Optimization of routine to find the correct zone for GPS coordinates

Simon Ternsjö
Abstract

**Optimization of routine to find the correct zone for GPS coordinates**

*Simon Ternsjö*

In this thesis a logistics program for keeping track of delivery cars was developed. The program was developed for Jetpak, a logistics company and customer of the IT consulting company Softronic where this thesis work was preformed.

Jetpak has divided Sweden into areas called "trafikledningszoner" (TLZs) which are used to direct the most suitable car for a new delivery. The aim was to improve Jetpaks current system for keeping track of their cars by increasing the speed at which the cars' TLZs are tracked as well as by visualising the TLZs on a map and make them editable.

Two programs were developed to perform this task. The look up program uses the crossing method to calculate in what TLZ each car is in using its GPS coordinates. The editing program displays and enables editing of TLZs on a Google map.

This work has resulted in the possibility of a greater understanding of the exact layout of the TLZs as well as a possibility to edit them to better suit Jetpak's needs. This is an improvement over the current system because more understanding and more control is given to Jetpak. However, these programs can still be improved to ease the editing process by adding more usable functionality.
Populärvetenskaplig sammanfattning

Jetpak är ett logistikföretag som använder sig av ett antal budbilar för att transportera varor. För att kunna välja vilken bil som är bäst lämpad för att hämta upp en vara använder Jetpak trafikledningszoner, vilket är områden liknande postnummerzoner.

Detta examensarbete har utförts på Softronic, ett IT-konsultföretag som har Jetpak som kund. Arbete har syftat till att utveckla det system som Jetpak använder för att ta reda på i vilka trafikledningszoner deras bilar befinner sig. Det primära målet var att öka hastigheten med vilken Jetpak får reda på i vilka trafikledningszoner deras bilar befinner sig. Det sekundära målet var att visualisera trafikledningszonerna på en karta samt möjliggöra redigering av dessa.

"A map does not just chart, it unlocks and formulates meaning; it forms bridges between here and there, between disparate ideas that we did not know were previously connected."

- Reif Larsen, The Selected Works of T.S. Spivet
Acknowledgements

I would like to express my very great appreciation to Andreas Johansson for the patient help when introducing me to the routines at Softronic as well as for good discussions and ideas regarding the thesis work.

I would also like to offer my special thanks to Mattias Lindström for believing in the project and for being a good manager.

My special thanks are extended to the staff of the Jetpak team at Softronic for always providing help and guidance when needed.

My special thanks goes to Lars Oestreicher for taking on this thesis work.

I am particularly grateful for the support and good times given by my friends, for always energizing me on the dance floor!

To my family, for showing mild interest and always good support, I am particularly grateful.

Advice given by Anna Heaggblom Bjellå has been a great help in increasing the professionalism of this report.

To my beloved Dr Lisa Christiansson, for all the love, support and help in all the ways you can imagine, my warmest thank you and a hug.
Contents

Abstract i

Populärvetenskaplig sammanfattning ii

Acknowledgements iv

List of Figures vii

Abbreviations viii

1 Introduction 1
   1.1 A Brief History ........................................... 2
   1.2 The Project ............................................... 2
   1.3 Research question ....................................... 3
   1.4 Delimitation .............................................. 3
   1.5 Outline ................................................... 4

2 Background Theory 5
   2.1 Definitions ............................................... 5
   2.2 Methods of finding if a coordinate is in a polygon .......... 7
       2.2.1 The Crossing method .................................. 7
       2.2.2 The MacMartin method ................................ 7
   2.3 Comparison of the methods ................................ 8

3 Development Method 9

4 Explanation of the program 11
   4.1 The editor ................................................ 11
       4.1.1 The map .............................................. 12
       4.1.2 The info bar ......................................... 13
       4.1.3 The buttons ......................................... 14
       4.1.4 How the edits work ................................. 15
       4.1.5 How the edits can fail ................................ 16
       4.1.5.1 Missed 0 point .................................... 17
       4.1.5.2 Missing polygon .................................... 17
   4.2 The main search program .................................. 18
List of Figures

2.1 The Winding Number .............................................. 6
2.2 The Jordan Curve Theorem ........................................ 6
2.3 Two different pentagrams ......................................... 6

4.1 The editor .............................................................. 12
4.2 How to edit ............................................................ 13
4.3 Unallowed edit ....................................................... 13
4.4 Missing the 0 point .................................................. 17
4.5 Overlapping TLZs ................................................... 17
Abbreviations

API  Application Programming Interface
GIS  Geographic Information System
GPS  Global Positioning System
IT   Information Technology
TLZ  Trafikledningszon or traffic routing zone
Chapter 1

Introduction

Jetpak is one of the largest courier service in the Nordic countries with operations in Sweden, Denmark, Norway, Finland and Estonia, [1]. There is a total of 1500 people involved in Jetpak’s organisation doing everything from driving to organizing deliveries.

Jetpak is divided into smaller parts, each taking main care of a local area. For instance the Stockholm part of Jetpak mainly focuses on in-town deliveries in the Stockholm area.

To administer all the cars, flights and deliveries, Jetpak has 140 people responsible for overseeing the logistics. This involves, for instance, giving delivery orders to the cars and rerouting packages.

