CHALMERS

Vector-based map cutting

In the context of cloud-based navigation systems

Master of Science Thesis in Electrical Engineering

ANDREAS KARLSSON
KRISTOFFER WILHELMSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2015

Vector-based map cutting
In the context of cloud-based navigation systems

Andreas Karlsson
Kristoffer Wilhelmsson

Examiner: Marina Papatriantafilou

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2015

Abstract

Vector maps are used when describing real world geographical objects, such as lakes or roads, with the convenience of vectors and vector objects. Each vector, or coordinate, describes a geographical point. Two or more vectors describe a line, or polyline, and three or more cordinates can describe a polygon. A lake can thus be described by a polygon with coordinates mapping its boundary, while a road can be described by a polyline with coordinates mappings its stretch.

When dealing with vector map objects you often bound the geographical area of interest by a bounding polygon. In its simplest form this is a bounding box, but arbitrarily shaped polygons can be used as well.

This thesis report examines how vector map objects in form of polylines and polygons can be cut out from arbitrarily formed polygons. It also describes how the arbitrarily formed polygons in turn can be created by splitting up a corridor surrounding a polyline, the corridor created by offsetting the polyline to each of its sides.

It goes into details on algorithms used for polygon clipping and polyline offsetting. The report also covers implementation of selected approaches into a vector map based mobile navigation system, as well as benchmarking and testing of it.

Contents

1 Introduction **3**

1.1 Background . . . 3

1.2 Case study: Cloud-based navigation application Wisepilot . . . 4

1.3 Analysis of old implementation . . . 4

1.3.1 Rectangular enclosure of vector maps . . . 4

1.3.2 Structural overview . . . 5

1.3.3 System performance . . . 6

1.4 Formulation of problem . . . 7

1.5 Aim of this thesis . . . 7

1.5.1 Project aim . . . 8

1.6 Limitations . . . 8

1.7 Geometric terminology conventions . . . 8

1.8 Human vs computational perception of figures . . . 9

1.9 Description of remaining sections . . . 9

2 Related work **10**

2.1 Offsetting algorithms . . . 10

2.2 Poly object simplification algorithms . . . 10

2.3 Clockwise polygons and polygon area algorithm . . . 11

2.4 Linear algebra . . . 11

2.4.1 Line intersection . . . 11

2.4.2 Relative positioning . . . 12

2.5 Locating algorithms . . . 13

2.5.1 Ray-edge intersection algorithm . . . 13

2.5.2 Angle-summation algorithm . . . 13

2.6 Polygon set-operation algorithms . . . 14

2.6.1 Weiler-Atherton algorithm . . . 15

2.6.2 Binary space partitioning tree algorithm . . . 16

2.6.3 Vatti's generic algorithm . . . 18

2.6.4 Experimental polygon tree algorithm . . . 19

3 Methodology **24**

3.1 Implementation of offset algorithms . . . 24

3.1.1 Equi-angular algorithm . . . 24

3.1.2 Encapsulation polygon algorithm . . . 25

3.2 Application of polygon set-operations . . . 28

3.3 Implementation of locating algorithm . . . 28

3.4 Computing relative position . . . 29

3.5 Implementation of polygon set-operation algorithm . . . 29

3.5.1 Inserting intersections . . . 30

3.5.2 Generating output . . . 31

3.5.3 Filtering intersections . . . 32

4 Results and discussion **35**

4.1 Measured results and system performance . . . 35

4.1.1 Discussion . . . 40

5 Conclusions and possible future work **41**
5.1 Conclusion 41
5.2 Possible future work 41

6 References **43**

1 Introduction

1.1 Background

Mobile navigation applications are common on today's mobile platforms. There are basically two different types of navigation applications; online and offline. Online, or cloud-based, is when the route calculation and map handling is done by an online server, as opposed to offline when routing and maps are handled locally on the device. Cloud-based car navigation applications have many advantages over offline applications, e.g. always fresh map data, no need for local storage on the mobile device and no cumbersome download procedures. There are however drawbacks that need to be considered in a cloud-based approach. The obvious drawback is that the map data needs to be downloaded on demand from the Internet, thus the device must have network coverage. To rely as little as possible on the network, but still enjoy the benefits of a cloud-based solution, it's important to keep the downloaded data amount as small as possible. Another reason to keep the downloads small is the cell phone data plan, which in the end the user of the car navigation application has to pay for.

This thesis is based on a hands-on project that was carried out at Appello Systems AB. Appello is developing and selling a mobile navigation system called Wisepilot. Wisepilot is a cloud-based car navigation application. Thus, for Wisepilot, low data consumption is a selling point compared to both offline car navigation as well as competing cloud-based solutions.

Before presenting the thesis's problem formulation and aims, here is some general terminology that will be used in the report.

Map server An online storage and service that can be queried for e.g. vector maps and navigable routes between two points.

Client The car navigation application Wisepilot, running on a mobile device.

Map data Map data is stored on a map server that can be queried for bounding box representations, resulting in vector map objects inside the bounding box. On the client-side the map data is held in memory by the application. Throughout the thesis, map data will always be referred to as being vectorized and represented by coordinates, as opposed to a rasterized map representation.

Map coverage Map data for a particular geometrical area. Could be a bounding box represented by two coordinates but could also be of a more general geometrical form, e.g. the geometry of a route with a given offset from the route.

Map object A vector representation of a road, water area, park area or anything found on a map.

Route A path between two or more geographical coordinates calculated by the navigation system. The route holds spatial information of the path, as well as navigational instructions used to guide the user while driving.

Corridor The area surrounding a route, defined by a given offset at every point on the route.

Map tile Or simply tile. A part of a an offsetted route, a set of vector map objcets bounded by a bounding box or polygon.

Wisepilot A cloud-based navigation system for cellular phones and smartphones. It consists of an end user application running on a device (a thin client), and a server side application feeding the client with navigable routes and map data surrounding the routes.

1.2 Case study: Cloud-based navigation application Wisepilot

Wisepilot is a cloud-based navigation application for cellular phones and smartphones. Being cloud-based, the application running on the device only shows what it downloads from the server. Wisepilot shows maps and navigable routes that it gets by querying an online server. To find a navigable route between two destinations the user submits its desired destination and the application queries the server for a route calculated from current position to the desired destination. The server responds with the route and the nearby pieces of map information needed to graphically represent the current location of the user. When the user progresses along the route the application detects this through GPS coordinates provided by the device, and updates the display of the map and the user's position accordingly. This way the user will always be timely informed about coming turns, roundabouts and other landmarks needed to navigate the route. During this progression, map information along the route ahead is prefetched and held readily available for the application to use when the user's scope reaches its perimeter.

Throughout this thesis references will be made to "the old implementation", which is the implementation used for querying and cutting vector map objects before the outcome of this thesis was implemented. The old implementation used a very simple algorithm deciding what vector map data should accompany the route. The algorithm used bounding boxes as map tiles to supply the route with map coverage along the route, which had the consequence of a an uneven map coverage and also the same data being sent twice when the rectangular tiles where overlapping eachother. The workings, and the limitations, of the old implementation are more thoroughly described in section 1.3.

1.3 Analysis of old implementation

In this section we analyze the implementation that was used by Wisepilot prior to this thesis project.We describe its bounding box algorithm to show how it works and to present its limitations. We also analyze and present its data structure and finally the metrics used to measure system performance.

1.3.1 Rectangular enclosure of vector maps

The old implementation used an algorithm which seeks to enclose the route with rectangles. These rectangles in turn represent the bounding boxes used by the lookup requests to the map server.

As figure 1 shows this algorithm provides a very uneven map coverage around the route since this coverage is not related to the geometrical properties of the route itself. This leads to the application intermittently showing insufficient map coverage and also overlapping tiles, resulting in unnecessary data transfers.

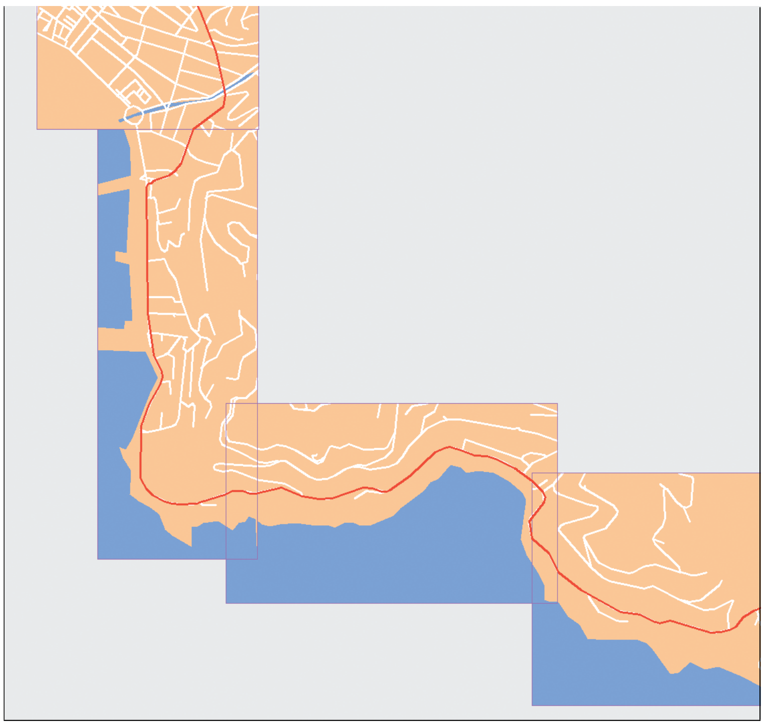

Figure 1: A part of a route near Girona, Spain, with map rectangles outlined in violet. Notice the sometimes very short distances from the red route to the gray field which holds no map coverage.

1.3.2 Structural overview

In the old implementaion map data is strucuted in map tiles formed as rectangles. These rectangles are fetched from a map server and holds infrastructural and cartographic data stored as polygons and polylines, representing objects on the map, e.g. buildings, water, roads, ferry lines etc. As the user travels along the route, new squares are passed down to the client. The map server can only be queried using bounding boxes, meaning that trimming of polylines and polygons will have to be done in the new implementation, as the new implementation will use arbitrarily shaped bounding polygons instead of rectangles.

The route (shown as a red line in figure 2) consists of vertices, defined in a class called ShadePoint, which holds positional data in terms of longtitude and latitude. ShadePoints are in turn, in terms of Java terminology, an extended version of the IntPosition class. To emphasize a vertex where the route makes a turn there is a class called WayPoint, which extends ShadePoint. The prefix 'Int' in IntPosition implicates that longtitude and latitude are stored as integers, multiplied with 100000 to have sufficient precision. Having positions stored as integers makes for easier handling on different types of mobile devices, which may have inefficient floating point handling. The shadepoints, which represents the route, are accompanied by map objects which are either PolyLineObjects or PolygonObject, which in turn also consists of IntPositions.

A map tile containing map data is represented by the VectorMap class, where functions for removing and adding map objects are found together with functions for writing the map to a stream or finalize objects before actually adding them. Figure 2 shows an example of a VectorMap object, with a route object drawn on top of it. The orange background in the figure is also a part of the vector map, spanning the vector map.

It's sufficient to know that a vector map is a structure of polygons and polylines and that a route is defined by shade points with coordinates, with occasional way points where the route makes a turn, instructing the user how to navigate at that turn.

