Efficient Localization Of Points In A Set Of Concave Polyhedra

Master Thesis
Stream Infmath
Univ. Prof. Dr. Otmar Scherzer
Institute of Informatik
Faculty Mathematik, Informatik und Physik
University of Innsbruck

presented by
Philipp Glatzl

Supervisors: Dr. Harald Grossauer (UIBK)
Dr. Michael Schnell (DLR)
Dipl. Ing. Christian Bauer (DLR)

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum Unterschrift

Zusammenfassung

Im Rahmen dieser Diplomarbeit wurde eine Software erstellt, mit der es möglich ist effizient Flugzeuge ihren ATC-Sektoren zuzuweisen. Zu diesem Zweck wird in dieser Arbeit näher auf sphärische Trigonometry eingegangen, da die Flugzeuge auf der Erdoberfläche lokalisiert werden müssen. Insbesonders durch die Verwendung von sphärischen Dreiecken, ist es möglich geometrische Berechnung auf einer Kugel durchzuführen. Desweiteren liegt der Fokus dieser Arbeit auf der Punktlokalisierung, bei der ein Punkt ein Flugzeug darstellt und ein ATC-Sektor in Form von Polygonen vorliegt. Um dies zu bewerkstelligen wurde in diesem Zusammenhang die Software Bibliothek CGAL verwendet, die mehrere Algorithmen zur effizienten Punktlokalisierung enthalten. Diese Arbeit wurde im Zuge eines Praktikums bei und für das "Deutsche Zentrum für Luft- und Raumfahrt (DLR) "angefertigt.

Abstract

In the scope of this Master thesis, a software has been developed, which is able to perform an efficient aircraft assignment to the ATC-Sector, in which this aircraft actually is. For this purpose, this thesis contains a chapter about spherical trigonometry, as the aircrafts should be localized on the earth surface. Particularly the subject of spherical triangles is important to assure the possibility for geometric calculation on a sphere. Another major focus is on point localization, which handles an aircraft as a point and an ATC-Sector as polygon respectively polyhedron. In this context the software library CGAL has been used, because zhis library contains several point location algorithms. This Master thesis has been elaborated out during a internship at the "Deutsches Zentrum für Luft- und Raumfahrt (DLR) ", which will use the results of this thesis.

Contents

Chapter 1.

Motivation

This chapter explains the need for *efficient point location in a set of concave polyhedra*, which starts with a closer look on the DLR project FACTS.

1.1. FACTS

FACTS, the abbrevation for Future Aeronautical Communications Traffic Simulator, is a project of the German research center *Deutsches Zentrum fuer Luft- und Raumfahrt e. V. (DLR)* (DLR). The main purpose of the project, is to develop a realistic simulator to determine the arise of future aeronautical communication, which is concerned with air traffic management (ATM). The goals of the project are

- realistic implementation of the current and future air traffic in Europe, including Air Traffic Control (ATC) Sectors
- Implementation of all relevant ATM applications
- Implementation of protocols for ATM communication systems
- Modeling and implementing the physical transmission between aircraft and base stations

This simulation environment is based on the OMNeT++ simulation framework with a GUI based on NASA Worldwind Java (see section A.4). ATM applications are defined in COCR, which will be treated in the next section.

1.2. COCR

COCR is the abbreviation for *Communications Operating Concept and Requirements for the Future Radio System*. This document (EF06) is the result of a combined study of EUROCONTROL and the Federal Aviation Administration (FAA). This study has two goals:

- Identifying the communication services for future ATM
- Identifying the technology requirements to fullfill these guidelines.

These emerging communication services have been divided into two major categories:

- Air Traffic Services (ATS), which contains the major ATM services
- Aeronautical Operational Services (AOS), which is relevant for safety related purposes.

In the following the services of the first category services are described. The deployment and the kind of use of this ATS services depends on the domain, in which the aircraft are operating. Aircraft in different domains use also different ATS services. COCR has

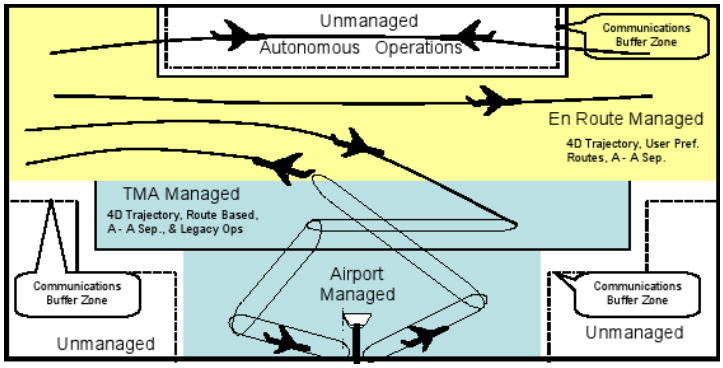


Figure 1.1.: ATS domains

identified 5 different domains (see figure 1.1):

- *Airport (APT):* 10 miles diameter around the airport and up to $\sim$ 5000 feet height
- *Terminal Maneuvering Area (TMA):* $\sim$ 100 miles diameter around the airport and starting from $\sim$ 5000 feet up to $\sim$ 24500 feet height
- *EnRoute (ENR):* Horizontal limit of 300 NM by 500 NM (Nautical mile) and starting from $\sim$ 24500 feet up to $\sim$ 60000 feet height.

- *Oceanic, Remote, Polar (ORP):* Areas outside domestic airspace with a horizontal limit of 1000 NM x 2000 NM.

- *Autonomous Operations Area (AOA):* Areas in which aircraft operate autonomous and have a horizontal limit of 400 NM x 800 NM

A closer look to the services of the ATC category helps to illustrate the relationship between COCR and this thesis: It can be seen in Figure 1.2, that the services are divided

Data Communications Management Services (DCM)	Clearance/ Instruction Services (CIS)	Flight Information Services (FIS)	Advisory Services (AVS)	Flight Position/ Intent/ Preferences Services (FPS)	Emergency Information Services (EIS)	Delegated-Separation Services (DSS)	Miscellaneous Services (MIS)
Data Link Logon (DLL)	ATC Clearance (ACL)	Data Link Automatic Terminal Information Service (D-ATIS)	Arrival Manager Information Delivery (ARMAND)	Surveillance (SURV)	Data Link Alert (D-ALERT)	In-Trail Procedures (ITP)	Air-to-Air Self Separation (AIRSEP)
ATC Communication Management (ACM)	Departure Clearance (DCL)	Data Link Operational Terminal Information Service (D-OTIS)	Dynamic Route Availability (DYNAV)	Flight Plan Consistency (FLIPCY)	Urgent Contact (URCO)	Merging and Spacing (M&S)	Auto Execute (A-EXEC)
	Downstream Clearance (DSC)	Data Link Operational En Route Information Service (D-ORIS)	Data Link Flight Update (D-FLUP)	Flight Path Intent (FLIPINT)		Crossing and Passing (C&P)	
	ATC Microphone Check (AMC)	Data Link Significant Meteorological Information (D-SIGMET)		System Access Parameters (SAP)		Paired Approach (PAIRAPP)	
	Data Link Taxi (D-TAXI)	Data Link Runway Visual Range (D-RVR)		Wake Broadcast (WAKE)			
	Common Trajectory Coordination (COTRAC)	Data Link Surface Information and Guidance (D-SIG)		Pilot Preferences Downlink (PPD)			
				Traffic Information Service-Broadcast (TIS-B)			

Figure 1.2.: ATS services

into eight subcategories with differen emphasis. For example, the DSS subcategory for example has the task to separate and merge aircraft in the pre-landing phase. Some services in this figure have been marked with a light yellow color and a dark red color. These services are triggered by the aircraft position. The yellow ones are active, when the aircraft is in a specific domain and the red ones should be called at least once in an ATC-Sector. So a major task for the simulation system is to locate aircraft and to trigger these service applications shown in Figure 1.2, based on the relationship of an aircraft to its current ATC-Sector.

1.3. Problem Statement

As emerged in the previous section, the main task in this thesis is to locate aircraft in a set of ATC-Sectors. The shape of the ATC-Sectors are presented in section 5.3 and they have a (concave) polyhedral form. The overall goal of this thesis can be explained as the mathematical problem of locating points in a set of concave polyhedra. Another major feature is to follow the flight route of an aircraft to minimize the number of point location queries (explained in detail in section 5.7).

Chapter 2.

Fundamentals

Inside this chapter the mathematical basics are explained, which are required for this thesis.

2.1. Polygons

Preperata and Shamos (PS85) gave a detailed definition of polygons in the 2-dimensional euclidean space. They first define a *point* p in the euclidean space to be a tuple of coordinates $p = (x_1, x_2, \cdots, x_d)$, whereas its length depends on the dimension d of the euclidean space E^d.
They define a *line* l in E^d as parametrized linear combination of 2 points p and q, having α as parameter:

$$l = \{p' : p' = \alpha p + (1 - \alpha)q \text{ with scalar } \alpha \in \mathbb{R}\} \tag{2.1}$$

As the length of lines is not limited, this requires the definition of *line segments* s, which means in particular adding a restriction for α:

$$s = \{p' : p' = \alpha p + (1 - \alpha)q \text{ with scalar } \alpha \in [0, 1]\} \tag{2.2}$$

In Figure 2.1 a line in a 3-dimensional euclidean space can be observed. The part of the line, that lies between the two points is equal to the corresponding line segment.

Now it is possible to define a polygon in the 2-dimensional euclidean space E^2. A *polygon* consists of a finite set of distinct line segments, with the condition that every line segment extreme (a segment extreme has α value of either 0 or 1, which denotes an endpoint) belongs to exactly two line segments. These two line segments are then called adjacent. It can easily be seen that a polygon must consist at least of three distinct line segments (PS85).

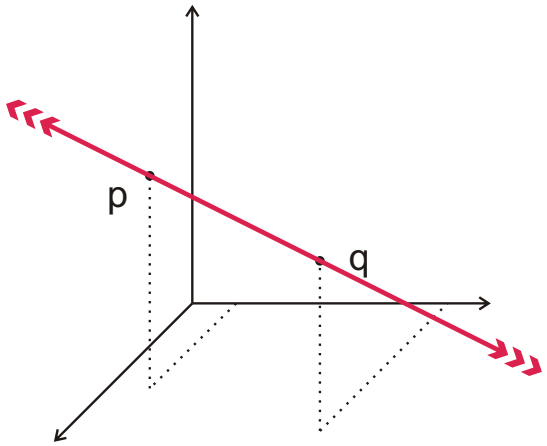


Figure 2.1.: Line in 3 dimensional euclidean space

2.2. Types of Polygons

simple polygon A polygon is called *simple* if it fulfills the following condition: For every two line segments there exists no common point they share, except it is an endpoint of both. So this means that the polygon doesn't intersect itself (PS85). In Figure

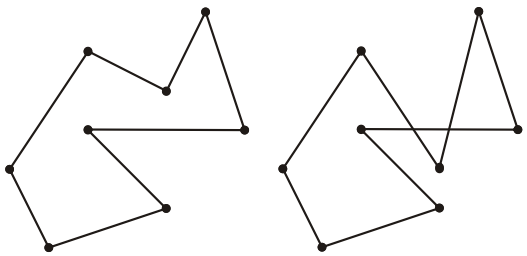


Figure 2.2.: simple(left) and non simple(right) polygon

2.2 two polygons are outlined. The left polygon is simple while the right is not, since there are intersections.

convex polygon A polygon is called *convex* if every point lying on a line segment, which can be generated by two arbitrary endpoints of the polygon, is inside the interior of the polygon. The interior of a polygon is determined by the jordans curve theorem (see section 4.1.1). If the polygon is not *convex* it is called *concave* (PS85). In Figure 2.3 two polygons can be seen. The left polygon is convex while the right is concave.

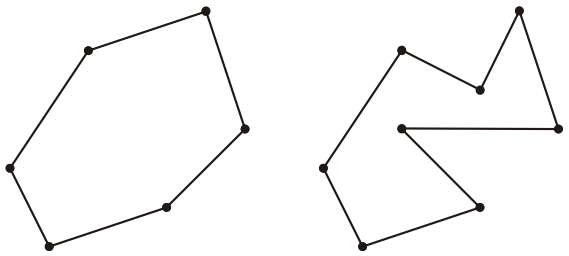


Figure 2.3.: convex and concave polygon

2.3. Spherical Polygon

Inside this section, the spherical polygon will be defined. Therefore spherical points, great circles and great circle segments have to be defined.

2.3.1. Spherical Points

A sphere S is defined inside the 3-dimensional euclidean space as a set of points which fulfills the following condition ((Fil93)):

$$S = \{p : |Mp| = r\} \tag{2.3}$$

where the 3-dimensional euclidean point M represents the midpoint of the sphere, r is an arbitrary scalar and the function $|ab|$ returns the distance between two points (namely a with coordinates (x_a, y_a, z_a) and b with coordinates (x_b, y_b, z_b)) of the euclidean space, which is given by :

$$|ab| = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2 + (z_a - z_b)^2} \tag{2.4}$$

A point of this sphere can now be expressed as 3-dimensional vector between the midpoint M and a point $p \in S$

$$\overrightarrow{v_p} = \overrightarrow{Mp} = \begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} - \begin{pmatrix} x_M \\ y_M \\ z_M \end{pmatrix} \tag{2.5}$$

A spherical point p can also be expressed in a spherical coordinate base ((Sig77)), which

is defined by the following bijective mapping m:

$$m : \mathbb{R}^3 \rightarrow [-\pi \cdots \pi] \times [-\frac{\pi}{2} \cdots \frac{\pi}{2}] \times \mathbb{R}_+ \tag{2.6}$$

$$\begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} \mapsto \begin{pmatrix} \lambda_p \\ \phi_p \\ r_p \end{pmatrix} \tag{2.7}$$

where λ_p, ϕ_p and r_p are given by:

$$\lambda_p = \begin{cases} \quad arccos\frac{x_p}{\sqrt{x_p^2+y_p^2}} \text{ if } y_p >= 0 \\ - \quad arccos\frac{x_p}{\sqrt{x_p^2+y_p^2}} \text{ if } y_p < 0 \end{cases} \tag{2.8}$$

$$\phi_p = arctan\frac{z_p}{\sqrt{x_p^2+y_p^2}} \tag{2.9}$$

$$r_p = \sqrt{x_p^2+y_p^2+z_p^2} \text{ which is the norm of the Cartesian vector} \tag{2.10}$$

Note that all those spherical vectors whose points belong to a specific sphere S have the same r_p value if the midpoint M is the origin $M = (0,0,0)^T$. Then λ_p and ϕ_p are also known as longitude and latitude, if they have been transformed in degree representation. With the reverse mapping the original point representation can be restored:

$$x_p = rcos\phi_p cos\lambda_p \tag{2.11}$$

$$y_p = rcos\phi_p sin\lambda_p \tag{2.12}$$

$$z_p = r_p sin\phi_p \tag{2.13}$$

Each spherical point p has a dual spherical point p_d, which also lies on the sphere S ((Sig77)).

$$p_d = M - \overrightarrow{v_p} \tag{2.14}$$

2.3.2. Great Circles

A great circle is the corresponding construct to a line in the euclidean space. The set of points belonging to a greatcircle GC is defined by the intersection of the set of points belonging to a sphere S and the set of points belonging to a plane PL in the 3-dimensional euclidean space, which contains the midpoint of the sphere ((Fil93)). A plane in the 3-dimensional euclidean space is defined by one point (e.g. sphere midpoint M) with two linear independent vectors ($\overrightarrow{v_p}$ and $\overrightarrow{v_q}$):

$$PL = \{p : p = M + \mu\overrightarrow{v_p} + \nu\overrightarrow{v_q} \text{ with scalars } \mu \text{ and } \nu \in \mathbb{R}\} \tag{2.15}$$

Now the points belonging to a great circle GC can be defined:

$$GC = \{p : (p \in S) \wedge (p \in PL)\} \tag{2.16}$$

Those two vectors, which have been used for the definition of the great circle, can also be derived from two spherical points p and q with:

$$\overrightarrow{v_p} = \begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} - \begin{pmatrix} x_M \\ y_M \\ z_M \end{pmatrix} \tag{2.17}$$

$$\overrightarrow{v_q} = \begin{pmatrix} x_q \\ y_q \\ z_q \end{pmatrix} - \begin{pmatrix} x_M \\ y_M \\ z_M \end{pmatrix} \tag{2.18}$$

Each great circle has two corresponding polar points, which also lie on the sphere. Those are defined by the intersection of the points defined by the sphere and the 3-dimensional line LO in the euclidean space, which is orthogonal to the plane PL and contains the midpoint of the sphere M. This line is given by one point (e.g. sphere midpoint M) and a vector, which is orthogonal to the two vectors, which have defined the plane PL

$$LO = \{p : p = M + \mu(\overrightarrow{v_p} \times \overrightarrow{v_q}) \text{ with scalar } \mu \in \mathbb{R}\} \tag{2.19}$$

And so the two polar points PP are:

$$PP = \{p : (p \in S) \wedge (p \in LO)\} \tag{2.20}$$

One polar point is always the dual point of the other polar point.

Two distinct great circles intersect each other always in two dual points ((Sig77)). These two points are given by the intersection of the sphere s with the 3-dimensional line LS, which is the intersection line given by the intersection of the two planes on whom the great circles lie (see Figure 2.4). The intersection line LS for two planes PL_1 (with vectors $\overrightarrow{v_{p_1}}$ and $\overrightarrow{v_{q_1}}$) and PL_2 (with vectors $\overrightarrow{v_{p_2}}$ and $\overrightarrow{v_{q_2}}$) is then:

$$\overrightarrow{v_{n_1}} = \overrightarrow{v_{p_1}} \times \overrightarrow{v_{q_1}} \tag{2.21}$$

$$\overrightarrow{v_{n_2}} = \overrightarrow{v_{p_2}} \times \overrightarrow{v_{q_2}} \tag{2.22}$$

$$\overrightarrow{v_s} = \overrightarrow{v_{n_1}} \times \overrightarrow{v_{n_2}} \tag{2.23}$$

$$LS = \{p : p = M + \mu\overrightarrow{v_s} \text{ with scalar } \mu \in \mathbb{R}\} \tag{2.24}$$

Now the two intersection points PI are

$$PI = \{p : (p \in S) \wedge (p \in LS)\} \tag{2.25}$$

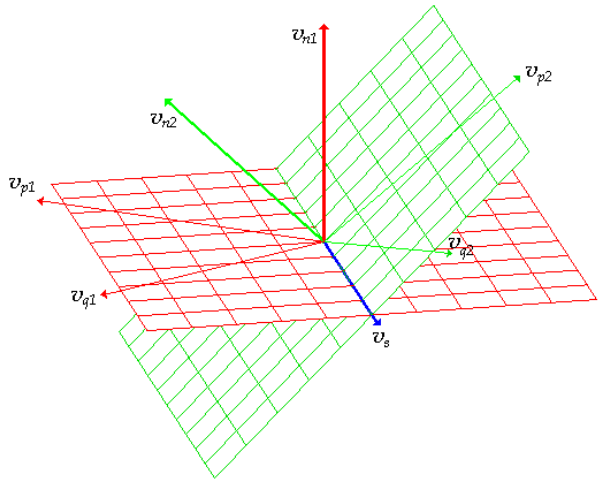


Figure 2.4.: Intersection of two planes

2.3.3. Great Circle Segments

A great circle segment is the corresponding construct to a line segment in the euclidean space. It is a subset of a greatcircle, which is bounded by two spherical points p and q. Usually these two spherical points can be used to construct the great circle itself, if those two points together with the midpoint M are not collinear (if p isn't the dual point of q). To define such a great circle segment, the angle $c_{p,q}$, which lies between the two vectors v_p and v_q, must at first be defined which will be given by the spherical points p and q ($c_{p,q} = \angle pMq < \pi$). Using the inner vector product, the angle $c_{p,q}$ is given by:

$$c_{p,q} = arccos(\frac{\langle v_p, v_q \rangle}{|v_p| \cdot |v_q|}) \tag{2.26}$$

Next the greatcircle segment is defined as parametrized curve. The points along this great circle segment can be retrieved, if a rotation in the 3-dimensional euclidean space along the circle defined by the sphere S and the plane PL will be performed. Therefore the rotation axis has to be determined. The corresponding rotation axis vector $\overrightarrow{v_R}$ will be determined by the two spherical points p and q (and the midpoint M), which define the great circle containing this segment.

$$\overrightarrow{v_R} = \frac{\overrightarrow{v_p} \times \overrightarrow{v_q}}{|\overrightarrow{v_p} \times \overrightarrow{v_q}|} \tag{2.27}$$

Now the rotation of the point p with rotation angle α around the rotation vector $\overrightarrow{v_R}$ is defined by ((Koe83))

$$R_{\alpha,\overrightarrow{v_R}}p = cos\alpha \cdot \overrightarrow{v_p} + sin\alpha \cdot (\overrightarrow{v_R} \times \overrightarrow{v_p}) + M \tag{2.28}$$

Therefore a great circle segment can be defined as parametrized curve:

$$GCS = \{p' : p' = R_{c_{p,q}t,\overrightarrow{v_R}}p \text{ with scalar } t \in [0,1]\} \tag{2.29}$$

Additionally the angle $c_{p,q}$ between the two points p and q of a great circle segment can be computed using spherical trigonometry (Sig77). The sphere midpoint M must be at the origin $M = (0,0,0)^T$ for this consideration. To deal with spherical trigonometry, 2 simple cases have to be considered. If the two points p and q of the great circle segment have the same longitude coordinate $\lambda_p = \lambda_q$, or one of the points is either the north pole ($\phi_x = \frac{\pi}{2}$) or the south pole ($\phi_x = -\frac{\pi}{2}$) then the angle between those two points is given by the difference of their latitude coordinates:

$$c_{p,q} = |\phi_p - \phi_q| \tag{2.30}$$

Now the angle $c_{p,q}$ of an arbitrary great circle segment can be calculated by constructing a spherical triangle, with the endpoints of the great circle segment p and q and the north pole n as corners. The sides of the spherical triangle are great circle segments.

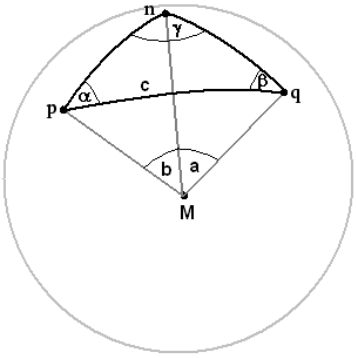


Figure 2.5.: Spherical triangle

It can be seen in Figure 2.5, that the spherical triangle consists of 6 relevant angles. Three of them are angles on the surface of the sphere, namely α, β and γ. The other three angles are those constructed by two points out of the three corner points and the midpoint M of the sphere, namely a (by $\angle nMq$), b (by $\angle pMn$) and c (by $\angle pMq$, which is equal to $c_{p,q}$). Three of this angles can be calculated, if the latitude and longitude coordinates of the two endpoints of the great circle segment are given:

$$a = \phi_n - \phi_q = \frac{\pi}{2} - \phi_q \tag{2.31}$$

$$b = \phi_n - \phi_p = \frac{\pi}{2} - \phi_p \tag{2.32}$$

$$\gamma = |\lambda_q - \lambda_p| \tag{2.33}$$

The other three can be obtained using the spherical trigonometry Neper equations (Sig77):

$$\frac{\alpha+\beta}{2} = tan^{-1}\left(cot\frac{\gamma}{2}\cdot\frac{cos\frac{a-b}{2}}{cos\frac{a+b}{2}}\right) \tag{2.34}$$

$$\frac{\alpha-\beta}{2} = tan^{-1}\left(cot\frac{\gamma}{2}\cdot\frac{sin\frac{a-b}{2}}{sin\frac{a+b}{2}}\right) \tag{2.35}$$

$$c = 2tan^{-1}\left(tan\frac{a+b}{2}\cdot\frac{cos\frac{\alpha+\beta}{2}}{cos\frac{\alpha-\beta}{2}}\right) \tag{2.36}$$

Two great circle segments can intersect each other at almost one point, if their corresponding great circles are distinct. As the intersection of two great circles delivers two intersection points, only one of them can be inside the set of a greatcircle segment. The reason for this is that all points p' belonging to the great circle segment have an angle $\angle pMp'$ between them and the first endpoint p, which is $\angle pMp' < \pi$ (guaranteed by the definition of the great circle segment as rotation in the 3-dimensional space), but the angle between two dual points q and q_d is always π.

2.3.4. Spherical Polygon Definition

A spherical polygon will be equally defined as polygons with line segments. It consists of a finite set of distinct great circle segments with the condition that every great circle segment extreme (a segment extreme has t value of either 0 or 1, which denotes an endpoint) belongs to exactly two great circle segments. This two great circle segments are then called adjacent (Hec94).