Softronic is a consulting company, specialized in IT management. The company has offices located in some of the larger cities in the Nordic country’s, for instance Stockholm, Gothenburg and Copenhagen and has around 500 employees [2]. One of their customers is Jetpak, and in Stockholm, a group of around a dozen people is managing the Jetpak’s IT-solutions. This involves making sure the cars have working Global Positioning System (GPS)-receivers and can communicate with their headquarter, as well as to make sure that the Internet booking systems and customer interface works.

To allow visualization of pickup and delivery places, as well as the cars positions, Jetpak is using Kartenas web based map services.

For easier management of the cars, Jetpak has divided Sweden into smaller areas called ”trafikledningszoner” (TLZ’s) or traffic routing zones. Each TLZ is made up of a number of zip-codes. For instance, the combined area of the zip-codes 696 76, 696 81 and 696 96 is the TLZ ”ZIN” which is an area called Zinkgruvan in Örebro.
1.1 A Brief History

In order to be a quick courier service, with short delivery-times, the headquarters need to know in which TLZ’s each car is located, and thereby be able to give new deliveries to the most suitable car. In the past, before the age of GPS-receivers, this was done manually by the drivers; they reported in to their headquarter in what TLZ they where, and the lists managing the cars where updated manually.

In 2006, Softronic introduced GPS-receivers in the cars which enabled an automatic update of which TLZ each cars was in. This reduced the workload of the drivers and enabled a more precise positioning of the cars.

The TLZ’s are made up of a number of zip-codes, which are managed and updated by the state. The state don’t release the information of how the zip-codes look like for free. Because of this, the exact area that each zip-code has, and thereby also each TLZ has, is not known. Hence, to determine the TLZ of a car from its GPS-coordinates, a look-up table is used. First, the GPS coordinates of the car are sent to Softronic, and then to Kartenas map service from where the address of that coordinate, or the closest address, is returned to Softronic. From this, the zip-code is extracted and then the TLZ that contains that zip-code is looked up from the database. This information will then be sent to Jetpak, and the TLZ in which the care is located is displayed at the headquarter.

This is a 5 step procedure and there are two downsides to this method. First, nobody knows exactly how the TLZ’s look like. Since the zip-codes are managed and updated by the state, this also means that it is impossible to manage the TLZ’s to better suite Jetpak’s needs. Second, Kartenas servers are not capable of handling that many look-up requests, roughly 1400 requests, per minute. Since Jetpak has 700 cars, the TLZ for each car get updated at an average only once every 35 second. Hence, a switch to a faster and more robust map service to manage the locations of Jetpaks cars is warranted.

1.2 The Project

The primary objective of this thesis work was to increase the frequency of the TLZ-update for each car by completely redesign the way to determine in which TLZ a car is located from its GPS-coordinates. This was done by removing the look-up request from Kartenas, and instead check which TLZ a cars coordinates is in from a database of stored TLZ coordinates.

This required a good algorithm for testing if a coordinate is inside a TLZ. It would hopefully lead to increased performance in finding the TLZ of a car. Subsequently, this
can increase the frequency of the TLZ-checks and thereby improving the accuracy when choosing which car will do what delivery. This is also in line with the plan of completely remove Kartena from Jetpacks administration.

The secondary objective is to create an interface consisting of a map in which the TLZ’s will be shown. In this interface the TLZ’s should be able to be edited. This visualisation of the TLZ’s will increase the knowledge and understanding of the areas of the different TLZ’s, which should further help the coordination of the cars when distributing deliveries.

Being able to edit the TLZ’s will make it possible to optimize the TLZs for Jetpaks needs instead of the Swedish mail-services needs.

1.3 Research question

This thesis focuses on the localization of delivery vehicles within specified zones of irregular shapes on a map. With this perspective, the central research question will be as follows:

- How do you check if a coordinate is inside a specific polygon in a sufficiently fast way?

The requirement ”sufficiently fast” in this question will be developed in collaboration with the company. The speed with which the coordinates are checked is however a critical part of the project.

1.4 Delimitation

Because of money and time constraints the focus of the thesis will be delimited as follows.

Testing of the programs developed will be limited to data from a sample of zip-code areas freely available. The rest of the zip-code areas need to be bought from the Swedish state and may be done at later state if Jetpak decides to buy this new system from Softronic. This gives a constraint that it is not possible to compare the speed of the look up requests to Kartena and the Main Search program.

Jetpaks area of interest involve most of the Nordic countries. This project will mainly focus at the TLZs in Sweden, and the Swedish zip-codes. This is because the free zip-code area sample is only available for Swedish zip-codes. Even though there is no
technical limitation of extending the TLZs to other countries, these areas will not be covered in this thesis.

This thesis will not contain usability tests or interviews regarding the editor. This is because the new system developed in this thesis will be administered by the Jetpak employees. Since Jetpak has not bought the service yet, the employees have not had a chance to test the program nor to be interviewed about it.

1.5 Outline

This thesis will start by introducing definitions and theories involved regarding polygons. It will also discuss different methods for testing if points are inside polygons or not.

Within software development there are two basic different methods for development. These, as well as the method used for this thesis, are introduced and explained further in chapter 3.

