algorithm as shown in figure 3.12. The other four functions support the first three functions. Listing 5.6 gives an overview of the seven custom function.

Listing 5.6: overview of functions for the second phase of the overlay algorithm

```
// Framework functions:
// -----

void insert_obj(long object_id);
Vertex_handle insert_edge(const Vertex_handle vt1, Point pt2, long
object_id);
Vertex_handle insert_section(Vertex_handle start_vertex, Vertex_handle
end_vertex, long object_id);

// Support functions:
// -----

Vertex_handle insert_vertex(const Point pt, long object_id);
Vertex_handle insert_vertex_in_edge(const Point pt, Face_handle f, int i,
long object_id);
void c_flip(Edge flip_edge);
Face_handle get_start_face(const Vertex_handle vstart, Vertex_handle vend);
```

The `insert_obj` function is called each time an object has been loaded into the object triangulation. In section 5.1 it was explained that, due to efficiency considerations, the pre-processing step of determining the convex hull of a feature is performed during the inserting features phase. The `insert_obj` function is the function that determines the convex hull. When an edge of the hull is determined, it is passed to the `insert_edge` function.

The `insert_edge` function takes the steps which are described in sub-section 3.4.2. It first uses the `insert_vertex` function to insert the start and end nodes of the edge into the overlay triangulation. It then inserts all sections of the edge with the `insert_section` function. Because of the earlier mentioned shortcoming of the CGAL triangulation data structure (it doesn't contain edges explicitly) the sections can't be determined before hand. The `insert_section` function determines each section on the fly by processing the edge without the previously processed sections.

The `insert_section` function does the bulk of the processing needed to insert features into the overlay triangulation. First it uses the `get_start_face` function to initialise the ray shooting algorithm. It then performs the ray shooting process until it reaches a constraint. When a constrained edge is found, it uses the `insert_vertex_in_edge` function to insert the end point of the section. The sections start and end vertex are known then, so it can reconstruct the section by using the `c_flip` function.

In the rest of this sections the seven custom functions will be discussed in the order in which they are used to insert features into the overlay triangulation. First the `insert_obj` function will be discussed with the functions it uses (the `insert_vertex` and the `insert_edge` functions). Then the `insert_section` function will be discussed with the functions it uses (the `get_start_face`, the `insert_vertex_in_edge`, and the `c_flip` functions).

5.3.1 Processing edges

All edges of a feature are processed by the `insert_obj` function. The complete code of the `insert_obj` function is shown in listing 5.7.

Listing 5.7: insert a feature into the overlay triangulation

```
void insert_obj(long object_id)
{
    Vertex_handle first_inserted; // the first vertex that is inserted
    Vertex_handle last_inserted;  // the last vertex inserted
    Vertex_circulator vc(obj_tr.infinite_vertex());
    Vertex_circulator vc_end(vc);

    // if tar_tr is empty, simply copy obj_tr into it
    if (tar_tr.number_of_vertices() == 0) {
        det_hull(); // determine it's convex hull
        tar_tr.swap (obj_tr); // swap the triangulations
        obj_tr.clear(); // clear the object triangulation
    }

    // if tar_tr isn't empty, add the edges of the object one by one
    else {
        // insert the first vertex into the target triangulation
        first_inserted = insert_vertex ((vc)++->point(),object_id);
        last_inserted = first_inserted;
        object_vector.back().locate_face = first_inserted->face();
        // insert the edges upto the last edge
        do {
            last_inserted = insert_edge (last_inserted,vc->point(),object_id);
        } while (++vc != vc_end);
        // insert the last edge
        insert_edge (last_inserted,first_inserted,object_id);
        // clear the obj_tr so that another object can be added
        obj_tr.clear();
    }
}
```

The four variables declared at the start of the function are used to circulate over the vertices on the convex hull of the feature if it isn't the first feature that is processed. When it is the first feature, the `det_hull` function is used to create the overlay triangulation (`tar_tr` in the code). All the edges of the convex hull of the feature are processed in this loop:

```
do {
    last_inserted = insert_edge (last_inserted,vc->point(),object_id);
} while (++vc != vc_end);
```

Each pass to the loop the circulator moves to the next vertex on the convex hull (`++vc`) until it reaches the last unprocessed vertex on the convex hull (`vc_end`). The convex hull can be seen as a directed sequential list of nodes. The next edge that needs to be processed during each pass of the loop is therefor constructed with the last processed node (`last_inserted`) and the node that was selected by the circulator.

The last edge of the feature is processed in exactly the same way as all the other edges. The `insert_edge` function which is used for the previous edges can't be used however. The end node of the last edge is already present in the overlay triangulation as a vertex. In stead of passing a vertex and a point to the `insert_edge` function, two vertices are passed to the overloaded `insert_edge` function. The latter one skips the code for inserting points into the overlay triangulation.

Listing 5.8: insert one feature edge into the overlay triangulation

```
Vertex_handle insert_edge(const Vertex_handle vt1, Point pt2,
                          long object_id)
{
    // insert the start and end point of the edge into the triangulation
    Vertex_handle vh1 = vt1;
    Vertex_handle vh2 = insert_vertex (pt2,object_id);
```

```

// insert all sections of the edge into the target triangulation
do {
    vh1 = insert_section (vh1, vh2, object_id);
} while (vh1 != vh2);

// return handle of the last inserted vertex
return (vh2);
}

```

The code for the vertex-point version of the `insert_edge` function is shown in listing 5.8. The difference between this version of the `insert_edge` function and the vertex-vertex version lies in the second line of code. In the latter version the `insert_vertex (pt2,object_id)` call is replaced by the provided vertex handle `vh2`. When the end node of the edge has been inserted into the overlay triangulation, the sections are determined and inserted in the following loop.

```

do {
    vh1 = insert_section (vh1, vh2, object_id);
} while (vh1 != vh2);

```

The `insert_section` function determines and inserts the first section of the edge that is passed to it; here defined by the vertex handles `vh1` and `vh2`. After the `insert_section` function has inserted the section into the overlay triangulation, it returns a vertex handle for the end vertex of that section. The first vertex of the edge that is being processed is replaced by the vertex which was returned by the `insert_section` function, i.e. the first section was cut off the edge. The resulting edge is then passed to the `insert_section` function again. This process continues until the first vertex of the edge is the same as the second vertex, i.e. the length of the edge becomes 0 and there are no more sections to insert.

There is a function for inserting vertices into a triangulation available in the CGAL. As a result of the lack of explicit edges in the triangulation data structure in CGAL triangulations this function has a shortcoming however. It doesn't take the constraints on the edges into account, i.e. when a new vertex is inserted, the constraint can jump from one edge to another. A new extended function had to be designed that would check for changed constraints after a vertex is inserted and repair them if they are found. The code for this function can be found in listing 5.9. Note that this function doesn't check for the exceptions of inserting a vertex on an edge or inserting a vertex on a vertex.

Listing 5.9: insert one vertex into the overlay triangulation

```

Vertex_handle insert_vertex(const Point pt, long object_id)
{
    // insert the vertex into the target triangulation
    Vertex_handle vh = tar_tr.insert (pt);
    vh->set_id(object_id);

    // if the point was inserted inside the convex hull of the triangulation
    // restore the scrambled constraints
    if (!tar_tr.is_edge(vh,tar_tr.infinite_vertex())) {
        // restore the constraints around the inserted vertex
        Face_circulator fc = tar_tr.incident_faces(vh);
        Face_circulator fc_done(fc);
        int vi;
        Vertex_handle evh1, evh2;
        int eil, ei2;

        // Circulate over all the faces adjacent to the new vertex
        do {
            fc->has_vertex(vh,vi);
            eil = tar_tr.cw(vi);
            ei2 = tar_tr.ccw(vi);
            if (fc->is_constrained(eil)) {
                Face_handle nfl = fc->neighbor(eil);
                nfl->set_constraint (nfl->index(vh),true);
                fc->set_constraint (eil,false);
            }
        } while (fc != fc_done);
    }
}

```

```

    }
    if (fc->is_constrained(ei2)) {
        Face_handle nf2 = fc->neighbor(ei2);
        nf2->set_constraint (nf2->index(vh),true);
        fc->set_constraint (ei2,false);
    }
    } while (++fc != fc_done);
}
// if the point was inserted outside of the convex hull of the triangulation
// constraints need to be set for the newly created faces
else {
    Face_circulator fc = tar_tr.incident_faces(vh);
    Face_circulator fc_done(fc);
    do {
        if (!tar_tr.is_infinite(fc)) {
            Face_handle nf = fc->neighbor(fc->index(vh));
            if (nf->is_constrained(nf->index(fc)))
                fc->set_constraint (fc->index(vh),true);
        }
    } while (++fc != fc_done);
}
// return the vertex handle of the inserted vertex
return (vh);
}

```

After the new vertex has been inserted into the triangulation with the function provided by CGAL, the scrambled constraints need to be restored. When determining the scrambled constraints, two situations are distinguished. In the first situation the new vertex was created inside the convex hull of the triangulation. As a result of this, any or all constraints in the face in which the vertex was inserted could have been scrambled and need to be checked. To check and repair the constraints the indices of the edges of the checked face are gathered:

```

fc->has_vertex(vh,vi);
ei1 = tar_tr.cw(vi);
ei2 = tar_tr.ccw(vi);

```

Both of the edges that have the newly inserted vertex as start or end vertex are tested for constraints. If a constraint is found it is moved back to the edge it is supposed to be on by removing the constraint from the edge ei1 or ei2 in the new face fc and placing the constraint on the edge opposite of the newly inserted vertex in neighbouring face nfc. The restoration of the constraints is illustrated in figure 5.2.

```

if (fc->is_constrained(ei1)) {
    Face_handle nfl = fc->neighbor(ei1);
    nfl->set_constraint (nfl->index(vh),true);
    fc->set_constraint (ei1,false);
}
if (fc->is_constrained(ei2)) {
    Face_handle nf2 = fc->neighbor(ei2);
    nf2->set_constraint (nf2->index(vh),true);
    fc->set_constraint (ei2,false);
}

```

When the new vertex was inserted outside of the convex hull of the triangulation only infinite faces are changed, i.e. constraints aren't scrambled. Constraints on the edges in the newly created faces will have to be set if the edges of the convex hull of the triangulation which are no longer on the convex hull after the insertion of the new vertex are constrained. In other words, the constrained edges on the convex hull have to be constrained in the faces on both sides of the edge if they are no longer on the convex hull.

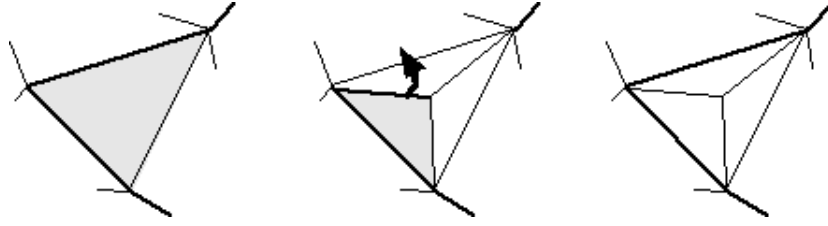


Figure 5.2: restoring a scrambled edge after a vertex insertion

5.3.2 Processing sections

After the end vertex has been inserted into the overlay triangulation the edge is inserted section by section with the `insert_section` function. The complete code of this function is shown in listing 5.10.