2.3.5. Spherical Points in the Plane

As the point location algorithms presented in chapter 4, are only applicable to polygons defined in the 2-dimensional euclidean space, spherical points must be treated as points in the plane. For this purpose, the following mapping for each spherical point $p \in S$ with sphere distance r and midpoint $M = (0, 0, 0)^T$ is applied:

$$[-\pi \cdots \pi] \times [-\frac{\pi}{2} \cdots \frac{\pi}{2}] \times \mathbb{R}_+ \to \mathbb{R}^2 \tag{2.37}$$

$$\begin{pmatrix} \lambda_p \\ \phi_p \\ r_p \end{pmatrix} \mapsto \begin{pmatrix} x'_p \\ y'_p \end{pmatrix} \tag{2.38}$$

where x'_p, y'_p are given by:

$$x'_p = \begin{cases} \pi \text{ if } \phi_p = \frac{\pi}{2} \\ -\pi \text{ if } \phi_p = -\frac{\pi}{2} \\ \lambda_p \text{ otherwise} \end{cases} \tag{2.39}$$

$$y'_p = \phi_p \tag{2.40}$$

In chapter 6, the use of these transformed spherical point belonging to great circle segments is shown.

Chapter 3.

Subdivision of Faces

Inside these chapter the term *Face* will be defined. Additionally two data structures are represented, which are able to store these faces.

3.1. Face Definition

A face is defined as a region inside a 2-dimensional euclidean space. A face can either be bounded or unbounded. If the face is bounded, the boundary is described by the line segments of a simple polygon. Points belongto a face, if they are located inside this polygon. The inside and the outside area of a simple polygon is well defined by Jordans curve theorem (see chapter 4.1.1). Each face can consist of several holes, where points inside the holes don't belong to the face. Holes can be described as a set of distinct simple polygons. Two simple polygons are distinct, if there exist no point which lies inside the interior of both polygons and no line segment of one hole polygon contains a point, which also lies on a line segment of another hole polygon ((dBvKOO00) pages 29-33). An example of a face, which is given by a border polygon and two hole polygons,

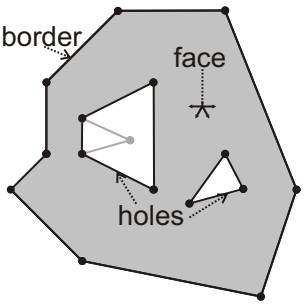


Figure 3.1.: Example of a face defined by a border polygon and two hole polygons

is pictured in Figure 3.1. Thereby a hole itselfcan consist of several faces, as indicated in the left hole in Figure 3.1. A point belongs now to the face, if it is in the gray area.

3.2. Representation of Faces

The main purpose inside this section is to find a data structure which is able to represent a subdivision of faces. A subdivision of faces consists of several faces, which may be adjacent among themselves at their hole and border polygons. The main restriction for this set of faces is, that they aren't allowed to intersect each other. A consequence of this restriction is, that no point of the underlying geometric space is in the interior region of more than one face. Hence there is at most one face, which is unbounded.

3.2.1. Independent Faces Representation

The simplest data structure to store several faces is the independent faces representation. The faces (this representation allows no holes) will be stored in a table. Each face itself consists of a list with coordinate tuples. The size of a tuple is n. where eac tuple defines a point in the underlying n dimensional euclidean space(e.g. the tuples have size 2 in the 2 dimensional euclidean space). These tuples represent the endpoints of the line segments, which are part of a simple border polygon. These tuples are ordered in such a way, that each two consecutive tuples form a line segment of the simple polygon, which bounds the face. The last and the first tuples will also be seen as consecutive and will therefore define a line segment too. So given three consecutive tuples, it is easy to determine, that the both line segments, which are defined by this three coordinate tuples are adjacent, because they share a common endpoint, which is given by the second coordinate tuple. Figure 3.2 gives an example for a data structure in the 2 dimensional space. On the

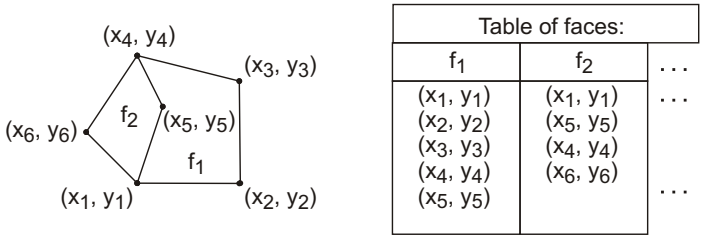


Figure 3.2.: Independent faces representation in 2 dimensional euclidean space

left side in Figure 3.2, two faces have been defined (labeled with f_1 and f_2). Each of these faces consists of several endpoints, which are described by the coordinate tuples

(x_i, y_i). On the right side, the face table of this polygon subdivision has been visualized and in which each face consists of a list of coordinate tuples. It is easy to see, that some coordinate tuples of the face table have been inserted twice (e.g. (x_1, y_1)). So there are several different tuple objects for a single point, which is somehow a redundancy.

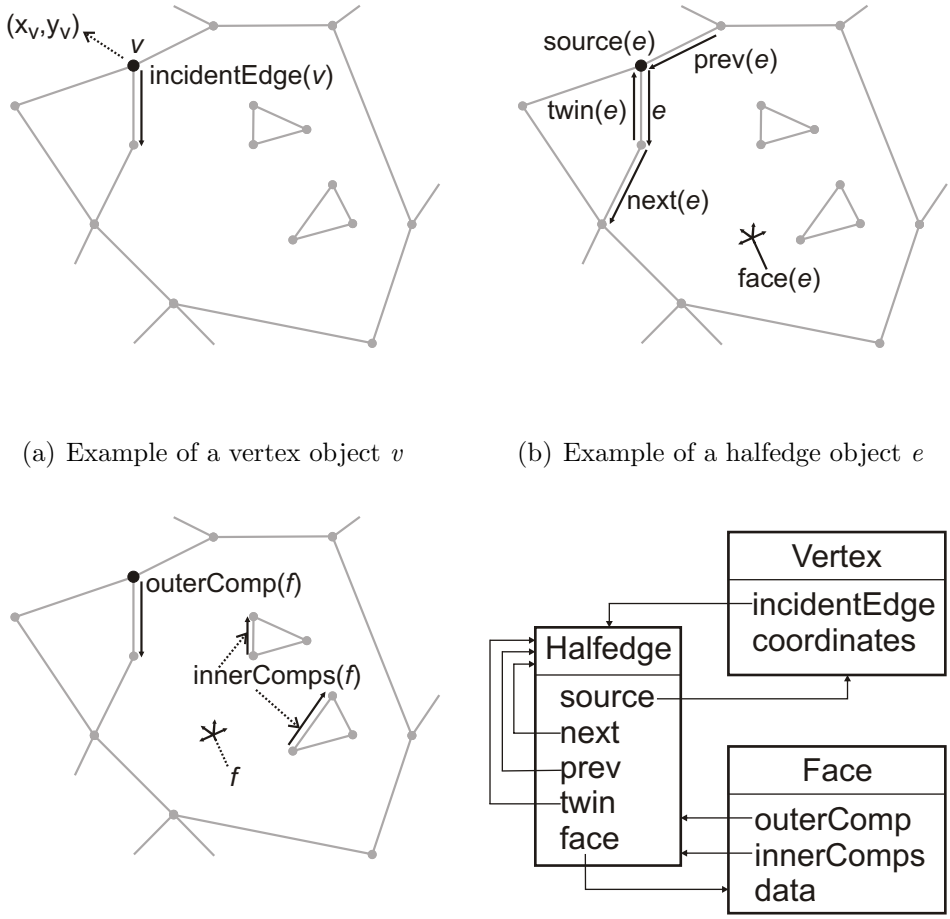
3.2.2. Doubly-Connected Edge List (DCEL)

The face table described in the last section contains rendundancy because coordinate tuples of points are inserted several times in the data structure. Other very important restrictions are that this data structure is not able to handle holes and topological informations. So the next step is, to integrate more topological information into the subdivision of face representation. For that reason the whole subdivision is considered as a graph, or more precisely as a directed planar graph (Tru93). The subdivision of faces is then also called a planar subdivision. The planar subdivision consists of vertices, which are equivalent to the nodes of the planar graph and the endpoints of the polygon, the halfedges which are equivalent to the directed arcs of the directed planar graph and the line segments of the polygon, and the faces which are the areas bounded by the arcs of the directed planar graph. If the directed planar graph is not connected and one subgraph is inside an area, then this subgraph is equivalent to a hole. As a subgraph, which is inside an area of the planar graph, is isomorphic to the situation where the subgraph lies outside this area, geometric properties must be added to prevent this.

The three different elements of a planar subdivision are described below:

Vertex A vertex object inherits a reference to its corresponding *coordinate tuple*. This reference is the only connection to the geometric properties of the whole structure. Additionally to its coordinate, the vertex has a reference named *incidentEdge* to one of the halfedges, whose *source* is identical with the actual vertex object. (see Figure 3.3(a))

Halfedge An edge between two vertices is given by two halfedges, which have an opposite direction. So the source (origin vertex of the arc arrow) of the first halfedge is equal to the destination (vertex, which the arc arrow points to) of the second halfedge and vice versa. To get from a specific halfedge to its corresponding opposite directed halfedge, each halfedge has a reference named *twin*. To connect the halfedge with its vertices, each halfedge object contains a reference to its *source* vertex, but not a reference to its destination. To get the destination of a specific halfedge, the source of the twin halfedge must be taken. As each edge borders two faces, it is possible to connect each halfedge of the edge to exactly one face. The two halfedges are ordered in the following way, that if one looks from the source to the destination of a halfedge, its corresponding *face* lies always on the left side of it. Furthermore the halfedge has two references to its *next* halfedge and its



(a) Example of a vertex object v

(b) Example of a halfedge object e

(c) Example of a face object f

(d) class diagram

Figure 3.3.: Vertex, Halfedge and Face examples and their class diagram

previous halfedge (*prev*). The next halfedge of a specific halfedge, must fullfill the following two properties (similar for the prev halfedge):

- The source vertex of the next halfedge is equal to the destination vertex of the actual halfedge.
- Both halfedges points to the same face object.

An example for a halfedge object is shown in Figure 3.3(b)

Face Each face is defined by one or zero border polygons on the outside and several hole polygons in its interior. To represent the whole polygons it is sufficient to just embed a reference to halfedges, which belong to a line segment of that specific polygon. The reference halfedges itself must point via its face reference to the actual face object. So each face object has one reference to a halfedge object belonging to the border polygon named *outerComp*, which stands for outer component and a set of references to halfedges corresponding to the hole polygons named *innerComps*, which stands for inner components. To traverse for example through the border polygon of a given face, one must first jump to a halfedge of the border polygon via the *outerComp* reference and follows the *next* references of the halfedges. It can be seen that the vertices of the border polygon are traversed in counterclockwise order, while hole polygons are traversed in clockwise order. Additionally a face can also have a reference to a *data* object, which corresponds to the face (This data reference has later been used to connect ATC-Sectors, which are in principle equivalent to faces, with its ATC-Sector information; see chapter 5.3). An example of a face object is given in Figure 3.3(c).

All these three types of objects are stored in three different sets. Additionally they are connected among themselves via the references just mentioned. An overview of the composition of these three object types (class information) is given in the class diagram 3.3(d). This definition of the DCEL data structure has been extracted from (dBvKOO00) pages 29-33.

Chapter 4.

Point Location

Inside this chapter algorithms will be presented to perform a point location in 2-dimensional euclidean space. Additionally some other algorithms (e.g. the map overlay algorithm) will be explained that yield a higher performance.

4.1. Point in Polygon

The main purpose of this section is to present two algorithms (sections 4.1.2 and 4.1.3), which are able to determine whether a point lies inside a simple concave polygon or not. This assumes that a specific simple polygon divides the underlying 2-dimensional space into two regions, which will be called the interior and the exterior region. The Jordans curve theorem shows that this is a valid assumption for the 2-dimensional euclidean space.

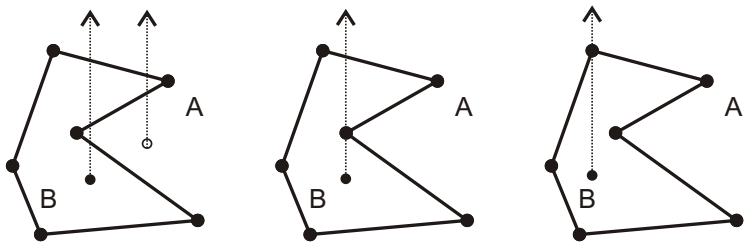
4.1.1. Jordans Curve Theorem

The Jordans curve theorem states:

> A simple closed curve C in the plane divides the plane into exactly two domains, an inside and an outside. (CRS96)

This theorem has been first proved correctly by Oswald Veblen (Veb05). Proving this theorem wasn't a obvious task as the definition of a simple closed curve is rather general. Another proof of the Jordans curve theorem for simple polygons P consisting of line segments can be found in (CRS96). The main idea of it will now be explained. The first thing to do is to choose a specific direction d, which isn't parallel to any line segment belonging to the polygon P. The next step is to divide the points of the plane into

two sets A and B, where a point p belongs to the set A, if a ray starting from point p with direction d would intersect the polygon P an even number of times. Otherwise if the ray starting from point p intersects the polygon an odd number of times, then p belongs to B (see Figure 4.1(a)). If the ray from a point p goes through a vertex point



(a) Defining sets A and B (b) No intersection at point lying on ray (c) Intersection at point lying on ray

Figure 4.1.: Jordans curve theorem

p_v of the polygon, then this would be counted as intersection, if and only if the two line segments of the polygon containing this point p_v lie on different sides of the ray (see Figures 4.1(b) and 4.1(c)). As the two sets A and B have now been defined, the following two properties must be proven seperately in order to prove the Jordans Curve Theorem completely:

- If any point $p_1 \in A$ is connected via a polygon path with a point $p_2 \in B$, then this polygon path must intersect P.
- Any two points belonging to the same set can be connected by a polygon path, which doesn't intersect P

The proof of these two statements will be omitted (see (CRS96) for details). The two sets A and B are now called the exterior and the interior of a polygon P.

4.1.2. Ray Crossing Method

The ray crossing method uses in principle the findings of the proof of Jordans curve theorem for polygons. A query point p is inside a polygon P, if a ray starting from this point with direction to a point definitely outside of the polygon would intersect the polygon an odd number of times. Usually the direction of the ray goes along an axis, consequently the computation can be speed up. The query point will be considered as the origin of the 2-dimensional euclidean space. The polygon P is given by a number of adjacent points. The direction of the ray points towards infinity on the positive y axis. Now the algorithm would traverse through all points of the polygon, until the sign of

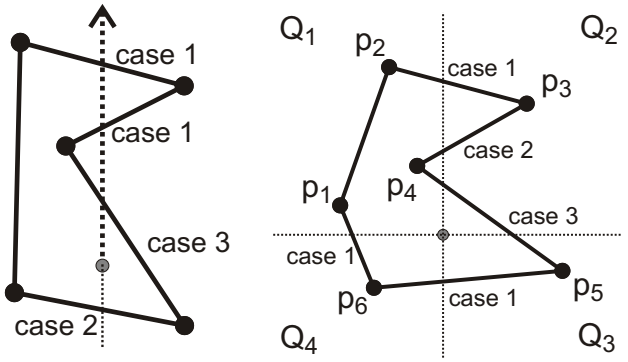
the x coordinate of the actual point differs from the sign of the x coordinate of the last point. If that case the y coordinate of both points will be considered. Three cases are possible (see Figure 4.2(a)):

1. If both y coordinates are positive, then an intersection has occurred (increment the intersection counter by 1).

2. If the two coordinates are negative, then no intersection has occurred.

3. Otherwise the intersection point between the line segment formed by those two points and the ray must be evaluated. If this intersection point lies on the positive y axis then an intersection has occurred (increment the intersection counter by 1).

After considering all line segments of the polygon, it will be clear whether the query point lies inside the polygon (intersection counter is odd) or not (intersection counter is even). This algorithm sketch has been retrieved from (Hec94) pages 24-46.



(a) 3 cases of the ray crossing method

(b) 3 cases of the quadrant method

Figure 4.2.: Examples for the ray crossing method and the quadrant method

4.1.3. Quadrant Method

Another method to check if a given query point p_q lies inside a polygon is, to sum up all angles $\angle p_i p_q p_{i+1}$ where p_i and p_{i+1} are the endpoints belonging to the i'th line segment of the polygon. If this sum isn't zero, then the point lies inside the polygon. Note that the polygon P must be given by a number of adjacent points. As the determination of such angles is a very costly operation, a simpler method can be applied, which is called the quadrant method. Therefore the query point p_q is considered again as the origin of the 2-dimensional space. Then the 2 dimensional space is divided into 4 quadrants, which are separated by the two axes (x-axis and y-axis). Next a *counter* variable isintroduced

which is initialized with 0. The quadrant algorithm would now traverse through all points of the polygon. Each time the actual point is in a different quadrant then the previous point, the *counter* variable will be modified. There are three different cases (see Figure 4.2(b)):

1. If the quadrants of the two points are adjacent along an axis and the line segment to the actual point crosses this axis clockwise, then the *counter* is increased by 1.

2. If the quadrants of the two points are adjacent along an axis and the line segment to the actual point crosses this axis counter-clockwise, then the *counter* is decreased by 1.

3. If the quadrants of the two points are diagonal, then at first the third quadrant through which the line segments traverses has to be determined. If this third quadrant is in clockwise order related to the quadrant of the previous point, then the *counter* is increased by 2, otherwise it is decreased by 2.

After considering all line segments of the polygon, the query point lies inside the polygon, if and only if the *counter* variable is not 0. This algorithm would only work for polygons with line segments. This algorithm sketch has been retrieved from (Hec94) pages 24-46.

4.2. Point Location Algorithms

The main purpose of this section is to present algorithms, which are able to perform a point location on a subdivision of faces defined in the 2-dimensional euclidean space (see chapter 3). For each query point the face which contains this point will be determined.

4.2.1. Walk along a Line Point Location

The main idea of this algorithm is equal to the idea used in the ray crossing method (see chapter 4.1.2). Initially a ray, directed from query point towards infinity on the positive y-axis, is introduced. For this algorithm, a subdivision of faces data structure is required, which can handle holes and can handle an unbounded face. Additionally it would highly improve the speed of the algorithm, if the data structure is able to switch quickly between adjacent faces (for example the DCEL data structure; see chapter 3.2.2). At the beginning of the algorithm the unbounded face is considered as actual face. Now the main issue of the following step is to retrieve the corresponding face, in which the query point lies:

1. In the first step it is checked if the query point lies inside the outer boundary polygon of the actual face. If the actual face is the unbounded face, then this is obviously true (continue with step 2). Otherwise the same strategy as with the ray crossing method is used. In detail this means, the number of intersections between the line segments of the outer boundary and the ray is determined. If this number is odd, then the query point lies inside the outer boundary polygon of the actual face and it will be continued with step 2 of the algorithm. If the number is even, then the query point doesn't lie inside the outer border polygon. Therefore the intersection point with the smallest y value is considered. Now the face, which is adjacent to the actual face along the line segment, on which this intersection point lies, will be focussed. This face is now the new actual face and the algorithm continues with step 1.

2. At this step of the algorithm, it is clear that the query point lies inside the outer boundary polygon of the actual face. Next it will be tested whether the query point lies inside any hole of the actual face. Therefore it will be determined for each hole polygon the number of intersection with the ray. If this number is odd for a specific hole, then the algorithm has to switch to the adjacent face along the line segment of the corresponding hole polygon, on which the intersection point with the smallest y value lies. This adjacent face will now be treated as actual face and it will be proceeded with step 1 of the algorithm. If the query point doesn't lie inside any hole polygon, then the correct face has been determined and the algorithm terminates.

The sketch of this algorithm has been retrieved from the CGAL library (CGA07). The original algorithm, which requires that all polygons are triangles, has been worked out by Devillers, Pion and Teillaud ((DPT01)). The complexity of the query time of the original algorithm is in average $\mathcal{O}(\sqrt{n})$, where n denotes the number of line segments involved.

4.2.2. Landmarks Point Location

The landmarks point location algorithm presented here is based on the algorithm described in (HH06). This algorithm uses the DCEL data structure. Before performing the point location queries, the algorithm computes the position of $\#p$ points, where position means the corresponding face, halfedge or vertex object of a specific point. These points will be called landmarks. After this step has been performed, these points together with their corresponding topology object will be stored into a data structure, which is able to quickly determine the landmark, which is nearest to a specific query point. $\#p$ has usually the same magnitude as $\#n$, which denotes the number of line segments inside the subdivision. Now a point location query performs the following two steps to determine the face, which contains the query point:

1. In the first step, the nearest landmark compared to the query point will be determined.

There are various possibilities of choosing this landmark points. Two of them are described in the following in more detail.

- One possibility is to consider all points of the subdivision as landmarks, as their corresponding topology objects are already known. A 2-dimensional kd-tree data structure ((dBvKOO00) pages 99-105) will be used to store this landmarks, to provide a fast access to them, which is required by the 1. step of the query algorithm. A 2-dimensional kd-tree with p points allows the determination of the nearest landmark in $\mathcal{O}(log(p))$ time and requires for its construction $\mathcal{O}(p)$ space and $\mathcal{O}(plog(p))$ preprocessing time.

- Another possibility is to use a grid of points ($\lceil\sqrt{p}\rceil$ different x coordinates $\times$ $\lceil\sqrt{p}\rceil$ different y coordinates) as landmarks. The next step is to consider the position of these p points. This will be achieved by using a sweep line algorithm, which is similar to the algorithm presented in chapter 4.3.1, using the line segments of the DCEL data structure (Note that an intersection test needs not to be performed). For this purpose the points of the grid will be treated as additional event points of the event queue. Each time the sweep line algorithm must handle a landmark event point, the line segment directly above this point in the status set will be determined. This line segment is now connected, via its corresponding halfedge object, to the adjacent face, which contains the landmark point. This operation of determining the corresponding faces for each landmark point has a complexity of $\mathcal{O}((p+n)log(n))$. This face objects can now be stored in a 2-dimensional array, which requires $\mathcal{O}(p)$ space. The benefit of choosing the landmarks in that way, is that the determination of the nearest landmark for a given query point can be performed in $\mathcal{O}(1)$ time.

4.2.3. Randomized Incremental Point Location

This algorithm has been developed by Mulmuley and Seidel ((Sei91)) and was presented in the book (dBvKOO00) pages 122-144. The main idea behind it is to transform a given subdivision of faces into a subdivision of faces with trapezoidal shape. This is also called the trapezoidal map. To achieve this two vertical line segments are added to each endpoint of the original subdivision, one above and one below the point. The length of these line segments is limited by the above and below line segments respectively. The main idea, which is known as trapezoidal decomposition, is shown in Figure 4.3. So each face in the original subdivision consists now of several faces with trapezoidal shape in the trapezoidal map. This means in a mathematical sense, that there exists a surjective mapping from the faces in the trapezoidal map to the faces of the original

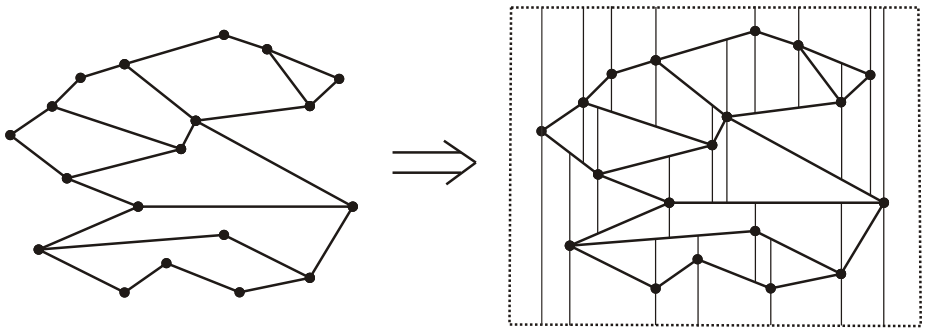


Figure 4.3.: trapezoidal decomposition

subdivision. While building the trapezoidal map, a search graph is constructed too, which is a directed acyclic graph with exactly one root node. To perform a point location query it is sufficient to traverse through the search graph (starting from the root node) until reaching one of its leaves. For each leaf in the search graph, there exists a bijective mapping to a face with trapezoidal shape in the trapezoidal map and so the original face is determined by the surjective mapping mentioned before. In more detail, the search graph consists of three types of nodes:

Point nodes Point nodes are always interior nodes and have an output degree of 2. For each such node exists a bijective mapping to the corresponding point in the original subdivision of faces. Their main purpose is to decide whether a given query point lies on the left(left child) or on the right(right child) side of the point corresponding to the point node.