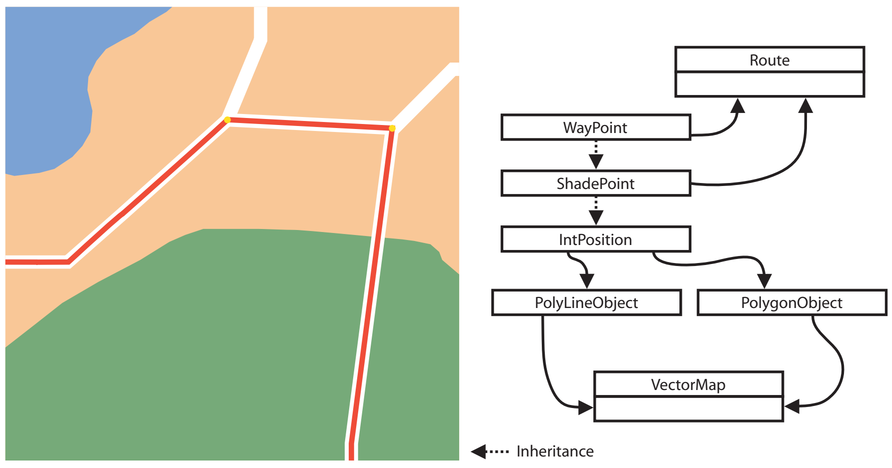

Figure 2: a) A vector map consists of polylines (white roads) and polygons (orange background, blue and green shapes). The route (red) is not part of the vector map, but is drawn and handled as an individual object. Waypoints are marked with a yellow dot. b) A simplified UML over how the classes are related to each other.

This report will not go into more detail about how each object is structured. Enhancements has been done to the existing structure to some extent, but this will be explained separately later on. Neither will the report explain how the algorithm in the old implemenation works when it determines the rectangles covering the route.

1.3.3 System performance

The two most important metrics are sufficient map coverage and small amounts of data sent to the client's handset. Using tiles shaped as rectangles presents a problem when combining these two. A route going from west to east, or north to south, or vice versa, is optimal for an algorithm with only rectangular tiles. However, the geographical and topological reality has a way of not allowing roads going solely in those directions. The most dreadful case for a rectangular enclosure is a diagonal route. In order to get good coverage on a diagonal road segment one has to use either a very large rectangle, including a lot of redundant data, or let smaller rectangles overlap each other. None of these alternatives are satisfying. With focus on keeping client downloads small the old implementation used little overlap and small rectangles, letting map coverage be somewhat scarce.

The system designed as part of this thesis can be guaranteed sufficient computational power by means of the powerful servers it's running on. Even though most design choices through out the project have been made with fast and efficient algorithms in mind, the fact that the new system will use more processor power has not been a major concern. Measurements on CPU cycles has not been done and discussion on the matter is not subject of this report.

1.4 Formulation of problem

In order to deliver map coverage to clients more efficiently several problems must be solved along the way from the map server to the client.

- When the map server is queried for vector map objects that will surround the route, the query must be performed in a way that guarantees complete enclosure of the route and its corridor. The amount of data generated by such a query might be massive if the route is long, and the number of queries made to satisfy the route must be carefully considered with regard to lookup overhead and data amounts.
- The map data surrounding the route must be trimmed according to a desired offset value, preserving the integrity of the vector map objects.
- The map data surrounding the route, i.e. the corridor, must be spliced into tiles before it's sent to the client. This is because the client can't handle too much vector map data at a given moment. When splicing the map data in to tiles, consideration must go into the size of these tiles in regard to the capabilities of the device and network overhead. Again, if such a splice were to occur just over a map object, the object's integrity must be preserved.

1.5 Aim of this thesis

The aim of this thesis is to describe the solution and implementation of the problems stipulated in 1.3.1. In short we want to develop and implement a route encapsulation corridor that doesn't result in much more data traffic than the old implementation, and that also presents an even coverage around the route. To accomplish this we need to:

- evaluate a method for calculating an offset around a route represented by a polyline
- evaluate a method for selecting map data that corresponds to the above mentioned offset geographically
- evaluate a method for splicing vector objects
- evaluate and test different map- query- and splice- sizing parameters with regard to network, system and device performance

Existing work related to the above problems will be investigated in order to find algorithms and methods useful for the solutions to the problems, this is described in section 2. If none is found, or deemed suboptimal, effort will be put into the creation of an algorithm that meet the demands.

1.5.1 Project aim

The project that is the base of this thesis is to provide new and improved algorithms for bounding vector maps, and also to implement the algorithms for use in Wisepilot. The main aim is to provide better map coverage, and if possible avoid increasing the data amount.

1.6 Limitations

This thesis does not cover route algorithms or route calculations. The routes mentioned and used in this thesis are expected to be served by any service providing a polyline describing the route with latitude and longitude coordinates. The scope of this thesis does also not cover the details of the old implementation's data structure and algorithms. Outside of the scope is also algorithm efficiency and comparisons in processing time between different algorithms, new and old. What is also not covered is comparison with other navigation systems or map software. The thesis focuses on the methods used in handling the map data itself.

1.7 Geometric terminology conventions

In the report a number of geometric expressions are used. For the reader's convenience they are briefly explained in this section. All geometry is two dimensional, so that a position can be defined by two coordinates x,y or more formally in geographical contexts: latitude and longitude. The following terminology will be used henceforth:

Vertex A two dimensional position. May be on its own or part of a structure of two or more vertices.

Edge A line segment spanned between two vertices.

Polyline A series of one ore more connected edges. A polyline may intersect itself.

Polygon A polyline where the first vertex has the same coordinates as the last vertex. A polygon may not intersect itself, then known as a *simple* polygon.

Merge A merge between two polygons is the result of the union of the sets represented by the span provided by their vertices.

Clip The intersection between two polygons.

1.8 Human vs computational perception of figures

Algorithms are used to enable computers to do what we as humans might see as trivial actions. This is especially true when dealing with geomtrical figures. Take for example the operation of clipping or intersecting two polygons. These are operations that are intuitive and trivial for a human simply by tracing a pen on a piece of paper. Completions of these tasks using computiation seems cumbersome in comparison because the human has instant access to all geometrical properties of the polygons in question simply by looking at them. As the human mind creates a perception of the image, information regarding all the vertices in the polygons becomes readily available: Which edges intersects which, which points lie inside which polygon and so on. Then the human uses this information (subconciously, or not) in guiding the pencil.

A computer, however, does not use visual percepts. Instead, a data-structural representation must be created in order to evaluate the image, which is then used to compute the result of the intended operation. These two parts are the basis of any polygon operation algorithm: the creation of a data structure representing the polygons and the subsequent traversal or manipulation of this strucure to create the desired result. Different algorithms have different ways of solving these two subproblems, leading to different complexities in the solutions of the subproblems. If the representation of the image contains a lot of useful information - it will be a lesser problem to compute the operation. On the other hand, if the representation of the image is sparse, more computation is needed to complete the operation. This means that computational complexity can be shifted between the two operations, but how much information (in total) is vital to the desired operation? Clearly, if the amount of information needed can be minimized - the operation will require less computation and that is always desirable.

With this in mind we will in the next chapter take a closer look at what previous work is available in the field of polygon and polyline operations and algorithms.

1.9 Description of remaining sections

Section 2 will cover related work in the field of linear algebra and algorithms for polygon and polyline computations. Section 3 will describe how the algorithms were implemented. In section 4 we will cover the results and provide a discussion of the findings. Finally, section 5 provides conclusions and propose some future improvements to the implementation and algorithms.

2 Related work

2.1 Offsetting algorithms

Offset, or tracing, algorithms are used to traverse a polyline and trace its shape, thus creating a corridor around the polyline. There are a number of related articles published in journals and in-proceedings which have been studied in order to get a grip of the field of vector manipulation in general, and outlining and offsetting especially. Much of the work done in this field is pointed towards the metal industry to be used in metal cutters and pocket-machining. The algorithms published often have constraints, i.e. it only works on closed polylines without holes. One article presents a complete algorithm for outlining curves and polylines. [9] Measurements presented in the paper shows that it outpeforms much of todays de facto standard CAD software products. Although this being a fully functional set of algorithms to create outlines to polylines it is way too complicated to be implemented with reasonable effort. It would also require heavy studying for someone to get acquainted with the implementation if some adjustments are to be made afterwards. [2] [7]

2.2 Poly object simplification algorithms

A normal route requst may result in a route consisting of thousands of vertices, many of them not at all necesarry to get a sufficient picture of how the route makes its way from start to destination. In order to avoid unnecessary amounts of calculations, one of the first things that is done is a line-simplification, or line-generalization, of the route. Among the line simplification routines available in the public domain one of the most renowned is the algorithm published by Douglas and Peucker in 1973 [3], and refined by others [5]. This is the algorithm used for line-simplification throughout this thesis. It is a recursive algorithm were a vertex's redundance is determined by its distance to a fictious line connecting the start and the end point of a polyline. If no intermediate vertex is distanced enough from the fictious line, the line is used as an approximation. Otherwise, new fictious lines are produced using the vertex farthest away from the previous line as starting and ending point, respectively. The routine is visualized in figure 3.

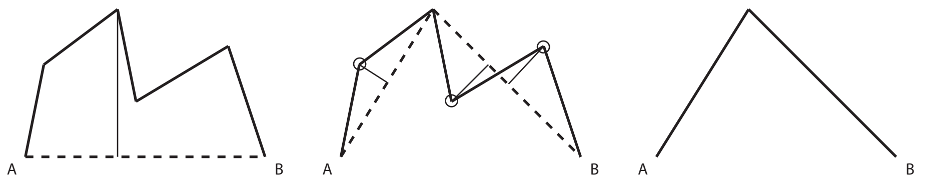

Figure 3: Example of a line-simplifcation for a line A to B. Note the fictious dotted line, wherefrom distances to intermediate points are calculated. Points with circles are placed closer than a certain distance, ϵ, from the fictious line and not present in the generalized line to the right.

Not only are line-simplification carried out on the route itself but also on other ojbects present in the cartographic representation of the map, such as roads, parks, water areas etc. The only variable that affects the end result of a line-simplification is the maximal allowed deviation, ϵ, the maximal distance between the fictious lines and the vertices labeled redundant. When dealing with this algorithm it's noteworthy that a line's starting point cannot be the same as its end point, causing the algorithm to not terminate. Hence, all polygons must be made polylines when conducting simplification on them.

2.3 Clockwise polygons and polygon area algorithm

When dealing with polygons it is important to know if they are oriented clockwise or counter-clockwise. It's also important to have consequent orientation on polygons sent to the client, where they are triangulated and drawn as triangles. The MIDP specification for Java on embedded devices can only draw triangles and it is important that the polygons sent for triangulation are all oriented in the same way. A method to check clockwiseness was already present in the existing framework but proved to be inefficeint and not fully trustable. To tell the clockwiseness of a polygon one can calculate the signed area and if found negative the polygon is clockwise, and vice versa. The signed area, A, of a polygon with N vertices is given by formula 1.

$$A = \frac{1}{2}\sum_{i=0}^{N-1}(x_i y_{i+1} - x_{i+1} yi) \tag{1}$$

However the calculated area of a polygon is not used in the final implementation it proved helpful in the development process when comparing map coverage between different tiles and overlapping tiles etc. [4] [1]

2.4 Linear algebra

2.4.1 Line intersection

To detect whether two edges intersect, the intersection point of two lines drawn coincident with the two edges are evaluated. As shown in figure 4 the coordinates of the edges are used to create two vectors. Using expression 4 the respective scalars defining the intersection point is evaluated. A scalar value including zero up to and including one asserts that the intersection point is on the vector. Zero puts the point on the base of the vector, and one on the point. Thus if both scalars are in that interval the two edges intersect eachother and the coordinates of the intersection point is evaluated using expression 5.