Listing 5.10: insert the first section of an edge into the overlay triangulation

```
Vertex_handle insert_section (const Vertex_handle start_vertex,
                             Vertex_handle end_vertex, long object_id)
{
    // declare variables
    Face_handle    start_face;
    Face_handle    current_face;
    int            current_index;
    Segment        s(start_vertex->point(),end_vertex->point());
    vector          edge_vector;
    VEdge          vertex_edge;
    Edge           face_edge;
    bool           constraint_found = false;
    Vertex_handle  found_end_vertex;
    CGAL::Object   result;
    Point          intersection_point;
    Segment        intersection_segment;
    Face_handle    opposite_face;
    int            opposite_index;

    // if the start and end vertex are connected directly
    // the section already exists in the triangulation
    if (tar_tr.is_edge (start_vertex,end_vertex,current_face,current_index)) {
        // if the face is infinite get the opposite face
        if (tar_tr.is_infinite (current_face)) {
            opposite_face = current_face->neighbor (current_index);
            opposite_index = current_face->mirror_index (current_index);
            current_face = opposite_face;
            current_index = opposite_index;
        }
        // set the constraint
        current_face->set_constraint (current_index, true);
        found_end_vertex = end_vertex;
    }

    // if the start and end vertex aren't connected directly
    // find the faces that intersect with its projection
    else {
        // locate the first face between start_vertex and end_vertex
        start_face = get_start_face (start_vertex, end_vertex);
        Line_face_circulator lfc = tar_tr.line_walk (start_vertex->point(),
                                                    end_vertex->point(), start_face);

        // locate the next constraint (end_vertex or constrained edge)
        do {
            // if the end vertex of the edge hasn't been reached
            if (!lfc->has_vertex(end_vertex)) {
                current_face = lfc++;
                current_index = current_face->index(lfc);
                // if a constraint has been found
                if (current_face->is_constrained (current_index)) {
                    result = intersection (tar_tr.segment
```

```

        (current_face,current_index), s);

    // intersection into the triangulation
    if (assign (intersection_point,result))
        found_end_vertex = insert_vertex_in_edge (intersection_point,
            current_face, current_index, object_id);
    // segment intersection hasn't been implemented yet
    else if (assign (intersection_segment,result)) {}

    constraint_found = true;
}
// if a non-constrained edge has been found add it to edge_vector
else {
    vertex_edge.va = current_face->vertex(tar_tr.cw (current_index));
    vertex_edge.vb = current_face->vertex(tar_tr.ccw (current_index));
    edge_vector.push_back(vertex_edge);
}

// if the end vertex of the edge has been reached
else {
    found_end_vertex = end_vertex;
    constraint_found = true;
}
} while (!constraint_found);

// if the edge_vector is not empty,
// flip the edges until edge_vector is empty
if (!edge_vector.empty()) {
    while (edge_vector.size() > 0) {
        vertex_edge = edge_vector.back();
        tar_tr.is_edge (vertex_edge.va, vertex_edge.vb,
            (face_edge).first, (face_edge).second);
        // only flip an edge if the hull of the adjacent faces is convex
        if (orientation((face_edge).first->vertex((face_edge).second)->point(),
            (face_edge).first->vertex(tar_tr.cw((face_edge).second))->point(),
            (face_edge).first->mirror_vertex((face_edge).second)->point()) ==
            CGAL::RIGHTTURN &&
            orientation((face_edge).first->vertex((face_edge).second)->point(),
            (face_edge).first->vertex(tar_tr.ccw((face_edge).second))->point(),
            (face_edge).first->mirror_vertex((face_edge).second)->point()) ==
            CGAL::LEFTTURN) {
            c_flip (face_edge);
        }
        // if hull of enclosing quadrangle isn't convex,
        // put the edge back on top of the list
        else {
            edge_vector.insert(edge_vector.begin(),vertex_edge);
        }
        // remove the processed edge from the vector
        edge_vector.pop_back();
    }
}

// set the constraint for the section
Edge constrained_edge;
// check if the edge exists in the target triangulation
if (tar_tr.is_edge (start_vertex,found_end_vertex,
    (constrained_edge).first,
    (constrained_edge).second)) {
    (constrained_edge).first->
        set_constraint ((constrained_edge).second,true);
    opposite_face = (constrained_edge).first->
        neighbor ((constrained_edge).second);
    opposite_index = (constrained_edge).first->
        mirror_index ((constrained_edge).second);
    opposite_face->set_constraint (opposite_index,true);
}
// if not, certain non-constrained edges may not have flipped
// out of the way so reinsert the section
else {
    insert_section (start_vertex,end_vertex,object_id);
}

```



```

    }

    // return the found constraint at the end of this section
    return (found_end_vertex);
}

```

First a check is made to see if the start and end vertex of the edge passed to the function are present in the triangulation as an edge. This situation occurs when the both start and end vertex have been inserted into the same face in the triangulation, or if both start and end vertex are neighbours in the convex hull of the triangulation. If the edge already exists, the following code places a constraint on it.

```

if (tar_tr.is_edge (start_vertex,end_vertex,current_face,current_index)) {
    // if the face is infinite get the opposite face
    if (tar_tr.is_infinite (current_face)) {
        opposite_face = current_face->neighbor (current_index);
        opposite_index = current_face->mirror_index (current_index);
        current_face = opposite_face;
        current_index = opposite_index;
    }
    // set the constraint
    current_face->set_constraint (current_index, true);
    found_end_vertex = end_vertex;
}

```

If the start and end edge aren't connected in the overlay triangulation, the faces that intersect with the projection of the edge need to be found, as described in sub-section 3.4.4. The function uses a standard function of CGAL, the line-face circulator, to do this. This function works in the same way as described in sub-section 3.4.4. The process of walking over the faces that intersect with the projected edge is performed by the following lines of the code:

```

do {
    // if the end vertex of the edge hasn't been reached
    if (!lfc->has_vertex(end_vertex)) {

        current_face = lfc++;
        current_index = current_face->index(lfc);

        ...

    }

    // if the end vertex of the edge has been reached
    else {
        found_end_vertex = end_vertex;
        constraint_found = true;
    }
} while (!constraint_found);

```

The code inside the loop checks if the edge between two successive faces is constrained. If it is constrained, the intersection between the edge and the projection of the new edge is calculated with the CGAL intersection function. This function returns a point result, or a segment result. If the first result is returned, the intersection point is inserted into the overlay triangulation with the `insert_vertex_in_edge` function, which will be discussed later on in this sub-section. The vertex returned by this function is saved and will be returned to the calling `insert_edge` function so that it can use that vertex as a new start vertex when invoking the `insert_section` function again. When a segment result is returned, the projected edge lies on top of an edge in the triangulation. This degeneracy is not handled here yet.

```

// if a constraint has been found
if (current_face->is_constrained (current_index)) {
    result = intersection (tar_tr.segment
                          (current_face,current_index), s);
}

```

```

// intersection into the triangulation
if (assign (intersection_point,result))
    found_end_vertex = insert_vertex_in_edge (intersection_point,
                                              current_face, current_index, object_id);
// segment intersection hasn't been implemented yet
else if (assign (intersection_segment,result)) {}

constraint_found = true;
}

```

If a non-constrained edge is found while walking along the faces in the line-face circulator, the edge is pushed onto a vector. A vector is an STL construct which contains an array of elements. In this case the elements of the vector are instances of a self defined type, two vertices that define an edge in the triangulation.

```

// if a non-constrained edge has been found add the edge to edge_vector
else {
    vertex_edge.va = current_face->vertex(tar_tr.cw (current_index));
    vertex_edge.vb = current_face->vertex(tar_tr.ccw (current_index));
    edge_vector.push_back(vertex_edge);
}

```

The next step in the overlay algorithm is the reconstruction of the section by flipping the non-constrained edges that intersect with the projection of the section. This is also the next step in the insert_section function. All edges that were pushed onto the edge_vector in the previous loop are now flipped one at a time using the c_flip function. This function is a custom made function for this algorithm. As with the insert_vertex function the CGAL flip function scrambles the constraints on the edges in the overlay triangulation. The c_flip function takes care of this problem by moving the constraints back to the correct edges.

```

if (!edge_vector.empty()) {
    while (edge_vector.size() > 0) {
        vertex_edge = edge_vector.back();
        tar_tr.is_edge (vertex_edge.va, vertex_edge.vb,
                       (face_edge).first, (face_edge).second);
        // only flip an edge if the hull of the adjacent faces is convex
        if (orientation((face_edge).first->vertex((face_edge).second)->point(),
                       (face_edge).first->vertex(tar_tr.cw((face_edge).second))->point(),
                       (face_edge).first->mirror_vertex((face_edge).second)->point()) ==
            CGAL::RIGHTTURN &&
            orientation((face_edge).first->vertex((face_edge).second)->point(),
                       (face_edge).first->vertex(tar_tr.ccw((face_edge).second))->point(),
                       (face_edge).first->mirror_vertex((face_edge).second)->point()) ==
            CGAL::LEFTTURN) {
            c_flip (face_edge);
        }
        // if hull of enclosing quadrangle isn't convex,
        // put the edge back on top of the list
        else {
            edge_vector.insert(edge_vector.begin(), vertex_edge);
        }
        // remove the processed edge from the vector
        edge_vector.pop_back();
    }
}

```

In section 3.4.5 Special cases and their implications, two degenerate cases were discussed in which the flipping of edges wasn't possible or wouldn't yield the intended result.

The first degenerate case is removed by the 8 lines long and seemingly complex if-statement. This statement checks if the quadrangle around the edge that needs to be flipped is convex. It does this by checking if all connecting edges of the quadrangle make right turns at the vertices when the edges are followed in a clockwise direction. If all turns are right turns the quadrangle is convex, otherwise it isn't. At the vertices that are part of the quadrangle, but not part of the edge that needs to be flipped

the edges always make right turns. This leaves to vertices at which the turns need to be checked. Figure 5.3 a) shows these turns for a convex quadrangle and figure 5.3 b) shows these turns for a non-convex quadrangle.

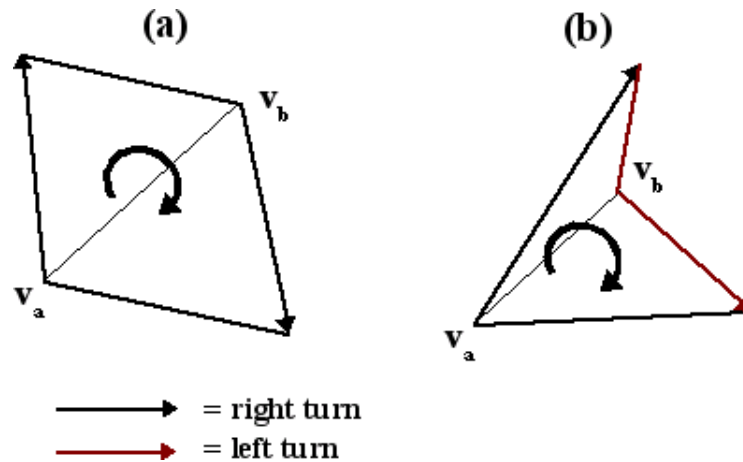


Figure 5.3: right turn test to determine if quadrangles are convex

Both the CGAL orientation function and the custom `c_flip` function use edges. These edges aren't defined as two vertices like the edges on the `edge_vector`. the CGAL `is_edge` function transforms the vertex-vertex definition to a face-index definition. The face is accessed through `(face_edge).first` and the index is accessed through `(face_edge).second`. These two different definitions and the resulting translations are needed because the CGAL triangulation doesn't contain explicit edges in its data structure. The edges can't be pushed onto the edge vector in the face-index notation, because inserting vertices and flipping edges changes the faces. When a face which was also pushed onto the `edge_vector` is changed, the edge on the vector changes as well. Since the vertices in the triangulation don't change a vertex-vertex representation can be used safely to store the edges that need to be flipped in the edge vector.

The second degeneracy which was discussed in section 3.4.5 occurs when there are two quadrangle vertices on each side of the projected section. When the edge is flipped, the flipped edge will still intersect with the projected edge. The `insert_section` function detects this degeneracy checking if the section appears in the overlay triangulation as an edge. If it doesn't the degeneracy occurred and the section is processed again by returning the original vertex in stead of the end vertex of the section. When the edge has been found, the constraints for that edge are set and the end vertex of the section is returned.

```
// check if the edge exists in the target triangulation
if (tar_tr.is_edge (start_vertex,found_end_vertex,
                    (constrained_edge).first,
                    (constrained_edge).second)) {

(constrained_edge).first->set_constraint((constrained_edge).second,true);
  opposite_face = (constrained_edge).first->
    neighbor ((constrained_edge).second);
  opposite_index = (constrained_edge).first->
    mirror_index ((constrained_edge).second);
  opposite_face->set_constraint (opposite_index,true);

}
// if not, certain non-constrained edges may not have flipped
// out of the way so reinsert the section
else {
  insert_section (start_vertex,end_vertex,object_id);
}
```

The `insert_section` function uses three custom functions. The first custom function used is the `get_start_face` function. This function is used to locate the first face for the line-face circulator. It does this by testing the faces around the inserted start vertex of the edge that is being processed. The code for the `get_start_face` function is shown in listing 5.11.