Segment nodes Segment nodes are always interior nodes and have an output degree of 2. For each such node exists a bijective mapping to the line segments of the trapezoidal map and therefore a surjective mapping to the line segment of the original subdivision of faces. All the segment nodes which points to the same line segment object of the original subdivision, represent disjunctive parts (the line segments in the trapezoidal map) of this line segment. The main purpose of the segment nodes is to find out, whether a given point, which must be inside the x domain of the corresponding line segment, lies above (left child) or below (right child) the segment.

Face nodes Face nodes are always leaf nodes. They have a bijective mapping to the trapezoidal shaped face of the trapezoidal map.

Now the two components (trapezoidal map and search graph), which will be needed by this point location algorithm, are described. Next the construction of these two components is regarded. The first thing to do is to border the face of the trapezoidal map. For this purpose a rectangular boundary polygon, which describes the first trapezoidal

polygon in the trapezoidal map (dashes boundary in Figure 4.3), will be introduced. The border polygon, if inserted into the original subdivision, would contain all faces in its interior. As there exists now a face in the trapezoidal map, a face node needs to be added to the empty search graph. In Figure 4.4, the result of this initial work can be observed.

(a) Trapezoidal map (b) Search graph

Figure 4.4.: Initial state of the trapezoidal map and the corresponding search graph.

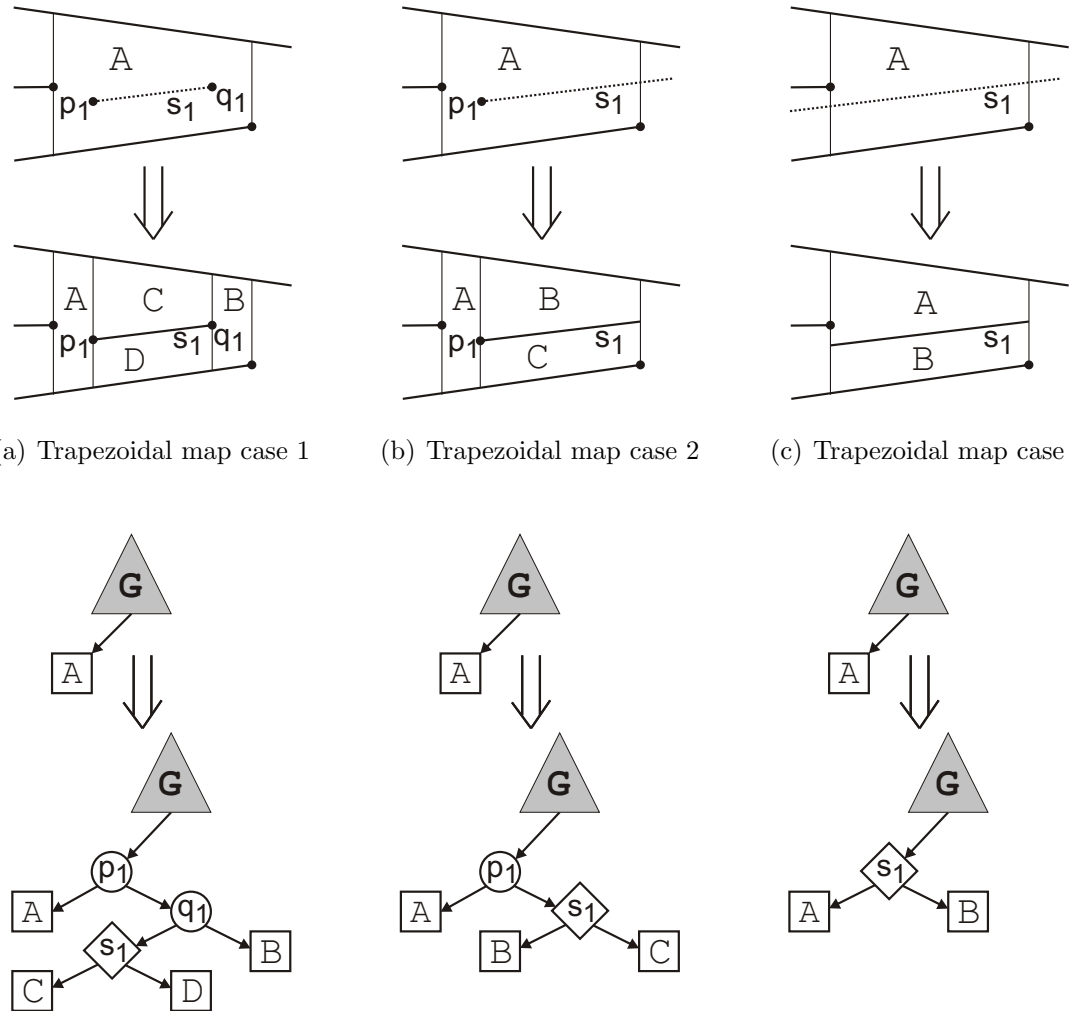
To build both data structures, all line segments of the original subdivision will be inserted separately and in randomized order. After each insertion of a single line segment, the algorithm prepares both data structures in a way that efficient point locations can be performed for the actual trapezoidal map. If a new line segment is added, it must be done in the following way:

1. Perform a point location for the left endpoint of the new line segment, to get the corresponding face.

2. Determine all faces, which contain parts of the new line segment, starting from the face, which has been determined in the previous state. These faces are horizontal adjacent (each face has at most 4 horizontal neighbors, if all points of the trapezoidal map have different x-coordinates).

3. Split each face into several faces, depending on the kind, how the line segment lies in the face. As the face will be splitted up, the corresponding face node must be removed from the search graph. There are three different cases, which can occur:

   - The whole line segment lies in a face. At first two new point nodes are generated, which correspond to the endpoints, and add them to the search graph (first the left endpoint and then the right endpoint as right child of the left endpoint). They will be added at the position of the deleted face node. Then vertical line segments for each endpoint must be added until reaching the upper / lower line segment. This generates a face to the left of the left endpoint and a face to the right of the right endpoint (setting left



(a) Trapezoidal map case 1 (b) Trapezoidal map case 2 (c) Trapezoidal map case 3

(d) Search Graph case 1 (e) Search Graph case 2 (f) Search Graph case 3

Figure 4.5.: Different cases

face node as left child of the left endpoint). Between the two endpoints lies the line segment which divides this area into an upper and lower face. The corresponding line segment node will be set as left child of the right endpoint and the two face nodes will be set as child nodes of the line segment node (see Figures 4.5(a) and 4.5(d)).

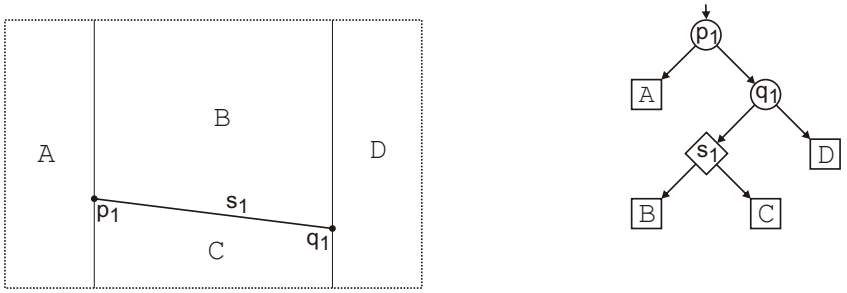
- Only one endpoint lies in the actual face. If it will be assumed that the left endpoint(similar for the right endpoint) lies in the face, then the corresponding point node have to be added to the graph at the position of the deleted face. Next, the vertical line segments must be added to the left endpoint which generates a new face at the left of it (setting face node as left child of point node). The right area will be divided into an upper and lower face, so

the line segment node must be added at first as right child of the left endpoint and the two new face nodes are the child's of this line segment node (see Figures 4.5(b) and 4.5(e)).

- No endpoint is inside the actual face. Therefore the corresponding line segment node has to be inserted at the position of the deleted face. The two new face nodes are added as child nodes (see Figures 4.5(c) and 4.5(f)).

4. As the vertical lines aren't allowed to intersect the new line segment, they have to be trimmed toward its original endpoint, if they would intersect the new line segment. As a result some faces will be merged to a single face. In the following example 4.7, it can be observed that the face node G represents a single face, altough there has been a vertical line segment generated by p_1 through it before (see Figure 4.6). If this line segment hasn't been trimmed, the vertical line segment would have intersected s_2.

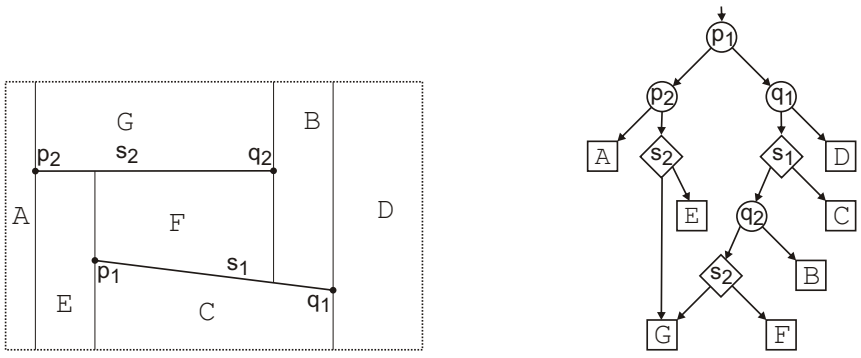
Next, a complete line segment insertion example will be explained, to provide a better understanding for functionality of this algorithm. Given a trapezoidal map and a search graph with one line segment inside (Figure 4.6), an additional line segment is added, which leads to the situation shown in Figure 4.7. In order to build the search graph for faces A and B, rule 2 has to be applied.



(a) Trapezoidal map (b) Search graph

Figure 4.6.: trapezoidal map and the corresponding search graph, with one line segment.

The complexity of this point location algorithm depends on the insertion order of the line segments. But considering the expected length value of a search graph for a specific query point, it can be shown that this length is on average $\mathcal{O}(log(n))$, which is also the complexity of the point location. The building of the search graph and the trapezoidal map can be achieved in $\mathcal{O}(nlog(n))$ time.



(a) Trapezoidal map

(b) Search graph

Figure 4.7.: trapezoidal map and the corresponding search graph, with two line segments.

4.3. Map Overlay

Inside this section the problem of two overlapping subdivisions of faces will be discussed. To perform such an operation efficiently, the intersections of line segments must be determined efficiently.

4.3.1. Line Segment Intersections

The problem can be postulated as follows: Determine all intersection points of a set of n line segments in the plane (2 dimensional geometric space). A first approach therefore would be to test each line segment with the others for intersection. This would take time $\mathcal{O}(n^2)$ and would be optimal if for every two line segments there exists such an intersection, but would be rather inefficient if there are only a few intersections. So an efficient algorithm should not only depend on the complexity of the input, but also on the complexity of the output (output-sensitive algorithm).

To avoid unnecessary intersection tests, only line segments which are close together are be compared. The first restriction can be made, if intersection tests will only be performed if the x domains of the two line segments overlap. An x domain of a specific line segment is the orthogonal projection onto its x axis. The second one is to compare line segments only if they are neighbors. Two lines are neighbors if there is no line

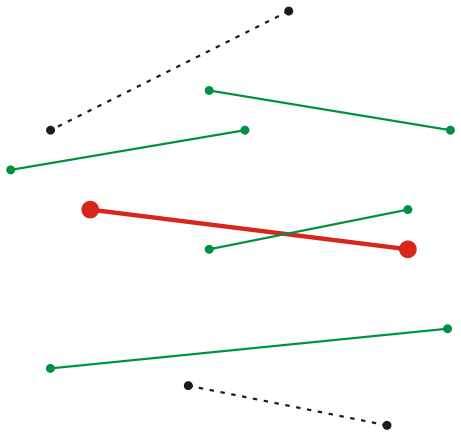


Figure 4.8.: neighbor lines(green solid) and non neighbor lines (black dashed) of a specific line (red solid thick)

segment between those two for a specific part of the x domain of both. This concept is shown in Figure 4.8.

Now the algorithm works with an imaginary vertical unbounded line which starts on the left of the leftmost line segment and propagates through all line segments to the right of the rightmost line segment. This line is called the sweep line. While sweeping through the line segments, the intersections between all line segments will be recognized. The invariant of the algorithm is, that every intersection point on the left of the sweep line has been detected. The sweep line has a status which is determined by the ordered set of line segments, which currently intersect the sweep line. The order of this set depends on the position of the sweep line and is determined by the intersection points of the sweep line and the line segments (see Figure 4.9). Note that these intersection points are different from those, which should be evaluated.

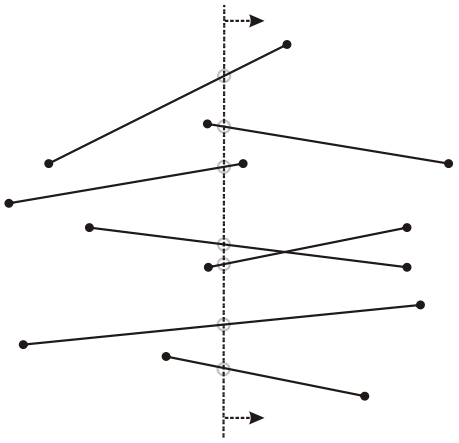


Figure 4.9.: sweep line (dashed) intersecting line segments (solid) at specific points (gray circles)

As suggested before, the status of the sweep line changes, while it moves forward. This changes in the status can only occur on endpoints of the line segments or at intersection points between them. These two categories of points are known as event points. So to perform a sweeping of the sweep line over the whole domain, it is sufficient to handle the event points from left to right. The event points will therefore be saved in an event queue. This queue will be initialized with the ordered set of endpoints of all line segments. For each event point the following operations must be performed, depending on the kind of the event point:

left endpoint If the event point is a left endpoint, then a new line segment will intersect the sweep line after this event point. So it must be added to the status set of the sweep line. The new line segment must be tested for intersection against its two neighbors. If there exists an intersection point, which is on the right of the sweep line, then this point is a new event point, which must be added to the event queue (with respect to its order), if it isn't already in the event list. This would be possible if a line segment intersects another line segment on its endpoint.

intersection point If the event point is an intersection point, then the two line segments, which have caused the intersection, change their order in the status set. So if the first line segment has been above the second line segment before the event point, it is below afterwards. As two line segments have changed their order inside the status set, each of the two line segments has got a new neighbor, which causes new intersection tests. If there exists an intersection point, which is on the right of the sweep line and hasn't already been computed, then this point is a new event point, which must be added to the event queue.

right endpoint If the event point is a right endpoint, then the segment containing it,

must be deleted from the status set and its two neighbors must be tested for intersection. Again, if there exists an intersection point, which is on the right of the sweep line and hasn't already been computed, then this point is a new event point, which must be added to the event queue.

Each time a new intersection point has been detected, this point will be stored in a separate output list. This algorithm requires two additional data containers during its run. The first, which embeds the event queue concept and the second, which must embed the actual status of the sweep line. The event queue must be able to perform the following operations efficiently:

- taking the leftmost event point from the queue
- inserting new event points
- determining, if a given event point has already been implemented (search)

This operation can be performend in time $\mathcal{O}(log(n))$, if a balanced binary search tree is used as data structure. The container for the sweep line status requires that following operations are performed efficiently:

- determining the neighbors for a given line segment
- removing line segments
- inserting line segments

This operations can be performed in time $\mathcal{O}(log(n))$ too, if a balanced binary search tree is used. Note that the order of the line segments inside the status container depends on the position of the sweep line.

If i is the number of intersections for a given set of line segments with n elements, then the algorithm must handle $(2n + i)$ event points. For each event point, it first must be popped from the event queue. Next the algorithm must insert and/or delete line segments. Then intersection tests are performed and eventually new event points must be inserted (if they haven't been inserted before). This operations take at most $\mathcal{O}(log(n))$ time so the overall complexity of the algorithm is $\mathcal{O}((n+i)log(n))$. Sorting the endpoints in the initial state doesn't change the asymptotical complexity. The sweep line algorithm sketched in this section is known as Bentley-Ottmann algorithm ((BO79)). A similar algorithm, which has been developed before, named the Shamos-Hoey algorithm (SH76), detects if a set of line segments has at least one intersection. The main idea behind it is identical to the just presented algorithm, except that if an intersection has been detected, the algorithm stops. The asymptotical runtime therefore is just

$\mathcal{O}(nlog(n))$, because the event points need not to be added to the event queue, which results from an intersection, as such an intersection would terminate the algorithm.

4.3.2. Map Overlay Algorithm

The map overlay algorithm computes the overlay of two subdivisions of faces, so it returns a new valid subdivision of faces. The algorithm presented here has been extracted from (dBvKOO00) pages 33-39. The definition of the map overlay inside this reference is given as

> ... define the overlay of two subdivisions S_1 and S_2 to be the subdivision $O(S_1, S_2)$ such that there is a face f in $O(S_1, S_2)$ if and only if there are faces f_1 in S_1 and f_2 in S_2 such that f is a maximal connected subset of $f_1 \cap f_2$. ((dBvKOO00))

To explain the functionality of this algorithm, the DCEL data structure is used as specific implementation of a subdivision of faces. The quintessence of this algorithm is to copy

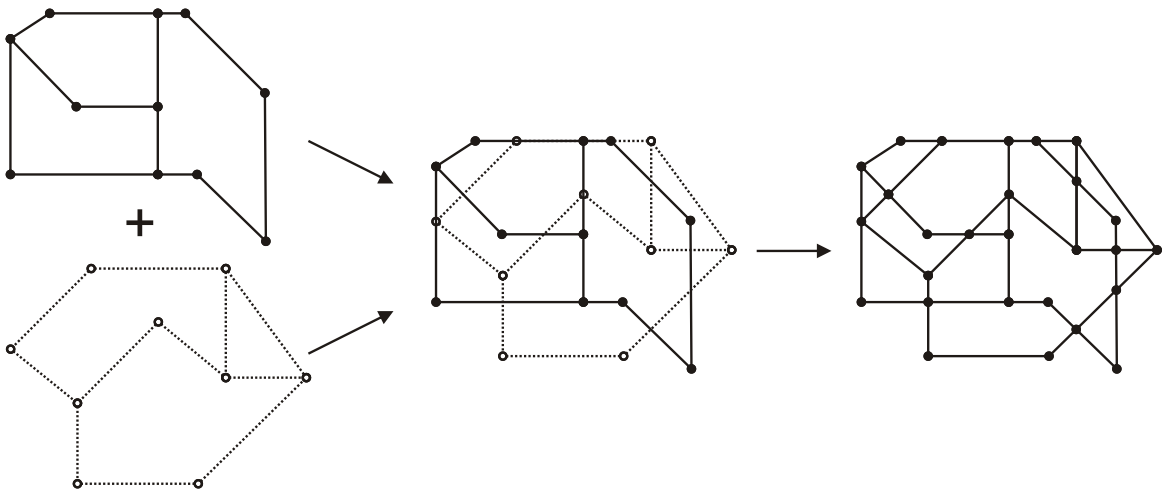


Figure 4.10.: Map overlay algorithm

both data structures in a new temporary not valid DCEL data structure and correct it in such a way, that it represents a valid DCEL data structure afterwards (see Figure 4.10). The vertex and the halfedge objects will be corrected by using a modified version of the line segment intersection algorithm (see section 4.3.1). Depending on the type of event point, 4 different cases can occur:

- If the event point has been an endpoint of only one of the original DCEL data structures, then nothing has to be done.

- The event point is an endpoint of one of the data structures and lies on a line segment belonging to the other data structure. In this case the line segment must be splitted up into two line segments cv_1 and cv_2 at the event point. This also means that the two halfedge objects of the original line segment will be substituted with four new halfedge objects which are corresponding to cv_1 and cv_2. The *next* and *prev* references of these four involved halfedges and some of the halfedges around the vertex must be modified, in order to point to the halfedges, which really are next and previous to them along the cyclic order around the vertex (see Figure 4.11).

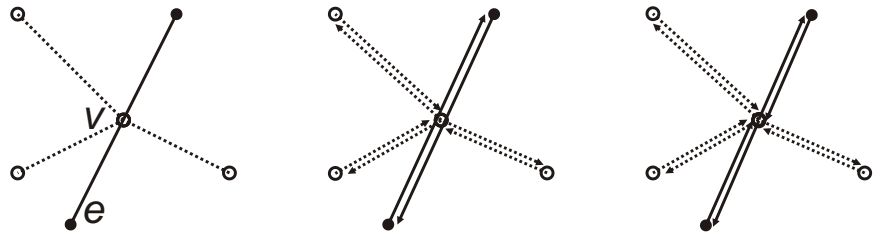


Figure 4.11.: Halfedge lies on vertex scenario

- If the event point is an intersection point between two line segments, then both line segments must be splitted up at the intersection point. Additionally a new vertex for this point must be introduced. Next the *next* and *prev* references of the 8 newly created halfedges have to be fixed up and one of this halfedges has to be assigned as *incidentEdges* to the new vertex object (see Figure 4.12).

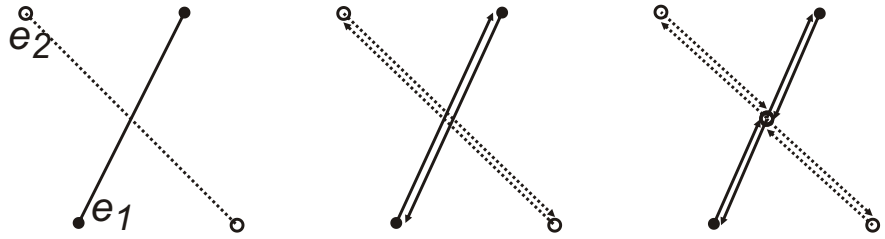


Figure 4.12.: Halfedge intersects other halfedge scenario

- If the event point belongs to endpoints in both data structures, then no new halfedge object will be generated. But again it is needed to fix up the *next* and

prev references of the involved halfedges to ensure the cyclic order around the vertex.

The next task is to update the face objects of the combined DCEL data structure. Therefore each new face f object will be named with the labels of the two face objects f_1 and f_2 (which form this new face f) of the two original DCEL data structures. Additionally a function g is needed, which connects the data from both faces f_1 and f_2 to a single data record for the face f. If for example ATC-Sector information objects are used as data for the original faces, the function g could therefore add both ATC-Sector information object to a list and assign this list as new data record to the newly generated face. To determine both faces f_1 and f_2 forming a new face f, the same modified line segment intersection algorithm is used again as above. Even the same 4 different cases of event points can occur. If the event point can be categorized in one of the later 3 cases, then the two involved DCEL objects (either a halfedge object or a vertex object) of both original data structures will be known. Therefore the two involved faces can easily be determined, which are used to construct the new face. In the first case, only the vertex object of one data structure is given. The determination of a DCEL object of the other one can easily be determined, if it will be traversed through the current status set of the sweep line algorithm (in upper direction) until a line segment is reached, which belongs to the other original DCEL data structure. As the two involved DCEL objects are now known, the two faces of the old data structures which build the new face, can be determined.
The last thing to do is to correct the *outerComp* and *innerComps* references of the new faces. A detailed explanation of this procedure is omitted here (see (dBvKOO00) pages 33-39 for details). The overall complexity of the map overlay algorithm is $\mathcal{O}((n + k)log(n))$, where n is the number of line segments of both original data structures and k is the complexity of the new data structures.

Chapter 5.

Implementation

In this chapter, the essential parts of the implementation of the localization module for FACTS will be presented.

5.1. Roadmap

Inside this section, the main milestones of the thesis implementation are presented.

1. The first thing was to determine all information of the simulated objects, which are available during the simulation. This information has been examined in their usability and helpfulness for aircraft localization in section 5.2.

2. Afterwards the characteristics of ATC sectors were assessed. One of the main tasks in this stage was to convert the ATC sector definition sourcefile, which has been extracted form the program SkyView2, to an appropriate structure. This will be presented in detail in section 5.3.

3. Comprehending existing point location algorithms has been the next task. This has been outlined in chapter 4

4. After all these steps have been performed, the main structure of the localization module has been defined (see section 5.4).

5. The first localizer, which has been developed, performs point location queries on a 2-dimensional Cartesian space. Its composition and functionality will be presented in detail in section 5.5.

6. The next step was to provide the possibility of 3-dimensional point location queries in the Cartesian space. For this purpose, three different point location strategies

have been used as outlined in section 5.6.

7. As the localization module of the FACTS simulator should offer point locations for aeroplanes, this requires the point location to be performed on the earth surface. So a spherical geometric kernel has been implemented (see section 6).

8. As the aircraft routes are given at the beginning of the simulation (see section 5.2), a so called dynamic point location algorithm has been developed, which follows the route of a specific aircraft to determine its position (ATC sector). The idea of this will be presented in detail in section 5.7.