In this thesis two programs, the Main Search program and the editor, have been developed. These programs are explained in detail in chapter 4. First, the editor is explained, with a focus on how the different parts of the program function and how to use them. Then, the underlying programming logic and functions are explained. This part is important to grasp to better understand how the editor can produce wrong results. Failed edits and wrong results are the last things explained before the Main Search program is elaborated upon.

The second section of chapter 4 describes functions of the Main Search program. The different classes of the program are first illustrated together with how they interact. The last part of this section explains how the program behaves when operated and how the program has been tested.

Chapter 5 dissect the programs and discuss how they can be improved in terms of existing functionality and new functionality. Moreover, chapter 5 elaborates upon different ways of solving the original problem, with corresponding strengths and weaknesses. The last section discusses the personal difficulties with this thesis work.

In the last chapter the new developed system and the importance for Jetpak is explained, and the findings of this thesis are elaborated upon.
Chapter 2

Background Theory

The primary goal of this thesis can be boiled down to the sentence "how do you check if a coordinate is inside a specific polygon in a sufficiently fast way?". In order to answer this question, one must first define polygons and what it means by being inside or outside a polygon.

2.1 Definitions

A polygon could be defined as a plane shape with straight sides, and it’s interior [3]. Mathematically, a polygon can be formulated by having an ordered set of coordinates that are combined with straight lines. A line must also go from the last coordinate to the first coordinate in the set, such that the polygon is closed.

Points can be defined as being inside a polygon if the polygon wraps itself around the point, at least once. For example, a square wraps itself around a point exactly once if the point is inside the square and zero times if it is outside. This is called Winding Number [3] which is illustrated in Figure 2.1.

The points inside a polygon can also be defined as the points from which there exists a ray, casted in any direction, that intersects the edge of the polygon an odd number of times. For example, all points inside a square has a ray cast upwards (or any other direction) that will intersect only one edge of the polygon. This is called the Jordan Curve Theorem [3], and is illustrated in Figure 2.2.

These two different definitions create different results with certain polygons. For example, a pentagram will be a solid star with the winding number method, but it will be a star with a pentagon-hole in the middle with the Jordan Curve Theorem, as is shown in Figure 2.3.
Figure 2.1: Winding Number illustrated

Figure 2.2: Jordan Curve Theorem illustrated

Figure 2.3: The Jordan Curve Theorem gives a pentagram a hole, while the Winding Number Method gives it a solid body
However, when dealing with "nice polygons", both these two definitions give the same result. In this thesis, "nice polygons" is defined as polygons without holes, that does not intersect itself and has only one coherent area. However, a "nice polygon" does not mean that the polygon has to be convex, it can still be concave. Throughout this thesis there will only be "nice polygons" which makes the difference in the definitions insignificant. The Jordan Curve Theorem will be the definition used since it most resembles the way the coordinates will be determined if there inside a TLZ or not.

2.2 Methods of finding if a coordinate is in a polygon

One way of finding what polygon a coordinate is in is to loop through all the different polygons stopping if the coordinate is inside the polygon. In order to do that, a method to determine if a coordinate is inside a polygon must be at hand.

Several methods, for instance MacMartin, Crossing, Triangle Fan, Barymcentric and Angle Summation, can determine if a coordinate is inside a polygon. Out of these, the MacMartin is the fastest for polygons with more than 10 edges, moreover it does not require any additional memory or preprocessing [3].

2.2.1 The Crossing method

The Crossing method is based on the Jordan Curve Theorem in that it uses a ray from the test point and counts how many times the ray intersects the edge of the polygon. To make calculations easy, a ray in only one dimension, usually in the positive X-direction, is used.

In order to count how many times the ray intersects the polygon, all the edges of the polygon are looped through and ray intersections of the edge-element are checked. There will most often be a lot of edge-elements that will not be intersected by the ray, however they will still be calculated upon and hence take unnecessary computer power. Because of this, MacMartin developed an improvement to this method [4].

2.2.2 The MacMartin method

In order to speed up the Crossing method, MacMartin suggested a way that would reduce the number of times that the intersection needed to be calculated. He suggested that only the Y-components of the edge-element should be checked against the test point. If
the edge-elements coordinates lie on either side of test point’s Y-component, then the X-components of the edge-element is retrieved and computations are made to check if the edge-element intersects the ray. These computations first checks if both X-components of the line segments edges are greater than the test points X-components. If they are, the line segment and the ray intersect. If both X-components are smaller than the test points X-components, the line segment and the ray do not intersect. If none of the first two outcomes are true, then and only then the intersection needs to be calculated and compared to the test point.

2.3 Comparison of the methods

Booth the Crossing and the MacMartin methods are based on the same theory, that a ray shot from a point will intersect the edges of a polygon an even number of times if the point is inside, and an odd number of times if the point is outside the polygon. The Crossing method is the easiest to implement because the only thing it does is to check the ray against every edge of the polygon by calculating the intersection point. Thus it has to calculate the intersection point quite a few times. The MacMartin method on the other hand is more complicated to implement. This is because it tries to reduce the number of times it needs to calculate the intersection of the edge-element and the ray. It checks every edge of the polygon against the ray, but uses different if-statements to try to determine whether or not the ray intersects the edge without calculating the intersection, with the goal to be faster than the Crossing method.

