Off-line processing of terrains structured by semantical hierarchies

D. Michelucci - J-M. Moreau
LISSE-ENSM.SE[1]

January 26, 1996

Technical report **RR 96.2**

[1]Address: 158, cours Fauriel, F 42 023 Saint-Étienne. Email: `moreau,micheluc@emse.fr`

Abstract

*This report describes efficient off-line solutions for processing $2\frac{1}{2}$ terrains structured by semantical hierarchies. It summarizes the results of over 7 years of joint research in the field of off-line terrain processing in Computational Geometry (*ENSM.SE*) and expertise in flight simulator technologies (*SOGITEC[1]*).*

From the standpoint of Computational Geometry, some of the main issues in terrain processing in presence of semantical hierarchies are: Dealing with heterogeneous data, solving precision issues, producing exact planar graphs, organizing hierarchies, and triangulating terrains with respect to the constraints it contains. These topics are interesting as such, but also have an influence on the design of applications that use geographic data.

It is hoped that this report will be useful to the community of researchers in Geographic Information Systems, as a basis for reflexion and future work.

Résumé

*Ce rapport décrit des solutions efficaces hors-ligne pour le traitement de terrains $2\frac{1}{2}$ structurés par hierarchies sémantiques. Il synthétise les résultats de plus de 7 ans de recherches communes dans le domaine du traitement hors-ligne de terrains en Géométrie Algorithmique (*ENSM.SE*) et d'expertise en simulateurs de vol (*SOGITEC*, cf. adresse en note de bas de page).*

Du point de vue de la Géométrie, les sujets d'intérêt principaux du traitement de terrains en présence of hiérarchies sémantiques sont: le traitement des zones hétérogènes, la résolution des problèmes de précision, la construction de graphes planaires exacts, l'organization de hiérarchies, et la triangulation de terrains asujettis à des contraintes géométriques. Ces sujets sont intéressants en tant que tels, mais ils ont aussi have une influence sur la réalisation d'applications exploitant des données géographiques.

Nous espérons que ce rapport permettra à la communauté des chercheurs en Systèmes d'Information Géographique de trouver une base de réflexion et de travaux futurs.

[1]Address: ZI Champ Niguel, 35 140, Bruz (France)

Contents

1

Goals and scope

1.1 Introduction

The task of designing large and accurate geographical databases for numerical or geographic processing is a long, tedious and time-consuming one, and the need for it is ever growing. Today's machines allow fast processing of larger and larger databases, but the human task of setting up all the details relative to each and every object remains one of the major problems, together with the handling of interactions between different objects at different times.

Computational Geometry is a new, developing field, in which problems are tackled from the point of view of geometry. The problems and solutions presented in this paper should give readers from the *G.I.S.* community some insight on how some of their own questions might find answers through the emergence of well-adapted techniques from pure geometry.

The rest of this paper is structured as follows: Section 1 describes the characteristics of a typical application to process a semantically-structured terrain. Sections 2 and 3 present, respectively, the objects that will be manipulated, and the data structures to manipulate them. In Section 4, we concentrate of precision issues, and how to solve imprecision problems with the help of a powerful paradigm. Section 5 describes the notions of hierarchy and priority, and how the semantics of priority may be implemented. Section 6 explains how river or road systems may be constructed. Section 7 presents one robust technique to make graphs planar, and the next section concentrates on how to exploit this information to build a complete "map" of the database. Section 9 outlines the principle of the divide-and-conquer construction of the Delaunay triangulation of a constrained graph, and the final section concludes on possible extensions. The ideal application we shall consider for formal support will have to perform the following basic tasks:

1. Input of data: Data come in as streams of information related to each object, including general attributes (class of the object, serial number, texturing attributes, *etc.*), and then specification of shape and altitudes. Special classes might require more details, as needed.

2. In the previous phase, objects are just lists of vertices or segments. It is now necessary to organize them into proper "regions", which is only possible if the underlying segment-vertex graph has been made planar, and "invalidated" sections have been eradicated (a process more commonly referred to as "clipping").