Listing 5.11: determine the first face for the projection of an edge onto the triangulation

```
Face_handle get_start_face (const Vertex_handle vstart, Vertex_handle vend)
{
    // declare variables
    bool found = false;
    Face_circulator fc = tar_tr.incident_faces(vstart);
    int v_index;
    int ei1, ei2;
    Vertex_handle evh1, evh2;
    Face_handle fstart;

    // Circulate over the faces around the starting point of the edge
    // that needs to be inserted
    do {
        v_index = fc->index(vstart);
        ei1 = tar_tr.cw(v_index);
        ei2 = tar_tr.ccw(v_index);
        evh1 = fc->vertex(ei1);
        evh2 = fc->vertex(ei2);

        CGAL::Orientation start_orient =
            orientation (evh1->point(), evh2->point(), vstart->point());
        CGAL::Orientation end_orient =
            orientation (evh1->point(), evh2->point(), vend->point());

        // if vhl and vh2 are on different sides of the line through evh1
        // and evh2, then (evh1, evh2, vhl) is the starting face.
        if (start_orient != end_orient) {
            found = true;
            fstart = fc;
        }
        ++fc;
    } while (found == false);
    return (fstart);
}
```

The `get_start_face` function uses the CGAL orientation function to determine the first face that intersects with the projection of the edge that needs to be inserted into the overlay triangulation. It does this by determining the orientation of the edges as shown in figure 5.4. If the orientations of $(v_1, v_2, vend)$ and $(v_1, v_2, vstart)$ are different, i.e. one is a left turn and the other is a right turn, the face defined by $vstart, v_1$ and v_2 is the start face. If not the search needs to continue with the next face that contains $vstart$. In the figure 5.4 the start face is found because the orientation $(v_1, v_2, vstart)$ is a right turn and the orientation of $(v_1, v_2, vend)$ is a left turn.

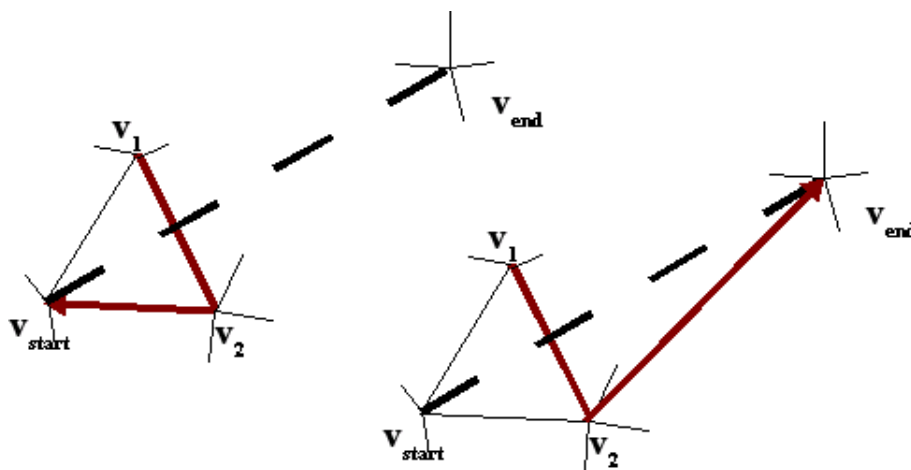


Figure 5.4: determining the start face

After the line-face circulator has produced a constrained edge, the intersection of that edge with the projection of the edge which is being processed needs to be inserted into the overlay triangulation. This is done with the `insert_vertex_in_edge` function shown in listing 5.12.

Listing 5.12: insert one vertex into a constrained edge of the overlay triangulation

```
Vertex_handle insert_vertex_in_edge(const Point pt, Face_handle f,
                                   int i, long object_id)
{
    VEdge c_edge; // the constrained edge that will be split

    // remove the constraint from the constrained edge and save the edge
    f->set_constraint(i,false); // remove the constraint in the cw face
    Face_handle nf = f->neighbor(i);
    int nb = nf->index(f);
    nf->set_constraint(nb,false); // remove the constraint in the ccw face
    c_edge.va = f->vertex(tar_tr.cw(i));
    c_edge.vb = f->vertex(tar_tr.ccw(i));

    // inset the point into the target triangulation
    long intersected_id = f->vertex(tar_tr.cw(i))->id();
    Vertex_handle vh = tar_tr.insert_in_edge(pt,f,i);
    vh->set_id(intersected_id);
    vh->set_second_id(object_id);

    // restore the constrained edge
    Edge_circulator ec = tar_tr.incident_edges(vh);
    Edge_circulator ec_done(ec);
    do {
        // if the edge is constrained move the constraint
        // to the correct edge in the face
        if ( (*ec).first->is_constrained ((*ec).second) ) {
            (*ec).first->set_constraint ((*ec).second,false);
            Face_handle opposite_face = (*ec).first->neighbor ((*ec).second);
            opposite_face->set_constraint (opposite_face->index(vh),true);
        }
        // else check if the edge is constrained in the adjacent face
        // and move the constraint if so
        else{
            Face_handle opposite_face = (*ec).first->neighbor ((*ec).second);
            int opposite_index = (*ec).first->mirror_index ((*ec).second);
            if ( opposite_face->is_constrained (opposite_index) ) {
                opposite_face->set_constraint (opposite_index,false);
                (*ec).first->set_constraint ((*ec).first->index(vh),true);
            }
        }
        // if the edge is the first part of the original constrained edge
        if ( (*ec).first->vertex(tar_tr.ccw ((*ec).second)) == c_edge.va ) {
            (*ec).first->set_constraint ((*ec).second,true);
            Face_handle opposite_face = ((*ec).first->neighbor ((*ec).second));
            int opposite_index = (*ec).first->mirror_index ((*ec).second);
            opposite_face->set_constraint (opposite_index,true);
        }
        // if the edge is the second part of the original constrained edge
        else if ( (*ec).first->vertex(tar_tr.cw ((*ec).second)) == c_edge.vb )
        {
            (*ec).first->set_constraint ((*ec).second,true);
            Face_handle opposite_face = ((*ec).first->neighbor ((*ec).second));
            int opposite_index = (*ec).first->mirror_index ((*ec).second);
            opposite_face->set_constraint (opposite_index,true);
        }
    } while (++ec != ec_done);

    // return the vertex handle of the inserted vertex
    return (vh);
}
```

This function works in the same way as the `insert_vertex` function which was discussed earlier. The standard CGAL function for inserting points into a triangulation is used to insert the intersection and then the scrambled constraints are restored. Because it is certain that the edge that is intersected is constraint, this edge is processed separately. The constraints on this edge are lifted temporarily and are placed on the split parts of the edge after the intersection has been inserted.

The details of the process of restoring the constraints has already been discussed in the previous sub-section. So a summary of the steps taken in the function will be sufficient. The function performs the following 4 steps:

1. remove the constraints of the intersected constrained edge
2. insert the intersection with the CGAL `insert_in_edge` function
3. rotate over the edges around the inserted vertex and restore the scrambled constraints
4. if an edge is reached that is part of the intersected constrained edge restore the constraint on that edge

The last custom function that the `insert_section` function uses is the `c_flip` function. As with the vertex insertion functions there is a flip function available in CGAL. This custom function adds an extra step to the original function in which the scrambled constraints are restored. The code of the `c_flip` is shown in listing 5.13.

Listing 5.13: flips edge `flip_edge` and restores the constraints in adjacent faces

```

void c_flip (Edge flip_edge)
{
    // get the faces sharing the edge
    Face_handle facel = (flip_edge).first;
    Face_handle face2 = (flip_edge).first->neighbor ((flip_edge).second);

    // save the constraints
    int amount_constr = 0;
    VEdge edges_constr[4];
    Edge c_edge;

    for (int i=0; i<3; i++) {
        if (facel->is_constrained (i)) {
            ++amount_constr;
            edges_constr[amount_constr].va = facel->vertex (tar_tr.cw (i));
            edges_constr[amount_constr].vb = facel->vertex (tar_tr.ccw (i));
        }
        if (face2->is_constrained (i)) {
            ++amount_constr;
            edges_constr[amount_constr].va = face2->vertex (tar_tr.cw (i));
            edges_constr[amount_constr].vb = face2->vertex (tar_tr.ccw (i));
        }
    }

    // remove the constraints
    facel->set_constraints (false,false,false);
    face2->set_constraints (false,false,false);

    // perform the flip
    tar_tr.flip ((flip_edge).first, (flip_edge).second);

    // restore the constraints
    for (int i=1; i<=amount_constr; i++) {
        tar_tr.is_edge (edges_constr[i].va, edges_constr[i].vb,
            (c_edge).first, (c_edge).second);
        (c_edge).first->set_constraint ((c_edge).second,true);
    }
}

```

During a flip the number of faces remains the same and the possible constraints in the quadrangle around the flipped edge don't move. Because of this it is easier to use a slightly different method of ensuring the correct placement of the constraints after the operation then with the previously discussed functions. The flip function saves the edges that are constraint in the same way the line-face circulator saved the edges that needed to be flipped, i.e. they are pushed onto a vector as vertex pairs.

```

for (int i=0; i<3; i++) {
    if (facel->is_constrained (i)) {
        ++amount_constr;
        edges_constr[amount_constr].va = facel->vertex (tar_tr.cw (i));
        edges_constr[amount_constr].vb = facel->vertex (tar_tr.ccw (i));
    }
    if (face2->is_constrained (i)) {
        ++amount_constr;
        edges_constr[amount_constr].va = face2->vertex (tar_tr.cw (i));
        edges_constr[amount_constr].vb = face2->vertex (tar_tr.ccw (i));
    }
}

```

```
    }  
}
```

After that the constraints are removed and the flip is performed using the CGAL flip function. The flip doesn't scramble any constraints now because all the constraints have been removed. When the edge is flipped the edges on the vector are converted from the vertex pair type to the face-index type and the constraints are placed back onto the edges of the quadrangle around the flipped edge.

```
for (int i=1; i<=amount_constr; i++) {  
    tar_tr.is_edge (edges_constr[i].va, edges_constr[i].vb,  
                    (c_edge).first, (c_edge).second);  
    (c_edge).first->set_constraint ((c_edge).second,true);  
}
```

5.4 Phase 3: re-assembling features

This implementation only needs to show the overlay triangulation with the result of the overlay. A complicated topology reconstruction isn't needed so the algorithm only needs to reassemble the features. The code that makes this happen is shown in listing 5.14.

Listing 5.14: reassemble the features

```
void reconstruct_objects()  
{  
    Object_locator_type current_object;  
    Face_handle first_face;  
  
    // reconstruct all objects that have an entry on the object vector  
    while (object_vector.size() > 0) {  
        // locate the first face that is part of the object  
        current_object = object_vector.back();  
        first_face = tar_tr.locate (current_object.locate_point,  
                                    current_object.locate_face);  
        // start the oil-spill algorithm that will reconstruct the object  
        locate_neighbors (first_face,current_object.object_id);  
        // remove the processed object from the object_vector  
        object_vector.pop_back();  
    }  
}
```

When the features were pre-processed, their feature identifier and a point within their convex hull was saved to a list. This list is now used to assign feature identifiers to the faces of the overlay triangulation. first the face is located that contains the saved point:

```
first_face = tar_tr.locate  
(current_object.locate_point,current_object.locate_face);
```

After this, the initialisation of the floodfill algorithm is complete. Both the first face and the feature identifier are known. The floodfill is performed by the locate_neighbors() function. Its code is shown in listing 5.15.

Listing 5.15: check if neighbouring faces belong to the same feature

```
void locate_neighbors (Face_handle current_face, long object_id)  
{  
    // set the id for the face  
    if (current_face->id() == 0) {
```

```

    current_face->set_id(object_id);
}
else {
    current_face->set_second_id(object_id);
}
// check all three neighbours of this face if they belong to the object
for (int i=0; i<3; i++) {
    // if the neighboring face doesn't have the object_id yet continue
    if ( !current_face->neighbor(i)->has_id(object_id) ) {
        // if the edge is not constrained continue
        if ( !(current_face->vertex(tar_tr.cw(i))->has_id(object_id) &&
            current_face->vertex(tar_tr.ccw(i))->has_id(object_id) &&
            current_face->is_constrained(i)) ) {
            // the neighbor also belongs to the object so process it
            locate_neighbors (current_face->neighbor(i),object_id);
        }
    }
}
}
}

```

The first thing that happens in this function is the assignment of the feature identifier to the current face. As mentioned in the introduction this implementation can only assign two different feature identifiers to a face. The function checks if the face already has a feature identifier assigned to it, if this is the case it assigns the new feature identifier to the second identifier of the face.