The MacMartin method is 1.8 times faster than the Crossing method for polygons with more than 100 edges [3]. However, in this project, the Crossing method will be used, with the option to improve it to the MacMartin method if the need arises.
Chapter 3

Development Method

There are a variety of methods for developing software programs, two usual categories are water fall methods and agile software development methods. These two can be considered as counterparts. In the water fall method, the project goes through a certain number of steps, where each step has to be completed in order for the next step to begin. The steps usually include some sort of requirement phase, a developing phase, a test phase and an installation phase. An advantage of this method is that it is easy to oversee how the budget is kept between each step, and decide whether or not more resources are needed. On the downside there is a great cost for changing something in steps that has already been completed. For instance, it is very expensive to add and change the system requirements once the development phase has begun.

The agile software development methods was created as a reaction to the way that many of the water fall methods limited the programmers [5]. Agile software development requires a close cooperation between the programmers and the client, with regular meetings. It is thought that direct communication between people in the project is the key to solve problems instead of writing guidelines and requirement specifications. This will also increase the possibility to change and add new requirements to the software in later stages. The method that has been used to develop the two programs in this thesis is a variant of agile software development.

Small incremental steps has been taken to develop the programs in this thesis. The requirements were discussed with the supervisor in the beginning and after that informal meetings have been taken place on a regular basis. During the course of the project, new ideas have been discussed and new details have been added to the project.
In order to maintain structure and flexibility of the program, the code has been written in small methods that perform a single key task. After these methods have been created they have been tested thoroughly to make sure that they do not contain any errors.

A great deal of inspiration have been sought at various Internet forums, this has generated a lot of new and smart ways of overcoming problems along the way. Another source for inspiration has been other programs from Softronic, from which the source code was available. The structure of how comments and methods were written could be seen in those programs, a style which was copied to the two programs in this thesis to make them follow the company standards.
Chapter 4

Explanation of the program

The problem at hand has resulted in mainly two programs; the main search program that will check in what TLZ each car is in, and the editor that will enable editing of each TLZ. The main search program should be fast and robust since it will be run once for each car every update-cycle, and was therefore written in C++. The editor should on the other hand be web based and more usable and was therefore written in JavaScript as well as HTML.

Both these programs will need to communicate with a database where the TLZ-data will be stored. The main search program will read the TLZs from the database, keep them in its memory and preform calculations on them from there. The editor will read the TLZs form the database and display them on a Google-map. On the map, users will be able to edit the TLZs and then save the new TLZs to the database, thus overwriting the old TLZs.

4.1 The editor

The program used to edit the polygons is simple in structure. This is because the basic editing should be easy and straight-forward. The program layout is made up of sex buttons, an info bar and the map, which takes up most of the space available, see Figure 4.1.

The way to edit the TLZ’s is by drawing a polyline on the map that the TLZ’s will form themselves after. This is explained in more detail in section 4.1.4.
4.1.1 The map

The map is a regular Google map with functionality such as zooming, panning, satellite/map-mode and labels on landmarks. On the map the TLZ’s are also shown as differently colored Google maps polygons. Google maps polygons are one of the features included in the Google Maps API v3 [6]. The polygons are 65% transparent with solid edges so it is easy to see the map underneath the polygons but still easy to see the edges and outlines of the polygons.

A TLZ can be clicked to ”mark” the polygon. A marked TLZ is only 20% transparent and has thicker edges, to show that it is marked. Several TLZ’s can be marked by clicking them, and unmarked either by clicking again or by clicking the ”unmark all” button.

The user edits the TLZs by drawing a polyline on the map. The TLZs will then change to follow the polyline, see Figure 4.2. The polyline has to fulfill some requirements to make editing possible. These requirements are:

- The length have to be greater than 2 nodes
- It can not cross itself
- It has to start in a TLZ
- It has to end in the same TLZ as it starts in
(a) The polyline needs to start and end in the same TLZ, and can cross multiple TLZs

(b) The TLZs will adjust themselves to follow the polyline

\textbf{Figure 4.2:} The user uses a polyline to change the TLZs

\textbf{Figure 4.3:} This edit would split the brown TLZ in two halves and is thus not allowed

- The first two points or the last two points can not be in the same TLZ
- The polyline cannot split a TLZ in two parts, as in Figure 4.3

If the requirements are fulfilled, the program will edit the TLZs and display the result in the info bar.

\subsection*{4.1.2 The info bar}

Above the map there is an info bar where certain information is displayed. The first bit of information is the number of TLZs that is loaded onto the map. If there is 27 TLZs loaded from the database it will say "Total number TLZs: 27", and if any TLZ is deleted, the information will be updated to display the current number of TLZs. The second bit of information is the number of marked TLZs, it starts with "Marked TLZs: 0" and is updated every time a polygon is clicked or the "unmark all" button is clicked. The third and last bit of information is the most dynamic. It has three types of information; messages, warnings and errors, displayed in black, orange and red text respectively. To increase the usability of the program they are all displayed consistently [7].

Messages are information about the the last thing that happened. For instance, when a TLZ is marked, it will read "Marked TLZ XXXX" where XXXX is the number of the TLZ. It will also tell which TLZs that is unmarked, if the program is waiting for database
communication, if the TLZ-import is complete, that TLZs have been deleted, that the last edit has been undone, which TLZs that where changed during an edit, if user enters or exits edit mode and if the TLZ’s where saved to the database in a successful way.