Forming the two lines' equations:

$$\begin{aligned} P_{AB} &= A + s_{AB}(B - A) \\ P_{XY} &= X + s_{XY}(Y - X) \end{aligned} \tag{2}$$

$P_{AB} = P_{XY}$:

$$\begin{aligned} A_x + s_{AB}(B_x - A_x) &= X_x + s_{XY}(Y_x - X_x) \\ A_y + s_{AB}(B_y - A_y) &= X_y + s_{XY}(Y_y - X_x) \end{aligned} \tag{3}$$

Solving for s_{AB} and s_{XY} yields:

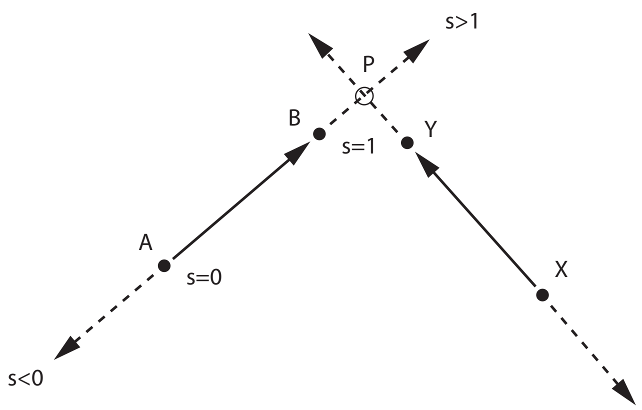

Figure 4: Two lines as described by four points

$$
\begin{aligned}
s_{AB} &= \frac{(Y_x - X_x)(A_y - X_y) - (Y_y - X_y)(A_x - X_x)}{(Y_y - X_y)(B_x - A_x) - (Y_x - X_x)(B_y - A_y)} \\
s_{XY} &= \frac{(B_x - A_x)(A_y - X_y) - (B_y - A_y)(A_x - X_x)}{(Y_y - X_y)(B_x - A_x) - (Y_x - X_x)(B_y - A_y)}
\end{aligned}
\tag{4}
$$

Using s_{AB} or s_{XY} accordingly, P can be calculated:

$$
\begin{aligned}
P_x &= A_x + s_{AB}(B_x - A_x) \\
P_y &= A_y + s_{AB}(B_y - A_y) \\
\\
P_x &= X_x + s_{XY}(Y_x - X_x) \\
P_y &= X_y + s_{XY}(Y_y - X_y)
\end{aligned}
\tag{5}
$$

If the denominator in expression 4 is zero, the lines are parallel. If the numerator and denominatior both are zero, the lines are coincedent.

2.4.2 Relative positioning

Relative positioning, "left of" or "right of" and relative distance between an edge and a point can be calculated by use of the cross product of the vector depicting the edge and the vector depicting the point. The cross product between two three-dimensional vectors are a third vector orthogonal to both its founding vectors. In our case the two vectors are always in the xy-plane which means that the resulting cross product will be parallel to the z-axis. Evaluating the cross products z-component thus tells us the sign of the angle between b and c.

$$\begin{aligned} a &= b \times c \\ a_x &= b_y c_z - b_z c_y \\ a_y &= b_z c_x - b_x c_z \\ a_z &= b_x c_y - b_y c_x \end{aligned} \tag{6}$$

Also, the angle between b and c affects the lenght of a accordingly:

$$|b \times c| = |b||c| \sin\theta \tag{7}$$

2.5 Locating algorithms

Clearly, it is of the highest importance to have a way of deciding whether any given point lies inside or outside of any given polygon. This method will become a vital tool in the solution of many subproblems that are encountered along the way. For example, if the method is applied to all the points in a polygon P with regard to any other polygon Q one can easily deduce some basic knowledge about these polygons spatial relationship to eachother. The application of this knowledge is further discussed in section 2.6.

2.5.1 Ray-edge intersection algorithm

The ray vertice algorithm is a simple assumption based on the Jordan Curve Theorem. The theorem states that every nonintersecting loop in a plane divides that plane in an "inside" and an "outside". The algorithm consists in counting the number of intersections between an arbitrarilly directed ray emanating from the desired point and the loop, see 5. The loop in this case being the edges of the polygon the point is being compared to. If zero or any even number of intersections are found, the point clearly is outside the polygon. It follows that if one or any odd number of intersections is found, the point is on the inside of the polygon. [6]

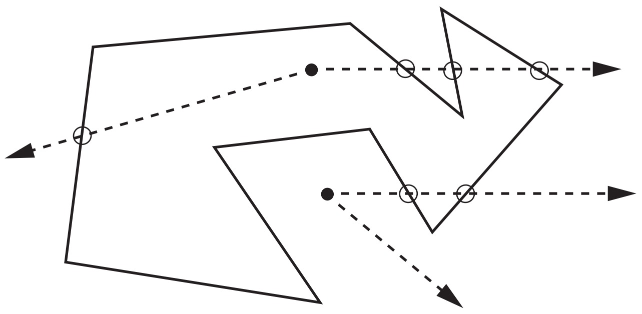

Figure 5: Rays for different points and their intersections

2.5.2 Angle-summation algorithm

In the angle summation algorithm, all the angles between the rays formed by the point and each vertice is summed. If the sum is zero the point is outside the polygon (7), otherwise it is inside the polygon (6). [6]

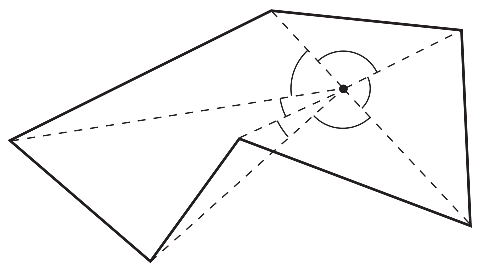

Figure 6: The angles formed by a point in the interior of a polygon.

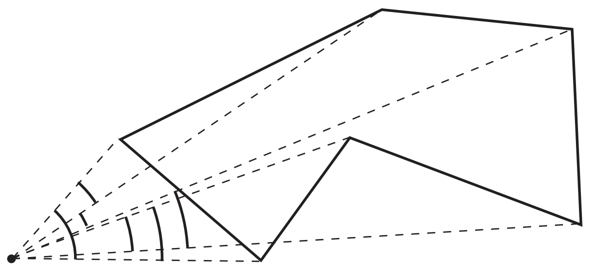

Figure 7: The angles formed by a point external to a polygon.

2.6 Polygon set-operation algorithms

It is clear that we need to perform two basic polygon operations, intersection (or clipping, denoted ∩) as shown in 9 and union (denoted ∪) as shown in 8. There's also the case where all points in one polygon are inside the other, but there still exists an intersection, as shown in 10.

If all the points in P are outside of Q, whilst all points in Q being inside P, and there are no intersections between any edges in P and Q it is easily understood that P∪Q = P and P∩Q = Q. Thus we can easily compute these polygon operations in the case above. For the sake of argument it is generally asserted that the two polygons discussed are intersecting one another, though we may not know where or how many such intersections exists.

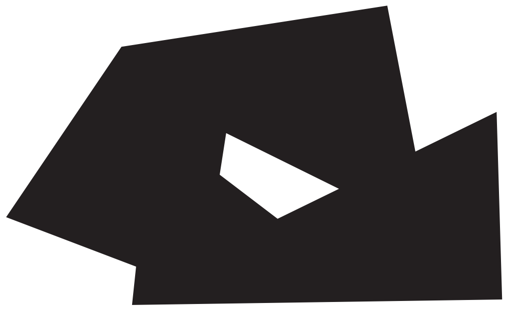

Figure 8: The union between two polygons.

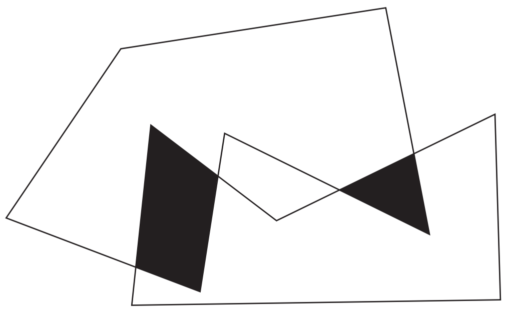
Figure 9: The intersection between two polygons.

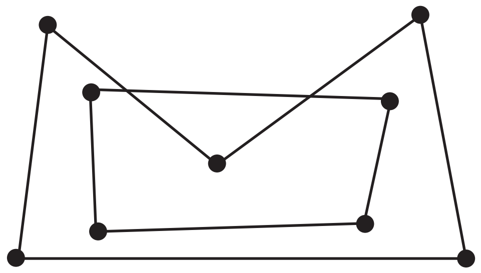
Figure 10: Although all the points in one polygon is inside the other, intersections do exist.

2.6.1 Weiler-Atherton algorithm

The Weiler-Atherton algorithm is the algorithm that closest mimics how a human would go about solving the problem at hand. For example, a person pen-tracing the union of two polygons would put her pen down on a point outside the other polygon and then start the trace, switching the pen between the two polygons' edges at each intersection, and stopping when reaching the starting point. If the intersection was the goal, she would simply start the trace on a point on the inside of the other polygon and carry on like above. [15]

Representation

The polygons are commonly represented as linked lists ordered so that list-traversal corresponds to walking clockwise along the edge of the polygon. The edges in P, constructed from two consecutive points in the list, are traversed checking for intersections against the edges in Q.

When an intersection is found, it is inserted in both lists between the points creating the intersecting vertices. This continues until all vertices in P has been checked against all vertices in Q, resulting in the lists now containing all intersection points as well as the original points of the polygons.

Operations

The desired operations are then carried out simply by applying the human pen-sweep to the lists. The union, for example, is attained by starting on any point

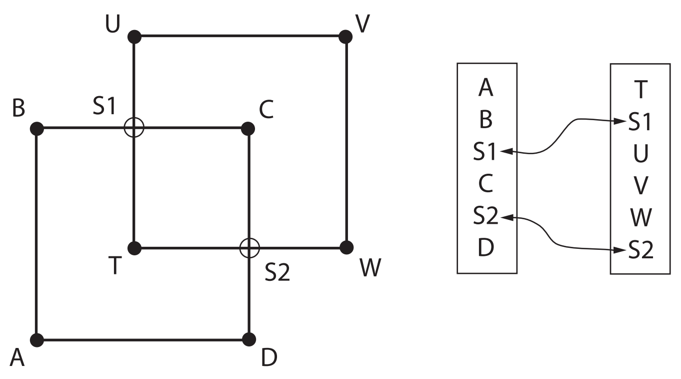

Figure 11: Polygons with inserted intersection points

in any list that is outside the other polygon, traversing the list and copying the visited points to a new list. If during the traversal, an intersection point is found, the traversal continues at the corresponding point in the other list. Thus mimicking the pen-tracing by switching list from which the result is read at every intersection point until the traversal reaches its starting point. Computing the clipping is the same as above, apart from starting the traversal on a point inside the polygon.

Considerations

The Weiler-Atherton algorithm is by far the simplest algorithm to comprehend due to its human-like behaviour but it has its drawbacks aswell. In implementing the Weiler-Atherton algorithm care must be taken to handle special cases that can arise from intersection detection. Inserting two or four intersections in the polygons depicted in figure 12 will yield correct results, but any odd number of intersections will generate incorrect results. This is the same problem as in figure 25. Counting the number of intersections between any two polygons is an easy way to check the integrity of the intersection detection. This since two closed loops in a plane intersect each other zero or an even number of times.

In order to achieve robustness, more information must be available than just the intersection points. Which points are inside or outside the other polygon? And if all are, but intersections still exist - further investigation is needed to decide where to start the traversal.

2.6.2 Binary space partitioning tree algorithm

The binary space partitioning tree algorithm is an algoritm for dividing and sorting a space using hyperplanes. This algorithm can be used to create a tree-structure representing a polygon, which can then be manipulated with regard to another polygon to obtain the desired set-operation. [11]

Representation

Generally, the BSP-tree algorithm takes any space and recursively divides it into two subspaces, associates a value with one space and the opposite to the

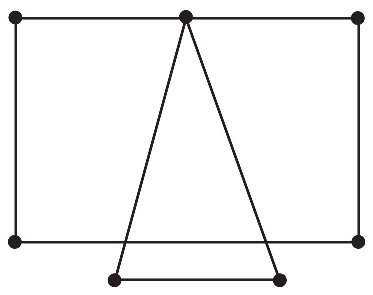

Figure 12: The number of detections between these polygons is dependent on how one defines an intersection. With regard to the Weiler-Athertor algorithm, two intersections is the "correct" number.