After the feature identifier has been assigned to the face, all three of the neighbouring faces are checked according to the procedure described in section 3.5. First a check is made to see if the neighbouring face has not been assigned a feature identifier before:

```

if ( !current_face->neighbor(i)->has_id(object_id) ) {

```

Then a check is made to see if the edge isn't constrained and if it is constrained to see if the constrained edge isn't part of the boundary of the feature which is being reassembled:

```

if ( !(current_face->vertex(tar_tr.cw(i))->has_id(object_id) &&
    current_face->vertex(tar_tr.ccw(i))->has_id(object_id) &&
    current_face->is_constrained(i)) ) {

```

If both of these checks are passed the face must be located inside the boundary of the feature which is being reassembled. This face is then passed to this function to get the feature identifier assigned to it and to check its neighbouring faces.

After this function finishes the last of the features in the object_vector the Qt library is used to generate an image of the final result of the overlay. Figure 5.5 shows a detail of a feature triangulation which was created using features from two different sources as input. The first data source was a set of 200 buildings of the Dutch city of Utrecht. The second data source was a set of 100 buildings from a different area in the same Dutch city. These buildings were shifted so that they would overlap partially with the buildings from the first data set. The blue lines are the unconstrained edges, the red lines are the constrained edges. The yellow faces and the grey faces are part of features from either of the two data sources. The white faces are part of a feature from both data sources. Black faces aren't part of any feature.

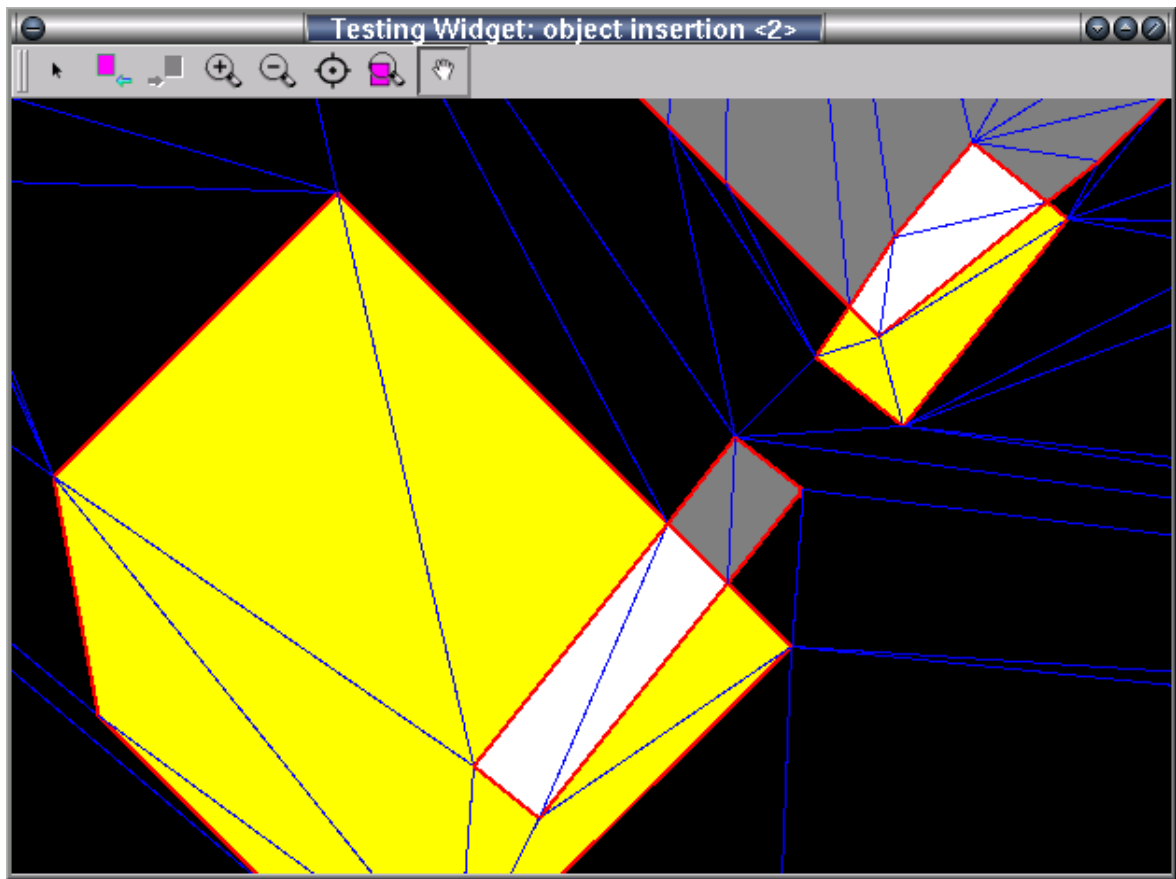


Figure 5.5: situation of neighbours and constraints in a face

6. Conclusions and recommendations

The main goal of the research explained in this thesis was to design an algorithm that could overlay three dimensional data using a tetrahedral network. To get to the results presented here, the main research question How can three dimensional data be overlayed using a tetrahedral network? was split up into the following five sub questions:

- What strategy should be used for overlaying data?
- What data would be required as input?
- How would that strategy work when applied to two dimensional data?
- Can the algorithm be implemented robustly in a finite precision system?
- Can the two dimensional overlay algorithm be translated to work in three dimensions using the same strategy?

The conclusions presented in section 6.1 will be structured in the same way as the research questions, the research strategy and the chapter structure of the thesis. First the research questions with respect to the two-dimensional case will be presented, followed by the three-dimensional case. Finally the results of the implementation will be analyzed. The time available for the graduation assignment was insufficient to explore every aspect of the overlay problem. The first steps to a mature three-dimensional overlay algorithm based on a tetrahedral data structure have been taken, but as with every research project undertaken there will always be unanswered questions and new ways of looking at something. Therefore section 6.2 contains some recommendations on future research.

6.1 Conclusions

Before any of the other research sub-questions could be answered, it was necessary to get an answer to the first question: What strategy should be used for overlaying data? A feature based approach was chosen. This kept the problem of the overlay itself relatively simple and made it possible to use a container algorithm that would describe the steps that needed to be taken in both two and three dimensions. Furthermore, instead of having to visualize large amounts of spatial data that needed to be overlayed, it was possible to start out with just two features and figure out how those could be combined in a compound view.

Although in general a feature overlay is less efficient than a map overlay because some data is processed more than once and because there are more exceptions that need to be handled, its use does have a big advantage. The overlay process can be paused or aborted when its running and there would still be a valid result for the features processed upto that moment. This makes it possible to start overlaying features even if not all features that need to be overlayed are available yet.

Two-dimensional overlay algorithm

The answer to the third research sub-question, 'Would that strategy work when applied to two dimensional data?', is yes. It is possible to overlay two-dimensional features by walking along the faces of a triangulation, calculating intersections and flipping edges. The algorithm as presented in chapter three has some nice properties:

- The algorithm only adds vertices at intersections of constrained edges
- Very little information about the features is needed to be able to overlay them
- With one exception, non-convex features, such as features with intrusions and holes are no problem for the algorithm
- the algorithm can easily be modified to behave as a map overlay algorithm instead of a feature overlay algorithm

By only adding vertices at intersections of constrained edges, the final overlay triangulation will contain the minimum number of vertices required to hold all the features that were overlayed. By keeping the number of vertices as low as possible, algorithms that depend on the number of vertices for their efficiency such as the triangle walk algorithm and the section insertion algorithm presented in chapter three perform at optimal speed. A downside to this property is that the shape of the faces in the final triangulation is not optimal, Delauney or otherwise. Without the ability to insert extra

vertices sliver polygons can even pose problems when implementing the algorithm in a finite precision system.

Not only does the algorithm produce the triangulation with the least amount of vertices as output, it also requires very little data as input. Only two things are required to make it possible for the algorithm to overlay features: directed feature boundaries with attached feature identifiers. With information about the neighbours of features the algorithm is easily adjustable to behave as a map overlay algorithm instead of a feature overlay algorithm. This makes it possible to process planar maps more efficiently.

Three-dimensional overlay algorithm

To make the three-dimensional overlay algorithm work in the same way as the two-dimensional algorithm the container algorithm needs to be extended with one extra step. Inserting edges is not sufficient to ensure that the boundaries of all three-dimensional features are present in the overlay tetrahedralization. After the edges of a feature have been inserted, the facets that make up the boundary of the feature in the overlay tetrahedralization need to be constructed explicitly.

The two-dimensional sub-algorithms used to fill the container algorithm to create the two-dimensional feature overlay algorithm can be translated to three-dimensional sub-algorithms with exception of the sub-algorithm that relies on flipping tetrahedrons to create constraints in the overlay tetrahedralization. In the two-dimensional situation it was possible to prove that flipping edges would result into the insertion of the feature boundaries. This proof doesn't exist (yet) for arbitrary tetrahedralizations.

At the moment this problem can be solved either by using conformal tetrahedralizations instead of constrained tetrahedralizations [Shewchuk, 2003] or by accepting that flipping won't work in 3% of the cases [Anwei, 2000]. Using conformal triangulations nullifies the attempt to keep the number of tetrahedrons needed to represent the overlay as small as possible because a lot of extra vertices will have to be added to make sure the boundaries of the features are represented in the overlay tetrahedralization [Cavalcanti, 1999]. A solution might be a combination of both strategies: use the flipping algorithm as intended in this thesis when possible, and when a situation occurs in which that doesn't work add extra vertices to get the representation of the boundary in the tetrahedralization.

This solution as well as a strategy that only involves flipping if that becomes possible both have the same downside as the two-dimensional algorithm. The resulting triangulation is not optimized, Delaunay or otherwise, in any way. Even if the triangulation of the feature boundaries is optimized, this optimization won't be carried on to the result of the feature overlay.

Implementation of two-dimensional overlay algorithm

Due to development setbacks, the implementation of the two-dimensional algorithm was not completed to the point where it could be used to test the algorithm for errors that might occur in a finite precision environment. The development was invaluable to the design process of the algorithm however and the problems that arose gave some insight in the complexity of the algorithm.

Implementation showed that the algorithm leans heavily on the explicit availability of edges in the data structure. The triangulation data structure used by CGAL did not contain edges explicitly. Edges were defined implicitly by the vertices opposite to them in the same face. This made it difficult to implement the triangle walk algorithm and many CGAL functions had to be rewritten to be able to implement it.

Early in the implementation process it also became clear that all the exceptions mentioned in section 3.4.5 need to be handled meticulously before the algorithm becomes robust enough to be able to process real data sets. Although most potential problems with finite precision could not be tested, it did become clear that it is important to define when a vertex lies on an edge and functions that validate these situations should be available. Both during the insertion of sections and the reconstruction of the boundaries after the overlay it is important to know if a vertex lies on an edge or not.

6.2 Recommendations

The method for overlaying features presented in this thesis seems promising. A lot of work still needs to be done before it can be used effectively though. Before an algorithm that overlays features by flipping in a tetrahedral network will produce meaningful results more research will have to be

done on at least the following for subjects:

- possible finite precision problems
- proving or refute the conjecture presented in [Joe, 1992]
- speed of the algorithm for different data sets
- implementation of the algorithm in 3D to see if a 97% of the cases is sufficient to turn the concept of flipping into an effective overlay algorithm [Anwei, 2000]

During the implementation of the two-dimensional overlay algorithm it already became clear that finite precision can become a problem if not handled with care. So even if proof is given that the conjecture presented in [Joe, 1992] is true and that theory is implemented, the implementation will still need some way to deal with issues such as floating point errors and sliver polyhedrons.

When you know an algorithm produces the required results, it is also useful to know how fast it produces these results. Especially if the algorithm is as complicated as a feature overlay algorithm and if there are huge amounts of data that the algorithm needs to be able to process. This algorithm uses a triangle (or tetrahedron) walk algorithm. Although it is possible to give the time complexity of such algorithms it is not possible to give the theoretical efficiency of the algorithm [Sundareswara, 2003]. So the speed of the feature overlay algorithm will have to be determined by testing it out with lots of different data sets.

If no proof is found for the conjecture presented in [Joe, 1992], or if it is refuted. The strategy of locally allowing the tetrahedralization to become conformal when features can't be overlayed using flips is worth looking at. The same issues of finite precision and speed also apply to this scenario.

Glossary

Boundary

Description of the geometry of a feature. The boundary of a feature consist of a number of nodes connected by edges.

Cell

Largest building blocks of tetrahedral networks. A cell consists of four facets aranged in a tetrahedral shape.

CGAL

The Computational Geometry Algorithms Library. It makes the most important of the solutions and methods developed in computational geometry available to users in industry and academia in a C++ library

Constrained edge

An edge in the overlay triangulation which represents a section of the boundary of a feature.

Constrained facet

A facet in the overlay tetrahedral network which represents part of the boundary of a three dimensional feature.