Warnings are displayed when the user has used the program in such a way that the program can’t interpret what the user is trying to do. For instance, if the user tries to edit a polygon so that the polyline crosses itself. Other warnings include if no polyline is found when the user presses the ”Change polygon” button, if the polyline is not long enough, if it does not start and end in the same polygon or if the the polyline cuts a TLZ in two halves, like in Figure 4.3.

Errors are displayed when the program does things it is not supposed to do. For example, the program can check if a TLZ is at a certain coordinate. It can also check what TLZ is located at a certain coordinate, this check require a TLZ to be located at that coordinate. If something has gone wrong, and there is no TLZ at that coordinate, then an error will be displayed. Errors are mainly for debugging purposes and should not happen during regular usage.

4.1.3 The buttons

Above the info bar, at the top of the page, there are six buttons. The buttons are labeled with a name that describes its action and a letter that tells what keyboard shortcut generates the same action.

The different buttons are ”Change TLZ (C)”, ”Toggle editing (E)”, ”Unmark TLZs (U)”, ”Delete marked TLZs (D)”, ”Regret (R)” and ”Save changes (S)”.

Toggle editing, which, part from the button also can be toggled by the E-key, will change the cursor to let the user draw a polyline on the map. This line will be used to change the TLZs accordingly to the polyline, see Figure 4.2a. To end drawing a polyline, the user can either click the Toggle editing button again, double click on the map or click the last coordinate. When the user is no longer inputting a polyline, the existing polyline can still be modified by dragging the coordinates on the line.

The ”Change TLZ” button, the C-key or the Enter-key, will change the TLZs that the polyline intersects to match the polyline, see 4.2. It will first check if the polyline is created in a correct way. If it is not, a warning-message will inform the user of what went wrong.

Regret, R-key or Ctrl+z, will undo the last ”Change TLZ” edit, this is useful if the end result is not as the user expected, either because the user made an error inputting the
polyline, or if the program changes the TLZ’s in a undesirable way, more about that in section 4.1.5.

Deleting marked TLZ’s, either by the button or the D-key, will remove the marked TLZ’s from the map. It will also internally mark those TLZ’s as deleted, so that they can be deleted in the database later.

The unmark TLZ’s button, or the U-key, will simply unmark all TLZs, making the background map easy to see.

Saving changes, or the S-key, write the existing state of the TLZs to the database. In the database a TLZ is saved as a number of rows, one row for each coordinate of the polygon. Along with the longitude and latitude information, the number of the TLZ is also stored as well as in what order in the polygon that coordinate comes. When saving, each edited TLZ-will be saved in order. For each TLZ the existing rows of information about that TLZ is removed from the database. Then the new coordinates are written along with there order-number and the TLZ-number. After all edited TLZs are saved, the deleted TLZs are removed from the database, one at a time.

4.1.4 How the edits work

When the user clicks the ”Change TLZ” button, the program will check if the polyline fulfills the requirements. If the requirements are fulfilled, the program will start to edit the TLZ that the polyline intersects, one at the time. The program finds the TLZs by checking what TLZ is at each corner of the polyline, adding each of these TLZs to a ”to be edited”-list.

The first step is to find the intersections between the polyline and the TLZ that is to be edited; where the polyline enters and exits the TLZ. The interesting information here are the enter and exit coordinates and at what line segment of the TLZ the intersections occur.

The second step is hard to understand without the knowledge about step three and four. It will therefor be explained after step three and four.

The third step is to remove the coordinates in the TLZ that lie between the enter- and exit-coordinates. For instance, if the enter-coordinate is at the fifth line segment, and the exit-coordinate is at the ninth line segment, then the coordinates number 6, 7, 8 and 9 are removed.
The fourth step is to insert the new coordinates in the TLZ from the polyline. This is an easy operation if the order in which to insert the coordinates is known. It is the second step that determine this order.

The hard step is the second step, where the orientation of the polyline is determined. Since a polyline is a ordered set of coordinates, the coordinates can be numbered in a clockwise or anticlockwise manner. This is important because the fourth step needs to know in what order to insert the coordinates from the polyline, if it should be done ”from left to right” or ”right to left”.

The second step also deals with the case of ”going around the 0-point”. For instance, if a TLZ has 50 coordinates, and a polyline exits it at the 48'th line segment and re-enters it at the third line segment, then, the ”other” set of coordinates should probably be removed; the coordinates number 49, 0, 1, 2 and 3, not coordinates 4 to 48. Hence the program needs to know when the coordinates to be removed should be done in reversed order, which means to include the 0-point, or in regular order. The optimal way to check if the polyline includes the 0-point would be to connect the start and end-points of the polyline so that this line do not leave the TLZ. Then check if the 0-point is inside the closed curve.

In practice, the program connects the start and end-points with a straight line, and checks if the 0-point is in that closed region. If the 0-point is inside the closed region, the program will remember this and the ”other” set of coordinates will be removed in the third step.

4.1.5 How the edits can fail

In the editor, it should not be possible to generate incorrect TLZ's. An incorrect TLZ would for instance be one that loops around itself, or has a hole.