3. Any further operation on the database (be it numerical, as it is the case for fluid dynamics, or finite-element applications, or else geometrical –as in terrain visualization– or both geometrical and numerical, as in the *G.I.S.* or in the Simulation domains) require that the complete graph be triangulated. This triangulation is said to be *constrained*, in the sense that no final triangle is allowed to cross over any *constraint* present in the graph, and it is well-known that the constrained *Delaunay* triangulation is the one giving the best results for virtually all applications.

There might be plenty of other tasks, but these constitute some sort of least-common-denominator, and may be augmented without altering the basic principles enumerated here.

1.2 Handling heterogeneous data

Today's machines accept larger and larger databases. It is now frequent to merge data from various sources without having the (human) possibility to check their compatibility. Data are said to be highly heterogeneous. As a result, a lot of information is either redundant or over-detailed, and sometimes needs to be filtered out automatically.

The community of researchers in the closely-related fields of Computational Geometry and Geographic Information Systems are still debating one crucial issue regarding this problem: Should we take basic information (raw data) for granted, and derive all knowledge from Task 2 (making the graph planar, whatever the cost), or do we try to adapt the initial input to solve ambiguities introduced by *noise* in the data. The former process graphs as they are, even if they have to declare some of them ill-designed; the latter try to adapt the topology of the initial data by using specific induction laws, inherited from cartography or other deeply-rooted, sensible, field expertise.

Far from trying to settle the debate, we shall stand on the side of the former, preferring to trust the data for what they are, and using up as much information as possible to process the *true* topology of the database. Of course, there is some irony in this "*true*", since we all know that there definitely is some inaccuracy in the measurements made to collect initial data. But the philosophy behind this is more profound than it seems: Re-entrance is the stake!

It would be folly to let an application use information from every source but its own results. Re-entrance is the basics of modeling: Start with an initial version, process and improve it, save it, over and over (possibly using indications from previous executions to simplify data, or suppress problems), until stabilization. A typical example of this is local reconstruction: Take a section of a large (already processed) database, and modify it. If efficiency is the issue (and it surely is, when millions of vertices and edges are considered!), reprocessing the whole database is out of the question: Uniformity of processing is the only guarantee that the revised local section will exactly "fit in" the former database (provided the local section was correctly bounded by edges of the whole database).

Adapting initial graphs may only yield non-matching versions of otherwise perfectly coinciding collateral sections, if processed independently or at different times!

Of course, the price one has to pay for this uniformity is infinite precision, a notion unheard of by the zealators of the adaptive technique. We shall come back to this issue, showing that precision may be obtained with far less expense than would seem necessary...

1.3 Off-line *versus* on-line

Off-line applications read their entire input, and *then* perform all necessary computations, only allowing reviewing the input/output by displaying them, and maybe interacting passively with the objects (moving in databases, zooming, *etc.*).

On-line applications allow insertion and deletion of data at execution time (altering data being performed by deletion followed by insertion of new values). Such applications also allow displaying results and inputs, naturally.

Off-line *randomization* is a very powerful technique, which ensures that the algorithms will behave in a very stable, close to optimal, non-deterministic fashion depending only on the outcome of a random generator. But this becomes more hazardous when dealing with real on-line (randomized) applications, where the next action is not ruled by such a generator, but by the outcome of the previous one: To simplify, reading the contents of a structured database is nowhere near random!

In terrain simulation, it is possible to separate the construction phase (when the database is set up) from the subsequent use made of the database. Furthermore, because of the complexity of

interactions between objects in a truly hierarchical geographic database, and because the effects of such interactions might be different at different times in the execution –as we shall later see– off-line (deterministic) applications are particularly well-adapted to terrain simulation. Dealing with such complex variations is facilitated by having a "batch" processing of events: First investigate all events pertaining to a peculiar stage of evolution in objects, then consider next stage, collect all relevant evolution of objects, and so on. On-line processing would imply to have a complete knowledge, at any time, of all the versions of each object throughout its life-span, a tantalizing task, indeed!