Container algorithm

General description of the steps that need to be taken to perform a feature overlay. These steps, or containers, need to be filled with algorithms that perform the tasks that need to be performed in the described steps.

Facet

Triangular shaped entity in a tetrahedral network. Facets are part of the boundary of cells in a tetrahedral network.

Feature overlay

Placing multiple features from different thematic sources in precise registration, with the same scale, projections, and extent, so that a compound view is possible.

Feature set

A feature set is a collection of features which are all available at the moment the feature overlay algorithm starts. Features in these feature sets can originate from different sources such as different tables in the same database, different databases, text-based files and map-layers, as long as all the features can be accessed by the feature overlay algorithm at any time after the algorithm has started.

Image rendering pipe-line

This term is used in computer graphics to denote all the algorithms that are used sequentially to create a flat image (usualy on a computer screen) from a multi-dimensional model in memory.

Node

Point on the boundary of a feature where two edges intersect.

Non-constrained edge

An edge in the overlay triangulation which doesn't represents a section of the boundary of a feature.

Non-constrained facet

A facet in the overlay tetrahedral network which isn't part of the boundary of a three dimensional feature.

Overlay triangulation

A triangulation which is used by the algorithm to overlay the features from the feature set. The triangulation consists of a collection of vertices connected by edges. These edges form a network of triangles: the faces of the triangulation.

Point

The geometric properties of a vertex or node, i.e. its co-ordinates.

Qt

Qt is a complete C++ application development framework, which includes a class library and tools for multiplatform development and internationalization.

Section

Part of an edge of the boundary of a feature. Sections are created by projecting an edge onto the overlay triangulation and cutting the edge up at every intersection of the projection of the edge with a constraint (vertex or constrained edge) in the overlay triangulation.

Segment

The geometric description of an edge (either of a feature or in a triangulation).

STL

The STL library, or Standard Template Library, is a general-purpose library of generic algorithms and data structures.

Triangle

The geometric description of a face, i.e. the co-ordinates of the points of the vertices that make up a face.

Vertex

Point in a triangulation where two edges intersect.

References

- [Aichholzer, 2002], Aichholzer, O., Alboul, L.S., & Hurtado, F., 2002, *On flips in polyhedral surfaces*, International Journal of Foundations of Computer Science (IJFCS), special issue on Volume and Surface Triangulations, 13(2):303-311, Institute for Theoretical Computer Science, Graz University of Technology, Graz, Austria
- [Amanatides, 1996], Amanatides, J., Choi, K., 1996, *Ray tracing triangular meshes*, Dept. of Computer Science, York University, Canada
- [Anwei, 2000], Anwei, L., Baida, M., 2000, *How far flipping can go towards 3D conforming/constrained triangulation*, IMR 2000, New Orleans, Louisiana, USA, 307-315
- [Arens, 2003], Arens, C., Stoter, J.E., Van Oosterom, P., 2003, *Modelling 3D spatial objects in a geo-DBMS using a 3D primitive*, proceedings of the 6th AGILE, Dares-Lieu, Lyons
- [Baker, 1998], Baker, T.J., Vassberg, J.C., 1998, *Tetrahedral Mesh Generation and Optimization*, Numerical Grid Generation in Computational Field Simulations, Proceedings of the 6th International Conference, held at the University of Greenwich, pp.337-349
- [Bernal, 1995], Bernal, J., 1995, *Inserting line segments into triangulations and tetrahedralizations*, National Institute of Standards and Technology, Gaithersburg, MD, U.S.
- [Bronsvoort, 2001], Bronsvoort, W.F., Noort, A., Post, F.H., 2001, *Geometrisch modelleren (dutch)*, Faculteit Informatietechnologie en Systemen, Delft University of Technology, Netherlands
- [Cavalcanti, 1999], Cavalcanti, P.R., Mello, U.T., 1999, *Three-dimensional constrained delauney triangulation: a minimalist approach*, Proceedings, 8th International Meshing Roundtable, South Lake Tahoe, CA, U.S.A., pp.119-129
- [Chen, 1994], Chen, X., Ikeda, K., Yamakita, K., Nasu, M., 1994, *Three-dimensional modeling of GIS based on Delaunay tetrahedral tessellations*, proceedings of SPIE, volume 2357, pag. 132-139
- [Clarke, 1999], Clarke, K.C., 1999, *Getting Stated with Geographic Information Systems, 2nd ed.*,
- [Clementini, 1993], Clementini, E., Di Felice, P. & Van Oosterom, P., 1993, *A small set of formal topological relationships suitable for end-user interaction*, Advances in Spatial Databases, Third International Symposium SSD, Singapore
- [De Floriani, 1998], De Floriani, L., Magillo, P. & Puppo, E., 1998, *Applications of computational geometry to geographic information systems*, Universita di Genova, Genova, Italy
- [Dodge, 1997], Dodge, M., Smith, A., Doyle, S., 1997, *Visualising urban environments for planning and design*, Centre for advanced spatial analysis (CASA), University College London
- [Field, 1986], Field, D.A., 1986, *Implementing Watson's algorithm in three dimensions*, ACM on computational geometry, New York, US
- [Franklin, 1992], Franklin, W.R., 1992, *Volumes from overlaying 3D triangulations in parallel*, Rensselaer Polytechnic Institute Troy, New York, US
- [Goodrich, 1997], Goodrich, M.T., Orletsky, M., & Ramaiyer K., 1997, *Methods for Achieving Fast Query Times in Point Location Data Structures*, th ACM-SIAM Symp. on Discrete Algorithms (SODA)
- [Hurtado, 1999], Hurtado, F., Noy, M. & Urrutia, J., 1999, *Flipping edges in triangulatoins*, Discrete and Computational Geometry 22:333-346
- [Joe, 1992], Joe, B., 1992, *Construction of three-dimensional improved-quality triangulations using local transformations*, University of Alberta, Edmonton, Canada
- [Kanaganathan, 1991], Kanaganathan, S. & Goldstein, N.B., 1991, *Comparison of four-point adding algorithms for Delaunay-type three-dimensional mesh generators*, IEEE transactions on magnetics, vol 27 no. 3, 3444-3451
- [Kriegel, 1992], Kriegel, H.P., Brinkhoff, T. & Schneider, R., 1992, *An efficient map overlay algorithm based on spatial access methods and computational geometry*, Geographic Database Management Systems, Workshop Proceedings, Capri, Springer-Verlag, 194-211

- [Masumoto, 2002], Masumoto, S., Raghavan, V., Nemoto, T., Shiono, K., 2002, *Construction and visualization of three dimensional geologic model using GRASS GIS*, Proceedings of the Open source GIS-GRASS users conference 2002, Trento, Italy
- [Mehlhorn, 1999], Mehlhorn, K., Naher, St., 1999, *The LEDA platform of combinatorial and geometric computing*, Cambridge University Press
- [Midtbo, 1993/1], Midtbo, T., 1993, *patial modelling by Delaunay networks of two and three dimensions, dr.ing. thesis*, Norwegian Institute of Technology, University of Trondheim, Trondheim, Norway
- [Midtbo, 1993/2], Midtbo, T., 1993, *Incremental Delaunay tetrahedrization for adaptive data modelling*, Proceedings of the Fourth European Conference and Exhibition on Geographical Information Systems, EGIS '93, vol. 1: 227-236, Genoa: EGIS Foundation
- [Molenaar, 1992], Molenaar, M., 1992, *A topology for 3D vector maps*, ITC Journal 1992-1, 25-33, Wageningen Agricultural University, Wageningen
- [Mucke, 1996], Mucke, E.P., Saias, I. & Zhu, B., 1996, *Fast randomised point location without pre-processing in two- and three-dimensional Delaunay triangulations*, Proceedings of the 12th annual symposium on computational geometry, Los Alamos, New Mexico
- [Musser, 2001], Musser, D.R., Derge, G.J., Saini, A., 2001, *STL Tutorial and reference guide, Second edition, C++ Programming with the Standard Template Library*, Addison Wesley
- [Nigro, 2002], Nigro, J.D., Ungar, P.S., de Ruiter, D.J., Berger, L.R., 2002, *Developing a geographic information system (GIS) for mapping and analysing fossil deposits at Swartkrans, Gauteng Province, South Africa*, Journal of Archaeological Science (2003) 30, 317-324
- [O'Rourke, 1998], O'Rourke, J., 1998, *Computational Geometry in C second edition*, Cambridge university press, Cambridge university press, Cambridge, UK
- [Oosterom, 1994], Van Oosterom, P., Vertegaal, W. & Van Hekker, M., 1994, *Integrated 3D modelling within a GIS, revised paper for AGDM'94*, Delft, The Netherlands
- [Pigot, 1991], Pigot, S., 1991, *Topological models for 3D spatial information systems*, Environmental systems research institute, New York, US
- [Raper, 1992], Raper, J.F., 1992, *Key 3D modelling concepts for geoscientific analysis*, Turner K. (ed) three-dimensional modelling with geoscientific information systems, 215-32, Dordrecht, Kluwer
- [Rossignac, 1999/1], Rossignac, J., 1999, *Edgebreaker: connectivity compression for triangle meshes*, IEEE transactions on visualization and computer graphics, vol. 5, no. 1, 47-61
- [Rossignac, 1999/2], Rossignac, J. & Szymczak, A., 1999, *Wrap&Zip decompression of the connectivity of triangle meshes compressed with Edgebreaker*, Computational Geometry 14, 119-135, Elsevier Science BV. The Netherlands
- [Rossignac, 2001], Rossignac, J., Safonova, A. & Szymczak, A., 2001, *3D compression made simple: edgebreaker on a corner-table*, lecture at Shape Modelling International Conference, Genoa, Italy
- [Schenkelaars, 1995], Schenkelaars, V., Van Oosterom, P., 1995, *Map-Overlay within a Geographic Interaction Language*, presented at Auto-Carto 12, Charlotte NC, pag. 281-290
- [Shewchuk, 1996], Shewchuk, J.R., 1996, *Robust Adaptive Floating-Point Geometric Predicates*, Proceedings of the Twelfth Annual Symposium on Computational Geometry, pages 141-150, Philadelphia, Pennsylvania, Association for Computing Machinery
- [Shewchuk, 1997], Shewchuk, J.R., 1997, *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*, Discrete & Computational Geometry 18, (3):305-363
- [Shewchuk, 2002], Shewchuk, J.R., 2002, *Constrained delaunay tetrahedralizations and provably good boundary recovery*, Proceedings, 11th international meshing roundtable, Sandia National Laboratories, p.193-204
- [Shewchuk, 2003], Shewchuk, J.R., 2003, *Updating and Constructing Constrained Delaunay and Constrained Regular Triangulations by Flips*, Nineteenth Annual Symposium on Computational Geometry, Berkeley, US

- [**Silva, 1997**], Silva, C.T., Mitchell, J.S.B., 1997, *The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids*, IEEE transactions on visualization and computer graphics, vol. 3, no. 2, p.142-157
- [**Snoeyink, 1997**], Snoeyink, J., 1997, *Point location*, Handbook of discrete and Computational Geometry, Discrete Mathematics and its Applications, chapter 30 p559-574, by Goodman, O'Rourke, CRC Press, Boca Raton
- [**Stoter, 2004**], Stoter, J.E., 2004, *3D Cadastre*, Publications on Geodesy 57, NCG, Delft, the Netherlands
- [**Sundareswara, 2003**], Sundreswara, R., Schrater, P., 2003, *Extensible point location algorithm*, 2003 International Conference on Geometric Modeling and Graphics (GMAG'03), Londen, England, pag. 84-89
- [**Van Oosterom, 1994**], Van Oosterom, P., 1994, *An R-tree based Map-Overlay Algorithm*, proceedings EGIS/MARI94, Paris, pag. 318-327
- [**Van Oosterom, 2004**], Van Oosterom, P., Quak, W., Tijssen, T., 2004, *About invalid, valid and clean polygons*, Delft University of Technology, OTB, GIS Technology, The Netherlands
- [**Vries, 2001**], De Vries, J., 2001, *Driedimensionale buffering op basis van tetraeder netwerken (Dutch)*, Delft University of Technology, Netherlands
- [**Watt, 2000**], Watt, A.H., 2000, *3D computer graphics, 3rd ed.*, Addison-Wesley publishing company