These incorrect TLZs are avoided by making sure that the polyline follows the restrictions put on it. However, to make sure that a polygon don't overlap any other polygon, and that there is no gap between the polygons, therein lies the hard challenge. These types of errors can occur in mainly two ways; if the 0 point is missed or if a polygon is missed when editing.

As of now, the program has no way of prevent these errors, or warn the user that either of these errors has or is about to occur.
Explanation of the program

4.1.5.1 Missed 0 point

Because the program checks if the 0-point is included in the way that it does, a edit can be performed where the program thinks that the 0-point is not included, but it actually is, or the other way around. This will cause the freshly edited TLZ to almost be turned inside out, see Figure 4.4, and will become useless. It is then the undo-button is handy.

4.1.5.2 Missing polygon

Every TLZ that is intersected by the polyline should be edited by the program. However, the program only checks the coordinates of the polyline’s corners. Hence, when the polyline intersects a TLZ without having a corner in that TLZ, the TLZ will not be added to the ”to be edited”-list, and it will not change to match the polyline. This will result in the nearby overlapping TLZs, see Figure 4.5. Then the undo-button is again handy.
4.2 The main search program

The main search program is written in C++ and does not have a user interface. Instead, it is supposed to run constantly on the server that tracks Jetpack’s cars. From the program, the lists of cars will be updated at the headquarters.

To handle the TLZ’s, the program has three classes; Coordinate, Polygon and PolygonList.

4.2.1 The Coordinate class

The Coordinate class is the most simple class because it only represents a single coordinate. The coordinate is made up of two doubles, longitude and latitude, and no useful methods. There is one method for printing the coordinates on the screen, which was mainly used for programming and debugging purposes. There are two constructors, one default which sets both coordinate values to -1, and one where the caller chooses the coordinate values.

4.2.2 The Polygon class

In order to represent a TLZ, a polygon class, which store all important information, is used. The TLZ’s coordinates are stored in a std::vector, in the same order as they were received from the database. The id of the TLZ and the number of coordinates it contains are stored as two separate integers. Moreover, information on whether or not the polygon is ”good” or ”bad” is stored. This indicates if the polygon comes from data from the database or if it is a return-polygon from a search where no polygon was found.

This class has more methods than the Coordinate class. Beside methods for programming and debugging, ”print all coordinates” and ”print id”, there are four important methods. The first is a method to add a single coordinate to the list of coordinates, the second is a method to set the id of the TLZ. Both these methods are used when the polygon is filled with coordinates from the database. The third important method checks if the polygon is ”good” or ”bad”.

The fourth method is the most important method of this class. It is the method that checks if a coordinate is inside the polygon or not. This method takes a coordinate and returns a boolean. The return variable starts being false, and will alternate between false and true through the method. The program loops through all the sides of the polygon. For each side of the polygon, the loop calls a private method. This method
checks if the ray in the positive X-direction from the coordinate crosses the current side of the polygon. If the ray crosses the side, the return variable will be inverted, hence, if the return variable is true the return variable will become false, and if it is false it will become true. When the loop has called the method for each side, the return variable will have the correct value and can be returned. This is one implementation of the Crossing method.

There are two constructors in this class; one standard, that constructs an empty polygon, and one with a boolean input, with which it is possible to create a "bad" polygon.

4.2.3 The PolygonList class

The PolygonList class stores all the different TLZs in a std::vector, and keeps track of how many polygons are stored.

This class has the programming and debugging useful methods "print” that prints all polygons and its coordinates, and "print number of polygons” that prints how many polygons that are stored.

The two most important methods are "add polygon”, which adds a complete polygon to the vector, and the method that checks which polygon that contains a certain coordinate. To check this, the method loops through all polygons, calling their method to see if the coordinate is inside that polygon. If it is, a reference to that polygon is returned, otherwise the loop through the polygons is continued. If no polygon is found, the method returns a "bad” polygon, which means that the coordinate was not found in any of the stored polygons.

4.2.4 The flow of the program

Since this program is not yet incorporated with Jetpack’s system, a test program has been created that test the functionality of all classes.

First the program selects all appropriate data from the database. Next, two nestled while-loops inserts all polygons in the PolygonList. The inner loop inserts the coordinate data from each row from the database to a TLZ. This TLZ is then inserted in the PolygonList along with the TLZ’s id-number, and the outer loop continues with the next TLZ.

Then 50 test coordinates are generated inside of a square area where the TLZs from the database are located. This is to increase the probability that most of the coordinates
Explaination of the program

will lie inside one of the TLZs. Then the PolygonList checks which TLZ each coordinate is in. If the coordinate is inside a TLZ, the coordinate is printed along with the id of the TLZ, and if it is outside all polygons the coordinate is printed along with a message that no TLZ was found.
Chapter 5

Discussion

Even if the two programs fulfill the requirements that was originally stated, there are a few improvements that could make the programs even better. The programs can become easier to use. Since the programs only contain basic functionality, new features can be added to facilitate the use of the programs.

5.1 The main search program

The possibility to improve the main search program lie mostly in making it faster, because the accuracy with which it determine what polygon a test point is located in is already very high. To increase the speed of the search one can improve the algorithm that checks if the test point is inside the TLZ and change the order of which TLZ is checked first.