With the advent of randomized algorithms, this situation might evolve. However we shall concentrate on the off-line aspects of the problems in this paper. One good reason is that most applications from the realm of terrain processing work in two stages:

1. Compute and construct the terrain
2. Fetch a section and either visualize it, or perform local (numerical, topological, logical, geometrical, *etc.*) computations on it, but do not modify data.

This is surely true of the finite-element method, since such a technique might be described as inferring the behaviour of a given particle at a particular moment from its previous position. Flight simulating applications only use partial sections of the face-database the plane or vehicle evolves in. Guessing the evolution of water levels in flood prediction applications follows the same two-part scheme: First construct the database, then use it to navigate in it. Roughly speaking, moving inside a database does not mean altering its contents, and only applications requiring to modify data interactively should call for true dynamicity.

Still, randomization might be a good solution to allowing dynamicity in the processing of large databases in the future years.

2

Objects

Our ideal application will manipulate objects. This vocable is not to be mistaken with its homonym from the field of object-oriented programming. Incidentally, the underlying programming for the algorithms presented in this paper will indifferently be *C* or *C++*, depending on the reader's sensibility to the questions of encapsulation and overall design techniques.

Basically, an object is a 3-D polygon, a 3-D polygonal chain or segment, or else a 3-D point. These objects live in 3-D space, but we're only interested in their projection onto a plane, namely the so-called xy-plane. z will represent altitudes. Hence, an object may be symbolized through a list of (1 or more) vertices. A (closed or open) N-gonal chain requires exactly N (x, y, z) triples. In the case of a closed N-gon, the first vertex must be linked to the last with an edge. Furthermore, it is important that the successive (projected) vertices of a given N-gon be listed in, say, counter-clockwise order, so the interior of the object always lies to the left of a hypothetical observer walking along its boundary. From now on, we shall only consider the projected version of the objects on the reference xy plane.

In fact, a given object may have several connected components. This is only a question of syntax in the input. All connected components may share the same attributes as those of the object, or not: The pillars of the same bridge might differ in appearance.... For example, the specific attributes of one given connected component might override that of the general object, if need be.

Attributes are listed in whatever form is preferable, and are references to specific entries in a general look-up table, as far as we are concerned. Clearly, some information about priority might be retrieved through this information. What is important is to know that objects may explicitly refer to their own attributes, a constant-time operation. Among these is the *class* the object –and *all* its connected components– belongs to. Typical classes are: Surface objects (fields, water, woods), Solid objects (buildings, villages,*etc.*), Linear objects (roads, rivers, motorways), Domains (containing all other objects), Altimetric points, and *Constraint* lines.

This final class, which includes contour lines, is used to influence the *planimetry* of terrains, and to impose their altitudes to them. For example, if a constraint line crosses the boundary of a field, the altitude of their intersection point is interpolated along the z equation of the constraint line. Constraint lines, therefore, may be regarded as a designer's privileged way to alter natural features of the terrain. As an application, constraint lines will be useful to flatten out the ground on either side of a planned motorway, the motorway itself only imposing its altitude along its path.

Another important attribute is the *serial number*. This integer might be the rank of an object within its class, the rank of an object in the overall list of objects, or a rank in any other appropriate ordering. Actually, this serial number might even be a *list* of serial numbers (one within its class, one for the global rank, and so forth). This is implemented by using an array for the serial information.

Polygonal chains, or segments, are primarily constraint lines. These are generally open, but may be cycles if the first and last of their vertices coincide. However, they do not define regions, contrary to polygons.

There might be as many classes of closed objects as needed in one particular application. We shall concentrate on two major subclasses:

2.1 Static and dynamic objects

Static objects are completely defined at input time, and don't normally change shape in the sequel. Dynamic objects are only partially defined at input time, and are dynamically constructed at run-time. Static classes include the Domain, fields, houses, buildings, blocks, villages, trees, forests *etc.*. Some of these classes may have object instances coming from other pieces of software: It is quite frequent for buildings or houses to be designed using extern specialized modules. This makes no difference here, since it is only the imprint of such objects on the xy-plane we shall be interested in.