Appendix A: source code for 2D feature overlay algorithm

```
// -----
// constrained_face_2.h defines a new face class that can
// be used to keep track of constrained faces within a
// triangulation.
//
// This class inherits all properties from the CGAL
// Triangulation_face_base_2 class. Methods involving
// vertex and or edge manipulation have been overridden.
//
// This class does not enforce the constraints. Enforcing
// and managing the constraints should be done by the
// triangulation class in which this face class is used.
//
// Edge i is defined as the edge opposing vertex i. This
// is the same way of identifying edges in a face as
// suggested in the CGAL documentation.
//
// -----

#include <CGAL/Triangulation_short_names_2.h>

// Add the new class to the CGAL namespace
CGAL_BEGIN_NAMESPACE

template <class Gt>
class Constrained_vertex_2
{
public:
    typedef typename Gt::Point_2 Point; // The point definition from the geometric traits class

    // Constructor (empty): creates regular triangulation_vertex_base_2
    // and sets the id to 0
    Constrained_vertex_2()
        : _p(Point()), _f(NULL), _id(0) { }

    // Constructor (point (face) ID): creates triangulation_vertex_base_2 at the
    // given point and sets the id to vertex_id
    Constrained_vertex_2(const Point & p, void *f = NULL, long vertex_id = 0)
        : _p(p), _f(f), _id(vertex_id) { }

    void set_point(const Point & p) { _p = p; }
    void set_face(void* f) { _f = f; }
    void set_id(const long vertex_id) { _id = vertex_id; }
    void set_second_id(const long vertex_id) { _sid = vertex_id; }

    const Point& point() const { return _p; }
    void* face() const { return _f; }
    const long id() const { return _id; }
    const long sid() const { return _sid; }
    const bool has_id(const long vertex_id) {return((vertex_id == _id) || (vertex_id == _sid));}

    // the non const version of point() is undocumented
    // but it is needed to make the point iterator work
    // using Lutz projection scheme
    Point& point() { return _p; }

    bool is_valid(bool /* verbose */ = false, int /* level */ = 0) const
    {return true;}

private:
    Point _p;
    void * _f;
    long _id; // The id of the vertex
    long _sid; // The second id of the vertex when it is contained in two objects
};

CGAL_END_NAMESPACE
```

```

// -----
// constrained_face_2.h defines a new face class that can
// be used to keep track of constrained faces within a
// triangulation.
//
// This class inherits all properties from the CGAL
// Triangulation_face_base_2 class. Methods involving
// vertex and or edge manipulation have been overridden.
//
// This class does not enforce the constraints. Enforcing
// and managing the constraints should be done by the
// triangulation class in which this face class is used.
//
// Edge i is defined as the edge opposing vertex i. This
// is the same way of identifying edges in a face as
// suggested in the CGAL documentation.
//
// -----

#include <CGAL/triangulation_assertions.h>
#include <CGAL/Triangulation_short_names_2.h>
#include <CGAL/Triangulation_face_base_2.h>

// Add the new class to the CGAL namespace
CGAL_BEGIN_NAMESPACE

template <class Gt>
class Constrained_face_2
: public Triangulation_face_base_2<Gt>
{
public:
    typedef Gt Geom_traits; // Geometric traits used for the
    points
    typedef Triangulation_face_base_2<Gt> Fab; // The parent face class
    typedef Constrained_face_2<Gt> Constrained_face_base; // The new constrained face class
    typedef typename Gt::Point_2 Point; // The point definition from the
    geometric traits class

protected:
    bool C[3]; // The constraints of the edges of the face (edge is constrained if true)
    long _id; // The ID of the object this face belongs to
    long _sid; // The ID of the second object if this face belongs to 2 objects

public:
    // Constructor (empty): creates regular triangulation_face_base_2
    // and sets all constraints to false
    Constrained_face_2()
    : Fab()
    {
        set_constraints(false,false,false);
        set_id(0);
    }

    // Constructor (vertices): creates triangulation_face_base_2 through the
    // given vertices and sets constraints to false
    Constrained_face_2(void* v0, void* v1, void* v2)
    : Fab(v0,v1,v2)
    {
        set_constraints(false,false,false);
        set_id(0);
    }

    // Constructor (vert. neighb.): creates triangulation_face_base_2 through the
    // given vertices and sets neighbours and sets
    // constraints to false
    Constrained_face_2(void* v0, void* v1, void* v2,
        void* n0, void* n1, void* n2)
    : Fab(v0,v1,v2,n0,n1,n2)
    {
        set_constraints(false,false,false);
        set_id(0);
    }

    // Constructor (vert. neighb. constr.): creates triangulation_face_base_2 through
    // the given vertices and sets neighbours and

```

```

// sets constraints
Constrained_face_2(void* v0, void* v1, void* v2,
                  void* n0, void* n1, void* n2,
                  bool c0, bool c1, bool c2,
                  long face_id = 0)
    : Fab(v0,v1,v2,n0,n1,n2)
{
    set_constraints(c0,c1,c2);
    set_id(face_id);
}

bool is_constrained(int i) const ;           // returns true if edge i is constrained
void set_constraints(bool c0, bool c1, bool c2) ; // sets the constraints of all edges
void set_constraint(int i, bool b);           // set constraint of edge i

void set_id(long face_id);                   // sets the first id for the face
void set_second_id(long face_id);            // sets the second id for the face
long id();                                   // returns the first id for the face
long sid();                                  // returns the second id for the face
bool has_id(const long face_id);              // returns true if the face has face_id as an id

void reorient();                             // flip the face (cw to ccw and vv)
void ccw_permute();                           // rotate vertices counter clock wise
void cw_permute();                             // rotate vertices clock wise
bool is_valid(bool verbose = false, int level = 0) const; // true if face is valid

};

// sets the constraints of all edges
template <class Gt>
inline void
Constrained_face_2<Gt>::
set_constraints(bool c0, bool c1, bool c2)
{
    C[0]=c0;
    C[1]=c1;
    C[2]=c2;
}

// sets the constraint of edge i
template <class Gt>
inline void
Constrained_face_2<Gt>::
set_constraint(int i, bool b)
{
    // don't know what this line does (think it has something to
    // do with debugging) but it is used in parent class when accessing
    // individual vertices.
    CGAL_triangulation_precondition( i == 0 || i == 1 || i == 2);
    C[i] = b;
}

// returns true if edge i is constrained
template <class Gt>
inline bool
Constrained_face_2<Gt>::
is_constrained(int i) const
{
    return(C[i]);
}

// sets the first id for this face
template <class Gt>
inline void
Constrained_face_2<Gt>::
set_id(long face_id)
{
    _id = face_id;
}

// sets the second id for this face
template <class Gt>
inline void
Constrained_face_2<Gt>::
set_second_id(long face_id)
{

```

```

    _sid = face_id;
}

// returns the first face id
template <class Gt>
inline long
Constrained_face_2<Gt>::
id()
{
    return(_id);
}

// returns the second face id
template <class Gt>
inline long
Constrained_face_2<Gt>::
sid()
{
    return(_sid);
}

// returns true if the face has face_id as an id
template <class Gt>
inline bool
Constrained_face_2<Gt>::
has_id(const long face_id)
{
    return( (face_id == _id) || (face_id == _sid) );
}

// changes orientation of the face by swapping vertex 0 and 1
template <class Gt>
inline void
Constrained_face_2<Gt>::
reorient()
{
    Fab::reorient(); // reorient vertices using parent function
    set_constraints(C[1],C[0],C[2]); // reorient the constraints to match
}

// rotate the vertices of the face in counter clockwise direction
template <class Gt>
inline void
Constrained_face_2<Gt>::
ccw_permute()
{
    Fab::ccw_permute(); // rotate vertices using parent function
    set_constraints(C[2],C[0],C[1]); // rotate the constraints
}

// rotate the vertices of the face in clockwise direction
template <class Gt>
inline void
Constrained_face_2<Gt>::
cw_permute()
{
    Fab::cw_permute(); // rotate vertices using parent function
    set_constraints(C[1],C[2],C[0]); // rotate the constraints
}

// returns true if the face is valid
// copied this part from CGAL source (is used for debugging)
// didn't feel like deciphering this and compiler doesn't
// seem to complain so there's no need for me to do so at
// the moment.
template <class Gt>
inline bool
Constrained_face_2<Gt>::
is_valid(bool verbose, int level) const
{
    bool result = Fab::is_valid(verbose, level);
    CGAL_triangulation_assertion(result);
    if (dimension() == 2) {
        for(int i = 0; i < 3; i++) {
            Constrained_face_base* n =
                static_cast<Constrained_face_base*>(neighbor(i));

```

```

        if(n != NULL){
            int ni = n->face_index(this);
            result = result && ( is_constrained(i) == n->is_constrained(ni));
        }
    }
}
return (result);
}

CGAL_END_NAMESPACE

```

```

// -----
//
// 2D_feature_insertion.C implements the insertion of
// features into a triangulation using flips
//
// -----

```

```

#include <CGAL/Homogeneous.h>
#include <CGAL/Triangulation_2.h>
#include "Constrained_face_2.h"
#include "Constrained_vertex_2.h"
#include <fstream>
#include <iostream>
#include <vector>
#include <CGAL/predicates_on_points_2.h>
#include <CGAL/Object.h>
#include <CGAL/intersections.h>
#include <CGAL/leda_real.h>

```

```

#include <qapplication.h>
#include <qmainwindow.h>
#include <CGAL/IO/Qt_widget.h>
#include <CGAL/IO/Qt_widget_standard_toolbar.h>
#include <CGAL/IO/Qt_widget_Triangulation_2.h>

```

```

//Define point and segment types based on integer coordinates
typedef CGAL::Homogeneous<leda_real> Rep;
typedef CGAL::Constrained_vertex_2<Rep> Vb;
typedef CGAL::Constrained_face_2<Rep> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb> Tds;
typedef CGAL::Triangulation_2<Rep,Tds> Triangulation;
typedef Rep::Point_2 Point;
typedef Rep::Segment_2 Segment;

```

```

typedef Triangulation::Vertex_circulator      Vertex_circulator;
typedef Triangulation::Edge_circulator        Edge_circulator;
typedef Triangulation::Face_circulator        Face_circulator;
typedef Triangulation::Finite_vertices_iterator Finite_vertices_iterator;
typedef Triangulation::Finite_edges_iterator  Finite_edges_iterator;
typedef Triangulation::Finite_faces_iterator  Finite_faces_iterator;
typedef Triangulation::Line_face_circulator   Line_face_circulator;

```

```

typedef Triangulation::Vertex_handle  Vertex_handle;
typedef Triangulation::Face_handle    Face_handle;
typedef Triangulation::Edge           Edge;
typedef Triangulation::Face           Face;
typedef Triangulation::Triangle        Triangle;

```

```

// structure used to create lists of edges
typedef struct{
    Vertex_handle va;
    Vertex_handle vb;
} VEdge;

```

```

// structure used to reconstruct the objects after they have been inserted
typedef struct{
    Point      locate_point; // a point inside the object
    Face_handle locate_face; // a face that contains, or is close to the locate point
    long       object_id;    // the id of the object

```

```

} Object_locator_type;

Triangulation tar_tr; // The target triangulation (holds the final result)
Triangulation obj_tr; // The object triangulation (temp. holds triangulated separate objects)

//Define the window in which the object will be shown
class My_window : public QMainWindow{
    Q_OBJECT

public:
    My_window(int x, int y)
    {
        //Set up the window and its properties
        widget = new CGAL::Qt_widget(this);
        setCentralWidget(widget);
        resize(x,y);
        widget->show();
        widget->lock();
        widget->setBackgroundColor(CGAL::BLACK);
        widget->setPointSize(10);
        widget->setPointStyle(CGAL::DISC);
        window_leftx = 999999999;
        window_rightx = 0;
        window_lowery = 999999999;
        window_uppery = 0;
        loaded_object_id = 1;

        // ask the file name of the file with object data and the number of objects to load
        char inp_file[256];
        std::cout << "Give filename of file with object data: ";
        std::cin >> inp_file;
        long maxobjects = 0;
        std::cout << "Give number of objects to load: ";
        std::cin >> maxobjects;

        //Load the objects from the file and insert them into the target triangulation
        while ((load_next_object(inp_file,loaded_object_id)) && (loaded_object_id <= maxobjects))
        { // load an object into obj_tr
            std::cout << "Object " << loaded_object_id << " loaded..." << std::endl;
            insert_obj(loaded_object_id); // insert the object into tar_tr
            std::cout << "Object " << loaded_object_id << " inserted successfully." << std::endl;
            ++loaded_object_id;
        }

        // if not all objects are loaded, remove the last entry on the object_vector
        if (loaded_object_id > maxobjects) object_vector.pop_back();

        // reconstruct the object topology
        reconstruct_objects();

        widget->set_window (window_leftx,window_rightx,window_lowery,window_uppery);

        //How to attach the standard toolbar
        std_toolbar = new CGAL::Qt_widget_standard_toolbar(widget, this);
        this->addToolBar(std_toolbar->toolbar(), Top, FALSE);

        //Set up the redraw function
        connect(widget, SIGNAL(custom_redraw()),
            this, SLOT(redraw_win()) );
        widget->unlock();
    }

private slots: //functions

    // -----
    // Loads the points of the next object into obj_tr
    // -----
    bool load_next_object(char* input_file, long object_id)
    {
        // set-up the iterator for the input file if not already set-up
        static ifstream data(input_file);
        static std::istream_iterator < Point > it(data);
        static std::istream_iterator < Point > end;
        Vertex_handle new_vertex;

        // if EOF hasn't been reached load points, until (1,1,1) is read
    }