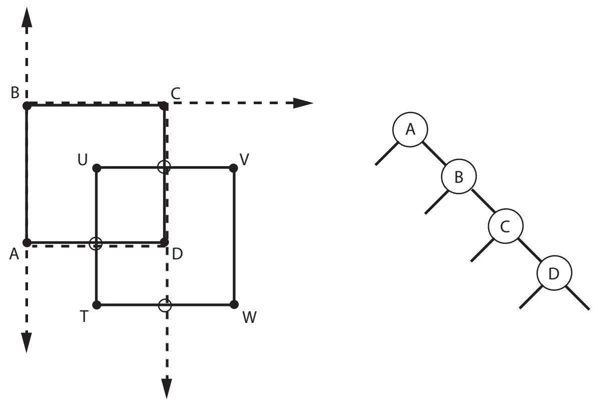

Figure 13: The subspaces created by the planes based in the polygon's edges. Note that the tree containing the edges are spatially sorted accordingly.

other. E. g. left and right. Thus the tree generally describes subspaces of any space, and their relative spatial relationship. If these spaces boundaries were to coincide with the edge of a polygon, clearly it can be used to obtain information about that polygon's spatial relationship. This by asserting whether the subject polygon lies wholly or partially within a subspace of the other polygon represented in the tree. [8]

The polygon is recursively divided, using its own vertices as basis for the planes and the two parts are inserted into the left/out and right/in of the node representing the plane. Thus creating a binary tree representation of the polygon. Inserting the faces of another polygon into the tree is done by checking its relationship to the node-plane and moving down the tree accordingly. If an edge were to intersect a node's plane it is split accordingly and the pieces passed down their corresponding side. When all faces of the subject polygon is inserted, the relative spatial relationship between the two polygons are known. This since all vertices in the subject polygon has been sorted into subspaces created by the faces of the clip polygon.

Operations

The tree representation of the two polyogons can now be manipulated and traversed. If, for example, the union of two polygons is wanted the desired result is actually the edge describing the edges of the polygons that are mutually external to eachother. Which edges of the subject polygon that are external to all subspaces created by the clip polygon is already defined by the tree - so traversing the tree and puzzling together the external edges is obviously part of the solution, but as can be seen in figure 13 the edges of the clip polygon also needs manipulating in order to obtain the the union's edges. In that case edges C-D and D-A needs partitioning according to their intersection with the subject polygon. There are several ways of solving this. For example node-pushing as described by [13] can be used to split the edge of the clip polygon. Another way is to construct two BSP-trees, one for each polygon, and then merge them. [10] When the tree is finally manipulated to contain all faces needed for the desired result a polygon can be constructed using a tree-traversal algorithm.

Considerations

While harder to intuitively comprehend than the Weiler-Atherton algorithm, the Binary space partitioning tree algorithm has advantages. The subdivision and sorting of space is a logically assertive way of approaching the problem (if, for example a point is "outside" every subspace created by the polygon - logically it must be outside the polygon itself). Also, optimization for specific operations can be done simply by discarding faces instead of passing them down the tree. If, when inserting the faces of the subject polygon, the union is desired - a face asserted to be on the interior of one subspace does not need checking against the subspaces lower in the tree [13].

2.6.3 Vatti's generic algorithm

Vatti's generic algorithm processes both polygons at once in a horizontally partitioned fashion. The polygons, in turn, are assigned types according to their edge-profile. The algorithm searches for events and determines the output based on the type of the edges constituting the event.

Representation

The polygons are thought of as having local maximas and minimas as described by figure 14. The edges connectining a local minima and a local maxima are asserted to be either left or right intermediates. This depending on their respective orientation compared to the local minima. Both polygons are scanned, bulding a table containing local maximas and minimas.

Operations

When the polygons have been scanned, they are in effect partitioned according to the table containing the local minimas. The polygons are then processed in a horizontally partitioned fashion using "scanbeams". A scanbeam is defined as the area between two succesive fictional lines which are drawn horizontally through all the vertices in the polygons. Each scanbeam is then processed by

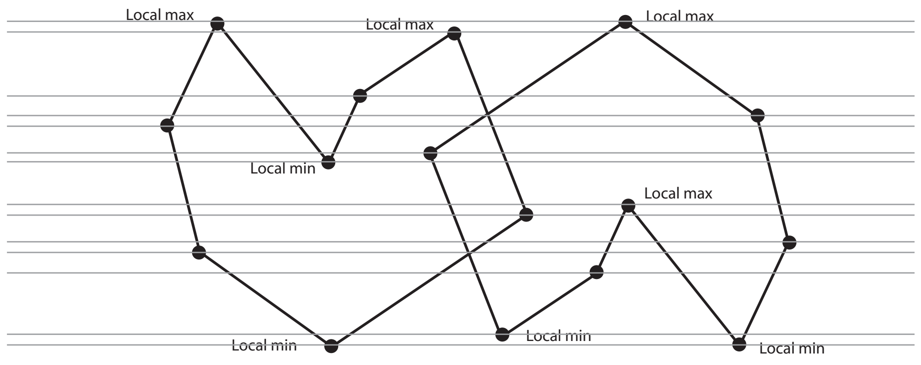

Figure 14: Two polygons after preprocessing, partitioned according to local maximas/minimas. Also shown are the scanbeams.

checking the faces intersected by the scanbeam for intersections with eachother. The scanbeam methodology is a simple way of asserting if two faces can intersect eachother. Two lines that does not share the same horizontally infinite space cannot intersect eachother, but two that does may or may not. To generate the desired output polygon(s) the scanbeams are sequentially processed according to a set of rules. The rules stipulate how different events found in a scanbeam shall be processed and, if appropriate, what should be written to the output. E. g. "clip's left intermediate intersects subjects right intermediate: generate local maximum in output". [14] Thus the output is incrementally generated as the scanbeams are processed.

Considerations

Vatti's generic algorithm is the least intuitive of the three studied algorithms due to its sequential processing of events, whose result is simply asserted by way of processing rules. However, it is capable of handling self-intersecting polygons and have only one documented special case. This is in regard to horizontal lines being able to lie on the edge of a scanbeam.

2.6.4 Experimental polygon tree algorithm

This algorithm was developed as attempt to find out why, what kind of and how much information is needed in order to compute the union and intersection between two polygons. The goal was to create an algorithm that partitions and sorts one polygon by intersecting it with another, thus creating a tree structure representing both polygons from which the union and intersection could be extracted through a simple traversal. Instead of using hyperplanes (like in the BSP-tree algorithm) only the edges of the clipping polygon acts as the partitioning agents, and the two resulting parts of the subject polygon are sorted either to the left or the right side of that edge. The main difference being that whilst the BSP-tree algorithm always sorts the edge in question to the left or right, the experimental algorithm omits clip- and subject polygon edges that does not intersect from the sorting. This in an attempt to minimize the amount of information that is needed to achieve the result.

This tree-representation of the two polygons is then to be traversed using a set of rules according to the operation that is performed. The investigation initiates assuming only relative spatial knowledge of "left of" and "right of" is needed. Also the tree should give sufficient information for both union and intersection operations. The tree-generation algorithm checks each of the edges in the clip polygon against the edges in the subject polygon. For each intersection found, the subject polygon is partitioned to the left and right of the intersecting edge. It is trivial that a point lying to the right of every edge in a convex polygon is inside it, but this is not true in the concave case as shown in figure 15. Thus it is necessary to obtain this information by other means for at least one point in each polygon. By knowing whether the starting point for a polygon walk is either inside or outside the oter, one can keep track of whether the next point is inside or outside depending on the number of intersections detected. For the sake of argument it is assumed that the polygon traversals start on a point outside the subject polygon and inside the clip polygon. Thus the tree can be constructed with full information about direction (left of/ right of) and locality (inside/outside other polygon). The polygons are processed in alphabetical order respectively in all examples. Eg. A-B vs T-U, A-B vs U-V and so on.

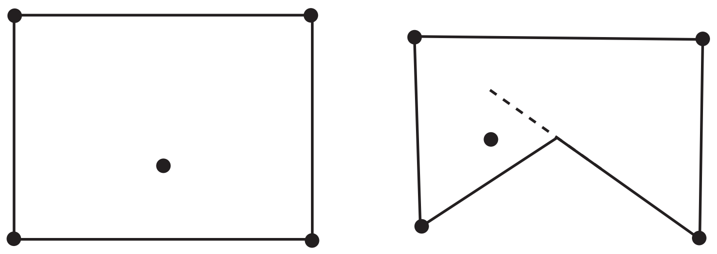
Figure 15: A point can be to the left of an edge, but still inside the polygon.

Pseudo code for generating the tree

```
Cn is the current edge in the clip polygon c0-cN

if Cn is the last point in the clip polygon {
   return a leaf containing the subject polygon
}

for each edge S:s1-s2 in the subject polygon s0-sN {
   if Cn intersects S {
      create intersection point X
      if s1 is right of C {
         append s0-s1,X to right polygon
      } else {
         append s0-s1,X to left polygon
      }
   }
}

if C did not intersect any subject vertice {
   append C to current node's vertices
   return this(Cn+1, subject polygon)
}
```

```
return node( this(Cn+1, left polygon), this(Cn+1, right polygon))
```

Note that the two polylines created by the intersection and subsequent partitioning are appended to either the right or left polygons, this in an attempt to create as much useful output as possible in the leafs of the tree.

Some polygons and their representation

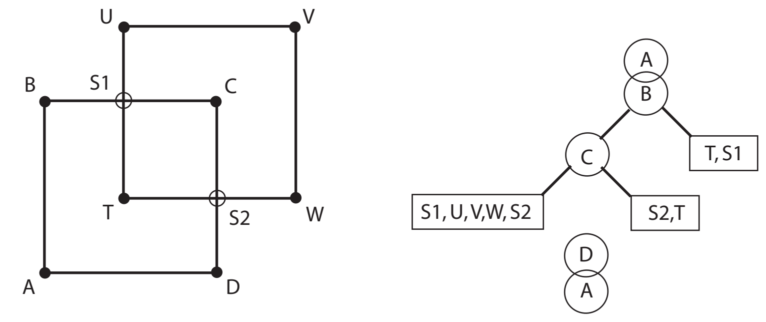

Figure 16:

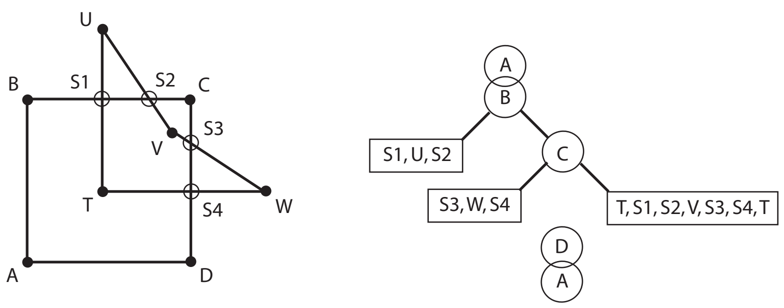

Figure 17:

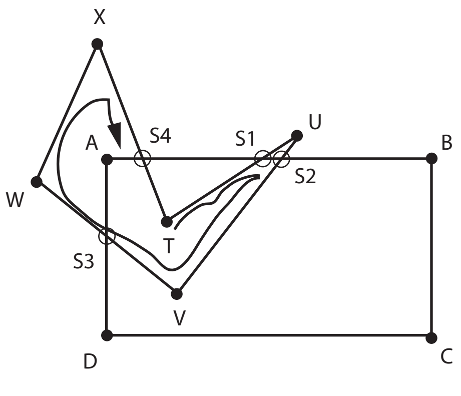

Figure 21: The curve marks the edges partitioned by A-B that cannot be said to be either left or right of A-B.