5.2. Determination of Usable Information for Point Location

This section provides information on the status of the already existing FACTS simulator. The FACTS simulator contains a list of every aircraft simulated by it. For each aircraft the actual 3-dimensional position (latitude, longitude and height value) and the corresponding time value, at which this aircraft is in this position, will be given (also called 4-dimensional waypoint). Additionally to the actual waypoint, the list of all other waypoints along the flight route of this aircraft are stored with it. So it is possible to extract from two adjacent waypoints the direction vector between them. The localization module should be designed in such a way, that at least per specific waypoint a localization should be performed. To implement a realistic localization module, it will be also very useful to know the approximate number of aircraft, on which the simulation should run. The simulator should be able to handle about 2000 aircraft. Another key parameter is the average number of waypoints per aircraft, which is about 1000 waypoints per aircraft.

5.3. ATC Sectors

Inside this section, the characteristics and properties of ATC sectors will be considered. Additionally the mechanism of transforming the ATC sector information, retrieved from the program SkyView2 (Sky07), to an appropriate data structure will be explained.

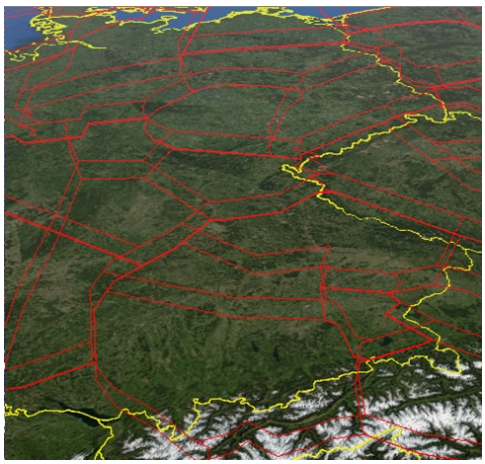


Figure 5.1.: ATC sectors - screenshot from Java Worldwind

5.3.1. ATC Sector Characteristics

The FACTS simulator and therefore the localization methods use the ATC (Air Traffic Control) sector definition of the program SkyView2 (Sky07). The main structure of a specific ATC sector is described by at least one 3-dimensional concave polyhedron (An ATC sector can also consist of several disjoint polyhedra). These polyhedra have a simple structure and consist mainly of two faces, which are parallel to each other along the height axis. All points which lie on one of these faces have exactly the same height coordinate (height level), thus making those faces horizontal. Additionally all points of the polyhedra lie on one of these two faces. Each point has a corresponding point, which has the same longitude and latitude coordinate, that lies on the other face. If there is a curve segment between two points of one of these horizontal faces, then there is also a curve segment between the two corresponding points on the other horizontal face. So the two faces are equal, except that they have different height coordinates. A 3-dimensional point lies now in the polyhedra if the 2-dimensional point (latitude and longitude) lies inside one of its faces and its height coordinates lie in the height level interval given by the two different height levels. In Figure 5.1 several examples of ATC sector polyhedra are illustrated. So to describe the geometry of such a sector polyhedra completely, it is enough to consider the following three parts:

- The upper height level value. This value is equal to the height coordinate of the horizontal face, which lies above the other.

- The lower height level value. This value is equal to the height coordinate of the horizontal face, which lies below the other.

- A description of one of these horizontal faces using concave polygons (see section 3). As the complete height information of the polyhedra is completely determined

by the preceding two parts, the height information can be discarded (projection to the ground).

There are about 3000 polyhedra using the ATC sector definition from SkyView2. In

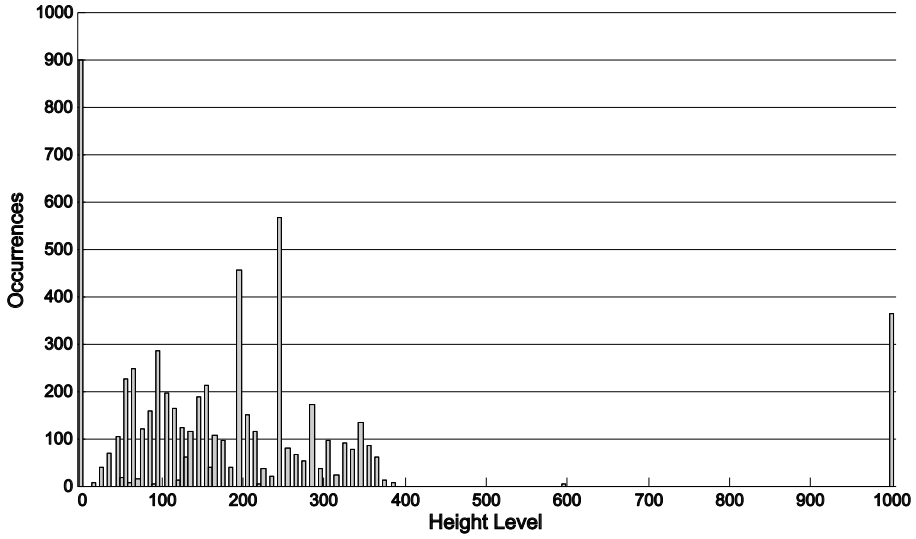


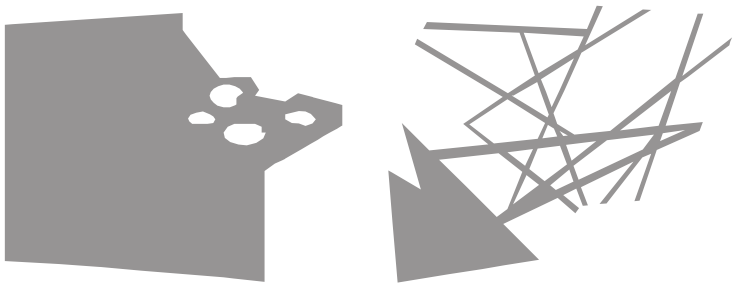
Figure 5.2.: Histogram of the different height levels

Figure 5.2 a histogram can be seen that shows the distribution of the different height levels of those ATC sectors. Overall there exist 49 different height levels.

As the ground structure of the polyhedra is given as face, holes can appear. In Figure 5.3 the ground projection of two examples of polyhedra are visualized, which contain holes.

5.3.2. ATC Sector Definition

As mentioned in the previous section, the ATC sector definition will be extracted from the program SkyView2. This program has the functionality of exporting the ATC sector definition that is used. The definition export file uses the XML based Geography Markup Language GML (GML04). The export file has the following DOM tree structure:



(a) ATC sector EISNLSS (b) ATC sector EGTTRED

Figure 5.3.: examples of ATC sector holes

```
− <ns1:FeatureCollection xsi:schemaLocation=...">
  + <ns1:boundedBy>
  − <ns1:featureMembers>
    − <SECTOR>
        <ICAO>BI</ICAO>
        <IDENT>BIRDFIS</IDENT>
        <UPPERLIMIT>55</UPPERLIMIT>
        <UPPERLIMITUNIT>FL</UPPERLIMITUNIT>
        <LOWERLIMIT>0</LOWERLIMIT>
        <LOWERLIMITUNIT>FL</LOWERLIMITUNIT>
      − <ns1:multiGeometryProperty>
        − <ns1:MultiGeometry>
          + <ns1:geometryMembers>
    + <SECTOR>
    + <SECTOR>
    + ...
```

It can be observed that it mainly consists of a node list of `<SECTOR>` tags. Each of these tags consists of the following essential attribute tags:

`<ICAO>`: ICAO state location identifier.

`<IDENT>`: Identifier of the ATC sector

`<UPPERLIMIT>`: Upper height level of the ATC sector (see section 5.3.1). The unit of the height level is determined by the `<UPPERLIMITUNIT>` tag. In this example the unit is FL, which stands for Flight Level.

`<LOWERLIMIT>`: Lower height level of the ATC sector.

Additionally to these attribute tags, each ATC sector consists also of an `<ns1:geometryMembers>` tag, in which the face (if an ATC sector corresponds only to one polyhedron) or the faces (if an ATC sector has several polyhedra) repectively will be specified. If the ATC sector consists only of one polyhedra and the corresponding ground face has no holes, then the `<ns1:geometryMembers>` tag would contain the following subnodes:

```
– <ns1:geometryMembers>
  – <ns1:Polygon>
    – <ns1:exterior>
      + <ns1:linearRing>
```

The `<ns1:linearRing>` tag of the `<ns1:exterior>` tag, which represents the boundary polygon of the face, would now contain the list of 2-dimensional coordinates representing the points of that polygon. The list has the following format:

```
{<longitude> <latitude> }*
```

Otherwise the tree structure would contain these subnodes:

```
– <ns1:geometryMembers>
  – <ns1:Surface>
    – <ns1:Patches>
      – <ns1:PolygonPatch>
        – <ns1:exterior>
          + <ns1:linearRing>
        – <ns1:interior>
          + <ns1:linearRing>
        + <ns1:interior>
        + ...
      + <ns1:PolygonPatch>
      + ...
```

For each polyhedron contained in the ATC sector, there exists one corresponding `<ns1:PolygonPat` tag. Therein the corresponding face of the specific polyhedra is given by one boundary polygon (specified by the points contained in the `<ns1:linearRing>` tag of the `<ns1:exterior>` tag) and several hole polygons (specified by the points contained in the `<ns1:linearRing>` tag of the `<ns1:interior>` tags).

5.3.3. ATC Sector Transformation

As mentioned in the previous section, the ATC sector information is stored in a GML structure. The first task was to convert this GML structure in a data structure, which can represent a subdivision of faces and is also able to contain additional information belonging to that ATC sector. After this transformation there is a bijective mapping between a face of the new data structure and a polyhedron (which is given by the data contained inside a `<ns1:PolygonPatch>` tag) belonging to an ATC sector.
The main transformation program (called Parser) is based on Java converts the information given by the GML input file into a independent faces representation (see section 3.2.1).
Figure 5.4 shows the class diagram of the parser. It mainly consists of two classes:

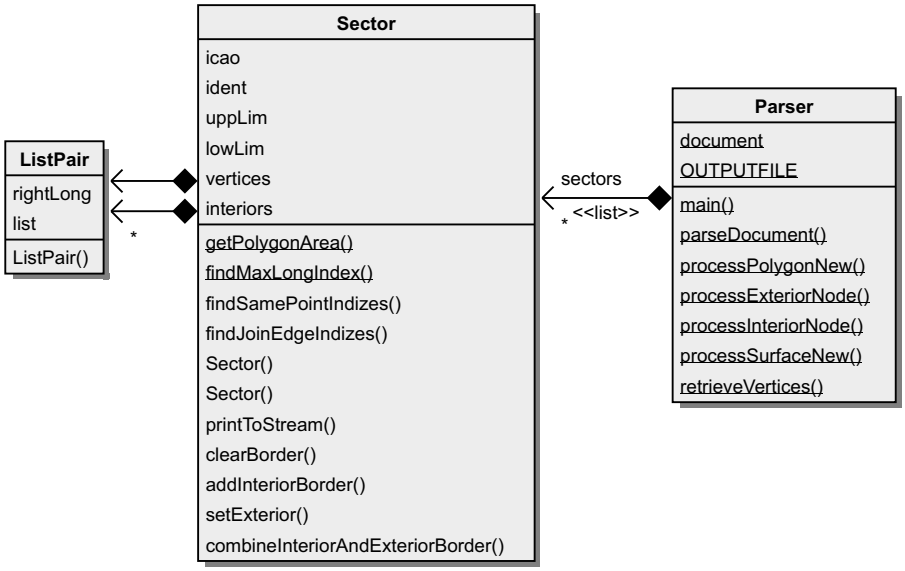


Figure 5.4.: class diagram of the Parser

Parser

The static Parser class is mainly responsible for parsing through the source file, which contains the ATC sector information, using the static method *parseDocument*. The source file itself is given as input stream referenced by the static variable *document*. Each time *parseDocument()* detects a `<SECTOR>` a new Sector object will be generated using the data read by the attribute tags (e.g. `<ICAO>`) of the `<SECTOR>` tag. Afterwards it will be decided, if the sector geometry is given by a `<ns1:Polygon>` tag, which will cause

the Parser to call the static method *processPolygonNew*. Otherwise the static method *processSurfaceNew* will be called. Inside the static method *processPolygonNew* the outer boundary polygon of the face, which will be read by the *retrieveVertices* method, will be set to the actual Sector object using its *setExterior* method. On the other hand if the Parser proceeds with the *processSurfaceNew*, then all hole polygons (inside a `<ns1:interior>` tag), which will also be retrieved by the *retrieveVertices* method, will also be added to the Sector object using its *addInteriorBorder* method. If there are several polyhedra inside an ATC sector, then the actual Sector object will be cloned and for the new object the polygons will be added separately. Each Sector that has been generated by the parsing operation, will be added to a list (*sectors*). These Sectors will then be written to an output file, where the corresponding output stream is defined by the *OUTPUTFILE* variable. For each Sector object there exists exactly one line in the output file.

Sector

A specific Sector object does not represent an ATC sector, moreover it represents one polyhedron of an ATC sector. So it might be said that the ATC sectors are splitted up into several subsectors, in which each subsector has a bijective mapping to a polyhedron of the original ATC sector. Each Sector object contains the attributes *icao*, *ident*, *uppLim* and *lowLim*, which correspond to the attributes defined by the GML input file of the original ATC sector. Additionally they have a attribute named *vertices*, which is a reference to a ListPair object containing the points of the outer boundary polygon. They also contain a list of ListPair objects, containing the interior polygon points of the holes, named *interiors*. One of the most powerfull methods is the *combineInteriorAndExteriorBorder* method, which combines the interior hole polygons and the exterior boundary polygon to one new boundary polygon by introducing bridge edges between the polygons (MH02). This method must be called at least before the Sector object will be printed (*printToStream* method), as the Independent faces representation can't handle holes.
The output format for one Sector is given by the following format, where `<LAT>` and `<LON>` represents the latitude and the longitude of a specific point belonging to the boundary polygon:

$$<icao>;<ident>;<lowLim>;<uppLim>;\{<LAT>;<LON>;\}*$$

5.4. Main Structure of the Localizer

Inside this section, the main structure of the localization module will be defined. The Localizer provides the following three main functionalities:

- *BUILD phase:* The Localizer provides the functionality to import ATC-Sectors in order to perform point location queries afterwards. This means in detail, that the ATC-Sectors will be stored in one or more DCEL data structures, provided by the CGAL library. To do this, the face, which describes partly an ATC-Sector polyhedron, will be inserted into the existing DCEL data structure. The remaining ATC-Sector information, including the two height levels, will be stored in an additional object (SectorInfo), which will be available via the *data* reference of the face object (see chapter 3.2.2 for further details). Defining these ATC-Sectors can either be done manually, using the insert functions, or can be performed for all ATC-Sectors at once. For the later case the definition of those sectors must be given, by a sourcefile, where the sector geometric are given in an independent face representation. Independently from the insertion mode, it is possible to check the validity of the imported sectors.

- *ATTACH phase:* When all sectors have been inserted, the next task of the Localizer is, to connect the constructed DCEL data structure to a selectable localization algorithm. If for example the randomized incremental point location algorithm has been chosen, then the trapezoidal map and the search graph will be built in that stage.

- *LOCATE phase:* The last functionality of the Localizer to performm efficient point location queries.

The main reason for using DCEL data structure is that it contains many topological information, to quickly find adjacent faces. This is important, because some of the localization algorithms (such as the walk along a line algorithm 4.2.1), require this property. The other point is that this data structure is used by the CGAL library, which is a main component of the localization module.

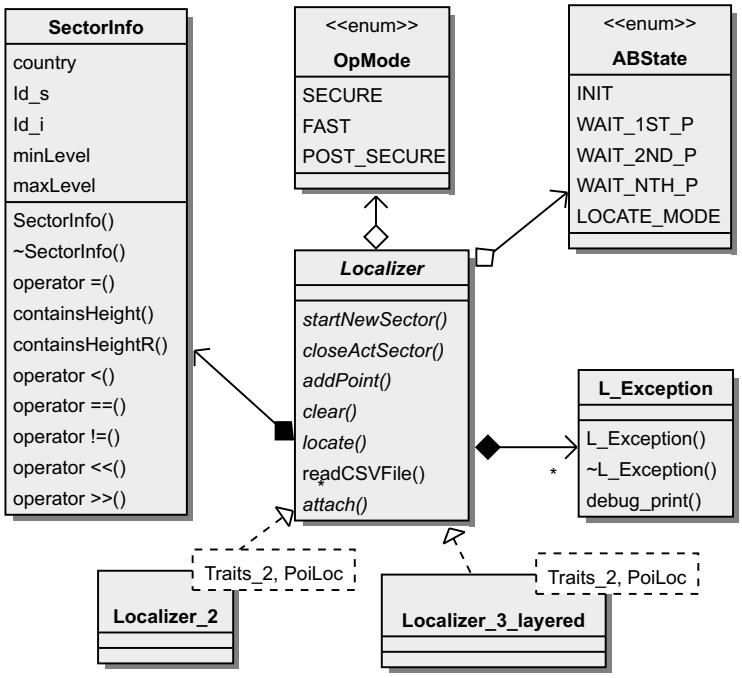


Figure 5.5.: Overview of the Localizer module

Figure 5.5 shows an overview of the localization module. It can be seen that it consists of the following classes:

5.4.1. Localizer Class

This class represents the abstract base class of all concrete Localizer classes (Localizer_2 for the 2-dimensional case and Localizer_3_layered for the 3-dimensional case). It defines the essential interface methods, which every concrete Localizer must contain. The method *readCSVFile* is the only method, which is not abstract, and is responsible for importing the ATC-Sectors from an input file, which contains the sectors in an independent face representation. The method itself uses some of the other interface methods, which are defined by a concrete subclass of the Localizer class, to import those sectors into a Localizer instance of that subclass. Additionally to these methods, the Localizer instances of the subclasses must contain two attributes:

OpMode This attribute defines the mode, how a concrete Localizer instance should check the validity of the inserted ATC-Sectors. Three different choices are possible:

- FAST: Using this option, the Localizer performs no validity checks. Only the used CGAL library performs some essential checks, but those checks can be turned off using the compiler option *-DNDEBUG*.

- SECURE: If *OpMode* is set to *SECURE*, then the Localizer tests each time, when a new curve segment is inserted into one of the DCEL data structures, if an intersection with a nearby curve segment in the same data structure has occurred.

- POSTSECURE: Using this option, the localizer performs no validity checks during the importation phase. But when the DCEL data structure[s] are attached to a point location algorithm, then the validity checks are performed for all contained DCEL data structures.

ABState This attribute determines, in which state the concrete Localizer instance is. The state transitions are given by the abstract methods, which every subclass must implement. The state transitions are represented as state machine in Figure 5.6.

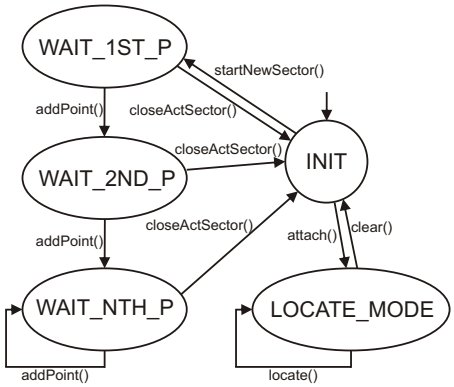


Figure 5.6.: State machine of the localizer

Every concrete Localizer instance must be in the state *INIT* after its initialization. The ATC-Sectors can now be defined. An ATC-Sector will be started by a *startNewSector* method call, which has a SectorInfoObjectt as parameter. The Localizer object is after this call in state *WAIT_1ST_P*. The next step is to define the ground face of the ATC-Sector. This will be done by several *addPoint* method calls, where each method call requires a tuple, defining the point by its latitude and longitude, as parameter. These points define then the outer border polygon of the face. During these method calls the Localizer arrives in the state *WAIT_NTH_P* passing the state *WAIT_2ND_P*. When all points of the outer border polygon have

been added, the ATC-Sector will be closed with *closeActSector*, which causes the localizer object to be again in state *INIT* after this method call. The *readCSVFile* method performs this methods call automatically. For every ATC-Sector defined in the input file, this method calls at first *startNewSector*. This will be followed by several *addPoint* calls and at the end the current sector will be closed with one *closeActSector* method call. If all ATC-Sectors have been inserted into the Localizer object, then this information, given in one or more DCEL data structures, will be attached (using the *attach* method) to the the point location algorithm. The localizer object is now in the *LOCATE_MODE* state after this operation. Now it is possible to perform point location queries with the *locate* method, which needs the 3-dimensional coordinates of the query point as parameter and returns the SectorInfo object of the ATC-Sector, in which the query point lies. This operation doesn't change the state.

5.4.2. SectorInfo Class

A SectorInfo class contains all attributes, which define an ATC-Sector, except its horizontal geometric shape. These parameters are *country*, *Id_s*, *minLevel* and *maxLevel*, which are equivalent to the attributes *ICAO*, *ident*, *uppLim* and *lowLim* of the Java class *Sector* of the Parser (see chapter 5.3.3). If a SectorInfo object is initialzed with the default constructor, then this object represents an unmanaged ATC-Sector (*Id_s* = *Unmanaged*). Additionally to the output and comparison operators, this class contains the method *containsHeight*, which determines if a given height value lies inside height level interval (including the lower height level *minLevel*, but excluding the upper height level *maxLevel*) of the SectorInfo object.