```

```

if (it != end) {
    Point dummy(1,1,1);
    while (*it != dummy) {
        // update the coordinates for displaying the objects
        if (to_double ((*it).x()) < window_leftx) window_leftx = to_double ((*it).x());
        if (to_double ((*it).x()) > window_rightx) window_rightx = to_double ((*it).x());
        if (to_double ((*it).y()) < window_lowery) window_lowery = to_double ((*it).y());
        if (to_double ((*it).y()) > window_uppery) window_uppery = to_double ((*it).y());

        new_vertex = obj_tr.insert (*it); // insert point into the object triangulation
        new_vertex->set_id (object_id);    // set the object id for the vertex
        ++it;                             // move to the next point in the file
    }

    // move past the dummy (1,1,1) in the file
    ++it;

    // determine a point that lays inside of the object and store it in the object_vector
    Finite_faces_iterator ffi = obj_tr.finite_faces_begin();
    Face_handle object_face = ffi;
    Point p0, p1, p2;
    p0 = object_face->vertex(0)->point();
    p1 = object_face->vertex(1)->point();
    p2 = object_face->vertex(2)->point();
    // determine the centre of mass of the face ((x0+x1+x2)/3, (y0,y1,y2)/3)
    Point object_point ((leda_real)(to_double(p0.x())+to_double(p1.x())+to_double(p2.x())),
                       (leda_real)(to_double(p0.y())+to_double(p1.y())+to_double(p2.y())),
                       (leda_real) 3);
    Object_locator_type object_locator; // set the values for the object locator
    object_locator.locate_point = object_point;
    object_locator.object_id = object_id;
    object_locator.locate_face = object_face;
    object_vector.push_back(object_locator);

    return (true);
}
else {
    return (false);
}
}

// -----
// determine the convex hull
// -----
void det_hull()
{
    int infinite_index;          // index of the infinite vertex
    Face i_face;                 // an infinite face
    int opposite_index;          // index of the vertex indicating the constrained edge
    Face_handle opposite_face;    // handle to a face neighbouring an infinite face
    Vertex_handle iv = obj_tr.infinite_vertex(); // handle to the infinite vertex
    Face_circulator fc = obj_tr.incident_faces(obj_tr.infinite_vertex());
    Face_circulator fc_end(fc);  // around the infinite vertex

    // remove the vertices that aren't on the convex hull
    Finite_vertices_iterator fvi = obj_tr.finite_vertices_begin();
    Finite_vertices_iterator fvi_end = obj_tr.finite_vertices_end();

    do {
        if ( !obj_tr.is_edge (fvi,obj_tr.infinite_vertex()) ) obj_tr.remove (fvi);
    } while (++fvi != fvi_end);

    // set constraints for convex hull
    if (fc != 0) {
        do {
            i_face = *fc;
            infinite_index = i_face.index (iv); // extract face from iterator
            opposite_face = fc->neighbor (infinite_index); // get index of infinite vertex
            opposite_index = fc->mirror_index(infinite_index);
            opposite_face->set_constraint(opposite_index,true);
        } while(++fc != fc_end);
    }
}

```



```

// -----
// iterate over the edges in the object triangulation and add them to the triangulation
// -----
void insert_obj(long object_id)
{
    Vertex_handle first_inserted; // the first vertex that is inserted
    Vertex_handle last_inserted;  // the last vertex inserted upto the current vertex
    Vertex_circulator vc(obj_tr.infinite_vertex());
    Vertex_circulator vc_end(vc);

    // if tar_tr is empty, simply copy obj_tr into it
    if (tar_tr.number_of_vertices() == 0) {
        det_hull(); // determine it's convex hull
        tar_tr.swap (obj_tr); // swap the triangulations
        obj_tr.clear(); // clear the object triangulation
    }
    // if tar_tr isn't empty, add the edges of the object one by one
    else {
        // insert the first vertex into the target triangulation
        first_inserted = insert_vertex ((vc)++->point(),object_id);
        last_inserted = first_inserted;
        object_vector.back().locate_face = first_inserted->face();
        // insert the edges upto the last edge
        long edgenr = 0;
        do {
            std::cout << " Inserting edge: " << ++edgenr << std::endl;
            insert_edge (last_inserted,vc->point(),object_id);
            std::cout << " Edge " << edgenr << " inserted successfully..." << std::endl;
        } while (++vc != vc_end);
        // insert the last edge
        std::cout << " Inserting last edge..." << std::endl;
        insert_edge (last_inserted,first_inserted,object_id);
        std::cout << " Last edge inserted successfully." << std::endl;
        // clear the obj_tr so that another object can be added
        obj_tr.clear();
    }
}

// -----
// inserts one vertex into the target triangulation and restores the constraints
// -----
Vertex_handle insert_vertex(const Point pt, long object_id)
{
    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    // situations of point on edge and point on vertex are not detected and handled yet
    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    // insert the vertex into the target triangulation
    Vertex_handle vh = tar_tr.insert (pt);
    vh->set_id(object_id);

    // if the point inside the convex hull of triangulation constraints need to be restored...
    if (!tar_tr.is_edge(vh,tar_tr.infinite_vertex())) {
        // restore the constraints around the inserted vertex
        Face_circulator fc = tar_tr.incident_faces(vh);
        Face_circulator fc_done(fc);
        int vi; // index of the inserted vertex in the tested face
        Vertex_handle evh1, evh2; // handles to the remaining two vertices in the tested face
        int eil, ei2; // indices of the remaining two vertices in the tested face

        do {
            fc->has_vertex(vh,vi);
            eil = tar_tr.cw(vi); // Get the indices of the remaining 2 vertices in the tested face
            ei2 = tar_tr.ccw(vi);
            evh1 = fc->vertex(eil); // Get the remaining 2 vertices in the tested face
            evh2 = fc->vertex(ei2);
            if (fc->is_constrained(eil))
            { // if an incorrect constraint is found in the first tested vertex, then:
                Face_handle nf1 = fc->neighbor(eil);
                nf1->set_constraint (nf1->index(vh),true); // restore the constraint
                fc->set_constraint (eil,false); // remove the incorrect constraint
            }
            if (fc->is_constrained(ei2))
            { // if an incorrect constraint is found in the second tested vertex, then:
                Face_handle nf2 = fc->neighbor(ei2);
                nf2->set_constraint (nf2->index(vh),true); // restore the constraint
                fc->set_constraint (ei2,false); // remove the incorrect constraint
            }
        }
    }
}

```

```

    }
    } while (++fc != fc_done);
}
// if the point outside convex hull of triangulation constraints need to be set
else {
    Face_circulator fc = tar_tr.incident_faces(vh);
    Face_circulator fc_done(fc); // circulator used to end the circulation
    do {
        // if it's not an infinite face, set the constraints if there are any
        if (!tar_tr.is_infinite(fc)) {
            // set the constraint for edge in the new face
            Face_handle nf = fc->neighbor(fc->index(vh));
            if (nf->is_constrained(nf->index(fc))) fc->set_constraint (fc->index(vh),true);
        }
    } while (++fc != fc_done);
}
// return the vertex handle of the inserted vertex
return (vh);
}

// -----
// inserts one vertex in a constrained edge into the triangulation and restores constraints
// -----
Vertex_handle insert_vertex_in_edge(const Point pt, Face_handle f, int i, long object_id)
{
    VEdge c_edge; // the constrained edge that will be split

    // remove the constraint from the constrained edge and save the edge
    f->set_constraint (i,false); // remove the constraint in the cw face
    Face_handle nf = f->neighbor(i);
    int nb = nf->index (f);
    nf->set_constraint (nb,false); // remove the constraint in the ccw face
    c_edge.va = f->vertex (tar_tr.cw (i));
    c_edge.vb = f->vertex (tar_tr.ccw (i));

    // inset the point into the target triangulation
    long intersected_id = f->vertex(tar_tr.cw(i))->id();
    Vertex_handle vh = tar_tr.insert_in_edge (pt,f,i);
    vh->set_id(intersected_id);
    vh->set_second_id(object_id);

    // restore the constrained edge
    Edge_circulator ec = tar_tr.incident_edges(vh);
    Edge_circulator ec_done(ec);
    do {
        // if the edge is constrained move the constraint to the correct edge in the face
        if ( (*ec).first->is_constrained ((*ec).second) ) {
            (*ec).first->set_constraint ((*ec).second,false);
            Face_handle opposite_face = (*ec).first->neighbor ((*ec).second);
            opposite_face->set_constraint (opposite_face->index (vh),true);
        }
        // else check if edge is constrained in the adjacent face and move constraint if it is
        else{
            Face_handle opposite_face = (*ec).first->neighbor ((*ec).second);
            int opposite_index = (*ec).first->mirror_index ((*ec).second);
            if ( opposite_face->is_constrained (opposite_index) ) {
                opposite_face->set_constraint (opposite_index,false);
                (*ec).first->set_constraint ((*ec).first->index (vh),true);
            }
        }
        // if the edge is the first part of the original constrained edge
        if ( (*ec).first->vertex (tar_tr.ccw ((*ec).second)) == c_edge.va ) {
            (*ec).first->set_constraint ((*ec).second,true);
            Face_handle opposite_face = ((*ec).first->neighbor ((*ec).second));
            int opposite_index = (*ec).first->mirror_index ((*ec).second);
            opposite_face->set_constraint (opposite_index,true);
        }
        // if the edge is the second part of the original constrained edge
        else if ( (*ec).first->vertex (tar_tr.ccw ((*ec).second)) == c_edge.vb ) {
            (*ec).first->set_constraint ((*ec).second,true);
            Face_handle opposite_face = ((*ec).first->neighbor ((*ec).second));
            int opposite_index = (*ec).first->mirror_index ((*ec).second);
            opposite_face->set_constraint (opposite_index,true);
        }
    } while (++ec != ec_done);
}

```

```

    // return the vertex handle of the inserted vertex
    return (vh);
}

// -----
// flips edge flip_edge and restores the constraints in adjacent faces
// -----
void c_flip (Edge flip_edge)
{
    // get the faces sharing the edge
    Face_handle facel = (flip_edge).first;
    Face_handle face2 = (flip_edge).first->neighbor ((flip_edge).second);

    // save the constraints
    int amount_constr = 0; // number of edges that are constianed in face 1 and 2
    VEdge edges_constr[4]; // the constrained edges defined by their end vertices
    Edge c_edge;          // constrained edge that needs to be restored

    for (int i=0; i<3; i++) {
        if (facel->is_constrained (i)) {
            ++amount_constr;
            edges_constr[amount_constr].va = facel->vertex (tar_tr.cw (i));
            edges_constr[amount_constr].vb = facel->vertex (tar_tr.ccw (i));
        }
        if (face2->is_constrained (i)) {
            ++amount_constr;
            edges_constr[amount_constr].va = face2->vertex (tar_tr.cw (i));
            edges_constr[amount_constr].vb = face2->vertex (tar_tr.ccw (i));
        }
    }

    // remove the constraints
    facel->set_constraints (false,false,false);
    face2->set_constraints (false,false,false);

    // perform the flip
    tar_tr.flip ((flip_edge).first, (flip_edge).second);

    // restore the constraints
    for (int i=1; i<=amount_constr; i++) {
        tar_tr.is_edge (edges_constr[i].va, edges_constr[i].vb, (c_edge).first,
(c_edge).second);
        (c_edge).first->set_constraint ((c_edge).second,true);
    }
}

// -----
// determines the first face on the path from vstart to vend
// -----
Face_handle get_start_face (const Vertex_handle vstart, Vertex_handle vend)
{
    bool found = false; // indicates if the starting face has been found
    Face_circulator fc = tar_tr.incident_faces(vstart); // Set-up a circulator over the faces
    int v_index; // index of the inserted vertex in the tested face
    int ei1, ei2; // indices of the remaining two vertices in the tested face
    Vertex_handle evh1, evh2; // handles to the remaining two vertices in the tested face
    Face_handle fstart; // handle to the first face on the path from vstart to vend

    // Circulate over faces around the starting point of the edge that needs to be inserted
    do {
        v_index = fc->index(vstart); // Get the index of vstart in the tested face
        ei1 = tar_tr.cw(v_index);
        ei2 = tar_tr.ccw(v_index);
        evh1 = fc->vertex(ei1); // Get the remaining 2 vertices in the tested face
        evh2 = fc->vertex(ei2);

        // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        // case of collinearity not handled yet
        // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        CGAL::Orientation start_orient = orientation (evh1->point(),evh2->point(),vstart->point());
        CGAL::Orientation end_orient = orientation (evh1->point(),evh2->point(),vend->point());

        // if vhl and vh2 are on different sides of the line through evh1 and evh2
        if (start_orient != end_orient) {