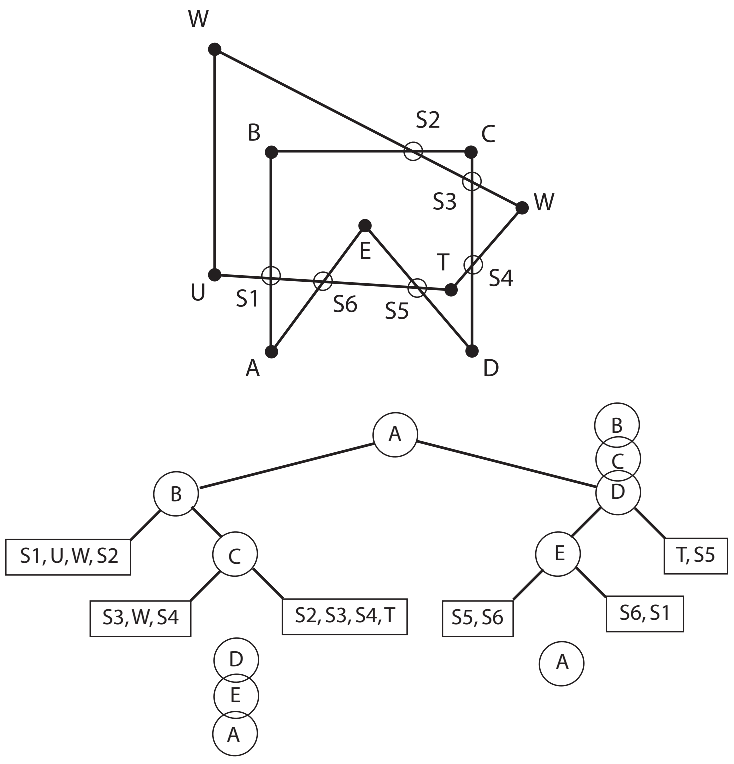

Figure 18: A polygon generating two subtrees

Generating output

As a first attempt at generating output a simple breadth first search is used. Depending on whether the intersection or the union is sought each level is processed in different directions. From left to right for the union and from right to left for the intersection. A leaf on the operation's side of the tree is simply appended to the output polygon whereas points in nodes are appended if they are outside of the clip polygon for the union, and inside for the intersection.

This approach generates a correct output for figure 16 and figure 17, but as is visible in figure 18, a way of handling double nodes is required. As it is desirable to keep the tree-traversal from jumping back and forth in the clip polygon the subtree that is first in the polygon must be processed entirely before the other. An other way to view this problem is by examining the two subtrees tail-section. In the subtree rooted in point B the tailsection contains points D, E, and A as if they would not generate any leafs. This, however, is plainly visible as D-E intersects T-U. Thus the subtree with the longer tailsection must be processed wihtout regard to the tailsections contents, whereas the subtree with the shorter tailsection is processed in normal fashion.

Considerations

As can be seen in figure 16 special care is needed when handling right leafs. If a leaf is a closed polygon, the other leafs are thus not connected to it and

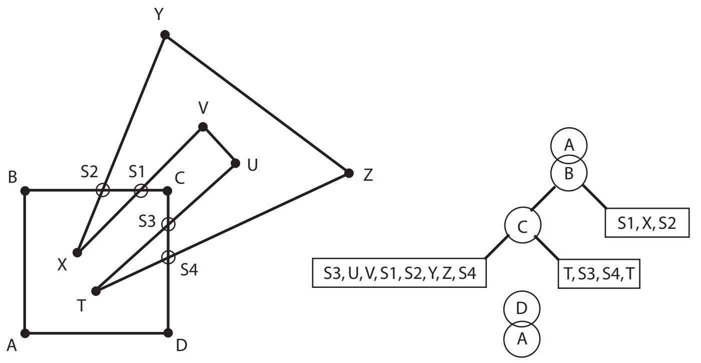

Figure 19:

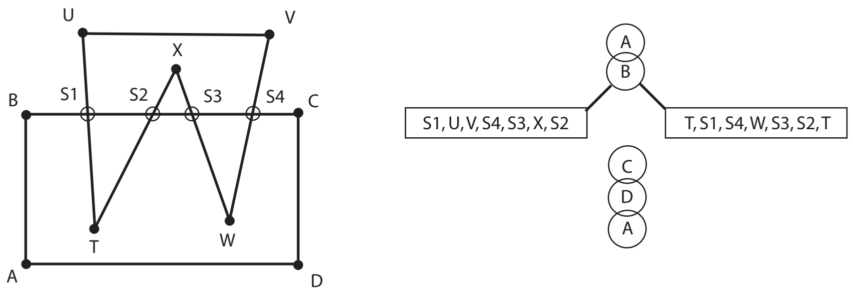

Figure 20:

generates additional polygons, which should be closed. If a leaf contains one endpoint it should be paired with the leaf containing the other.

Should the two subtrees be rooted in the same point the leafs must be processed in the order in which they are found when walking along the edge corresponding to the node. This could be acheived by spatially sorting the leafs intersection points with regard to the point of the node. Thus enabling the piecing-together of the output.

At this point the algorithm uses no less information or spatial processing than the Weiler-Atherton algorithm. Unlike the BSP-tree algorithm there is no spatial sorting done per edge, but only on the edges intersecting the clip polygon. For the algorithm to be a serious contender we must do away with the "inside or outside"-preprocessing needed in order to enforce the starting-point rules stipulated in the beginning. This, however causes the algorithm to break down. As can be viewed in figure 21, the edges marked by the curve cannot be asserted to be either on the left or right of the A-B edge. This could possibly be solved by terminating the tree, generating only the unambiguous leafs around the A-node and then passing the ambigouos partition down a new tree rooted in B. Then we would have two trees and a new set of puzzling-together problems. The conclusion of this is that in order to achieve a robust algorithm, one must keep track of the entire subspace encapsulated by a polygon. Either by enclosing it with partitioning planes related to eachother, or by asserting which points in the other polygon lies inside or outside it.

3 Methodology

3.1 Implementation of offset algorithms

As stated in section 2.1 offseting algorithms are used to trace a polyline. In the case of this project, as the algorithm traces down the route it creates a corridor around the route that will be used to create the desired map coverage. Two algorithms were tested and the implementation ended up using a combination of the two.

3.1.1 Equi-angular algorithm

Between three consecutive vertices in the route two line segments can be drawn, seperated by the angle θ. The equi-angular algorithm uses θ to place an offset point on both sides of the two connected line segments, so that the offset line on one side has the same angular distance to both line segments. Offset points are sited for all line segments in the route and forms a corridor around the route. Figure 22 shows an example of a line, with a corridor formed by this algorithm. An obvious problem arises when the angle between three consecutive points is smaller than 90°, which the figure demonstrates. A solution is to add extra points on the outside of narrow angles, or to increase the offset. These were tested in combination with each other, but the coverage could even now be insufficient on the inside between line segments with narrow angles.

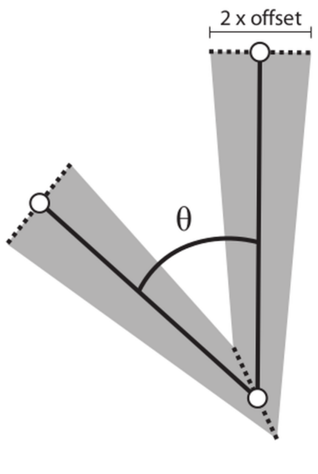

Figure 22: Example of the equi-angular algorithm done on three points. Illustrates the obvious problem with narrow angles.

Even though this algorithm didn't fully meet the needs with sufficient coverage, it's easy to use when dividing a longer corridor into tiles. With a tile ending with a fixed angle towards the route, it's convenient to start next tile with that same fixed angle. Thus eliminating redundancy in coverage, and in extent minimizing redundant data down to the client.

3.1.2 Encapsulation polygon algorithm

Unlike the equi-angular algorithm, which uses three consecutive vertices, this method forms an edge of two vertices and encapsules them with a polygon, e.g. a hexagon as in 23a. A new polygon is calculated from the next line segment and merged with the former using a polygon merging algorithm. Pilot studies at Appello regarding a corridor shaped map coverage included this method. The main issue with this algorithm, and also what kept Appello from starting a development of it, is to create a robust merging algorithm that handles all kinds of polygons. The merging algorithm itself is described in section 3.5. This method guarantees even coverage for the entire route but doesn't provide a intuitive way to split the corridor into tiles. One could simply merge a number of polygons created from encapsulation of the same number of edges, and let the result of this merge be a tile. The problem is that there will be some overlapping coverage between all adjacent tiles (23 b), not minimizing data sent to the client.

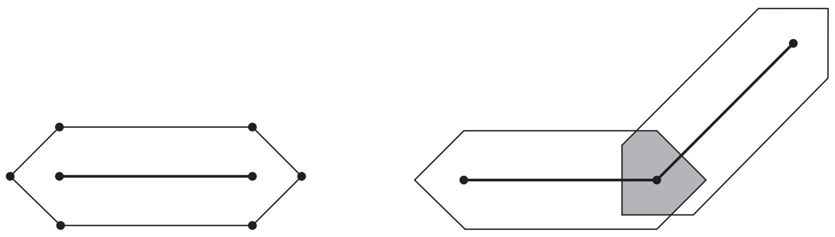

Figure 23: a) Example of an edge encapsuled by a hexagon and b) two hexagons overlapping each other.

As mentioned above the equi-angular algorithm for offsetting the route polyline proved insufficient in providing decent coverage when the route made sharp turns. On a lighter note, the fashion it delimited one offsetted edge from another was unambiguous, making it suitable for splitting a corridor into smaller pieces.

The polygon encapsulation algorithm did in fact provide a satisfying coverage but did not have built-in functionality to split the corridor into tiles, without generating redundant overlapping map coverage.

Taking the strength of both algorithms you end up with something purposive for making a corridor and split in into convenient sized tiles. The coming section will explain what preparations are made and how the tiles are constructed from the route.

Preparation of the route

First of all is the inserting of so called *splice points*, a boolean mark on those vertices where a tile splitting will occur later on. The splice points are set up at a preferred distance from one another, marking what is called the *preferred tile distance*. The distance between vertices might be too long so that the preferred tile distance marking fails. Leaving this untreated would create very large tiles, which in turn could lead to the downloading of tiles larger than a mobile device can handle. To avoid this behaviour intermediate vertices are inserted at the preferred tile distance. There is also another parameter in the working, a *precision* parameter. It defines how precise the tile length should be. The precision parameter ranges from 0 to 1, with a value close to 1 inserting many intermediate vertices. If a tile is to have a preferred distance of l kilometers, and the precision parameter is set to 0.9, it can use existing vertices in the route polyline to have a distance between $0.9l$ and $1.1l$ kilometers. For the case where no such vertices exist, intermediate vertices will be inserted at distance l.

Now when the vertices that start and end all tiles are marked, a polyline simplification is performed on the route. Not all vertices is needed in order to present the route properly. Fewer edges also means lighter computational load when merging the polygons that will encapsule each edge, and fewer vertices sent to the client. A special version of the existing line simplification has to be done, one that preserves the newly marked splice points. It is solved by handling the vertices between each splicepoint as separate polylines and using the existing polyline simplifier at each one of the semi-routes.