5.5. 2-Dimensional Localizer

The 2-dimensional localizer presented here is the first concrete implementation of the abstract Localizer class. The class diagram of it is represented in Figure 5.7. The 2-dimensional localization module consists of the following classes:

5.5.1. Localizer_2 Class

This class is the main class of the 2-dimensional localization module and is responsible for point locations. This localizer is limited, as the name indicates, to the 2-dimensional case. Thereby it would need a height value at the initialization, which will be stored in

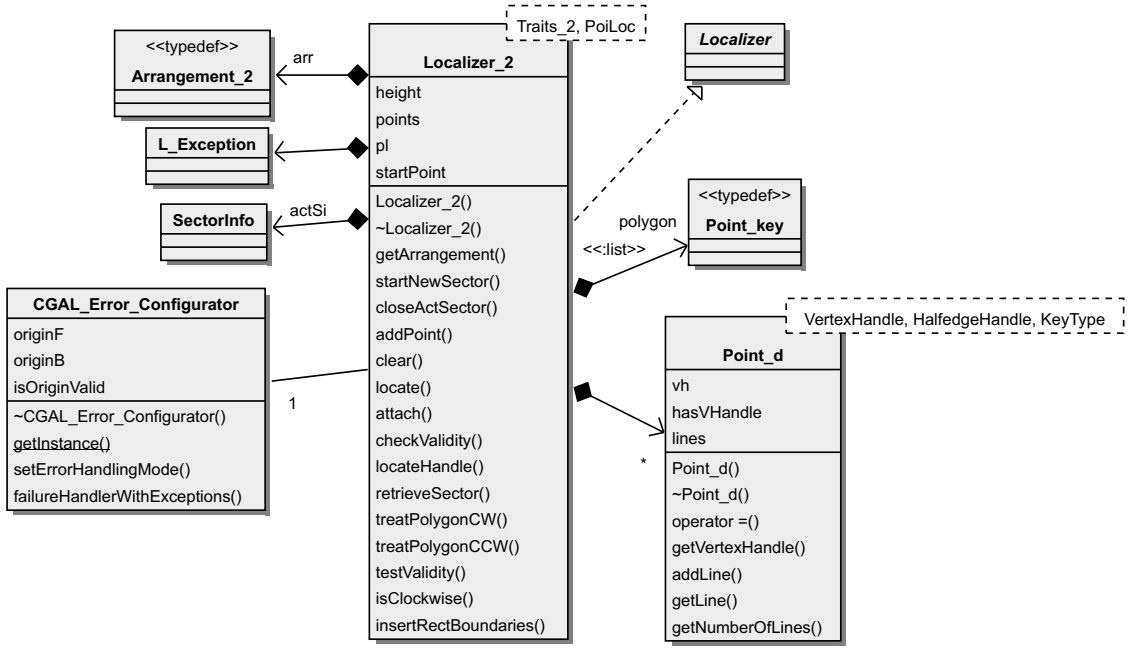


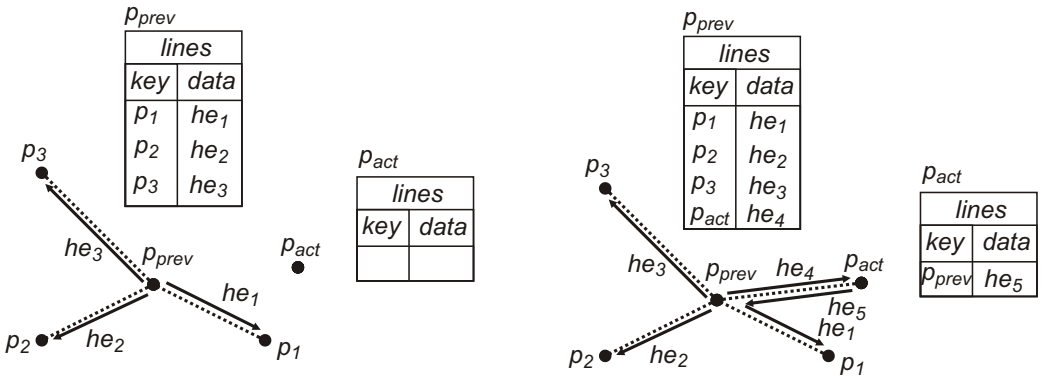
Figure 5.7.: Class diagram of the Localizer_2

the attribute *height*. The DCEL data structure of CGAL is reachable through the *arr* attribute. The interface class, which provides access to the geometrical kernel, can be defined by the template *Traits_2*. Whereas this geometric kernel (e.g. spherical kernel; see chapter 6) defines the properties of the points and curve segments, used from the Localizer_2 and the CGAL algorithms. The following three paragraphs describe now the implementation of the three main functionalities:

BUILD Phase of the Localizer_2

Every new ATC-Sector will be started with the *startNewSector* method, which has a SectorInfo object as parameter. This SectorInfo object will then be assigned to the *actSi* attribute. This method checks at first if the *height* value lies inside the height level interval of the Sectorinfo object. If this is the case, the points defining the ground face of the ATC-Sector can be defined using the *addPoint* method. The type of the Point class will be defined by the geometric kernel, which is reachable through the *Traits_2* template. Those points will then be added to the *polygon* list, which will be cleared during the *startNewSector* method call.Additionally it will be determined if the current point has been already included in the *points* map. The key values of this map are the points, which have already been included in the DCEL data structure. The data values of this map are the corresponding *Point_d* instances, which provide a proximity

relationship to the adjacent points for a specific point (see chapter 5.5.2). A point is adjacent to another point, if there is a curve segment between them. So if the current point is already available as key value in the *points* map, then a reference to this point in the *polygon* list will be remembered in the *startPoint* attribute.
During the *closeActSector* method call, the points contained in the *polygon* list are inserted together with their associated curve segments, into the DCEL data structure. Ao at first the method *isClockwise* determines whether the definition of the polygon is in clockwise or in counterclockwise order. Depending on the return value of this method, either the *treatPolygonCW*, which inserts the points in reverse order, or the *treatPolygonCCW*, which inserts the points in the given order, will be called. To insert these polygon points the position of one point inside the DCEL data structure must be specified first. If the *startPoint* has been set during the *addPoint* method calls, then the position of the point, on whom the *startPoint* reference points, is known. Otherwise the position of one of the points must be determined with a point location query. Based on this position the other points will be inserted. Before a new point will be instert, it is necessary to look in the *points* map, whether this point has already been inserted. In case it is not the point in combination with the curve segment, which lies between this point and the previous point, will be inserted in the DCEL data structure starting from the position of the previous point. After this operation the new point will be added to the *points* data structure. The next thing to do, is to update the *Point_d* objects, which correspond to these two points. Therefore two Halfedge objects are added, which describe the inserted curve segment, together with their destination points into the respective *Point_d* objects. This procedure is illustrated in 5.8. If the new point has



(a) Situation before curve segment has been added

(b) Situation after curve segment has been added

Figure 5.8.: Updating the two involved *Point_d* Objects

already been inserted, the remaining thing to do, is to check if a curve segment defined by this point and the previous point has also been defined. In case it is not this curve segment must be inserted. After this insertion the two involved *Point_d* objects must be updated.

After all curve segments of the outer border polygon have been inserted, the *data* reference of the new face, which is bounded by this polygon, will be assigned to the Sectorinfo Object given by *actSi*. All the other faces inside the DCEL data structure, which are created during the insertion of the current face, will be handled as unmanaged ATC-Sectors and will therefore contain an appropriate SectorInfo object (see Figure 5.9).

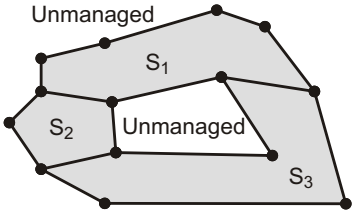


Figure 5.9.: Faces with unmanaged SectorInfo objects

ASSIGN Phase of the Localizer_2

After all ATC-Sectors have been inserted, the DCEL data structure *arr* will be assigned to the localization algorithm (attribute *pl*). The type of the algorithm can be chosen via the *PoiLoc* template. The CGAL library provides for this purpose the possibility to choose between the *walk along a line*, the *landmarks* and the *randomized incremental* point location algorithms. The assigning to the point location algorithm will happen during an *attach* method call. Additionally to this task, this method is responsible to check for all curve segments inside the DCEL data structure if there is an intersection possible between them (this will only be done if the *POST_SECURE* mode has been chosen). To do this, the CGAL implementation uses the Shamos-Hoey algorithm (see chapter 4.3.1).

LOCATE Phase of the Localizer_2

The localization of the correct ATC-Sector, which contains a given 2-dimensional point, will be done by the *locate* method. To call this method, the Localizer_2 instance must be in the state *LOCATE_MODE*. Otherwise an appropriate exception are thrown. The *locate* method itself determines first (using the selected point location algorithm) the face object, which contains the 2-dimensional point, respectively the halfedge or vertex object, if the point lies on such an object. If this DCEL object is a face, then the SectorInfo object, which is accessable through its *data* reference, will be returned. If the DCEL object is either a halfedge or a vertex object, then one of the adjacent faces of this object will be chosen. Now the corresponding SectorInfo object of this adjacent face

determines the ATC-Sector and will therefore be returned. The determination of the SectorInfo object for a given DCEL object is done inside the *retrieveSector* method.

5.5.2. Point_d Class

This class is an extension to the DCEL data structure (see chapter 3.2.2). An instance of this class, contains for a specific point the corresponding vertex object (attribute *vh*) of a DCEL data structure instance. Additionally it contains the halfedge objects, which have the vertex *vh* as origin, if the appropriate curve segments for these halfedges have been defined in the DCEL data structure instance. The point itself, for which this proximity relationship are built, will not be stored inside this class. The adjacent halfedges are stored as data values in the *lines* map. The key value for a specific halfedge, is defined by the destination point, on whose vertex the halfedge points. In other words, this point is the other endpoint of the curve segment which lies between this endpoint and the considered point. The class has the following interface:

getVertexHandle This method returns a reference to the vertex object, which corresponds to the point, for which this *Point_d* instance has been created.

getLine This method determines for a given point a reference to the halfedge (if such a halfedge exists), which orginates from vertex *vh* and ends in the vertex of the given point.

addLine Within this method a new Halfedge together with its destination point will be added to the *lines* map.

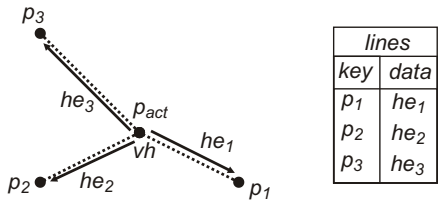


Figure 5.10.: Example for a Point_d instance

In Figure 5.10 an example of a Point_d instance for a specific point p_{act} is illustrated. The dashed lines inside this figure represent the curve segments between the points.

5.6. 3-Dimensional Localizers

Inside this chapter, the three different localizers, which are able to perform a point location for 3-dimensional points are introduced. The class diagram of these 3 localizers is visualized in Figure 5.11. Similar like for the 2-dimensional localizer, the used geometric

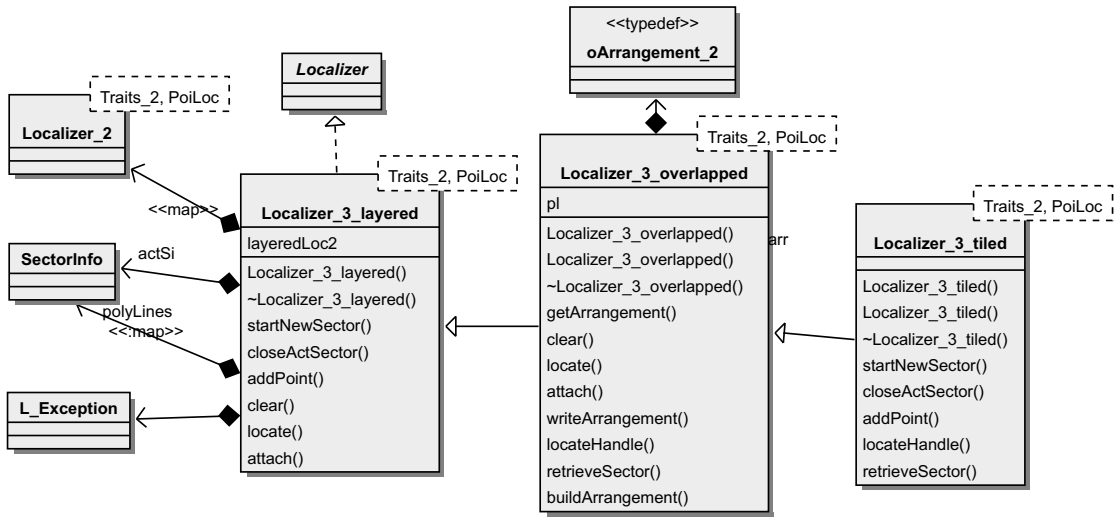


Figure 5.11.: Class diagram of the Localizer_3

kernel can be chosen via the *Traits_2* template. The template *PoiLoc* specifies the type of the point location algorithm. As illustrated in the class diagram (Figure 5.11), the three localizer classes are built on each other. Their functionality will be explained in the further subchapters:

5.6.1. Localizer_3_layered

This localizer uses, as its name implies, several 2-dimensional localizers, to perform a point location query for a 3-dimensional point. The number of the used 2-dimensional localizers depends on the number of the different lower height level values of the ATC-Sector, which should be inserted. In doing so, each different lower height level requires an own Localizer_2 instance (see Figure 5.12). If the ATC-Sector definition extracted from the program SkyView2 (Sky07) is used, then 48 different Localizer_2 instances (see chapter 5.3 for details) are needed. Those 2-dimensional localizers will be stored as data values in a map (attribute *layeredLoc2*). The key values of this map, are given by the *height* values of the Localizer_2 instances. Additionally to this map, there exists also another map (attribute *polyLines*), which stores the already inserted ATC-Sectors.

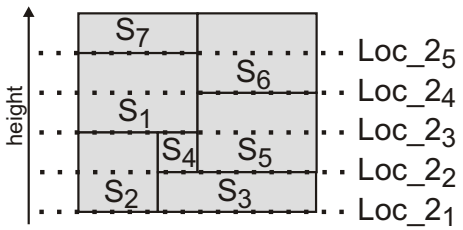


Figure 5.12.: Height levels of the contained Localizer_2 objects

The key values of this map are defined by the SectorInfo objects, whereas each data value contains a list with the outer border polygon points inside it. The following three paragraphs describe now the implementation of the three main functionalities:

BUILD Phase of the Localizer_3_layered

Each time when a new ATC-Sector is started (using the *startNewSector* method), the localizer determines, if there is already a Localizer_ instance in the *layeredLoc2* map, whose *height* value is equal to the lower height level of the SectorInfo object. Otherwise a new 2-dimensional Localizer must be generated, which will then be inserted into the *layeredLoc2* map. The *height* of this localizer is now equal to the lower height level of the SectorInfo object. As it is possible that already ATC-Sectors have been defined, which would fit into this new localizer, these ATC-Sectors must be inserted into the new Localizer using the *polyLines* map. Now the new ATC-Sector can be included into all involved 2-dimensional localizers. This new ATC-Sector will also be stored into the *polyLines* map.

ASSIGN Phase of the Localizer_3_layered

During this phase (*attach* method), all contained 2-dimensional localizers are attached to the point location algorithms. Attaching each 2-dimensional localizers can be very inefficient (see chapter 7).

LOCATE Phase of the Localizer_3_layered

To determine the SectorInfo object for a given 3-dimensional point, the 2-dimensional localizer, whose *height* value is close to but still below the height value of the point, will be determined via binary search first. Then a 2-dimensional point location is performed on this Localizer_2 instance.

5.6.2. Localizer_3_overlapped

This localizer is a subclass of the Localizer_3_layered class (see Figure 5.11). The Importation of the ATC-Sectors is done in exactly the same way as with the Localizer_3_layered instance would do it. The other functionalities will be described in the next two paragraphs:

ASSIGN Phase of the Localizer_3_overlapped

If all ATC-Sectors have been imported, all DCEL data structure instances of the 2-dimensional localizers are merged to a single DCEL data structure (using a *buildArrangement* method call inside the *attach* method). This merging is done with the map overlay algorithm (see chapter 4.3.2). As the map overlay algorithm can only merge two different DCEL structures, this algorithm must be called several times. The strategy of choosing the merge order of this DCEL data structures, has a relevant influence on the performance of this phase (see chapter 7.3.2). As mentioned in chapter 4.3.2, merging

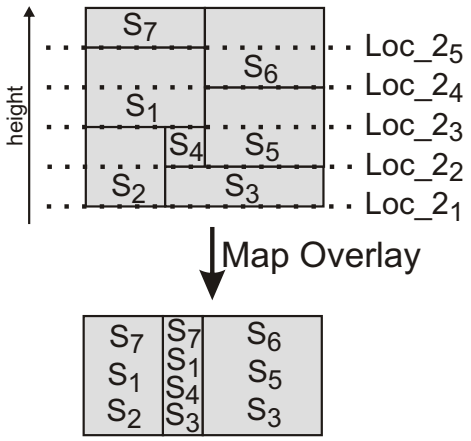


Figure 5.13.: Combining DCEL data structures of several Localizer_2 objects

several faces belonging to different DCEL data structures needs a function g, which dictates the procedure how the *data* objects of the faces should be combined. In our case this function inserts all SectorInfo objects contained by the involved faces of the original data structures into one list, which will then be assigned to the appropriate face of the final DCEL data structure. This process is illustrated in Figure 5.13. If the final DCEL data structure has been computed, it will be attached to the point location algorithm. As the combination of these data structures is rather complex, the *writeArrangement* method has been added to this localizer, which is able to store the final DCEL data structure into a file, which preserves the DCEL structure. This file can then be read by

another localizer instance during its initialization, so that no map overlay needs to be computed. Additionally this data structure will immediately be attached to the point location algorithm.

LOCATE Phase of the Localizer_3_overlapped

To determine the corresponding SectorInfo object for a given 3-dimensional point (inside the *locate* method), a point is splitted up into a 2-dimensional point and a height level. Now the corresponding face for the 2-dimensional point (*locateHandle* method) will be determined. This will be done in the same way as for the 2-dimensional localizer (see section 5.5.1). Next, the corresponding SectorInfo object out of the list of SectorInfo objects has to be determined, which is accessable via the *data* reference of the located face. The SectorInfo object, whose height level interval contains the height of the point, has to be choosen. This will be done in the *retrieveSector* method.

5.6.3. Localizer_3_tiled

The last localizer presented here is a refinement of the Localizer_3_overlapped class. Instead of inserting ATC-Sectors several times in the different 2-dimensional localizers, the ATC-Sectors will be only inserted at these Localizer_2, whose *height* value is equal to the lower height level of the corresponding SectorInfo object. The following two paragraphs describe the implementation of the other two main functionalities:

ASSIGN Phase of the Localizer_3_tiled

The difference between this phase and the corresponding phase of the Localizer_3_overlapped class is, that the function g, which is required to merge several faces, will only insert those SectorInfo objects into the *data* list of the face of the final DCEL data structure, which don't represent unmanaged sectors. This is essential, because otherwise the height level intervals of the SectorInfo objects belonging to the list would intersect each other. The reason for this is that a specific ATC-Sectors has been inserted only in one Localizer_2 instance and not in others, where it would also fit. In those other 2-dimensional localizers the face, which would bound the appropriate ATC-Sector, will not be defined anymore and would therefore be a part of an unmanaged area.

LOCATE Phase of the Localizer_3_tiled

Similar to the Localizer_3_overlapped class this localizer would first retrieve the face (*locateHandle* method) of the DCEL data structure, which contains the query point. The difference lies in the *retrieveSector* method. Inside this method, it will be considered whether the height value of the point lies inside the height level interval of a SectorInfo object. If this is the case, this SectorInfo object is returned. In the other case, a new unmanaged SectorInfo object must be created. The next thing remaining is to adjust both height levels of these Sectors in such a way, that they would cover the whole height level interval which is unmanaged (see Figure 5.14).

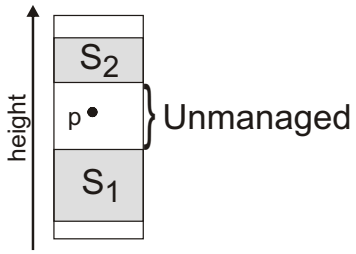


Figure 5.14.: List of SectorInfo Objects with Unmanaged areas

5.7. Dynamic Localizer

Inside this chapter a closer look to the two dynamic localizers, which has been implemented, will be performed. The first localizer is responsible for determining the ATC-Sector of an aircraft for the 2-dimensional case and the second performs this localization for the 3-dimensional case. Both localizers are able to follow the routes of aeroplanes, detect their ATC-Sector transition and report it afterwards. The route of a specific aircraft is given as a list of 4-dimensional waypoints. Whereas each waypoint consists of a 3-dimensional spatial position (longitude, latitude and height) and a time value on which the aircraft is at this spatial position. In Figure 5.15 the class diagram of both localizers is visualized. It can be seen from that class diagram that both localizers have a *LocateAlongRoute* method, which is able to determine the localization of ATC-Sectors for a given aircraft along its route. To do this, this method requires two adjacent waypoints (the actual waypoint wp_{next} and the next waypoint wp_{next} of an aircraft) and the identifier of an aircraft, which flies across this two waypoints. The result of this method, is the corresponding SectorInfo object through which the part of the route lies. If several ATC-Sectors would be crossed along this route part, then this method returns the ATC-Sector, which would be crossed first. For this case the method would also compute the waypoint wp_{trans}, at which the route part leaves the first time this ATC-Sector.

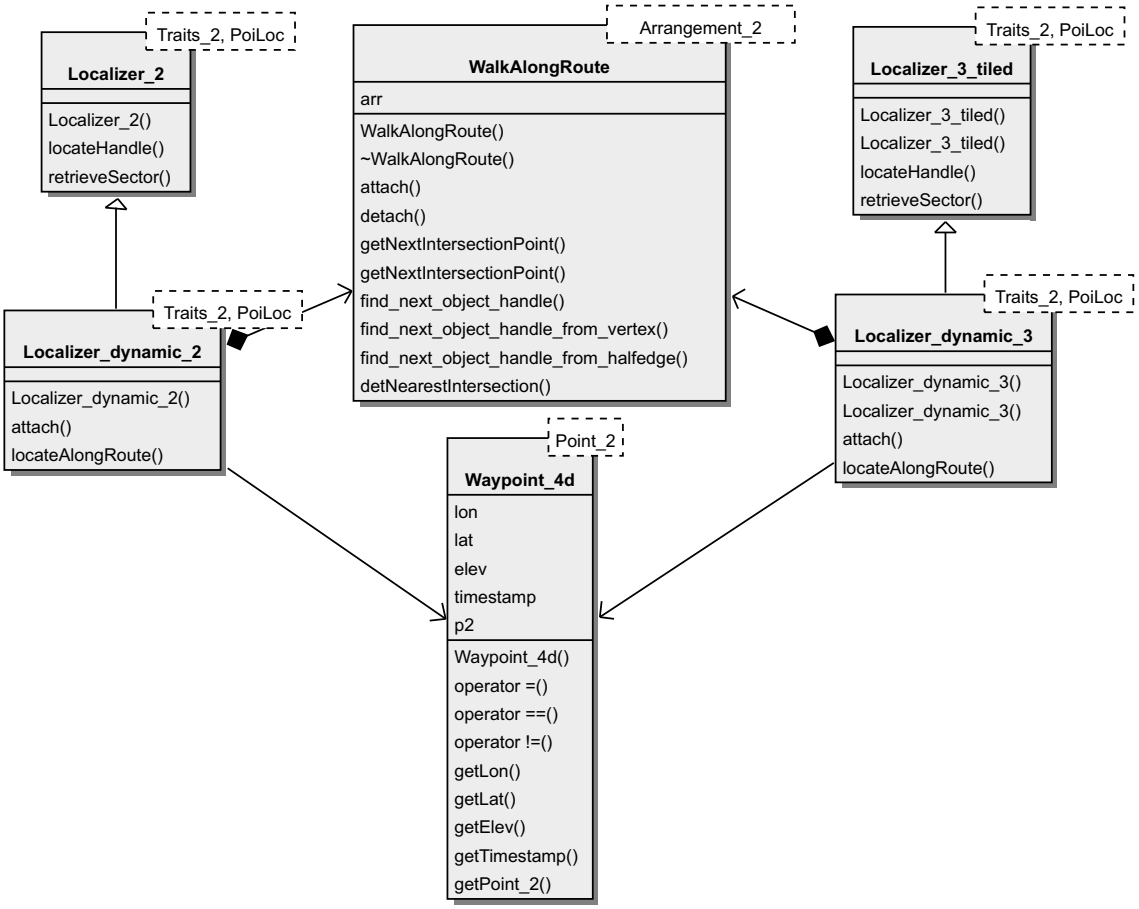


Figure 5.15.: Class diagram of the dynamic localizers

This waypoint will then be assigned to the next waypoint parameter, which would be delivered by reference. The next call of this method for the same aircraft, should now containt the wp_{trans} as actual waypoint and the original wp_{next} as next waypoint. The essential classes of the class diagram will be explained in the following paragraphs:

5.7.1. WalkAlongRoute Class

Every instance of this class contains an attribute *arr*, which represents a DCEL data structure. Every face inside this data structure is related to the ground projection of an ATC-Sector (for the 2-dimensional case) or to the intersection set of the ground projections of several ATC-Sectors (for the 3-dimensional case; see chapter 5.6). This DCEL data structure will be set during an *attach* method call. The main functionality of this class is given by the *getNextIntersectionPoint* method, which requires amongst others two 2-dimensional points p and q as parameters. This method would now determine if the curve segment defined by p and q would intersect the DCEL data structure in one or more intersection points. If this is the case, then this method returns the intersection point i, which is nearest to p (internal *detNearestIntersection* method call). This is illustrated in Figure 5.16. If there is no intersection, then the method returns q. To ensure

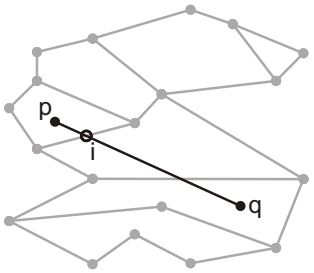


Figure 5.16.: Walk along route example

that this method is efficient, it requires that the position of p inside the used DCEL data structure would be provided as parameter. This position can either be a reference to a face, if p lies inside this face, or a reference to a halfedge respectively vertex, if p lies on such a DCEL primitive. This reference will be denoted in the following as obj_1. After the *getNextIntersectionPoint* is finished, obj_1 would contain a reference to the position of the nearest intersection point i respectively q (if there is no intersection). Additionally to this DCEL primitive the method determines also another reference to a DCEL primitive obj_2. obj_2 will be determined by an internal *find_next_object_handle* method call and contains the position of the points inside the DCEL data structure of the curve segment, which lies between p and i respectively q. This reference will then be used from both localizers to determine the corresponding ATC-Sector. In Figure 5.17 a concrete example is illustrated. The DCEL primitive reference obj_1 is in that case the halfedge e_1. After the determination of the intersection point i, obj_1 will be set to e_2. Int this example obj_2 is given by the face f_1.