Dynamic objects are frequently used to model roads or rivers. More generally, they represent *linear* objects, defined by successive planar sections. Each linear object has a *medial* axis (succession of segments in space), and possibly a profile. A profile is defined as a list of segments on either side of the medial axis. For instance, a motorway may be seen as a symmetric combination of verges, ditches, traffic lanes and (half) a central reservation. This may be modeled by giving the corresponding succession of 3D segments (orthogonal to the current medial axis section), which will have to be "expanded" along successive sections of the medial axis.

Where dynamic and static objects interact, there might be problems, from the point of view of the database designer: As it is not known at design time exactly what the expanded version of a given dynamic object will look like, the predicted intersection point between it and a neighbouring static object might be slightly offset compared to its actual position, which might, in turn, result in micro-edges. We shall give some hints on how to solve this problem later.

2.2 Hierarchies and priority

The notion of class is crucial for that of priority between objects, and hence for that of hierarchy: Classes are naturally ordered using priority relationships, according to the laws edicted by the user of the application. It must be understood that these relationships are dynamic in two fashions:

1. They may vary with time: one given class may precede another at a certain time of construction, while the order will be reversed later.
2. They may vary according to the semantics of the situation. For instance, the order presiding over the intersection of two segments (from two different objects), might be different from the order presiding over the operation of clipping two objects. To give but one example, if the objects are all supposed to lie inside a maximal box (what is usually referred to as the *Domain*), intersecting this bounding box is correct for any object (some of its contents is inside the bounding box, after all), while no (part of any) object is allowed to lie outside it.

2.2.1 Semantic rules

In normal conditions a field should give way to a country road, a country road to a major one, a major road to a motorway; on the other hand, a major road might exceptionally yield to a canal, or to another major road, depending on pre-established rules, that constitute a set of fixed and well-established common-sense regulations.

Such rules pertain to the subject of "cartographic" hierarchy. Another set of hierarchy rules is related to altimetry: Since two-dimensional databases (the only ones we shall be dealing with in this paper) require for any point to be processed to have exactly one altitude value (no two points in space may project to the same point in the plane), it is necessary to break ties between adjacent objects sharing points with different altitudes. For instance, two adjacent buildings sharing one plane edge will have to be made compatible, by lowering one or the other, so their altimetries coincide.

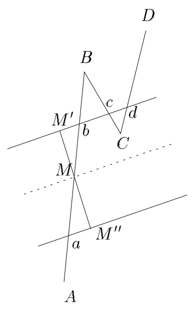

Figure 2.1: *Constraint lines and linear objects.*

Inclusion relationships induce a third set of hierarchy rules: For instance, the inclusion of one object in another one may result in two different outcomes. If the inner object has "greater priority" than the other, it is allowed to be seen in the final output; otherwise, it will have to be removed from the output.

Priority semantics might even vary with time: Consider Figure 2.1, where a constraint line in three segments meets one section of a (single segment-profiled) linear object. In a first stage, the intersection of one section of the constraint line with the medial axis of the unexpanded linear object is detected (M). This produces two orthogonal projections (M', and M'') through which the expanded version of the linear object is made to pass. At that time of the process, constraint lines impose their altitudes to linear objects: M will have its altitude interpolated using the constraint line, and both M' and M'' will be given this altitude.

On the same figure, small letters indicate indesirable interactions between the constraint line and the *expanded* linear object. All of these are artefacts and should not be taken into account in the final output: Once linear objects have been expanded, constraint lines must lose all influence on them.

There might even be subtler variations on the theme of hierarchy: one might wish to override locally the previously accepted effects of hierarchy because the user decides it is so. For example, fields usually give way to motorways, but what about verges and grass sections between opposite lanes? Along the same line, it is very unadvisable that a house should be allowed to stand right in the middle of a motorway, except if it is a toll booth, or any other object belonging to a specialized version subclass of the class "house", such as a lamp-post or even a tree!

Basically, semantic hierachies are extremely varied and numerous, and any application might have good reasons for introducing more of them than were planned in the first place.