This concludes the preparation of the route. The following pseudo code does a preparation of a route, without doing any special treatment for first and last vertex in polyline:

```
loop through all vertices
    if length of current tile <= preferred tile length
        if length of current tile + distance to next vertex >= preferred tile length
            mark next vertex as splice point
            reset tile distance
    else if tile not long enough
        add distance between vertices to this tile, check next vertex
else if distance to tile was too long
        add intermediate point and mark it as splice point
        reset tile distance
end loop

simplify polyline with regards taken to splice points
```

Creating the actual tiles

Next step, after the route is prepared with splice points, the corridor itself must be created. This is where the polygon encapsulation comes in. The encapsulation algorithm starts from the beginning of the route and creates one tile at a time, as opposed to creating the entire corridor and then splitting it into tiles. The method will be explained using hexagons as the encapsuling polygons, however the algorithm works with any higher edged, even numbered, polygon. The algorithm is best described using a series of figures. Figure 24 describe it in nine steps for a short route with one splice point, creating two tiles. Below is a list that walks through and explains the figures.

a) N=0, where N is what vertice in the route that is discussed. This figure is a short route polyline with five vertices, ordered from left to right. The third vertice is marked as a splice point. New vertices in every figure will be filled with white.

b) N=1, in the first step two polylines are shaped around the first vertice. Direction of polylines/polygons are always clockwise around the vertice/edge. Notice that the right hand side (in the route direction) is only one vertice long.

c) N=2 is not a splice point, so the encapsulation will carry on. One vertex is added to the left hand polyline, and two on the right hand polyline. Notice that vertices added to the left is added last, while vertices added to the right is added first.

d) N=2, still on the second vertice. The hexagon around the first edge is completed by creating it from the left and right polyline. Two new polylines are being started on, again the right with only one vertice.

e) N=3, which is a splice point. Instead of doing the other side of the hexagon, one vertex on either side of the route is added. A fictious line from the route vertice to these vertices is splitting the angle that is formed by the two adjacent edges in two equal parts. The hexagon around the first edge and this pentagon is then merged to one big polygon, namely the first tile.

f) N=4, the next vertex is not a splice point, but we must use the two equi-angular vertices from the splice point to start next polygon. The polygon formed share the two equi-angular points with the first tile.

g) N=4, the start of a normal encapsulation.

h) N=5, the last vertex is handled like a normal vertex and a hexagon is created by the two polylines.

i) N=5, finishing off by merging the last two polygons, creating the second tile.

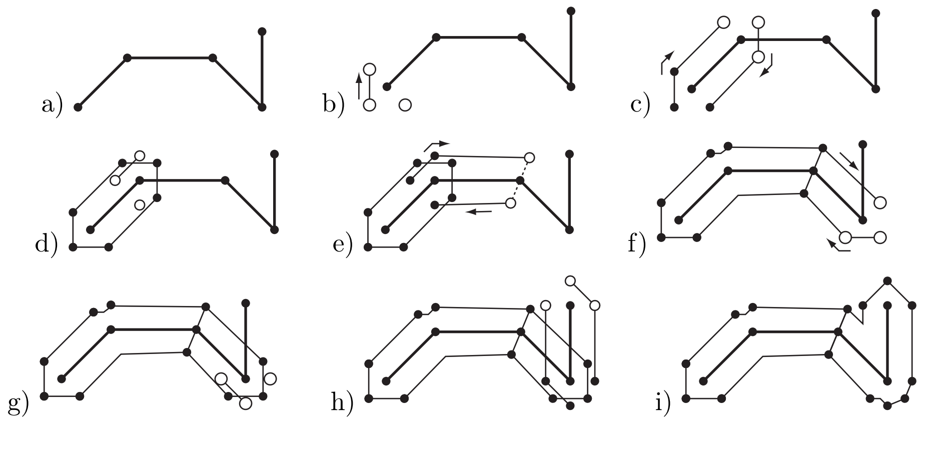

Figure 24: Creation of tiles from polygon encapsulation and equi-angular splitting.

So far the method is described for a standard case but there is a special case when the splicepoint is located at a vertice where the route creates a narrow angle. In this special case the algorithm locates the outer part (e.g. the right hand side of the route when it makes a left hand turn) of the curve and extends the offset with 30%. Nothing is done about the inside of the curve because extending the offset may create a non-simple polygon, i.e. a polygon intersecting itself.

Hexagon or higher edged encapsulation

The example with hexagon encapsulation and merging gives the idea of how a route can be covered by a corridor. If the system would allow curved lines one could use half circles on each end of the two lines parallell to an edge and encapsule with a fixed offset around the edge. This is however not plausible in the current system. What one can do is to increase the number of edges in the encapsuling polygon and use for example eight or ten edges. By doing this a more even coverage can be provided, but also creating more data that has to be sent to the client. A trade-off between even coverage and amount of data has to be done.

Simplification on tiles

After a tile has been created it may have many small edges not contributing to the overall perfomance of good coverage. By doing a simplification on created tiles along the route a lot of those small edges can be removed. All vertices that are created on either side of a splicepoint must be preserved in the simplification, otherwise gaps between tiles can occur. This is done by marking those vertices as splice points, so that they are preserved if ran through the new simplification algorithm. Again, a trade-off has to be done. Data sent to the client is decreasing with stronger simplification, but to strong a simplification calls for a very poor experience.

Tiles overlapping each other

Consider a route taking a sharp turn and then continuing on a path parallell and close to the first one. If the distance also is long enough so that a new tile is beginning somewhere near the sharp turn, the two adjacent tiles will then share a lot of coverage. Much of the information sent to the client will only be a duplicate of what was already sent. To get rid of such redundant data a check is done upon all created tiles. If the area spanned by the newly created tile shares 50% or more of the area spanned by the previous tile, those tiles will be merged before moving along with the rest of the route.

3.2 Application of polygon set-operations

This section describes why and how the polygon set-operations are used in order to create the desired map output. At this point the shape of the tile has been asserted, what is left to do is to filter the map output. This is done by requesting a square of map-data covering the tile and subsequently computing the intersection between the polygon formed by the tile and the polygons and polylines in the map data. The result of the intersection-computation is packaged into an object ready to be sent to the client.

3.3 Implementation of locating algorithm

Since the ray-vertice intersection algorithm will be dependent upon the implementation (and success) of the intersection-detection and also what is defined as an intersection (E. g. Does a vertice with one point on anoter vertice intersect that vertice?), the choie is made to use the sum of angles algorithm. This in order to ensure a high level of modularity in the program, meaning that one can modify the conditions for a detected intersection when dealing with polygon operations, thus minimizing the risk of inconsistent behavoiur originating from the locating algorithm.

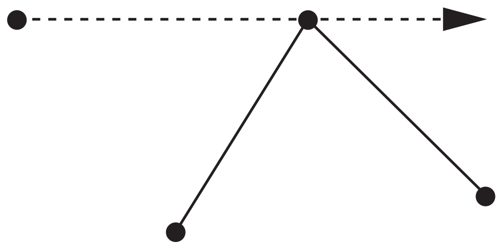

Figure 25: Zero, one or two intersections?

The implementation of the algorithm was part of the system supplied by our tutor.

3.4 Computing relative position

Three points are passed to a function, A, B, and X. The edge that is the basis of the comparison is described by A→B. The point's coordinates are treated as they were cartesian coordinates (x,y,0) and are shifted uniformly so that A is in origo and B describes a vector emanating from origo, →B, and X describes another vector emanating from origo →X. Using expression 6 →B× →X's z-component is evaluated. As given by 7 it will be zero if X is in line with A→B, greater than zero if X is left of, and lesser than zero if X is right of A→B. Note that for X to be on the edge A→B, the cross-product's z-component has to evaluate to zero and lie inside the bounding box formed by A and B's respective coordinates.

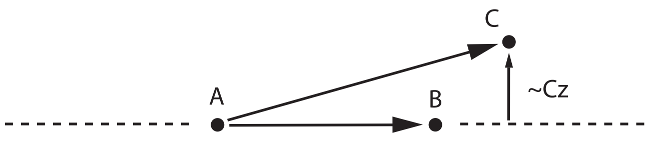

Figure 26:

3.5 Implementation of polygon set-operation algorithm

Since the number of polygons that are to be processed in each iteration are relatively small, 0-20, and the number of edges in those polygons are relatively small aswell, 3-15, performance of the algorithms was deemed of little importance. The algorithm should be easy to understand and modify in order to facilitate easy modification of the system's behaviour by any other person, and since the investigation of the experimental polygon tree algorithm had yielded further insight into the possible ambiguous intersections, and how they could be detected, the Weiler-Atherton algorithm was the final choice for implementation.

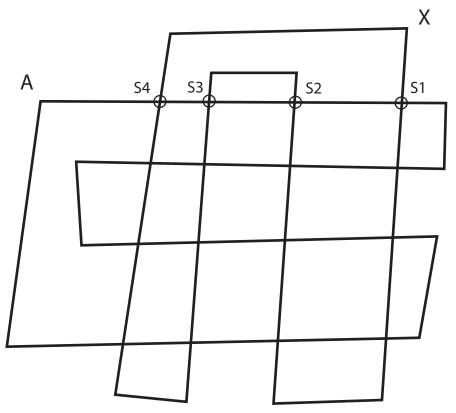

Figure 27: The intersections S1-S4 will be found in the enumerated order, but if they are inserted in that order into the list that represents the polygon X it will degenerate.

3.5.1 Inserting intersections

Firstly, the polygons' intersection points must be found and inserted into the polygons. This is not as simple as brute-forcing each edge in one polygon against the other, inserting the points when they are found. This is because of two properties of the problem. Firstly, the points in the polygons lie on integer coordinates, but two lines described by four integer coordinates may intersect eachother in a point that lies between integers. When inserting an intersection point, one must make sure that it does not alter the original intersection-behaviour of the polygon by moving it around. In this implementation the points are therefore inserted but kept inactive. This allows for them to be correctly ordered in the data structure without altering the polygon itself. Secondly, one must make sure that the points are inserted in correct order. If the intersection points in figure 27 are inserted into the clipping polygon the same order as they are found by a brute-force algorithm the resulting polygon will be out of order. Therefore it is necessary to either spatially sort the intersection points along the edge, or insert them in a smarter way. In order to solve this, a recursive way of inserting intersection points in a polygon was developed, thus mimicking a binary search. In order to calculate the intersection point expression 5 is used, at this point the scalar interval for a valid intersection is choosed as $0 \leq \mathrm{s} < 1$. Thus not covering the endpoints (which is the succeeding edge's starting point) of the edges that are compared. This in order to avoid overlapping intervals which would create duplicate intersection detection events.

Pseudo code for inserting intersection points

```
Cn is the edge n -> n+1 in the clip polygon
Sp is the edge p -> p+1 in the subject polygon
```

```
lowerBound and upperBound are the scalar values defining a valid intersection
on an edge in the clipping polygon, these values are always 0 and 1 for the subject polygon.

for each Cn : Cn = [n, generate(Cn, 0, 1, subject), n+1];

generate(Cn, lowerBound, upperBound, subject){
    for each edge Sp in polygon
        if Cn intersects Sp in point P and lowerBound =< scalarVal < upperBound
            return(generate(c1-P,lowerBound,scalarVal,subject),
                 P,generate(P-c2,scalarVal,upperBound,subject));
}
```

Thus the intersection points are inserted in both polygons in the correct order. Note the use of scalar values in restricting the interval on Cn for which an intersection is valid.

3.5.2 Generating output

Generating output is a fairly straightforward procedure called polygon walking. First, a suitable starting point for the polygon walk is selected. If the intersection between two polygons is to be attained either a point inside the other polygon or an entering intersecion-point is used as a starting point. For the union any point on the outside or an exiting intersection-point is selected. The polygon walk then commences from the starting point along the edges of the polygon until an intersection with the other polygon is found. At this point the walk swithces polygon and continues along the intersecting edge, continuing in this manner until returning to its starting point. The walk is now complete and the path taken by the walk is the result of the desired operation. Since this behaviour is completely independent of the direction of the edges it is crucial that the polygons are ordered in the same way. In this thesis and its implementation the convention of clockwise-ordered polygons are used. So what does this do in effect? The real idea here is that when walking two clockwise polygons, and "turning left" in each intersection the edge creating the outmost contour will be chosen. This since walkin along a clockwise polygon, its interior will always be to the right. Thus the innate directionality of the clockwise polygons are used in order to turn left or right in each intersection.

Generating output for polylines is done as above, with the only difference that no output is generated when walking in the clip polygon.