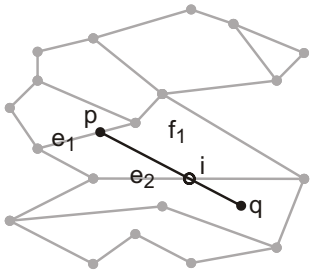


Figure 5.17.: Walk along route example with DCEL Objects

5.7.2. Localizer_dynamic_2 Class

The Localizer_dynamic_2 is a subclass of the Localizer_2 class (see chapter 5.5). Therefore its functionality is in principle the same, except for a few differences.For example a Localizer_dynamic_2 instance would assign the constructed DCEL data structure also to a WalkAlongRoute instance during its *attach* method. This localizer has also a map structure (*actAircraftPos*), which stores the next waypoint for all aircrafts, for whom a location along a route has already been performed. The key value of this map is the identifier of the aircrafts and the data value is a container, which consists of a waypoint, the appropriate 2-dimensional point and the position (reference to a DCEL primitive) of this point inside the DCEL data structure.
For each *LocateAlongRoute* method call the current waypoint wp_{act}, the next waypoint wp_{next} (which will be handled as reference, as the method may change this value) and the identifier of the aircraft must be provided as parameters. The first thing, which this method does, is to generate two 2-dimensional points p_{act} and p_{next} out of the two waypoints. Then this method checks whether an entry for the current aircraft exists in the *actAircraftPos* map. If this not the case, or the waypoint inside the map differs from the actual waypoint, then the position of p_{act} inside the DCEL data structure has to be determined. This will be done with *locateHandle* method call, which performs a static point location query. Now the *getNextIntersectionPoint* method of the WalkAlongRoute instance would be called, with p and q and the position of the p inside the DCEL data structure as parameters. This method returns the next intersection point p_{int} together with its position in the DCEL data structure. Those two values will then be assigned to the appropriate entry in the *actAircraftPos* map. A reference to the DCEL primitive obj_1, in which the points between p and p_{int} ly, will be also retrieved. Together with the height value of the actual waypoint and obj_1, the actual ATC-Sector can be retrieved (*retrieveSector* method). If the intersection point p_{int} is unequal to q, then the next waypoint wp_{next} will be adjusted. As the latitude and the longitude are already given by p_{int}, the remaining thing to do is, to calculate the values of the height and the time of the 4-dimensional waypoint. This is done by linear interpolation on the length of the two involved curve segments. These two curve segments are the curve segment between p and p_{int} and the curve segment between p and q. If this new next waypoint has been

determined, this waypoint is assigned as part of the data to the appropriate entry in the *actAircraftPos* map.

5.7.3. Localizer_dynamic_3 Class

The Localizer_dynamic_3 is a subclass of the Localizer_3_tiled class (see chapter 5.6.3). It works in principle in the same way as the 2-dimensional dynamic localizer. The difference is that ATC-Sector transition can not only occur along the horizontal projection. They can also occur in a vertical direction, as a DCEL face consists of a set of ObjectInfo objects. This situation is visualized in Figure 5.18. While the 2-dimensional

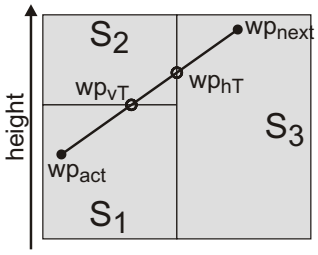


Figure 5.18.: Route inside a 3-dimensional space

dynamic localizer would detect wp_{hT} as transition and therefore as new next waypoint, the 3-dimensional dynamic localizer must detect wp_{vT} as transition point. To do this, this localizer will first determine the horizontal transition point wp_{hT} in the same way, as the 2-dimensional dynamic localizer would do that (including the determination of the current ATC-Sector). Then it will be evaluated, whether the height value of this waypoint lies inside the height level interval of the current ATC-Sector. If this the case, then the correct waypoint has been computed and the localizer is finished. If this is not the case, then the vertical transition point wp_{vT} must be evaluated. For this purpose the ratio β, which consists of two height level intervals, is computed. The first interval (enumerator) will be generated by the height level of the current waypoint and the upper or lower height level of the SectorInfo Object (depends on the fact, whether the horizontal transition point lies above or below the current waypoint). The second height level interval (denominator) will be constructed by the difference of the height level of the actual waypoint and the height level of the next waypoint. Then the 2-dimensional point p_{vT} is computed, which lies on the curve segment, defined by p_{act} and p_{next}, and has a relative distance of β to p_{act}. If the curve segment is a great circle segment, then this point with the parametrized great circle segment definition (see chapter 2.3.3) is computed. The last thing remaining is to compute the time value of this waypoint, which will also be accomplished with linear interpolation.

Chapter 6.

Implementation of the Spherical Kernel

Three different spherical kernels will be introduced inside this chapter, which allows the localizer to perform point locations on the earth surface. For this purpose there must be a point class and a segment class per spherical kernel. Thereby, the point class represents spherical points and the segment class represents spherical greatcircle segments. The diverse localizers and the CGAL library requires 2-dimensional points and curves, whereas the spherical points of a greatcircle segment must be treated as a 2-dimensional mapping (see section 2.3.5). As visualized in Figure 6.1, there is the possibility that there can occur spherical creatcircle segments, whose 2-dimensional mapping is not connected. To

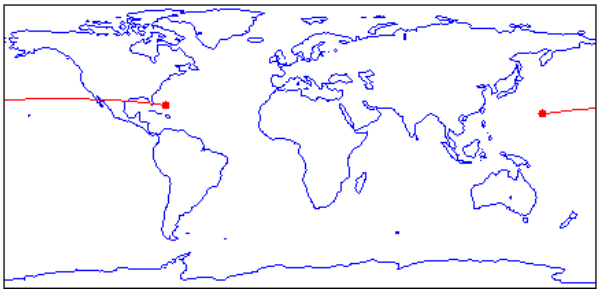


Figure 6.1.: Problematic greatcircle segment

overcome this effect, the number of greatcircle segment are restricted:

1. No greatcircle segments are allowed, which contain either the north or the south pole, otherwise there will be a jump in the 2-dimensional mapping. To fullfill this, the following restrictions to the endpoint p and q must be applied:

$$|\phi_p| \neq \frac{\pi}{2} \quad \wedge \quad |\phi_q| \neq \frac{\pi}{2} \quad \wedge \quad |\lambda_q - \lambda_p| \neq \pi \tag{6.1}$$

2. Greatcircle segments, which cross the $+/-180$ meridian ($\lambda = +/- \pi$) are not allowed, because the planar mapping of a greatcircle would consist of two distinct

and not adjacent subcurves. To guarantee this, the following restriction is needed:

$$|\lambda_q - \lambda_p| < \pi \tag{6.2}$$

A greatcircle segment, which fullfills the above restrictions, is transformed into the 2-dimensional plane either vertical line segment (if $\lambda_p = \lambda_q$) or a continuous curve. Every vertical line intersects such a continuous curve at most once. This can easily be shown by extracting a vertical line segment from this line, whose lower endpoint is infinitely near $-\frac{\pi}{2}$ but above it and whose upper endpoint is infinitely near $\frac{\pi}{2}$ but below it. These endpoints are defining a greatcircle segment, which intersects an other greatcircle segment only once. The other greatcircle segment's planar mapping is the given curve (already demonstrated in section 2.3.3). The next step is to provide a proof that this continuous curve will not be intersected at $\pm\frac{\pi}{2}$, but this is per definition not possible. Thus the set of this restricted greatcircle segment can be quoted *weakly x-monotone*, which has been defined inside the CGAL library:

> A continuous planar curve C is *x-monotone*, if every vertical line intersects it at most once. ... treat vertical line segments as *weakly x-monotone*, as there exists a single vertical line that overlaps them. ... (CGA07)

Consequently the localization algorithms and the map overlay algorithm not only work for line segments, but also for *weakly x-monotone* curves. So these set of greatcircle segments can be assigned to those algorithms.

6.1. Interface class

This section will give a closer look to the interface class, which will be assigned as template to the localizer and the CGAL algorithms. The class diagram of the interface is pictured in Figure 6.2. The Figure shows that this class needs the template T_kernel, with whom the geometric kernel should be chosen. The different spherical kernels, which have been implemented, will be described in section 6.2. The interface function objects of this class will call the corresponding methods of the point respectively greatcircle segment class, which are inherited by the given spherical kernel. The functionality of the most important function objects will be overviewed in the following itemization:

- *Compare_xy_2:* This function object compares two spherical points with each other. First of all the longitude values of these points are compared. If these values are equal, then the latitude values of these points will be compared. This function object returns the relative position of the first point compared to the second one

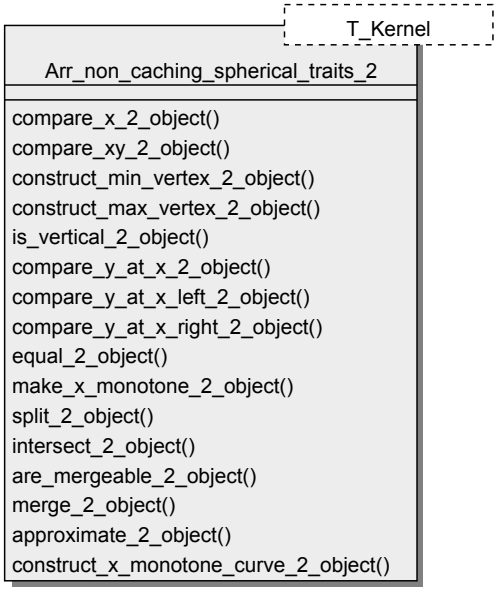


Figure 6.2.: Interface class

(smaller, equal or larger).

- *Compare_y_at_x_2:* Inside this function object, the relative position of a spherical point compared to a greatcircle segment is determined. To guarantee a correct mode of operation the longitude value of the spherical point must be inside the longitude range of the greatcircle segment. If this condition is fullfilled, the function object determines, whether the spherical point is below, above or on the greatcircle segment.

- *Compare_y_at_x_right_2:* With this function object, the relative position of two greatcircle segments, which have the same left endpoints, will be determined. Therefore, it will be determined whether the first greatcircle segment is below, above or lies along the second greatcircle segment. For that purpose the function object *Compare_y_at_x_2* can be used. But firstly the smaller of the both endpoints has to be determined (by using *Compare_xy_2*). Then this smaller endpoint will be compared, using the *Compare_y_at_x_2* function object, with the greatcircle segment, whose right endpoint is the larger right endpoint. When the smaller right endpoint belongs to the first greatcircle segment, then the result of *Compare_y_at_x_2* will be directly returned, otherwise this result has to be inverted (larger $\rightarrow$ smaller, smaller $\rightarrow$ larger).

- *Intersect_2:* This function object evaluates the intersection point of two greatcircle segments, if there exists one. The strategy of retrieving such an intersection point has been described in section 2.3.2.

6.2. Geometric Kernels

6.2.1. Kernel_sph_surf

The kernel *Kernel_sph_surf* is one of more concrete implementation of a spherical kernel. The classes of this kernel use the findings of the spherical trigonometry for answering geometrical issues.

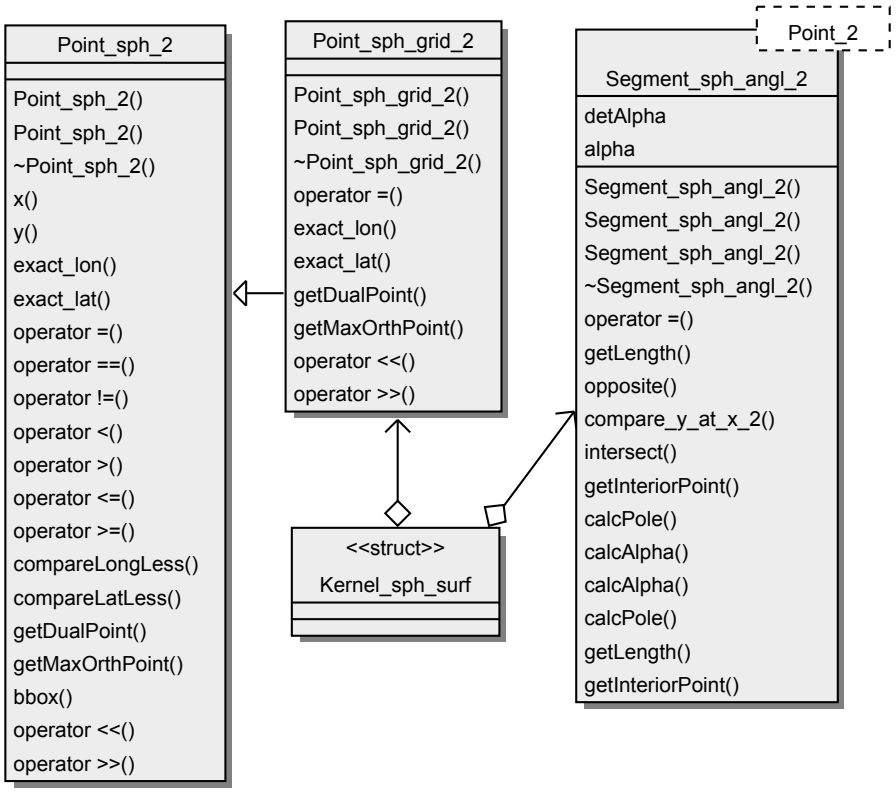


Figure 6.3.: Overview of the inexact spherical kernel

Every instance of a spherical point p (Point_sph_grid_2) contains an attribute for its latitude value (ϕ_p) and an attribute for its longitude value (λ_p), whose used data type is *double*. Additionally, two rounded values of these variables are stored in this class for matching in a predefined grid. The precision of these rounded values used in the implementation, which has been constructed for this thesis, are 17th digits. This procedure is essential for comparing the rounded latitude and longitude values of a point, which

can be derived from the intersection of two greatcircle segments with other points lying close to this one. This attitude is equal to all implemented compare methods.

Every instance of a greatcircle segment consists of two such spherical points. Additionally, one of the two possible polar points will be stored in the class, which is defined by the greatcircle on which the greatcircle segments lies. The calculation of this polar point will be determined by using the concept of spherical triangles (Sig77). First the angle α

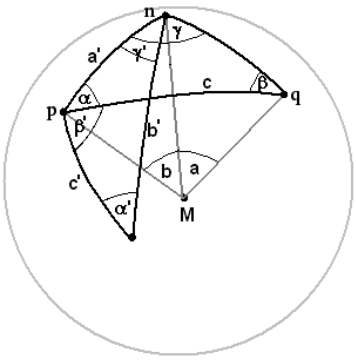


Figure 6.4.: Spherical triangles with polar point calculation

is dtermined by using the point p, which has the smaller longitude value. The angle α lies between the north pole n and the other point q on the surface, as it is illustrated in Figure 6.4. The definition of the angle α is given by the following formulas:

$$\frac{\alpha+\beta}{2} = tan^{-1}\left(cot\frac{\gamma}{2}\cdot\frac{cos\frac{a-b}{2}}{cos\frac{a+b}{2}}\right)$$
$$\frac{\alpha-\beta}{2} = tan^{-1}\left(cot\frac{\gamma}{2}\cdot\frac{sin\frac{a-b}{2}}{sin\frac{a+b}{2}}\right)$$
$$\alpha = \frac{\alpha+\beta}{2}+\frac{\alpha-\beta}{2} \tag{6.3}$$

whereas a, b and γ are defined in the following way:

$$a = \phi_n - \phi_q = \frac{\pi}{2} - \phi_q \tag{6.4}$$
$$b = \phi_n - \phi_p = \frac{\pi}{2} - \phi_p \tag{6.5}$$
$$\gamma = |\lambda_q - \lambda_p| \tag{6.6}$$

This angle α will be used to compute $\tilde{\beta}$, so that the angle between $\tilde{c}$ and b has the constant value of $\frac{\pi}{2}$:

$$\tilde{\beta} = |\alpha - \frac{\pi}{2}| \tag{6.7}$$

The distance between point p and polar point r is constantly $\tilde{c} = \frac{\pi}{2}$. The reason for this is that the polar point r is defined by the normal vector of the plane, which contains the greatcircle. As the point p lies on this plane (more precisely on the greatcircle), the vector starting at the sphere midpoint M to this point is always orthogonal to the normal vector of the plane. The next step is to consider a new spherical triangle, for whom $\tilde{\beta}$, $\tilde{c}$ and $\tilde{a} = b$ are defined (see Figure 6.4). The polar point r is determined by computing $\tilde{\gamma}$ and $\tilde{b}$ at first as described in the following:

$$\frac{\tilde{\gamma} + \tilde{\alpha}}{2} = tan^{-1}\left(cot\frac{\tilde{\beta}}{2} \cdot \frac{cos\frac{\tilde{c}-\tilde{a}}{2}}{cos\frac{\tilde{c}+\tilde{a}}{2}}\right) \tag{6.8}$$

$$\frac{\tilde{\gamma} - \tilde{\alpha}}{2} = tan^{-1}\left(cot\frac{\tilde{\beta}}{2} \cdot \frac{sin\frac{\tilde{c}-\tilde{a}}{2}}{sin\frac{\tilde{c}+\tilde{a}}{2}}\right) \tag{6.9}$$

$$\tilde{\gamma} = \frac{\tilde{\gamma} + \tilde{\alpha}}{2} + \frac{\tilde{\gamma} - \tilde{\alpha}}{2} \tag{6.10}$$

$$\tilde{b} = 2tan^{-1}\left(tan\frac{\tilde{c}}{2} \cdot \frac{cos\frac{\tilde{\alpha}-\tilde{\beta}}{2}}{cos\frac{\tilde{\alpha}+\tilde{\beta}}{2}}\right) - \tilde{a} \tag{6.11}$$

Now, the the latitude and longitude values of the polar point r can be computed:

$$\phi_r = \frac{\pi}{2} - \tilde{b} \tag{6.12}$$

$$\lambda_r = \begin{cases} \lambda_p - \tilde{\gamma} \text{ if } \alpha - \frac{\pi}{2} < 0 \\ \lambda_p + \tilde{\gamma} \text{ if } \alpha - \frac{\pi}{2} > 0 \end{cases} \tag{6.13}$$

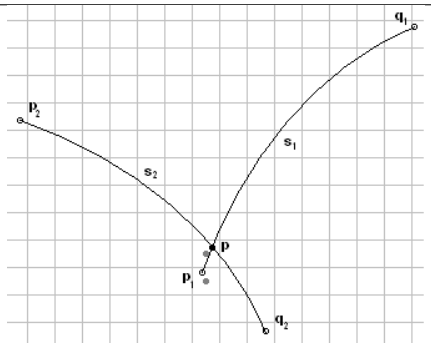
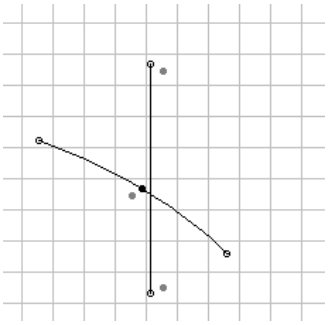
The computation of these both values is always based on the usage of the original values, not the rounded one.

Next, the method *compare_y_at_x* is used for calculating the relative position of a point r to a given greatcircle segment cv_1 (with the endpoints p and q). Therefore, a new greatcircle segment is introduced, which starts from point p and ends up in point r. A check will be done, whether point r is situated on the greatcircle segment cv_1. This will be done, by comparing the polarpoints of the two greatcircle segments. If the polar points are equal, then the point r lies on the greatcircle segment cv_1 and the end is reached. If the polar points are different, then the angles α_1 and α_2 have to be calculated (as described in formula 6.3). If $\alpha_1 > \alpha_2$, then the point r lies above the greatcircle segment, otherwise below it.

The method *intersect* calculates the intersection of two greatcircle segments. As mentioned in section 2.3.2, the intersection of two greatcircles is given by the intersection of two planes on which the greatcircle lie. Firstly, the normal vector of both planes and the normal vector based on these normal vectors will be calculated. This normal vector, the sphere's midpoint M and the radius of the sphere are used together to get two intersection points. As shown in section 2.3.2 this normal vector of a plane, on whom the greatcircle lies, can be used to compute the polar point of this greatcircle. If both definitions are combined, the intersection of two greatcircle segments can be calculated, by firstly computing their polar points. For this purpose the polar point for each greatcircle segement will be determined. Then this two polar points will be connected with a new greatcircle segment. The polar points of this new greatcircle segment is now one of the intersection points of the two greatcircles, on which the greatcircle segments are lying. Next it has to be determined whether one of these intersection points is lying inside the range of both greatcircle segments. This will be tested by using the method *collinear_has_on*. This method checks, whether the intersection point is greater or equal than the left endpoint and smaller or equal than the right endpoint.

Numerical stability

This kernel uses the imprecise calculation model as previously mentioned. In this section the error rate ant reason of errors will be discussed. Errors can occur, if for example two points will be compared, which are very close together, but not equal. To overcome this effect, the rounding operation can be made more exact by shifting the decimal place backwards. The precision after rounding in the thesis implementation will be 17 digits. The decision of using such a precision is based on two reasons. At first it has been checked that using the ATC-Sector definition extracted from SkyView2 will not cause such an error as described above. Secondly, the resolution using 17 decimal places resolves to less than 1 meter on the surface. Another error source is been generated by the *compare_y_at_x* method. Using the inexact spherical kernel, it can happen that a point which is close to a greatcircle segment, will be identified to lie on the wrong side of this segment. This error is not dramatic as long as the resulting ATC-Sector is adjacent to the correct one and the point is almost in this adjacent ATC-Sector. A more serious error can be caused by the intersection of two greatcircle segments. During testing the thesis implementation (see chapter 7), two major errors have been found, which may cause a wrong ATC-Sector definition:

Intersection near endpoint	*intersection at vertical greatcircle segment*
	
For this first error, it can be observed, that the intersection point p lies near the first endpoint p_1 of greatcircle segment s_1. As illustrated in the graphic above, the intersection point will be rounded to the position above p_1. If a *compare_y_at_x* call is now performed, the point p would not ly on s_1. A possible method to avoid this error would be to round this point to p_1, but therefore it would be possible, that this point may not lie on the other greatcircle segment s_2 anymore.	The next error can occur, if one of these greatcircle segments (for example s_1) is vertical and lies near the grid border. If now the calculated intersection point lies on the other side of this grid border, the *compare_y_at_x* method would once again fail. In this example this error can be avoided, by setting the longitude of this point to the longitude value of s_1.

The measurement of the frequency that these errors are occuring will be explained below. For that, a test routine has randomly generated greatcircle segments. Thereby, two greatcircle segments are always used for calculating an intersection point, if there exists one. The next step is to check with the *compare_y_at_x* method, whether the intersection point is on both greatcircle segments. If that is not the case, than an error will be registered. This testing scenario has been repeated until 10^6 intersection points have been detected. Using this mode of operation, the error rates, with the decimal place of the rounding operation as a variable parameter, have been measured as outlined in table 6.1. As it can be seen in this table that the error rate decreases, while the position of the rounding decimal place increases. The reason for this effect will be caused by the smaller tolerance which makes the first error source more probable. This effect is valid until the tolerance decimal place reaches the value 17. For 18th decimal place the error rate rises. The reason for this is, that the tolerance is so small that a small divergence can cause an error during the polar point comparison. If a less precise tolerance is used, the errors didn't appear.

tolerance decimal place	error rate in %
10	$1.288 \cdot 10^{-4}$
11	$6.123 \cdot 10^{-5}$
12	$3.016 \cdot 10^{-5}$
13	$1.736 \cdot 10^{-5}$
14	$7.311 \cdot 10^{-6}$
15	$3.655 \cdot 10^{-6}$
16	$1.828 \cdot 10^{-6}$
17	$9.138 \cdot 10^{-7}$
18	$1.828 \cdot 10^{-6}$

Table 6.1.: Error rate depending on the tolerance decimal place

6.2.2. Kernel_sph_vect

Compared to the inexact spherical kernel the *Kernel_sph_vect* is using the 3-dimensional coordinates of the spherical points as attribute instead of latitude and longitude. The transformation of latitude and longitude value into 3-dimensional Cartesian coordinates (and vice versa) has been defined in section 2.3.1. The benefit of this representation is that exact computations can be performed, which will be outlined below. This assumes that an exact number type is used to store this coordinates. The main disadvantage is, that the comparison operations become more complex and time consuming, because the order of the points still depend on their latitude and longitude values. The transformation of the Cartesian coordinate into spherical coordinates is not usefull, as the transformation uses trigonometric functions. As this functions deliver always irrational numbers, which may be infinitely long, it cannot be calculated exactly. To assure this relative order of two points though, the following scheme is used:

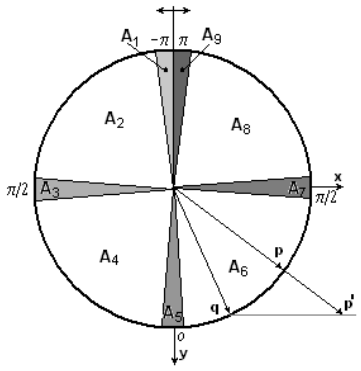


Figure 6.5.: Different possible areas with point comparison example

- At the beginning it has to be determined in which area the longitude value of these spherical points is. Therefore the x- and the y-coordiante of the point p are compared to zero. The result of these comparisons can be used for assigning a specific point to 9 possible areas A_1-A_9 (actually 10, if the case $p_x = 0$ and $p_y = 0$, which denotes either the north or south pole, is considered). This assignment has been visualized in Figure 6.5.