Applications should naturally allow some flexibility in handling objects of different kinds, but generally do not manage to offer a complete set of primitives to solve the problem at hand. This paper attemps to describe the geometric primitives that one should reasonably wish were available in the designing and processing of real-scale and realistic terrains.

3

Data structures

The first issue in this section is how to store the information relative to the underlying graph, both internally and externally.

3.1 Vertices

The basic structure is the `Vertex`, with three co-ordinates and additional fields, as required. For instance, it is crucial to record any given point only once. The natural structure allowing this is the well-known AVL balanced search tree ([10]), requiring two extra pointers on left and right children. Therefore, `Vertex` may be seen as a node in the AVL structure. If needed, this structure may be augmented to allow specific operations, such as retrieving all edges through one given vertex, although this is usually taken care of *via* the next structure.

This AVL structure is also augmented with a doubly linked list mechanism to allow its traversal in *lexicographical order* (also called xy-order: Compare x first, then y if equal).

3.2 Edges

Once the graph has been made planar, its elements may be easily accessed using the traditional *winged-edge* structure ([2]), `WEdge`. In short, the information support is the edge, and each edge has two vertices, represented by an array of two pointers on `Vertex`, and carries four "wing" pointers on `WEdge`, two for each vertex, as depicted on Figure 3.1.

In addition, the `WEdge` structure has to convey the notion of "direction", *i.e.* a two bit unsigned integer, with value 0 for one way, 1 for the other.

This structure is optimal for space considerations, but has the drawback of being slightly cumbersome to use, because of the direction. For instance, suppose that a given edge e has A and B as endpoints, and that we wish to move from it to one of its neighbours, say next clockwise edge around B; If current direction of traversal is d, with vertex $v[d] \equiv A$, and if 0 stands for clockwise, the next edge is

$$e = e \rightarrow wing[1-d][0].$$

But now, which vertex represents B? It is only possible to know by a test, such as:

$$d = (e \rightarrow endpoint[1] == B);$$

All navigation in the graph is punctuated by such instructions.

Another well-known structure, the *planar map* ([9]), suppresses this problem, while requiring that the size of the graph be known from the start –another cumbersome restriction! Still, some sort of compromise between both structures is possible, as we shall see shortly.

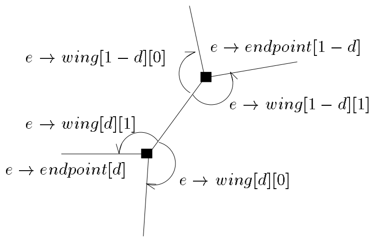

Figure 3.1: *Typical winged-edge structure,* `WEdge`

The principle of the planar map is the following: supposing we know the (planar) graph has M edges (and therefore at most $2M$ vertices), allocate an array of exactly $2M$ cells, each representing one "half- edge" with an implicit vertex (its origin), and containing a pointer to its endpoint vertex, one pointer to the next clockwise edge around its origin, and finally one pointer to the next counter-clockwise edge around its origin. These pointers are just indices in the array. To complete the structure, each planar map half-edge cell has an implicit "dual" half-edge cell, heading in the direction of the latter's origin. This implicit cell is easy to find: It is located symmetrically at the other end of the array. For instance, the dual for cell at index 0 is located at index $2M - 1$, that for cell at index 1 is located at index $2M - 2$, and so forth. Both half-edges combine to yield all the information relative to the complete edge, for the whole array uses the same space as the winged-edge structure.

Note one may increase or decrease the size of the underlying array dynamically, as this is known to be possible for hash-tables ([10]), while preserving *amortized* constant-time creation/destruction operations over all insertion/search/deletion operations performed by the application.

A very original consequence to the planar map structure is that it gives a straighforward means to write the structure of the graph into a file, for later use. Supposing the `Vertex` structure has an extra field for the rank of vertices in the lexicographical order (sort by x, and break ties using y), write the number of vertices of the graph in the file, then all vertices (together with their associated data) in lexicographical order; next write the number of edges of the graph, and finally write the contents of the array, using vertex indices instead of pointer adresses to the `Vertex` structure. Upon re-opening the file, it will be possible to allocate the space required for both the vertex and planar map structures, to use a recursive algorithm to reconstruct the *AVL*, and a straightforward iterative algorithm to fill the half-edges of the planar map array.