Pseudo code for output generation

```
P and Q are the processed polygons,
containing points p0-pN and q0-qN respectively

x is the current point in polygon P

generate(P, Q, x) {
    if x is marked
        terminate
    else
        mark x

    if p is an intersection point
        append p to output
        generate(Q, P, x+1) // this is a jump to the point after the intersection
                           //point in Q, a jump between the polygons
```

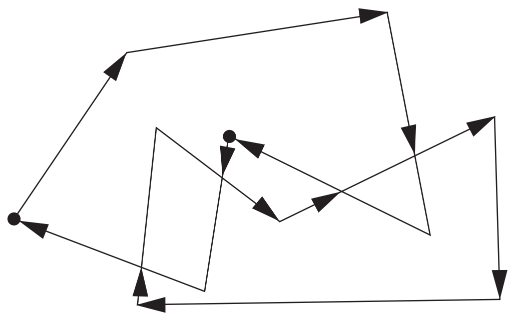

Figure 28: The two walks forming the union, note that the direction of the hole is counter-clockwise in accordance with the convention.

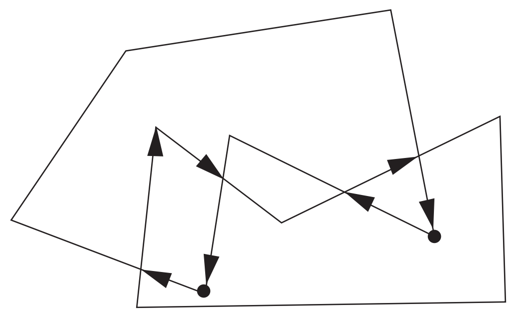

Figure 29: The two walks forming the two polygons that are the intersection.

```
        else
            append p to output
            generate(P, Q, x+1) // this is simply walking along the polygon
}
```

This generates both the intersection and the union depending on which point was selected as starting point. Also, if the polygons generates several polygons, the generation will terminate for each resulting polygon and a new starting point must be selected until no more starting point-candidates are unmarked.

3.5.3 Filtering intersections

Not all intersections detected are useful for for output-generation and introducing them into the representational structure of the algorithm will degenarate results in most cases. As can be seen in figure 12 only two intersections are useful for output generation, whereas the third, if introduced, will create a degenerative result. This is due to the algorithm's use of innate directionality of a clockwise polygon. In the general case , where two edges intersect each others interior, there can be no ambiguities with regard to the way the polygon walk will turn in the intersection. But when two polygons intersect eachother with one point lying on the edge of the other, special care is needed in order to avoid a wrong turn. As implemented in this thesis these ambiguous intersections are handled at the time of intersection detection and are either discarded or inserted. Since the directional assumptions for the general case is void, these special cases must be processed with regard to spatial directionality to eachother. Meaning that the decision whether to discard or insert an intersection point is based on mathematical evalutation of the directionality of the edges in question. Thus asserting whether the intersection is due to one polygon actually leaving or entering the other, instead of just reflecting on the edge.

Case 1: One point on the edge

In this case it is evaluated whether X and Z are on the same side of the plane created along edge A-B. If they are, the intersection is discarded. If not, an intersection point is inserted in A-B and Y is marked as an intersection point.

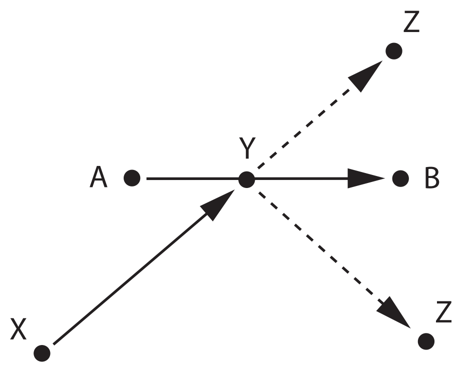

Figure 30: Case 1

Case 2: Point overlap

This case is slightly more complex since it involves more edges, it is possible to intersect the plane based in A-B without entering or leaving the polygon. Therefore the test must be augmented with the plane based in B-C to define the interior of the polygon. For example the left side, or exterior is formed as follows. If C is left of the A-B plane the left side of the polygon is what is left of the A-B plane intersected with what is left of the B-C plane. If C is righ of the A-B plane, the left side of the polygon is what is left of the A-B plane in union with what is left of the B-C plane. Thus the rule of "right-of = inside of" has been augmented to solve this special case.

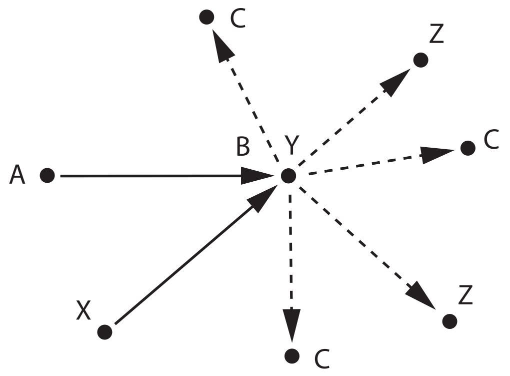

Figure 31: Case 2

Case 3: Case 1 and/or case 2 with padding parallel edges

This is in turn a special case of case 1 or case 2 which cannot be solved in the above ways since there is no useful information to be gained from parallel edges. Instead X is marked as being ambiguous, and the polygon is marked for special processing. If a polygon contiains an ambiguous intersection the one piece of directional information available from the ambiguous intersection is compared to that obtained from the first point in the polygon that useful information can be extracted from.

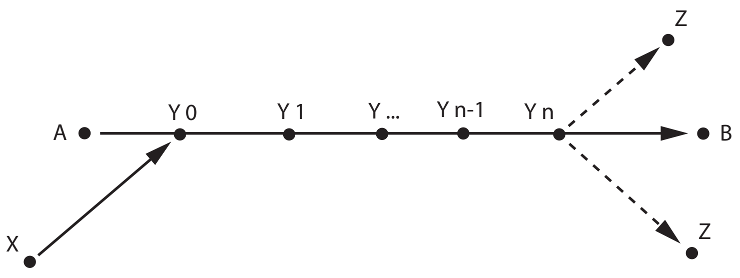

Figure 32: Case 3

4 Results and discussion

4.1 Measured results and system performance

This chapter is about the implementation and experimental evaluation and tuning of the methods presented in the previous chapter. The implementation was done in cloud-based navigation system Wisepilot, where a number of settings has been tuned to get the best possible performance out of the implementation. One major setting that won't be adjusted in any measurement is the offset distance parameter. This is the distance between the route and the edge of the map coverage. The offset distance is hence to be considered as a fixed value. This will be set to 200 m throughout all measurements that will be presented in this section. Using this offset gives a user experience that encompasses current needs in the client software. Discussions on how different offsets can be used to improve performance will be presented in section 5.2.

To get a fair picture of the performance, three different routes will be used in the measurements, specified in table 4.1. The first route is one within Gothenburg, an in-city route with some freeway and many roads beinged covered, not being the route itself. Next route is a typical Swedish long-distance route, main part being freeways, starting and ending in a city. Finally, an Austrian equivalent to the Swedish long-distance route, chosen because Austria is a big market for Appello.

The metrics that have been studied to meet the aims of the thesis are:

Tile simplification factor Decides how much the polygon that makes up each tile is simplified using the algorithm described in 2.2.

Route simplification factor Decides how much the route polyline to be encapsulated with a corridor is to be simplified, again using the algorithm described in 2.2.

Decagon or hexagon encapsulation Compares two geometric forms for encapsulation of the line segments making up the route polyline.

Merge overlapping tiles Compares merging of adjacent tiles that overlap geometrically.

Tile length When creating the corridor and splitting it up into tiles, the tile length describes how long each tile should be.

Tile length precision Decides how exact the tile length should be.

Start			**Destination**			**Route info**	
Name	Lat.	Long.	Name	Lat.	Long.	Length	Vertices
Järnbrott	57.65821	11.93172	Kålltorp	57.71594	12.0284	12.7 km	123
Gothenburg	57.70309	11.95902	Uppsala	59.85811	17.64463	459 km	2191
Vienna	48.20919	16.37279	Innsbruck	47.26267	11.39471	480 km	2349

Table 1: The routes that will be used in measurements.

Comparison between different tile simplification factors

Tile simplification is used to remove some of the angularity that arises when merging a lot of polygons together. Since every edge in the generalized route will be encapsuled there will be a lot of small edges in the corridor that doesn't add anything to the coverage. In fact, from a user perspective, the corridor looks a lot nicer without all the small edges around the route. A smoother corridor is to prefer not only because of its appearance but also of a somewhat reduced amount of data. Table 4.1 shows how the data amount for the test routes varies with different tile simplification factors.

Tile simpl. factor	**Järnbrott to Kålltorp**		**Gothenburg to Uppsala**		**Vienna to Innsbruck**	
	Bytes sent	Comp. to no simplification	Bytes sent	Comp. to no simplification	Bytes sent	Comp. to no simplification
0	8802	0%	88460	0%	116310	0%
10	8717	1%	87357	1%	114675	1%
20	8687	1%	87148	1%	114403	2%
30	8646	2%	86974	2%	114254	2%
40	8599	2%	86922	2%	114177	2%

Table 2: Shows how line simplification on the tile polygon affects the data amount sent to the client. Measurements are done with *route simplification* = 40, *tile length* = 2 km, *precision* = 0.9, *decagon encapsulation* and *merging of overlapping tiles*.

Using too much simplification on tiles leads to a problem with the encapsulation. In the above measurements a decagon encapsulation have been used. With a simplification factor of 40, edges in the decagon where removed, which of course can't be tolerated. Reducing the simplification factor to 30 preserves the decagon edges, but still does a little too much simplification. Figure 33 illustrates how simplification decreases the distance from the route to the corridor border, making it too short.

As a matter of fact, using no tile simplification at all doesn't guraantee a distance of 200 meters, since the corridor is based on a simplified route. Only when not using simplification on neither route nor tiles can one be sure to have a specified distance between the route and the corridor border. When using tile simplification set to 20, 10 or even no simplification, the distance from the route to the border is 173 meters. When doing distance calculations between two points on a curved surface, as the Earth, one cannot simpy apply Pythagora's formula. One can however apply Haversine's formula, explained in equation 8, where $R = 6371$ km is the approximated earth radius, d is the distance between two latitude - longitude pairs, $\Delta\phi = \phi_1 - \phi_2$ is the radial latitude separation and $\Delta\lambda = \lambda_1 - \lambda_2$ is the radial longitude separation. [12]

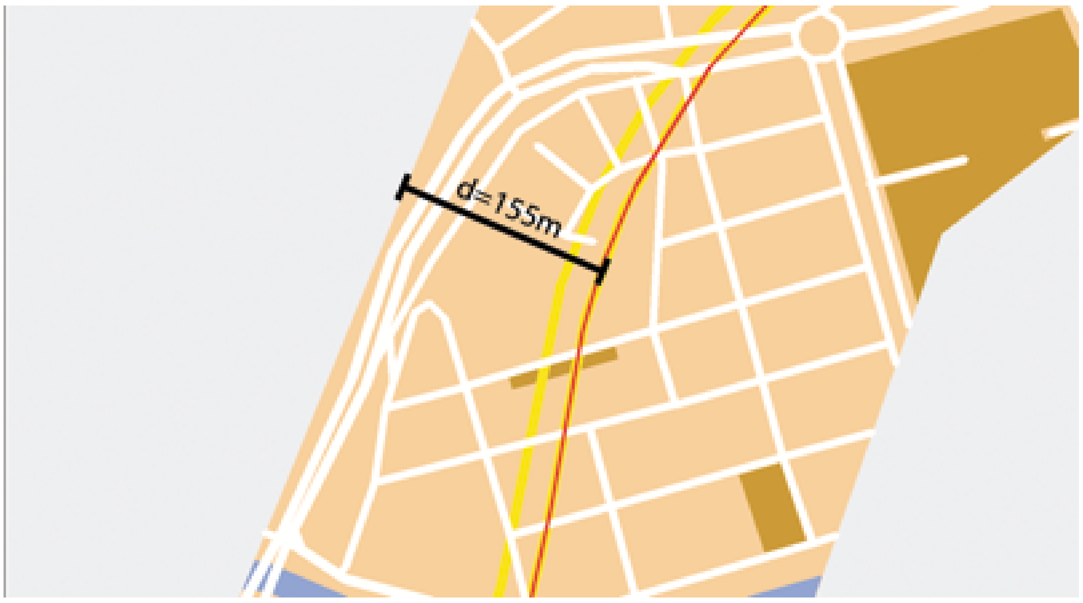

Figure 33: With the tile simplification set to 30 it generalizes the tiles too much, making the distance from the route to the corridor border too short.

$$\begin{aligned}
\text{Let }\ \text{haversin}(\theta) &= \sin^2\left(\frac{\theta}{2}\right) \\
\text{and let }\ \text{haversin}\left(\frac{d}{R}\right) &= \text{haversin}(\Delta\phi) + \cos(\phi_1)\cos(\phi_2)haversin(\Delta\lambda) \\
\text{then denote } h &= \text{haversin}\left(\frac{d}{R}\right) \\
\Rightarrow d &= R \cdot \text{haversin}^{-1}(h) = 2R \arcsin\left(\sqrt{h}\right)
\end{aligned} \tag{8}$$

Comparison between different route simplification factors

As mentioned in the previous section, route simplification has impact on the coverage. Since the route shown in the client's device is ungeneralized and the one that the corridor is built upon is more or less generalized, the distance from the route to the corridor border can vary. Aside from the coverage the route simplification also alters the data amount, since the number of encapsulations increases with decreasing simplification. Table 4.1 gives the details on how different simplification factors alters the data amount.