- If both points lie in different areas, then the indices of these areas define the relative order of the points. If for example the first point is lying in A_3 and the second point lies in A_6, then the first point is smaller than the second one.

- If both points are lying in the same area, then further calculations must be performed. For this purpose the coordinates of the first point have to be temporarily stretched or bulged in such a way that $p'_y = q_y$ (see Figure 6.5). Now both x-coordinates are compared with each other and if they are equal then the z-coordinates are equal too. If both points have been either in A_3 or A_7, then the first point has to be stretched respectively bulged such that $p'_x = q_x$.

Similar to the inexact spherical kernel an instance of a greatcircle segment of this kernel consists of the two endpoints of it. Additionally to the endpoint the polar point will also be stored too. The 3-dimensional coordinates of the polar point r are given by the cross product of the vectors $\overrightarrow{v_p}$ and $\overrightarrow{v_q}$ of the points p and q (see section 2.3.2). The vector $\overrightarrow{v_r}$, which corresponds to the polar point r, will not be normalized as the squareroot operation would return an irrational result. An exact result is achieved with disregarding the wrong distance between the polar point r and the sphere midpoint M, as the cross product only uses multiplications and subtractions.

The calculation of the intersection points (*intersect* method) of two greatcircle segments, would be computed similar to the inexact spherical kernel (beside the polar point calculation). So, the *compare_y_at_x* method of the greatcircle class follows another strategy. For this purpose, the greatcircle segment will be intersected with a new vertical greatcircle segment, which runs through the specified point. If there is an intersection, then this point will be compared with the intersection point.

6.2.3. Kernel_sph_vect_norm

This kernel works almost in the same way as the previous one. The only difference is, that the vectors, which corresponds to a spherical point, will be expanded in such a manner that $p_y = \pm 1.0$. If the vector points either into A_3 or A_7, then the vector will be expaned for fullfilling $p_x = \pm 1.0$. This mechanism is visualized in Figure 6.6. In doing

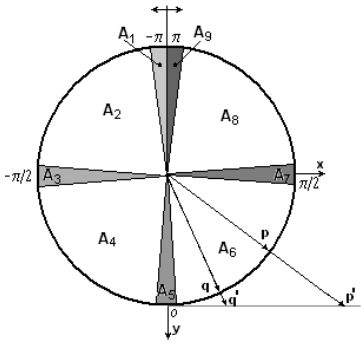


Figure 6.6.: Example for two spherical points using the normalized spherical kernel

so, the stretching of the points are not needed to compare their x-coordinates with each other. So a better performance will be yielded.

Chapter 7.

Tests and Evaluation

7.1. Evaluation Conditions

In this evaluation part the following point location algorithms were used:

- **Arr_naive_point_location** algorithm, which performs for each polygon of the DCEL data structure a point in polygon test (this algorithm works similar like the ray crossing method). This algorithm will be denoted in the following with the term NAIVE.

- **Arr_walk_along_a_line_point_location** algorithm, which has been represented in chapter 4.2.1. It will be denoted with WALK.

- **Arr_landmarks_point_location** algorithm, where the landmarks will be given by the points of the DCEL data structure instance (see chapter 4.2.2). This algorithm will be called VERT.

- **Arr_landmarks_point_location** algorithm, where the landmarks will be given by a grid of points named with GRID.

- **Arr_trapezoid_ric_point_location** algorithm (see section 4.2.3). This algorithm will be named RIC.

For the whole evaluation the ATC-Sector definition, extracted from the SkyView program (see chapter 5.3.1), has been used.

7.2. 2-Dimensional Localizer

In this chapter the performance of the three main functionalities of the 2-dimensional localizer are examined. For all tests inside this chapter a Localizer, which has a height level of 3050, has been used. Therefore only ATC-Sectors containing this height will be considered.

7.2.1. BUILD Phase

At first we take a look on the running times of the import phase of these localizers for all possible geometric kernels and several number types. For this evaluation, all ATC-Sectors, which contain the height level 3050, have been imported. These are 776 ATC-Sectors.

Number types	Cartesian	Kernel_sph_vect	Kernel_sph_vect_norm
Quotient	1.7	6.89	
Lazy_exact_nt <Quotient >	0.09	3.3	
Gmpq	0.28	6.87	1.17
Lazy_exact_nt <Gmpq >	0.09	3.3	1.09
Kernel_sph_surf <double >	0.09		

Table 7.1.: Importing ATC-Sectors (different geometric kernels and number types) times in seconds

It can be observed in table 7.1, that importing the ATC-Sector definition for the Cartesian kernel is faster, if the Gmpq (GNU Multiple Precision Arithmetic Library (GMP09)) number type is used instead of the Quotient number type. If both are connected with the Lazy_exact_nt class, which performs exact computation only when it's necessary, the importation speed can be improved. As those both values are identical, this suggests that exact computation is hardly ever performed. The same effect can also be observed for the exact spherical kernel Kernel_sph_vect. Additionally it can be seen that the Kernel_sph_vect_norm kernel is much faster than Kernel_sph_vect, but they are both slower than the inexact spherical kernel Kernel_sph_surf.

The next evaluation will analyze the complexity of the import phase, depending on the number of inserted ATC-Sectors. As for each ATC-Sector several curve segments have to be included, the number of curve segments are used as parameter. This test will only be performed for the Cartesian kernel with Gmpq number type, the inexact spherical kernel and the exact Kernel_sph_vect_norm also with the Gmpq number type. For both kernels, which have the Gmpq number type, the Lazy_exact_nt class is used. It can

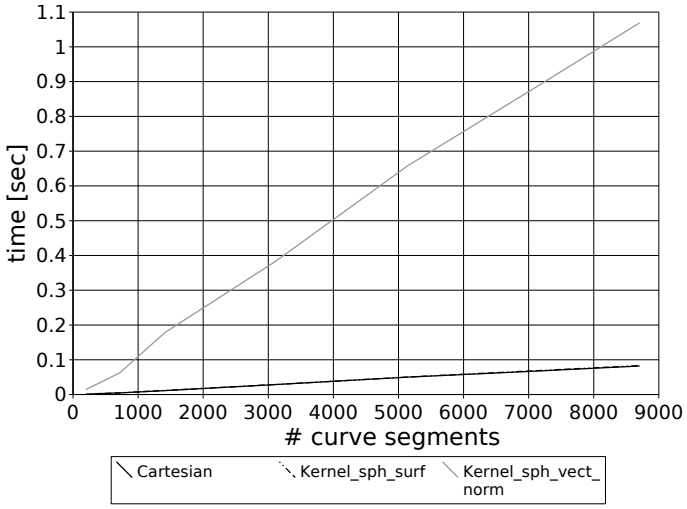


Figure 7.1.: Importing ATC-Sectors (different number of ATC-Sectors); times in seconds

easily be seen in Figure 7.1, that the localizer with Cartesian kernel requires almost the same time for importing the ATC-Sectors as the localizer with inexact spherical kernel. Using the exact spherical kernel requires much more time. It seems that the importation phase has a linear complexity depending on the number of ATC-Sectors. Theoretically the localizer should have a higher importation complexity, as it must evaluate for each ATC-Sector the position of it in the DCEL data structure. This requires at least one map access (in *points*) which has a complexity of $\mathcal{O}(log(p))$, where p is the number of points. As the insertion of an ATC-Sector is rather expensive, this map access for these small numbers of ATC-Sectors will be of no consequence.

Now a closer look to the complexity of the *SECURE* and the *POST_SECURE* mode of the localizer will be taken. For this test, the inexact spherical kernel is used. In Figure 7.2

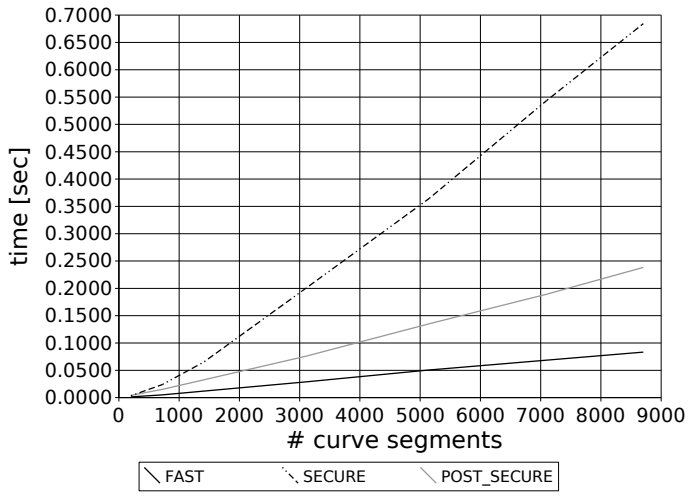


Figure 7.2.: Importing ATC-Sectors with different insertion modes

it can be seen that the importation time, if the *SECURE* importation is used, is nearly 80 times slower (note that the values for the *SECURE* mode have been divided by 10) than using the *FAST* mode. Additionally to this drawback, the importation is also not linear. The *POST_SECURE* mode itself is only about 3 times slower, but a nonlinearity occurs. The reason for this is, that this method uses the Shamos-Hoey algorithm (see chapter 4.3.1) for the intersection tests, which has a complexity of $\mathcal{O}(nlog(n))$.

7.2.2. ASSIGN Phase

Inside this section the running times, which are required to attach the DCEL data structure to the point location algorithms, will be examined. The first test will give an overview of the attaching times of all used point location algorithms. Again the three geometric kernels are used, namely the Cartesian kernel with Gmpq number type, the inexact spherical kernel and the exact Kernel_sph_vect_norm also with the Gmpq number type. For both kernels, which have the Gmpq number type, additionally the Lazy_exact_nt class is used.

It can be observed in table 7.2, that the two simple point location algorithms NAIVE and WALK require very little assigning time. The reason for this is that they have only to copy the DCEL data structure. The landmarks algorithm (GRID), whose landmarks belongs to a grid, is much slower than the landmarks algorithm (VERT), whose landmarks are the points of the DCEL data structure. The reason for this is that the

# Algorithms	Cartesian Lazy_exact_nt <Gmpq >	Kernel_sph_surf double	Kernel_sph_vect_norm Lazy_exact_nt <Gmpq >
NAIVE	0.003	0.002	0.005
WALK	0.003	0.002	0.004
VERT	0.012	0.011	0.024
GRID	0.107	0.117	0.804
RIC	0.419	0.845	5.658

Table 7.2.: Assigning ATC-Sectors to different point location algorithms; times in seconds

position of the landmarks inside the DCEL data structure for the GRID point location algorithm has to be computed, whereas the position of the VERT landmarks is already given. The VERT algorithm is therefore about 10 times faster then the GRID point location algorithm. For the exact spherical kernel this ratio is even higher. The last algorithm presented here (RIC) is the slowest algorithm to assign. It is about 8 times slower than the GRID algorithm.

Now a closer look to the performance of the last three point location algorithms for the inexact spherical kernel Kernel_sph_surf with different numbers of ATC-Sectors is taken. As the VERT algorithm uses the points of the DCEL data structure as landmarks, the only thing it has to do inside the assigning phase is to build up the KD-tree data structure, therefore it requires hardly any computation time. The GRID landmarks algorithm in contrast has to compute the position of all landmark points inside the DCEL data structure, which requires much more effort. Although it seems in Figure 7.3 that this task has a linear complexity, the determination of these points has a complexity of $\mathcal{O}(nlog(n))$. The RIC point location algorithm has the same complexity as the GRID

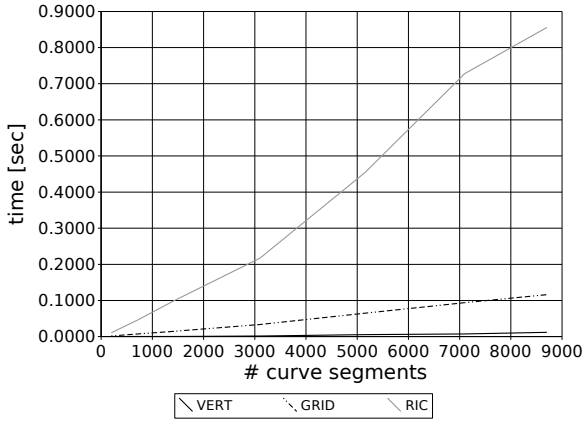


Figure 7.3.: Assigning ATC-Sectors to point location algorithm

algorithm, except that it requires much more time for the assigning operation. This suggests that it has a higher constant.

7.2.3. LOCATE Phase

Inside this section the performance of the point location queries will be examined. For the first evaluation again the three geometric kernels are used, namely the Cartesian kernel with Gmpq number type, the inexact spherical kernel and the exact Kernel_sph_vect_norm also with the Gmpq number type with the Lazy_exact_nt option. To determine the mean point location query time, several thousand points, which have been generated with a random generator, have been used for this test. All points lie in the range of the points of the DCEL data structure. For this test, all ATC-Sectors, whose height level interval contain the height value 3050, have been imported. The results are visualized in table 7.3. As expected, the NAIVE algorithm is the slowest point location

# Algorithms	Cartesian Lazy_exact_nt <Gmpq >	Kernel_sph_surf double	Kernel_sph_vect_norm Lazy_exact_nt <Gmpq >
NAIVE	$3.200 \cdot 10^{-3}$	$3.248 \cdot 10^{-3}$	$8.329 \cdot 10^{-3}$
WALK	$3.480 \cdot 10^{-5}$	$6.160 \cdot 10^{-5}$	$3.554 \cdot 10^{-4}$
VERT	$4.609 \cdot 10^{-5}$	$1.450 \cdot 10^{-5}$	$2.594 \cdot 10^{-4}$
GRID	$1.138 \cdot 10^{-5}$	$1.181 \cdot 10^{-5}$	$4.240 \cdot 10^{-5}$
RIC	$4.600 \cdot 10^{-6}$	$9.220 \cdot 10^{-6}$	$2.672 \cdot 10^{-5}$

Table 7.3.: Point location query time for different point location algorithms

algorithm. The WALK algorithm, which requires also almost no preprocessing time as the NAIVE one, is about 100 times faster than the NAIVE algorithm for the Cartesian kernel. If another kernel is used, then the speedup lies in the range of 50. The other algorithms including the WALK algorithm have similar query times. This indicates, that the problem size (number of curve segments) is to small, to reach significant differences. It is noticeable that the query time for the VERT localization logarithm is for this problem size slower than those of the WALK algorithm and also slower than the GRID algorithm, which uses almost the same strategy 4.2.2. The difference is that the VERT algorithm uses the points of the DCEL instance as landmarks and has therefore at first to determine in which adjacent face of a specific point the curve segment to the query point lies.

In the next test the change of the query time, when varying the number of inserted curve segments, will be examined. This test will be only performed for the inexact spherical kernel. To see the difference, the query times for a specific point location algorithm are given as ratio between them and the largest query time of that algorithm (see Figure 7.4). It can be observed that the NAIVE algorithm has almost a linear complexity for larger numbers of curve segments. The walk algorithm becomes more efficiently for

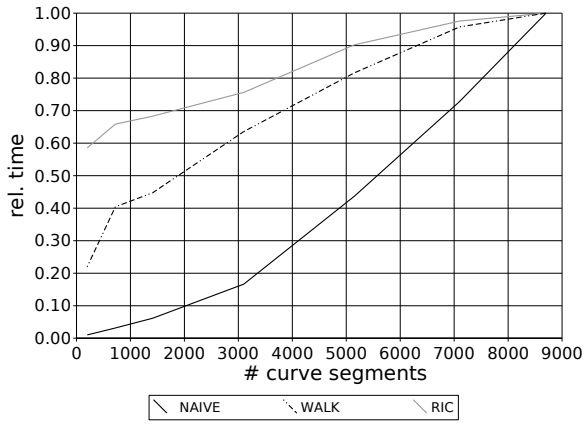


Figure 7.4.: relative query time for several point location algorithms

higher numbers of curve segments. The reason for this is that its complexity is $\mathcal{O}(\sqrt{n})$. For the RIC point location algorithm it can be seen that it behaves the same way, except that its slope is flatter. This can be explained by the complexity of the algorithm, which is $\mathcal{O}(nlog(n))$ and therefore better than those of the WALK algorithm.

7.3. 3-Dimensional Localizers

Inside this chapter the performance of the three main functionalities of the three 3-dimensional localizers will be examined. The Localizer_3_layered will be denoted as LAYERED, the Localizer_3_overlapped as OVERLAPPED and the Localizer_3_tiled localizer as TILED.

7.3.1. BUILD Phase

Next, a closer look to the performance of the importation times of the Localizer_3_layered class and the Localizer_3_tiled class will be taken. The Localizer_3_overlapped class will not be considered, as the importation procedure is equal to those of the Localizer_3_layered. The Cartesian, the inexact spherical (Kernel_sph_surf) and the exact Kernel_sph_vect_norm kernel are again used. Again the Gmpq number type with the Lazy_exact_nt option is used for the two exact kernels. All ATC-Sectors will be imported for this test. The results of this test are visualized in table 7.4. The TILED localizer imports all ATC-Sectors about 10 times faster into the DCEL data structures as the LAYERED one for the exact kernels. For the inexact spherical kernel the speedup is

Localizers	Cartesian Lazy_exact_nt <Gmpq >	Kernel_sph_surf double	Kernel_sph_vect_norm Lazy_exact_nt <Gmpq >
LAYERED	3.013	2.712	42.235
TILED	0.443	0.412	4.297

Table 7.4.: Importing the ATC-Sectors into the 3-dimensional localizers with different geometric kernels, values in seconds

only about 8. The reason for this is that the LAYERED localizer inserts a specific ATC-Sector in several 2-dimensional localizers. It can be seen that the importation with the exact spherical kernel requires much more time than the importation for the other two kernels.

In the next test, the change of the insertion speed of the two localizers, depending on the number of ATC-Sectors included, will be observed. These results have only been examined for the inexact spherical kernel. In Figure 7.5, the developing of the

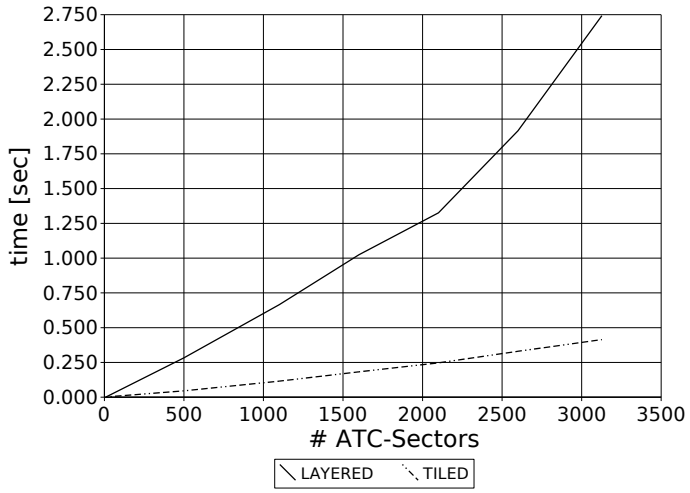


Figure 7.5.: Importation time for a different number of ATC-Sectors

importation times for several numbers of ATC-Sectors is outlined. The imporation times of the TILED localizer increase almost linear, while the times of the LAYERED localizer grow faster and are not linear. The reason for this nonlinear slope is, that the LAYERED localizer has to insert an ATC-Sector in several contained 2-dimensional localizers. As a higher number of ATC-Sectors implies also a higher number of 2-dimensional localizers, there exist more 2-dimensional localizers, in which a specific ATC-Sector could fit. Therefore the probability is higher, that ATC-Sectors, which will be inserted later, would be inserted into more 2-dimensional localizers than ATC-Sectors, which will be inserted at the beginning.

7.3.2. ASSIGN Phase

Now a closer look to the assigning complexity of the 3-dimensional localizers for all used point location algorithms is taken. These times will only be determined for the inexact spherical kernel. In Figure 7.6 it can be seen, that the LAYERED localizer is very

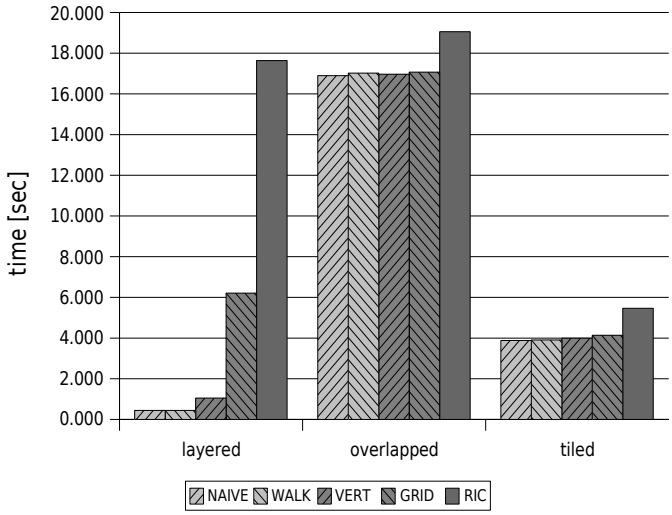


Figure 7.6.: Assigning time for all localizers and point location algorithms

efficient, if a point location algorithm, which require hardly no precomputation, is used. The reason for this is that this localizer simply assigns each contained 2-dimensional localizer to a different point location instance. As this delivers quite good results for simple point location algorithms, it is rather bad if the algorithm (primarily the RIC algorithm) requires much preprocessing time. However the assigning times of the RIC point location algorithm combined with the inexact spherical kernel have not been drawn. The assignment time of the Cartesian kernel for this algorithm has been visualized instead. The reason for this is that the localizer (using one of the spherical kernels) requires a very high amount of memory, which disposes the system to swap the memory to the harddisk. The Cartesian kernel in contrast uses an internal caching strategy, such that equal curve segments will be only stored once (CGA07). The OVERLAPPED algorithm computes in this assign phase the map overlay of all DCEL data structures of the 2-dimensional localizers and assigns only once the emerging DCEL data structure to the point location algorithm. Therefore it is good for point location algorithms, which have a high preprocessing time. As the TILED localizer avoids the redundancy of several included ATC-Sectors compared to the OVERLAPPED algorithm, it can be seen that this improves the assigning times considerably.

As mentioned in chapter 5.6.2, the map overlay algorithm can only combine two DCEL data structures at a certain time. Therefore this algorithm must be applied multiple times to combine all data structures to a single one. It can be seen that the order in which these DCEL data structures are combined has a major impact on the performance of this task. For this purpose the following three combining strategies are considered:

- The first strategy combines the DCEL data structures of the 2-dimensional localizers according to their height values. This means in particular that in the first step the two DCEL data structures, whose localizers have the smallest height values, will be combined. The newly emerged DCEL data structure will then be combined with the data structure of the localizer, which contains the next higher height value. This procedure will be repeated until the data structure of the localizer with the highest height value has been considered too. This strategy will be denoted in the following with MERGE_HEIGHT.

- Using the second strategy, in each step the two DCEL data structures, which contain the fewest curve segments, are combined. Therefore the number of curve segments of the emerging data structure has to be determined, as this data structure can be one of the two data structures with the fewest curve segments of the next step. This will be repeated until there exists only one data structure. This strategy will be named MERGE_SMALLEST.

- The last strategy works similar as the second one. The difference is that in every step always the DCEL data structures with the most curve segments are combined. This strategy will be denoted as MERGE_LARGEST.