5.1.1 Bounding Box

The algorithm can be improved by adding a ”bounding box-test” before the polygon is tested. The bounding box is the smallest rectangle with vertical and horizontal edges that the polygon will fit in. This bounding box can be stored as four doubles in the Polygon class. The four doubles can be found by finding the max- and min- longitude and latitude of the TLZ-coordinates. The bounding box-test is a simple test to make sure that the test point is inside that box, because if it is not in the bounding box it can not be inside of the TLZ. If the test point is inside the bounding box, then the test point needs to be tested against the TLZ.
The addition of the bounding box will reduce the number of times that the slower algorithm of testing the entire TLZ has to be run. However, the suggested way will increase the amount of memory that is needed by the program by the equivalent of four doubles per TLZ. This is not much compared to the total amount of memory needed for the program. On the other hand, the time needed to load the TLZs from the database will increase, regardless if the bounding box is updated every time a new coordinate is added to the TLZ, or if the bounding box is created after all coordinates are imported.

5.1.2 MacMartin

Another way to improve the speed of checking if a test point is inside the TLZ is to change the Crossing method to the MacMartin method as described in section 2.2.2. This would reduce the amount of calculation performed by the program and also the amount of coordinates X-components that are calculated upon by the processor. This improvement does not exclude using a bounding box as well.

Because only all the Y-components are needed, the storage of the coordinate data may be better off in another way. In current database every Y-component is stored together with its corresponding X-component. Thus, when the processor needs the Y-component, the X-component will follow along. If the coordinate-data were stored in two vectors, one for X-components and one for Y-components, then the data would be stored in a more suitable way for the MacMartin method.

5.1.3 Optimization of the order in which the TLZs are checked

Even if the method to check if a point is inside a TLZ is very fast, it will still take some time if there are a lot of TLZs, especially if the point to test is inside the last TLZ that is checked. In an ideal situation, the first TLZ to be checked would be the one containing the test point. This is obviously impossible to achieve all the time because then one need not to test that point. However, one can arrange the TLZ in such an order that the one with the highest probability to include the test point is checked first.

5.1.3.1 Start with closest

If all TLZs have a center point, then the TLZ with the center point closest to the test point can be tested. If the test point is not in that TLZ, then the TLZ with second closest center point is tested and so on. In most cases this addition would lead to that
the time consuming algorithm, to find if the test point is inside of a TLZ, only would be executed a few times.

To find the preferred order one can simply sort the TLZ-list after the square of the distance from the test point to the TLZ center point and test the closest first. This operation would probably be relatively fast compared to testing all TLZs with the crossing method.

5.1.3.2 Start with last

Since this program is designed to track cars, another way to optimize the order in which the TLZ are checked is to save the TLZ the car was registered in at the previous update. Then start to check that TLZ, and if the car has left the TLZ, the program would check nearby TLZs. This would take into account that the cars can not jump around from any TLZ to another. The TLZs are big enough not to allow a car to drive to a distant TLZ within one update cycle, or at least not very often. If that would happen, the program would obviously check all the other TLZs.

5.1.3.3 Start with most frequent

A third way of solving this problem would be to give the TLZs a counter that would increment every time a car is found in that TLZ. After a while the TLZ in which the cars spend most time in will have the highest counter, the second most visited TLZ will have the second highest counter and so on. Having the TLZs sorted by this counter will often result in fewer TLZs tested.

5.2 The editor

The possibility to improve the editor is different in nature from the main search program. It is possible to improve the speed of certain methods that are similar to those in the editor, however, the biggest advantages would be to make it easier to use and to add functionality.

The current version of the program allow the user to edit existing TLZs, but there might be other things that the user would like to be able to do. Moreover the edits could be more stable and more tolerant.
5.2.1 More edit options

Users can now edit the existing TLZs, and remove TLZs. Other functionality that could be added would be being able to add new TLZs. In order for such a functionality to work, a few problems need to be solved. First, the newly created TLZ cannot overlap any of the existing TLZs. This problem could probably be solved with the same algorithm that makes sure that none of the TLZs overlap when editing a TLZ. The newly created TLZ must also get an id that is permitted in the database.

Another functionality that can be added is the ability to split an existing TLZ in two. This function could be needed if there is much traffic in a large TLZ. Then it would be good to have more detailed information when handing out deliveries. The problems that arise with this functionality is that the user must decide which of the TLZ that keep the old id, and which id the new TLZ should get.

Another feature that might improve the editing would be if the name of existing TLZs could be changed. This would reduce the need for choosing the perfect name from the beginning when splitting or adding TLZs.

5.2.2 More of existing functionality

Some things in the existing version can be improved to increase the usability of the program. For instance, the ability to undo an edit. It would be better, from a user perspective, if more than one edit could be undone. To be able to undo more than one edit a more complicated way of keeping track of the past positions of the polygons need to be developed. This is possible but would also require more memory.

Moreover, adding the possibility to undo other things besides the edits would improve the program. For example, if deleting, renaming, splitting and creating new TLZs could be undone, it would make the program more usable.

To deliver information about what the program is doing, the program uses the info bar. However, as soon new information is displayed, the old information is lost. It would be preferable if one could review all the messages, warnings and errors that the program had displayed.

5.3 Other ways to solve the problem

There is a saying that if one want to improve a program, one should start to change the algorithm. This is because if a better algorithm is found it will probably improve the
speed of the program more than if each part of the current program gets slightly faster by optimization. Thus, this section is about how the original problem can be solved in other ways, than by slightly improving the existing programs.