We have said that both structures were optimal in space, the first one being ideal for dynamic situations, although slower with regard to access, the other being optimal for static situations (the size of the graph is not supposed to go beyond a possible extra room buffer allocated at the start), and optimal with regard to access. The best solution might be to accept to waste a little space and achieve optimality of access in either dynamic or static situations! The price to pay is to remove half-edges from their array habitat, and to augment them with an *explicit* pointer to their dual half-edges. If internal memory is not the issue, this is by far the fastest and most practical solution to navigating in the graph, while the algorithm for storing graphs represented with this structure is slightly more far-fetched, although still quite reasonable.

3.3 3D-Faces

There is another issue to consider when choosing the adequate structure for edges: Original 3D-faces must also be accessible from them. Access is made possible by using two pointers to the

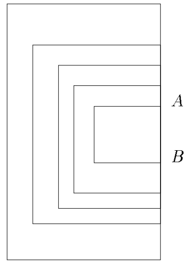

Figure 3.2: *Piling up of faces on one side of the same edge.*

faces on either side of any edge. The real problem regards the number of faces that may have to be accessed, for each side.

The problem with 3-D faces is that more than one polygonal face might pile up on either side of any edge[1]. Consider nested squares, all sharing one edge, as on Figure 3.2.

After the graph on Figure 3.2 has been made planar, there are 5 faces piled up to the left of edge AB. Hence, it is necessary to maintain another structure to keep track of all the faces lying on either side of a given edge. Such a structure is familiar to computer scientists, and is called a *priority queue* ([10], [16]). A priority queue allows inserting any element with a "priority key", as well as retrieving and deleting the element with highest (or alternatively, lowest) rank in the priority order, in optimal time proportional to the logarithm of its size. There is even a well-known data structure for supporting queues: the heap.

Dynamic heaps may be implemented with a binary structure which will be crucial in the rest of the paper: the *leftitst tree* ([16]). Although we have no space here to fully describe the mechanisms of such a structure, let us just say the following:

- A leftitst tree is a dynamic structure, supported by a rooted binary tree, with one basic optimal primitive, named `Combine`.

- This primitive takes two leftist trees as arguments and returns another leftist tree, combining the nodes of both, with no extra space and in optimal time proportional to the logarithm of the arguments size.

- To insert a new element in a (possibly empty) leftist tree, allocate a new node (by essence a leftist tree), and `Combine` the new node with the former leftist tree.

- Since every node in a leftist tree satisfies the so-called "heap property" (the key in every node ranks higher than any key in both its subtrees), the extremal element of a set represented by a leftist tree is always located in the root.

- To remove this element, delete the root and `Combine` its two leftist tree children.

So, the basic structure of a face is a priority queue holding a pointer to an object. When an edge is encountered in the data, it is explicitly associated with the current polygonal face or with the current polygonal chain. In the first case, it is straightforward to update its faces: one of them is `NULL`, the other refers to the current object. Finding out which face is not void follows

[1] $2\frac{1}{2}$-D terrains get their name from this piling up of 3-D faces. The action of clipping will reduce multiple 3D-faces to unique 2D-faces, an action only possible after the graph has been made planar. We will thefore use the same word, *faces*, to designate both notions, since there is no possible confusion.

directly from the orientation of the edge and the fact that boundaries are always traversed counter-clokwise. In the other case (a linear object or a constraint line), its "left" face is systematically set to point to the current object.

Since objects are read independently, adjacent faces share edges which must be eventually merged into a unique one. For reasons to become apparent later, the best solution is not to store edges in a balanced tree structure, which would have been ideal for preventing duplicates. Edges will themselves be stored in priority queues, implemented with leftist trees again. The nodes of edge queues are made up of any of the three data structures mentioned earlier, augmented with two pointers to leftist tree children.