Now these three strategies are applied to the TILED localizer to determine their impact. For this test the Cartesian, the inexact spherical (Kernel_sph_surf) and the exact Kernel_sph_vect_norm kernels are again used. For the two exact kernels the Gmpq number types with the Lazy_exact_nt option have been used. The results for this test are visual-

Combine strategies	Cartesian Lazy_exact_nt <Gmpq >	Kernel_sph_surf double	Kernel_sph_vect_norm Lazy_exact_nt <Gmpq >
MERGE_HEIGHT	15.290	13.920	132.690
MERGE_SMALLEST	4.591	3.905	46.730
MERGE_LARGEST	17.990	15.620	169.260

Table 7.5.: Complexity of the map overlays using different combine strategies; in seconds

ized in table 7.5. The different combine strategies have the same effect for all geometric kernels. The MERGE_SMALLEST method is by far the most efficient strategy followed by the MERGE_HEIGHT and the MERGE_LARGEST strategies. It can be argued with the map overlay complexity, why the MERGE_SMALLEST strategy is the fastest. The map overlay complexity is given by $\mathcal{O}((n+k)log(n))$, where n is the number of the

curve segments of both original data structures and k is the number of curve segments of the overlayed DCEL data structure. Next an overlay of three disjunctive DCEL data structures is considered (see example illustrated in 7.7), where $DCEL_1$ has n_1 curve segments, $DCEL_2$ has n_2 curve segments and $DCEL_3$ has n_3 curve segments. Additionally

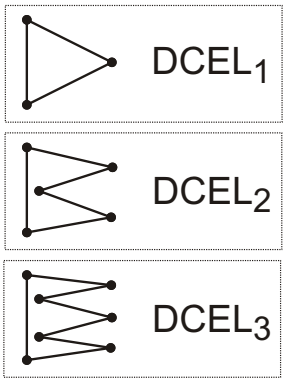


Figure 7.7.: Example of three different disjunctive DCEL data structure instances

a restriction is imposed saying that $n_1 < n_2 < n_3$. As the DCEL data structures are disjunctive, the complexity can be rewritten as:

$$\mathcal{O}((n+k)log(n)) = \mathcal{O}((\underbrace{n_1 + n_2}_{n} + \underbrace{n_1 + n_2}_{k})log(\underbrace{n_1 + n_2}_{n})) = \mathcal{O}((n_1+n_2)log(n_1+n_2)) \quad (7.1)$$

If the map overlay is applied to $DCEL_1$ and $DCEL_2$ and afterwards with $DCEL_3$ (MERGE_SMALLEST strategy), the following complexity will be achieved:

$$\mathcal{O}((n_1 + n_2)log(n_1 + n_2)) + \mathcal{O}((n_1 + n_2 + n_3)log(n_1 + n_2 + n_3)) \quad (7.2)$$

If the map overlay will instead be performed to $DCEL_2$ and $DCEL_3$ first and $DCEL_1$ (MERGE_LARGEST strategy) afterwards, this complexity will be achieved;

$$\mathcal{O}((n_2 + n_3)log(n_2 + n_3)) + \mathcal{O}((n_1 + n_2 + n_3)log(n_1 + n_2 + n_3)) \quad (7.3)$$

It can easily be seen, that the second terms of these complexities are equal, but the first term is for the first case smaller as $n_1 < n_2 < n_3$. Allthough both reaches the same asymptotic running times, using the first case will be faster

Although the MERGE_SMALLEST strategy achieves good results, an additional method has been implemented, which is able to write the combined final DCEL data structure instance into a file, which preserves the DCEL structure (see chapter 5.6.2). Now the running times of the TILED localizer which imports such a file (denoted with READ DCEL) and the running times determined before, where several map overlays using the

MERGE_SMALLEST strategy have been performed, will be compared. Additionally the importation times are added to the later running times, to provide a reasonable comparison. This mode will be called READ + MAP. The results are visualized in table 7.6. It can be seen that the importation of the final DCEL data structure instance, which

Building modes	Cartesian Lazy_exact_nt <Gmpq >	Kernel_sph_surf double	Kernel_sph_vect_norm Lazy_exact_nt <Gmpq >
READ + MAP	0.443 + 4.591	0.412 + 3.905	4.297 + 46.730
READ DCEL	0.713	0.212	0.405

Table 7.6.: Comparison of the running times between the two modes READ DCEL and READ + MAP

has been computed by another localizer instance, is much faster than importing the ATC-Sector definition given in an independent face representation and performing the map overlays afterwards. As the final DCEL data structures instance has been written to a file, which preserves the DCEL structure, this mode is for the two spherical kernels also faster than the importation of the ATC-Sector definition alone. The drawback of the READ DCEL mode is, that the final DCEL data structure instance must have been computed before with a localizer instance, that performs the building and the assigning of the ATC-Sectors.

7.3.3. LOCATE Phase

Now a closer look to the point location query times of the three localizers for all considered point location algorithms is taken. The Cartesian, the inexact spherical (Kernel_sph_surf) and the exact Kernel_sph_vect_norm kernel are again used. The Gmpq number type with the Lazy_exact_nt option is again used for the two exact kernels. The first results presented in table 7.7 visualize the location times of the LAYERED localizer.

# Algorithms	Cartesian Lazy_exact_nt <Gmpq >	Kernel_sph_surf double	Kernel_sph_vect_norm Lazy_exact_nt <Gmpq >
NAIVE	$3.101 \cdot 10^{-3}$	$2.114 \cdot 10^{-3}$	$7.617 \cdot 10^{-3}$
WALK	$5.898 \cdot 10^{-5}$	$6.398 \cdot 10^{-5}$	$2.579 \cdot 10^{-4}$
VERT	$1.070 \cdot 10^{-4}$	$6.298 \cdot 10^{-5}$	$3.139 \cdot 10^{-4}$
GRID	$2.599 \cdot 10^{-5}$	$1.999 \cdot 10^{-5}$	$6.298 \cdot 10^{-5}$
RIC	$7.500 \cdot 10^{-6}$		

Table 7.7.: Point location query time of the LAYERED localizer for different point location algorithms in seconds

It can be seen that the NAIVE algorithm is again the slowest localization strategy. It can also be observed that the VERT algorithm is slower than the WALK algorithm and again slower than the GRID algorithm. The reason for this is that the VERT algorithm has to determine the adjacent face of the landmark point, through which the curve segment to the query point lies, at first. The RIC algorithm provides again the best query time. The query times of the RIC algorithm for the spherical kernels have been omitted, as the localizer using that kernels requires too much memory ($\sim$ 800 Mbytes), as mentioned in section 7.3.2.

In the next table (table 7.8), the point location times for the TILED localizer are outlined. These times are equivalent to those of the OVERLAPPED algorithm, as

# Algorithms	Cartesian Lazy_exact_nt <Gmpq >	Kernel_sph_surf double	Kernel_sph_vect_norm Lazy_exact_nt <Gmpq >
NAIVE	$8.592 \cdot 10^{-3}$	$6.510 \cdot 10^{-3}$	$1.674 \cdot 10^{-2}$
WALK	$6.711 \cdot 10^{-5}$	$6.569 \cdot 10^{-5}$	$4.827 \cdot 10^{-4}$
VERT	$5.498 \cdot 10^{-5}$	$1.999 \cdot 10^{-5}$	$2.756 \cdot 10^{-4}$
GRID	$1.780 \cdot 10^{-5}$	$1.490 \cdot 10^{-5}$	$5.230 \cdot 10^{-5}$
RIC	$5.802 \cdot 10^{-6}$	$9.807 \cdot 10^{-6}$	$2.799 \cdot 10^{-5}$

Table 7.8.: Point location query time of the TILED localizer for different point location algorithms in seconds

they only differ in the BUILD and ASSIGN phase. Again the much slower behaviour of the NAIVE algorithm can be observed in table 7.7. Compared to the results of the LAYERED localizer, it can be seen that the NAIVE and WALK algorithms are faster for the LAYERED localizer, while the other algorithms achieve better running times for the TILED localizer. A possible explanation for these effects is that the DCEL instance of the TILED localizer is much more complex (contains more curve segments) than those instances of the LAYERED algorithm. As the complexity of the NAIVE algorithms is $\mathcal{O}(n)$ and $\mathcal{O}(\sqrt{n})$ for the WALK algorithm, this improvement of the complexity affects them much more than for example the RIC algorithm, which has a point location complexity of $\mathcal{O}(log(n))$. In contrast the LAYERED algorithm has at first to determine the appropriate DCEL instance, on which the point location algorithm should be applied, which increases also the point location time, whereas the TILED localizer needs not to perform this operation. Additionally it can be again observed that the VERT algorithm is much slower than the GRID algorithm.

7.4. Dynamic Localizers

Inside this section, a closer look to the performance of the two dynamical localizers, namely the Localizer_dynamic_2 (analyzed in section 7.4.1) and the Localizer_dynamic_3 (analyzed in section 7.4.2), is taken. As both classes have been derived from their static pendants, the BUILD and the ASSIGN performance is hardly the same. Therefore the LOCATE performance will only be examined. To get a meaningful evaluation, these tests consider the average location time for a single waypoint passing and also take in mind the time cost for a complete flight route for a specific aircraft. To avoid influences caused by the property of a specific flight route, these times will be derived as intersection of 26 different aircraft routes. Every time value is given in seconds. For each localizer (2-dimensional and 3-dimensional) two different tests have been performed:

- *Influence caused by the number of aircraft:*
  Using this test, reports the influence of the aircraft number on the location time of a specific aircraft route.

- *Influence caused by the number of waypoints:* This test determines, if the number of used waypoints has an impact on the location time along a specific aircraft route. For this purpose, the waypoint list of an aircraft will be reduced, by considering only every second, third, fourth, ... waypoint.

7.4.1. 2-Dimensional Dynamic Localizer

For this localizer, the results of the first test (*Influence caused by the number of aircraft*) will be presented at first. As mentioned in section 6.2, there exist several problems with the inexact spherical kernel. To avoid these problems, the *Kernel_sph_vect_norm* kernel with the exact *Gmpq* number type has been used for the following tests. This avoid failures against the location along a flight route, as an inexact kernel would cause a crash in this location strategy. The number of aircraft represents the parameter inside this test. As outlined in table 7.9, the number of aircraft has hardly any influence on the location time for a specific aircraft route respectively on the time for a *locateAlongRoute()* call.

For the next test the same spherical kernel has been used. So inside this test only the influence of the number of waypoints for a specific flight route is considered. The flight level of the 2-dimensional localizer has been set to 3050. As it can be observed in table 7.10, the number of waypoints along the flight route has significantly more influence on the location time than the number of aircraft. The number of waypoints in this table corresponds to the number of waypoints originally given for this test plus the number of crossing waypoints, which appear while following the flight route. The

# aircrafts	whole Flight-Route time	locateAlongRoute calling time
1	$4.813 \cdot 10^{-2}$	$7.121 \cdot 10^{-5}$
10	$4.835 \cdot 10^{-2}$	$7.154 \cdot 10^{-5}$
50	$4.846 \cdot 10^{-2}$	$7.170 \cdot 10^{-5}$
100	$4.868 \cdot 10^{-2}$	$7.203 \cdot 10^{-5}$
200	$4.873 \cdot 10^{-2}$	$7.211 \cdot 10^{-5}$
500	$4.885 \cdot 10^{-2}$	$7.227 \cdot 10^{-5}$

Table 7.9.: Location time for a specific waypoint ant the complete flight route of the 2-dimensional dynamic point locator

kind of waypoint selection	# waypoints	average Flight-Route time	locateAlongRoute calling time
all	676	$4.813 \cdot 10^{-2}$	$7.121 \cdot 10^{-5}$
every 2nd	343	$3.181 \cdot 10^{-2}$	$9.276 \cdot 10^{-5}$
every 3rd	232	$2.648 \cdot 10^{-2}$	$1.142 \cdot 10^{-4}$
every 4th	177	$2.401 \cdot 10^{-2}$	$1.360 \cdot 10^{-4}$
every 5th	143	$2.247 \cdot 10^{-2}$	$1.569 \cdot 10^{-4}$
every 6th	121	$2.143 \cdot 10^{-2}$	$1.771 \cdot 10^{-4}$
every 7th	105	$2.071 \cdot 10^{-2}$	$1.968 \cdot 10^{-4}$
every 10th	77	$1.934 \cdot 10^{-2}$	$2.518 \cdot 10^{-4}$
first & last	14	$1.775 \cdot 10^{-2}$	$1.307 \cdot 10^{-3}$

Table 7.10.: Location time for a specific waypoint ant the complete flight route of the 2-dimensional dynamic point locator

computation time for the average flight route shrinks, while the location time for a single *locateAlongRoute()* call rises. It can also be seen that the computation time for the average flight route shrinks at the beginning very fast, but is slowing down at the end of the table. Also for the extreme case, where only two waypoints are given, the running time only slightly differs from the case, where every tenth waypoint has been used.

7.4.2. 3-Dimensional Dynamic Localizer

For this section, the impact of the number of waypoints on the running time is considered, as the number of aircraft has hardly any influence on the location time (see table 7.9). It can be observed in table 7.11, that the same behavior as in the 2-dimensional case (see table 7.10) occurs. So the running time for the whole flight route shrinks at the beginning very fast and slows down for the later cases. The case, where only the first and the last waypoint are used, has been omitted, as this observation is similar to the 2-dimensional case, because it will only be traversed through the ATC-Sectors at flight

kind of waypoint selection	# waypoints	whole Flight-Route time	locateAlongRoute calling time
all	697	$4.769 \cdot 10^{-2}$	$6.846 \cdot 10^{-5}$
every 2nd	363	$2.951 \cdot 10^{-2}$	$8.126 \cdot 10^{-5}$
every 3rd	251	$2.275 \cdot 10^{-2}$	$9.077 \cdot 10^{-5}$
every 4th	196	$2.055 \cdot 10^{-2}$	$1.049 \cdot 10^{-4}$
every 5th	162	$1.868 \cdot 10^{-2}$	$1.152 \cdot 10^{-4}$
every 6th	139	$1.703 \cdot 10^{-2}$	$1.222 \cdot 10^{-4}$
every 7th	124	$1.670 \cdot 10^{-2}$	$1.348 \cdot 10^{-4}$
every 10th	95	$1.549 \cdot 10^{-2}$	$1.626 \cdot 10^{-4}$

Table 7.11.: Location time for a specific waypoint ant the complete flight route of the 3-dimensional dynamic point locator

level 0. Additionally the location times for the 3-dimensional case are smaller than those of the 2-dimensional test. The reason for this is that the ATC-Sectors for higher flight levels have a less complex shape and have larger scale. It can also be seen that the dynamic localizer is about 3 times slower than the TILED localizer.

Chapter 8.

Conclusion

During the work on this project several localizers have been programmed, which are able to locate aircraft and to follow aircraft along their flight routes. To do so, several point location algorithms have been modified so that they can be applied to these localizers. Additionally a new location algorithm for the dynamic localizer has been developed. Furthermore the point location algorithms have been adapted to spherical kernels, to provide an accurate point location. For this purpose, several kernels have been developed, which are able to perform fast but inexact and slower but exact point locations.

8.1. Recommendations

Inside this section a short overview of when to use which localizer is given. Therefore only the 3-dimensional localizers will be considered, as this usage is more realistic. In the simple case, where only less point location queries are needed, the best choice would be to use the LAYERED localizer with the WALK point location algorithm and the inexact spherical kernel. Another possibility is to take the TILED localizer, where the overlapped ATC-Sector definition is build and saved. Now this definition can be loaded with the inexact spherical kernel and applied to the queries. In the case, where point locations are heavily used, the TILED localizer with the GRID or RIC point location strategy should be used. As the drawbacks of the inexact spherical kernel with intersections has been demonstarted in section 6.2, it is recommended to use the exact spherical kernel during the BUILD and ASSIGN phase. Then this overlapped definition can be saved and loaded with the inexact spherical kernel to achieve a much more performant LOCATE phase. If it is needed to know the exact ATS-Sector transitions, the dynamic localizer should be used. Note that the dynamic localizer requires the exact spherical kernel.

8.2. Further Work

Similar to other projects, there is always the possibility for further improvements and ideas:

- One of this ideas would be to include more point location algorithms in this implementation and to evaluate their advantages and disadvantages in order to their location times. One of this point location algorithm, which guarantees an optimal query time, is the *Optimal Point Location in a Monoton Subdivision* from Edelsbrunner, Guibas and Stofli (EGS86). As indicated in the name, this algorithm requires a monotone subdivision ((dBvKOO00) pages 49-55). Another Algorithm is the Slab decomposition algorithm ((dBvKOO00) pages 122-124) which also guarantees a $\mathcal{O}(log(n))$ query time. In detail two binary searches are needed, which would be very efficient. The drawback of this method is, that it requires $\mathcal{O}(n^2)$ space.

- An example for an improvement would be to distribute the point location queries to several processors. One possibility would be to use the LAYERED localizer and assign each layer to a different processor.

- Another idea would be to use instead of a 2-dimensional data structure combined with a 2-dimensional point location algorithm a 3-dimensional data structure with a 3-dimensional point location algorithm. As such a 3-dimensional point location algorithm is much slower than a 2-dimensional one, the development of a new point location algorithm is needed, which take into account, that the ATC-Sectors have only an upper and a lower height boundary and describe therefore a simpler case than a general 3-dimensional subdivision.

Appendix A.

External program libraries and programs

A.1. CGAL

CGAL is an abbreviation for Computational Geometry Algorithms Library (CGA07). The goal of this open source library is to provide efficient and reliable geometric algorithms, which have been programmed in C++. The CGAL contains algorithms for triangulations, boolean operations on polygons, geometry processing, arrangement of curves and many other subjects. The arrangement of curves algorithms are the main one, which have been usefull to obtain the goals of my master thesis. This arrangement of curves do not only contains curves, furthermore they contain the 0-dimensional vertices, which are proportional to the geometric points, the 1-dimensional edges, which are the topological equivalent to curves, and the 2-dimensional faces. The most algorithms need some restriction to the shape of the used curves. They call this property x-monotone, which has been explained in chapter 6. All the algorithms are working on geometric object data structures like point and segment classes. As mentioned in spherical kernel chapter, some of these classes have been overwritten to provide point location on the earth surface. One major feature inside this library is its possibility to configure the algorithms either to make them work faster or more secure. The secureness will be provided by enabling the precondition and postcondition CGAL-macros, which perform validity tests and cause either an program abortion or throws an Exception in the case of error.

A.2. CppUnit

Beside unit-tests with JUnit, there exist several implementations for many programming languages (CPP06). CppUnit is a framework to provide software tests for the program-

ming language C++. To test a specific part of the software, a new class has to be written, which is extended by miscellaneous CppUnit macros, which will be registered as new test. Inside the methods of this test class, the methods of the testing classes will be called in such a way, that they return a correct output for a predefined input. During the programming of my localization module, this library has been used to achieve a more errorless software.

A.3. OMNeT++

OmNeT++ is an object-oriented modular discrete event network simulator (Omn08). This simulator stand under the Academic Public License, which allows it, to use this simulator free for academic teaching and research. This simulator is usefull for amongst other purposes: traffic modelling of telecommunication networks, modeling queuing networks and protocol modeling. A specific OMNeT++ example consists of several modules, which communicate via meassage passing to each other. Messages can be send directly to their destination or through gates or along predefined paths. Using module specific parameters enables customizing the functionality of a module and to design the topology of the model. The modules of OMNeT++ are hierarchically ordered. The modules at the lowest level are called simple modules and will be programmed in C++ to define their behavior. Modules on a higher level can now be used to connect lower leveled or simple modules together to achieve the required simulated network. This framework is designed to work on various operating systems (Windows, Unix like OS) and machines. With OMNeT++ it is also possible to make parallel distributed simulations, using MPI or named pipes. There exist several open source simulation models for internet purposes, which implement known protocols like IP, IPv6, MPLS and others. One of this models is the INET network, which is suited for wired and wireless networks. This framework contains beside the IP and the UDP/TCP protocols also the Ethernet, IPv6, RIP, OSPF and other protocols. Based on this module, the FACTS project has been implemented.

A.4. NASA Worldwind Java

NASA Worldwind Java is an application, which allows the user to zoom from outside to any place on the earth and has been developed by the NASA (wwj08). Amongst other applications like google earth, this program is able to be linked against other programs. In FACTS, this tool has been used to provide the GUI for the simulator and has been linked together with the OMNeT++ model.

A.5. SkyView2

SkyView2 (Sky07) is a free program charged by EUROCONTROL and is used for demonstration of geo-spatial interoperability in support of Aeronautical Information Management (AIM). The main features of it are:

- providing several layers to visualize aircraft waypoints, airports, country borders, acc and ATFM sectors (equivalent to ATC-Sectors)

- providing the ability to include multiple geo-spatial information sources, based on OpenGIS (Ope09) and the ISO19100. For example OpenGIS has been worked out by the Open Geospatial Consortium (OGC) to provide a general standard for exchanging geo-spatial informations. The included standards are freely available to provide abstract descriptions as well as detailed specifications of the implementation of services.

- Additionally its possible to extract the data for this layers mentionend before using the GML (GML04) XML scheme.

The data used in the contained layers provided by SkyView2 have been produced by EUROCONTROL. For this reason, the ATC-Sector definition has been extracted from this program to achieve a realistic simulation.

List of Figures

List of Tables

Bibliography

[BO79] Bentley, J. L. and T. A. Ottman: *Algorithms for reporting and counting geometric intersection.* IEEE Trans. Comput., 28(9):643–647, 1979.

[CGA07] *CGAL User and Reference Manual; Release 3.3.1*, August 2007. http://www.cgal.org/Manual/3.3/doc_html/cgal_manual/contents.html.

[CPP06] *Cppunit.* http://sourceforge.net/projects/cppunit, jun 2006.

[CRS96] Courant, R., H. Robbins, and I. Steward: *What Is Mathematics?: An Elementary Approach to Ideas and Methods.* Oxford University Press Inc, USA, 2rev edition, September 1996.

[dBvKOO00] Berg, M. de, M. van Kreveld, M. Overmars, and O.Schwarzkopf: *Computational Geometry.* Springer-Verlag, 2 edition, Februar 2000.

[DLR] *DLR - Deutsches Zentrum fuer Luft- und Raumfahrt e. V.* http://www.dlr.de.

[DPT01] Devillers, Olivier, Sylvain Pion, and Monique Teillaud: *Walking in a triangulation.* In *SCG '01: Proceedings of the seventeenth annual symposium on Computational geometry*, pages 106–114, New York, NY, USA, 2001. ACM.

[EF06] EUROCONTROL and FAA: *Communications Operating Concept and Requirements for the Future Radio System*, jun 2006. http://www.eurocontrol.int/communications/public/ standard_page/General_FCI_Library.html.

[EGS86] Edelsbrunner, Herbert, Lionidas J Guibas, and Jorge Stolfi: *Optimal point location in a monotone subdivision.* SIAM J. Comput., 15(2):317–340, 1986.

[Fil93] Filler, Andreas: *Euklidische und Nichteuklidische Geometrie.* Spektrum Akademischer Verlag, 1993.

[GML04] *OpenGIS Geography Markup Language(GML) Implementation Specification*, 3.1.1 edition, Februar 2004.

[GMP09] *Gnu multiple precision arithmetic library*. http://gmplib.org/, jan 2009.

[Hec94] Heckbert, Paul S.: *Graphics Gems IV*. Academic Pr, har/dsk edition, April 1994.

[HH06] Haran, Idit and Dan Halperin: *An experimental study of point location in general planar arrangements*. In *Proc. ALENEX 2006*, 2006.

[Koe83] Koecher, Max: *Lineare Algebra und analytische Geometrie*. Springer-Verlag, 1983.

[MH02] Moller, Thomas and Eric Haines: *Real-Time Rendering*. AK Peters, Ltd., 2 edition, July 2002.

[Omn08] *Omnet++*. http://www.omnetpp.org/, dec 2008.

[Ope09] *Nasa worldwind java*. http://www.opengeospatial.org/standards, 2009.

[PS85] Preparata, Franco P. and Michael Ian Shamos: *Computational Geometry - An Introduction*. Springer-Verlag, 1985.

[Sei91] Seidel, Raimund: *A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulation polygons*. Comput. Geom. Theory Appl., 1:51–64, 1991.

[SH76] Shamos, Michael Ian and Dan Hoey: *Geometric intersection problems*. sfcs, 0:208–215, 1976.

[Sig77] Sigl, Rudolf: *Ebene und sphaerische Trigonometrie mit Anwendungen auf Kartographie, Geodaesie und Astronomie*. Herbert Wichmann Verlag, 1977.

[Sky07] *SkyView User Manual*, 2.4.0 edition, Februar 2007.

[Tru93] Trudeau, Richard J.: *Introduction to Graph Theory*. Dover Publications, 1993.

[Veb05] Veblen, Oswald: *Theory on plane curves in non-metrical analysis situs*. Transactions of the American Mathematical Society, 6:8398, 1905.

[wwj08] *Nasa worldwind java*. http://worldwind.arc.nasa.gov/java/index.html, may 2008.

Danksagungen

An dieser Stelle bedanke ich mich beim DLR, für die Möglichkeit diese Diplomarbeit anfertigen zu dürfen. Hierbei gilt mein besonderer Dank Herrn Dr. Michael Schnell, der mir ein Übermaß an Verständnis und Unterstützung zukommen ließ. Weiters will ich mich bei Herrn Dipl.-Ing. Christian Bauer für seine fachliche Beratung und bei Herrn Dr. Harald Grossauer für seine organisatorischen Unterstützung bedanken. Eine sehr große Hilfe, ohne die ich diese Diplomarbeit nicht vollenden hätte können, war mir meine Familie und meine Freundin Simone. Mein Dank gilt auch meinen Freunden Florian Haider und Thomas Kluckner für ihre Sorgfalt während des Korrektur lesens. Schlussendlich gilt mein Dank auch Frau Dr. Claudia Thaler-Wolf und Herrn Ao.Univ.-Prof.Dr. Gerald Zernig-Grubinger für ihre mentale Betreuung.