5.3.1 The storage of data

If the areas were treated not as individual polygons, but as an area where different parts belong to different TLZs, then the entire data can be represented as a grid instead of a bunch of polygons. A grid is made up of two data sets, the first would store all the coordinates in a long list. This would disregard how many TLZ that share a coordinate, every coordinate would be in the list only once. The second data set would store a list of lists with all the TLZs. The coordinates would not be stored, but rather the position of the coordinate in the first data set.

In this way the data is stored more like the reality that the data represents. The TLZs are not individual polygons, but rather areas in a grid. With this way of storing the data, the change of one coordinate will not result in overlap or a gap between two TLZs, because the corner of every TLZ that share that corner will also be moved. Another advantage is that less memory is required to store the data this way, since each shared coordinate only needs to be stored once. However, the extra list with the number of coordinates that each TLZ is made up of will take some extra memory. This added memory will not be significant in comparison to the coordinates stored because the lists will only be integers, whereas the coordinates are doubles.

5.3.2 Finding where the test point is

The algorithm for finding in which TLZ a test point is located is based on the question "Is the test point inside this TLZ?", and this question is asked to every TLZ. If the system is seen as a grid instead, it should be possible to ask the question "In what TLZ is this test point?". One should be able to use the fact that there is no gap between the TLZs to make an algorithm optimized to find where a test point is. Searching published material, no such algorithm has been found. The algorithm at hand was deemed good enough given the time to develop the program and the demands on the system.

5.3.3 Other ways to edit

The current program uses a polyline to make edits possible, a method inspired by the system used by the open-source program "Quantum GIS". This is not the only possible
way. The first idea was to make all the corners of the TLZs movable. This would enable edits by dragging each individual corner of the TLZ. This idea is good in the sense that the users get a lot of control of where the corners go, and can adjust the corners back and forth to their liking.

This way of editing was quickly abandoned when considering the amount of corners that needed to be edited to move a side of a TLZ to another part of a city. The TLZs have a lot of corners with small edges between them. This way to edit would give the user more control but it would also make the edits much more tiresome. However, by choosing a polyline to edit the TLZs, it is not exclude to give the option of editing individual corners this way, it has just not been done.

5.4 Challenges

The challenges with this thesis work have been quite few and I have always had my supervisor at hand who has been able to help me with most difficulties. However, there has been challenges with doing things in different ways than what I am used to. This is mainly because I have followed the standard way of doing things at Softronic, which has included things like new programming languages, new Application Programming Interfaces (API), and database programming.

5.4.1 New programming languages

One challenge have been to convert my programming knowledge to the task at hand. Before I started I knew how to program C and Java, and to some extent HTML. One of the programs that I would create should be written in C++, which was a decision from my supervisor. I thought that it would be very much like C but found that it was enough different to give me some hard time. Nothing that I did not solve with some persistent work, and advise from my supervisor. The other program should be written in JavaScript, also a decision from my supervisor. Being fooled by the name and thinking it would be something like Java, I spent a good time on tutorials finding out that the name JavaScript was chosen as a marketing ploy to gain some of the popularity of Java [8]. After some good advise from my supervisor and a few different tutorials I could enough JavaScript to, together with my previous programming knowledge, get started.
5.4.2 APIs

Another thing that I had a hard time with at the beginning was Googles Maps API. When I learned to program we mostly created our own methods and classes from scratch, therefore using another API, and especially such a complicated API as Googles, was at the beginning challenging. However, when the right API version was loaded and I had gotten to work with it a bit, it became very intuitive.

5.4.3 Database

Before I started my thesis work I had no experience with databases. Softronic is a company quite fond of databases and therefore my program should be integrated with data from the database. This was at first the thing I had most questions about, because I had no idea how it would be done. It was quite hard to start working on the programs without knowing how it would work with the other systems. I just had to trust that my supervisor would help me with it when that day came.

It turned out to be easy enough. My supervisor gave me two different skeletons that made both the main search program and the editor communicate with the database. After studying them for a while I could tweak them to be optimized for the two programs and the communication worked without a hitch.
Chapter 6

Conclusion

In this thesis, two computer programs, a Main Search program and an editor, were developed. These programs were developed to ease the work for the operators at Jetpak. The program have not yet been tested at Jetpak but they will provide a different way to preform the task of giving deliveries to cars.

One difference is the editor that allows the operators to view and edit the TLZs. This is important because it gives the operator greater understanding of their area and thus giving them more information to base the decision of what car will get what delivery.

The possibility of the operators to edit the TLZs is the greatest improvement from the current system since the current "editing" of the TLZs is limited to the decision of which zip-codes that should make up each TLZ.

These two features alone are reason to conclude that the work of this thesis can improve the work of the operators at Jetpak. However, the research question was not about editing but how to check if a coordinate is inside a specific polygon in a sufficiently fast way. Thus, the main objective of this thesis is the Main Search program. The current version of the Main Search program should give the operators at the headquarters faster updates, even though this has not been tested, and more more up-to-date information.

In summary, these two programs and the features they bring, along with the performance improvements compared to the old system can give the operators at Jetpak a much better situational awareness.
Bibliography