Using more than a factor of 40 results in a too rough generalization. Especially where the route makes soft, big turns. When simplified with a factor of 60 the corridor is too much off-centered to get a good coverage. The extra vertices in the less generalized route makes the coverage sufficiently centered on the route.

Route simpl. factor	**Järnbrott to Kålltorp**			**Gothenburg to Uppsala**			**Vienna to Innsbruck**		
	Bytes	Comp. to factor 10	Vert's	Bytes	Comp. to factor 10	Vert's	Bytes	Comp. to factor 10	Vert's
10	9059	0%	67	89630	0%	997	116425	0%	1052
20	8940	1%	43	88786	1%	705	115902	0%	764
40	8687	4%	27	86974	3%	514	114403	2%	545
60	8641	5%	22	86402	4%	437	113574	2%	459

Table 3: Shows how data amount, percental data amount difference and number of route vertices varies with different route simplifications. Measurements are done with *tile simplification* = 20, *tile length* = 2 km, *precision* = 0.9, *decagon encapsulation* and *merging of overlapping tiles*.

Difference between decagon and hexagon encapsulation

Using a decagon encapsulation rather than a hexagonal is an easy choice given the data in table 4.1. For the short in-city route the amount of data is a little bigger, but the coverage of both hexagon encapsuled routes are not good enough.

Polygon edges	**Järnbrott to Kålltorp**		**Gothenburg to Uppsala**		**Vienna to Innsbruck**	
	Bytes sent	Comp. to 6 edges	Bytes sent	Comp. to 6 edges	Bytes sent	Comp. to 6 edges
6	7987	0%	87104	0%	114407	0%
6[a]	7948	0%	86935	0%	114238	0%
10	8687	9%	87148	0%	114403	0%

a) Tile simplification factor = 30

Table 4: Measurements are done with *route simplification* = 40, *tile simplification* = 20, *tile length* = 2 km, *precision* = 0.9 and *merging of overlapping tiles*.

Merging of overlapping tiles

The difference between using and not using merging of overlapping tiles is very small for long-distance routes. For a route within a city, or for a short route, the difference may be larger. To see if a tile is to be merged or not is a very small operation and it can be worth the effort. A comparison between the two cases are shown for the example routes in table 4.1. For the example routes there is no difference, but given a route with many intersections it's more likely to make a difference.

Merge	**Järnbrott to Kålltorp**		**Gothenburg to Uppsala**		**Vienna to Innsbruck**	
	Bytes	Diff. to yes	Bytes	Diff. to yes	Bytes	Diff. to yes
yes	8679	0%	87148	0%	114403	0%
no	8679	0%	87148	0%	114403	0%

Table 5: Shows the lack of difference between merging and not merging adjacent overlapping eachother. Measurements are done with *route simplification* = 40, *tile simplification* = 20, *tile length* = 2 km and *precision* = 0.9.

Impact of tile length precision

Tile length precision determines how far before and after an existing vertice a new intermediate vertice can be placed. With a low precision the tiles might end up with very different lengths, and a high precision normalizes the average tile length. Table 4.1 tells us that for a start, for longer routes, the higher the precision the lower the data amount. A turning point is the precision of 0.93, where after the amount of data increases. The short in-city route shows off a result where three different, consecutive, precisions all yields the same result. They all produces the same tiles, and the precision 0.93 is in the middle of these three precisions.

Tile length	**Järnbrott to Kålltorp**		**Gothenburg to Uppsala**		**Vienna to Innsbruck**	
prec.	Bytes	Tiles	Bytes	Tiles	Byte	Tiles
0.7	8783	10	91406	291	120682	307
0.8	8681	9	89061	261	117425	271
0.85	9132	8	87859	247	115470	260
0.9	8687	8	87148	234	114403	247
0.92	8588	8	86937	233	114341	245
0.93	8588	8	86879	232	113445	243
0.94	8588	8	88726	231	114537	242
0.95	8495	8	88771	230	114345	241

Table 6: A comparison of different tile length precisions. Measurements are done with *route simplification* = 40, *tile simplification* = 20, *tile length* = 2 km and *merging of overlapping tiles*.

Comparison between different tile lenghts

The tile length is the parameter that most gravely changes the byte/tile value, not considering the offset that is constant at 200 m. As mentioned before, a mobile device can't handle tiles that are too big. This is because of the limited computational power, in particularly when keeping the client software compatible with old devices. Table 4.1 shows how the the size of the tiles varies with the length.

Tile length (km)	**Järnbrott to Kålltorp**			**Gothenburg to Uppsala**			**Vienna to Innsbruck**		
	Bytes	Tiles	Bytes/tile	Bytes	Tiles	Bytes/tile	Byte	Tiles	Bytes/tile
0.5	10890	26	419	14404	909	159	181955	958	190
1.0	9470	14	676	104548	460	228	136965	485	282
1.5	8986	10	899	92936	312	298	122131	327	373
2.0	8687	8	1086	87148	234	372	114403	247	463
2.5	8168	7	1167	83685	191	438	108885	198	550
3.0	8523	6	1421	80953	160	506	103767	167	621

Table 7: A comparison of different tile lenghts. Measurements are done with *precision* = 0.9, *route simplification* = 40, *tile simplification* = 20 and *merging of overlapping tiles*.

4.1.1 Discussion

Above tables shows how the parameters can be tweaked in order to get the best performance out of the new implementation. If all parameters are adjusted to the value that gives the most desirable result the demands set up for the new implementation is fully met. Table 8 shows how the new implementation compares to the old one. All paramters are adjusted to minimize the amount of data with respect also taken to user experience. When this project was started the requirement was to develop a route encapsulation corridor that did not result in much more data traffic than before, and that also presented an even coverage around the route. The results presented here fulfil these requirements, but also leaves room for some future improvements that can enhance the system performance, as discussed in 5.2.

Implementation	**Järnbrott to Kålltorp**		**Gothenburg to Uppsala**		**Vienna to Innsbruck**	
	Bytes	Diff (%)	Bytes	Diff (%)	Bytes	Diff (%)
Old	10251		123646		166741	
New	8588	16	86879	30	113445	32

Table 8: A comparison between new an old implementation. Measurements are done with *route simplification* = 40, *tile simplification* = 20, *tile length* = 2 km, *precision* = 0.93 and *merging of overlapping tiles*.

5 Conclusions and possible future work

5.1 Conclusion

The thesis set out to examine if a simple bounding box based clipping algorithm could be replaced by a more advanced algorithm based on bounding polygons formed by the shape of the route polyline. The conclusion is that, yes, it is possible. Not only is it possible but also provides a more even map coverage compared to the bounding box alternative. The data amount needed to download the vector maps is also reduced by as much as a third.

Now, even some time after the thesis actually took place, it's interesting to point out that the algorithms that came out of this thesis, and their implementation, are busy serving vector maps to the users of cloud-based navigation application Wisepilot.

5.2 Possible future work

Presented here are number of adjustments that can be made to the corridor and tile creating algorithms to improve the overall performance. They are presented with a brief description and an approximation of the effort it would take to implement them.

Dynamic tile sizes

Since some devices running the client application is not suitable for large tiles, a differentiation could be made when calculating the corridor and tiles for different screen resolutions. For example could a high-end mobile phone with high-speed data access cope with longer tiles, thus increasing the data amount, while a lower-end mobile phone could have the tiles calculated as proposed in table 8. On the other hand, if one would rather desire a wider offset, instead of the 200 meters used in the measurements, a high-end phone with high screen resolution could increase the offset but use the same tile length.

Another use of dynamic tile sizes could be to let tiles covering a city map have bigger offset and those covering a freeway have smaller offset. When driving on a freeway it's not likely a user is intereseted in roads passing nearby. In the city, on the other hand, a user might find it useful to see the surrounding road net, in order to get a better orientation.

An implementation of dynamic tiles for different unit types would not be a very time demanding operation. All parameters that control the tile shape are easily changed for each corridor calculation and can thus be dependent of screen resolution and type of handset. To make different offsets for different kinds of roads are bit trickier. It would mean a change of corridor offset in the middle of a corridor calculation and also information on what kind of road that is currently encapsuled etc.

Merge map server requests

When a request is made to the map server there is a lot of overhead data transfered every request. If a method for making the map server request for n succeeding tiles at once, the number of map server request could be decreased and the result faster calculated and transferred to the user. The response for todays system is not a concern, but a slight increased performance can be expected.

To implement this functionality is not to be considered especially difficult. There are some obstacles in merging and selecting from vector map data that would have to be conquered, but it could be done with superable effort.

6 References

References

[1] R. Boland and J. Urrutia. Polygon area problems, 2000.

[2] Byoung Kyu Choi and Sang C. Park. A pair-wise offset algorithm for 2d point-sequence curve. *Computer-Aided Design*, 31(12):735–745, 1999.

[3] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

[4] JSR 118 Expert Group. *Mobile Information Device Profile for Java(TM) 2 Micro Edition*. Java Community Process, Sun Microsystems, 2 edition, November 2002.

[5] John Hershberger and Jack Snoeyink. Speeding up the douglas-peucker line-simplification algorithm. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, volume 1, pages 134–143, Charleston, South Carolina, 1992.

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

[7] G. Kalmanovich and Gregory Nisnevich. Swift and stable polygon growth and broken line offset. *Computer-Aided Design*, 30(11):847–852, 1998.

[8] Ravinder Krishnaswamy, Ghasem S. Alijani, and Shyh-Chang Su. On constructing binary space partitioning trees. In *CSC '90: Proceedings of the 1990 ACM annual conference on Cooperation*, pages 230–235, New York, NY, USA, 1990. ACM.

[9] Xu-Zheng Liu, Jun-Hai Yong, Guo-Qin Zheng, and Jia-Guang Sun. An offset algorithm for polyline curves. *Comput. Ind.*, 58(3):240–254, 2007.

[10] Bruce Naylor, John Amanatides, and William Thibault. Merging bsp trees yields polyhedral set operations. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 115–124, New York, NY, USA, 1990. ACM.

[11] William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. *SIGGRAPH Comput. Graph.*, 21(4):153–162, 1987.

[12] `http://en.wikipedia.org/wiki/Haversine_formula`. Haversine — Wikipedia, the free encyclopedia, 2015. [Online; accessed 03-May-2015].

[13] `http://www.faqs.org/faqs/graphics/bsptree-faq/`. Binary space partitioning trees faq, 2015. [Online; accessed 01-May-2015].

[14] Bala R. Vatti. A generic solution to polygon clipping. *Commun. ACM*, 35(7):56–63, 1992.

[15] Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 214–222, New York, NY, USA, 1977. ACM.