```

```

        found = true;
        fstart = fc;
    }
    ++fc;
} while (found == false);
return (fstart);
}

// -----
// inserts the first section of the edge through start_vertex and end_vertex after
// start_vertex to tar_tr
// -----
Vertex_handle insert_section (const Vertex_handle start_vertex, Vertex_handle end_vertex,
long object_id)
{
    Face_handle    start_face;           // first face between start_vertex and end_vertex
    Face_handle    current_face;
    int            current_index;        // index in current_face of the next edge/face
    Segment        s(start_vertex->point(),end_vertex->point());
    vector<VEdge>  edge_vector;
    VEdge          vertex_edge;
    Edge           face_edge;           // temporary container for flipping edges
    bool           constraint_found = false; // loop exit condition for locate constraint loop
    Vertex_handle  found_end_vertex;
    CGAL::Object   result;
    Point          intersection_point;
    Segment        intersection_segment;
    Face_handle    opposite_face;
    int            opposite_index;      // index of the section edge in opposite_face

    // if the start and end vertex are connected directly
    if (tar_tr.is_edge (start_vertex,end_vertex,current_face,current_index)) {
        std::cout << "- section already exists in the triangulation." << std::endl;
        // if the face is infinite get the opposite face
        if (tar_tr.is_infinite (current_face)) {
            opposite_face = current_face->neighbor (current_index);
            opposite_index = current_face->mirror_index (current_index);
            current_face = opposite_face;
            current_index = opposite_index;
        }
        // set the constraint
        current_face->set_constraint (current_index, true);
        found_end_vertex = end_vertex;
    }
    //if the start and end vertex aren't connected directly
    else {
        std::cout << "- locating start face..." << std::endl;
        // locate the first face between start_vertex and end_vertex
        start_face = get_start_face (start_vertex, end_vertex);
        std::cout << " ...face located." << std::endl;

        // set-up the line-face circulator for this section starting at the start_face
        Line_face_circulator lfc = tar_tr.line_walk (start_vertex->point(), end_vertex->point(),
start_face);
        if (lfc== (CGAL_NULL_TYPE) NULL) std::cout << "Something went really wrong: no line face
circulator available" << std::endl;

        // locate the next constraint (end_vertex or constrained edge)
        // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        // not tested for collinear edge and vertex
        // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        do {
            // if lfc (current face) isn't the last face between start_vertex and end_vertex
            std::cout << "- determining end of edge..." << std::endl;
            if (!lfc->has_vertex(end_vertex)) {

                std::cout << " ...edge determined." << std::endl;
                current_face = lfc++;
                current_index = current_face->index(lfc);
                // if a constrained edge has been found, insert the intersection
                if (current_face->is_constrained (current_index)) {
                    result = intersection (tar_tr.segment (current_face,current_index), s);

                    // if the result is a single point insert the intersection into the triangulation
                    if (assign (intersection_point,result))

```

```

        found_end_vertex = insert_vertex_in_edge
(intersection_point,current_face,current_index,object_id);
        // if the result is a segment give message that it hasn't been implemented yet
        else if (assign (intersection_segment,result))
            std::cout << " - segment intersection is not implemented yet." << std::endl;
        // if there was no intersection... panic ;)
        else
            std::cout << " - something went really really wrong!!!, there was no
intersection found" << std::endl;

        constraint_found = true;
    }
    // if a non-constrained edge has been found add the edge to edge_vector
    else {
        vertex_edge.va = current_face->vertex(tar_tr.cw (current_index));
        vertex_edge.vb = current_face->vertex(tar_tr.ccw (current_index));
        edge_vector.push_back(vertex_edge);
    }
}
// if lfc (current face) is the last face between start_vertex and end_vertex
else {
    std::cout << " ...edge determined and last face has been found." << std::endl;
    found_end_vertex = end_vertex;
    constraint_found = true;
}
} while (!constraint_found);

// if the edge_vector is not empty, flip the edges until edge_vector is empty
if (!edge_vector.empty()) {
    // while there are edges that need to be flipped
    std::cout << " flipping edges..." << std::endl;
    while (edge_vector.size() > 0) {
        vertex_edge = edge_vector.back(); // get the last unconstrained edge
        tar_tr.is_edge (vertex_edge.va, vertex_edge.vb, (face_edge).first,
(face_edge).second); // transform it to (face,index) notation
        // only flip an edge if the hull of the adjacent faces combined is convex
        if ( orientation((face_edge).first->vertex((face_edge).second)->point(),
            (face_edge).first->vertex(tar_tr.cw((face_edge).second))->point(),
            (face_edge).first->mirror_vertex((face_edge).second)->point()) ==
CGAL::RIGHTTURN &&
            orientation((face_edge).first->vertex((face_edge).second)->point(),
            (face_edge).first->vertex(tar_tr.ccw((face_edge).second))->point(),
            (face_edge).first->mirror_vertex((face_edge).second)->point()) ==
CGAL::LEFTTURN) {
            std::cout << "Flipping edge for real ;)" << std::endl;
            c_flip (face_edge); // flip the unconstrained edge
        }
        // if hull of adjacent faces combined isn't convex put edge back onto list at top
        else {
            edge_vector.insert(edge_vector.begin(),vertex_edge);
        }
        edge_vector.pop_back(); // remove the processed edge from the vector
    }
    std::cout << " edges flipped." << std::endl;
}

// set the constraint for the section
Edge constrained_edge;
// check if the edge exists in the target triangulation
if (tar_tr.is_edge
(start_vertex,found_end_vertex,(constrained_edge).first,(constrained_edge).second)) {
    std::cout << "- creating constraints..." << std::endl;
    (constrained_edge).first->set_constraint ((constrained_edge).second,true);
    opposite_face = (constrained_edge).first->neighbor ((constrained_edge).second);
    opposite_index = (constrained_edge).first->mirror_index ((constrained_edge).second);
    opposite_face->set_constraint (opposite_index,true);
    std::cout << " constraints created." << std::endl;
}
// if not, non-constrained edges may not have flipped out of way so reinsert section
else {
    std::cout << "A non-constrained edge didn't flip out of the way" << std::endl;
    insert_section (start_vertex,end_vertex,object_id);
}
}

// return the found constraint at the end of this section
return (found_end_vertex);

```

```

}

// -----
// inserts one edge into the target triangulation and updates the target triangulation
// -----
Vertex_handle insert_edge(const Vertex_handle vt1, Point pt2, long object_id)
{
    // insert the start and end point of the edge into the target triangulation
    Vertex_handle vh1 = vt1;
    Vertex_handle vh2 = insert_vertex (pt2,object_id);

    // insert all sections of the edge into the target triangulation
    do {
        std::cout << "Inserting section..." << std::endl;
        vh1 = insert_section (vh1, vh2, object_id);
        std::cout << "section inserted." << std::endl;
    } while (vh1 != vh2);

    // return handle of the last inserted vertex
    return (vh2);
}

// -----
// inserts one edge into the target triangulation and updates the target triangulation
// -----
Vertex_handle insert_edge(const Vertex_handle vt1, Vertex_handle vt2, long object_id)
{
    // insert the start and end point of the edge into the target triangulation
    Vertex_handle vh1 = vt1;
    Vertex_handle vh2 = vt2;

    // insert all sections of the edge into the target triangulation
    do {
        vh1 = insert_section (vh1, vh2, object_id);
    } while (vh1 != vh2);

    // return handle of the last inserted vertex
    return (vh2);
}

// -----
// reconstructs object topology by giving each face that belongs to an object its object ID
// -----
void locate_neighbors (Face_handle current_face, long object_id)
{
    // set the id for the face
    if (current_face->id() == 0) {
        current_face->set_id(object_id);
    }
    else {
        current_face->set_second_id(object_id);
    }
    // check all three neighbors of this face if they belong to the object as well
    for (int i=0; i<3; i++) {
        // if the neighboring face doesn't have the object_id yet continue
        if ( !current_face->neighbor(i)->has_id(object_id) ) {
            // if the edge is not constrained continue
            if ( !(current_face->vertex(tar_tr.cw(i))->has_id(object_id) &&
                current_face->vertex(tar_tr.ccw(i))->has_id(object_id) &&
                current_face->is_constrained(i)) ) {
                // the neighbor also belongs to the object so process it
                locate_neighbors (current_face->neighbor(i),object_id);
            }
        }
    }
}

// -----
// reconstructs object topology by giving each face that belongs to an object its object ID
// -----
void reconstruct_objects()
{
    Object_locator_type current_object;

```

```

Face_handle first_face;

// reconstruct all objects that have an entry on the object vector
while (object_vector.size() > 0) {
    // locate the first face that is part of the object
    current_object = object_vector.back();
    first_face = tar_tr.locate (current_object.locate_point,current_object.locate_face);
    // start the oil-spill algorithm that will reconstruct the object
    locate_neighbors (first_face,current_object.object_id);
    // remove the processed object from the object_vector
    object_vector.pop_back();
}
}

// -----
// redraws the objects from the file
// -----
void redraw_win()
{
    // draw the faces of the objects
    Finite_faces_iterator ff = tar_tr.finite_faces_begin();
    *widget << FillColor(CGAL::YELLOW);
    widget->setLineWidth(0);
    for(int j = 0; ff != tar_tr.finite_faces_end(); ++ff, ++j) {
        if (ff->id() > 0) {
            if (ff->sid() > 0) {
                *widget << FillColor(CGAL::WHITE) << tar_tr.triangle(ff);
                *widget << FillColor(CGAL::YELLOW);
            }
            else *widget << tar_tr.triangle(ff);
        }
    }
    widget->setLineWidth(2);

    Edge fac1_edge; // the edge defined in the face as given by the iterator
    Edge face2_edge;
    Finite_edges_iterator fe = tar_tr.finite_edges_begin();

    for(int i = 0; fe != tar_tr.finite_edges_end(); ++fe, ++i) {

        // determine the first face containing the edge
        fac1_edge = *fe;

        // if the edge is constrained in face 1 then draw it in red
        if ((fac1_edge).first->is_constrained((fac1_edge).second)) {
            widget->setLineWidth(2);
            *widget << CGAL::RED << tar_tr.segment(fac1_edge);
        }
        // else determine second face containing edge and draw it in red if it is constrained
        else {
            // determine the second face containing the edge
            (face2_edge).first = (fac1_edge).first->neighbor ((fac1_edge).second);
            (face2_edge).second = (fac1_edge).first->mirror_index ((fac1_edge).second);
            // if it's constrained draw it in red
            if ((face2_edge).first->is_constrained((face2_edge).second)) {
                widget->setLineWidth(2);
                *widget << CGAL::RED << tar_tr.segment(face2_edge);
            }
            // if the edge is not constrained draw it in blue
            else {
                widget->setLineWidth(1);
                *widget << CGAL::BLUE << tar_tr.segment(fac1_edge);
            }
        }
    }
}

private:        //members
CGAL::Qt_widget *widget;
CGAL::Qt_widget_standard_toolbar *std_toolbar;

double window_leftx;
double window_rightx;
double window_lowery;
double window_uppery;

```

```
    long loaded_object_id; // the object id of the object that is loaded from file
    vector<Object_locator_type> object_vector; // points that lay inside the objects (used for
object reconstruction)

}; //End of My_window

// moc_source_file: test.C
#include "standard_toolbar.moc"

int main( int argc, char **argv)
{
    //Set-up the window in which will be drawn
    QApplication app( argc, argv );
    My_window W(1024,768);
    app.setMainWidget ( &W );
    W.show();
    W.setCaption("Testing Widget: object insertion");
    return app.exec();
}
```