It may now become apparent that *C++* is the ideal language to implement all these structures, although *C* was the language used in the work presented here, for industrial reasons. Still, the template and heritage mechanisms of *C++* should do wonders here, since most structures used are binary trees with slight variants for priority queues or *AVL*.

3.4 Polar angles

As mentioned earlier, edges are manipulated by leftist-tree based priority queues. These queues rely on several order relations, which facilitate the imprecision-error-free implementation of graph traversals.

One such order relation for edges (the "birth" relation) R_b is be expressed as: Edge e comes before edge f if and only if

its leftmost endpoint has smaller x,

or its leftmost endpoint has same x, but smaller y,

or e and f have same origin but e has a smaller polar angle around their common left endpoint than f,

or e anf f have same origin and polar angle around it, but e is shorter than f.

Polar angles are recorded in the `Wedge` structure, for instance.

How do we manage to compute polar angles? The idea is to find a function that *behaves* like the standard transcendental function `atan`, but only performs rational operations. An example of such a function is given in [25], and may be extended as follows:

Consider any directed edge $\vec{AB}$, with origin A, and take the origin for polar angles around A to be the lower vertical directed half-line through A. Suppose B lies in the first open quadrant, so that

$$dx = B_x - A_x > 0, \quad dy = B_y - A_y > 0.$$

Define the (pseudo) polar angle of $\vec{AB}$ to be

$$\Theta_{\vec{AB}} = 1 + \frac{dy}{dx + dy}$$

Function $\Theta_A : B \to \Theta_A(\vec{AB})$ returns a "real" number in $]1,2[$ for all points B in the first quadrant, and behaves as the real polar angle. Similar expressions may be found for the three other quadrants, while 1, 2, 3 and 4 correspond to the East, North, West and South directions, respectively.

3.5 Graph traversals

Traversing a graph consists in enumerating all its elements using a specific order. One well-known traversal is referred as "sweep-line" ([24]). It consists in moving an ordered structure (for instance

a vertical line) across each successive vertex of the graph, in lexicographical order. We shall need a specialized version of this sweep, that will consider edges in their relative order around each vertex swept.

To do this, we design a priority queue PQ_b on the R_b order relation defined further up, and store all initial edges in it. We also need a "dual" priority queue, PQ_d, for storing edges already born, and awaiting to reach their death vertex. Such a priority queue is based on R_b's dual relation, R_d, applied to the rightmost vertices of all edges.

The sweep-line method may be summarized by the following generic scheme:

Algorithm 3.1 *Typical sweep-line algorithm:*

1. Start with initially empty PQ_b and PQ_d. Insert all edges in PQ_b.
2. For each vertex v in lexicographical order, do:
   (a) while $((e_d = \mathrm{First}(PQ_d))$ ends in $v)$
      i. Remove e_d from PQ_d.
      ii. ...
      iii. Output e_d.
   (b) while $((e_b = \mathrm{First}(PQ_b))$ originates in v:
      i. Remove e_b from PQ_b.
      ii. Insert e_b in PQ_d. ...

In the code above, the ... stand for all other actions that will have to be performed by specific algorithms using this mechanism.

4

Precision issues

Exact computations seem the only way to ensure the consistency of databases. Quite a lot of work has gone in the field of robust geometry, and the Computational Geometry literature of the past five years is full of accounts on this research (see for instance [19], [15], [28], [13], [30], [18]).

As the subject of this paper is the processing of terrains, we may not go too deeply into the problems of precision. Still, let us just mention a few basic facts to make our point of view clear.

4.1 A short historic note

Most problems in Computational Geometry may be expressed in terms of computing the sign of quantities. For example, determining the position of a point relatively to a polygon boils down to computing the sign of the power (the signed distance) of this point with respect to all the polygon edges. In most cases, double precision is enough. In very infrequent situations, double precision fails to yield the correct answer. The reason is that results are truncated and errors accumulate until they become too large for some results to be trusted. Were algorithms used of the "brute-force" type, such errors would go unnoticed, only affecting the solution locally. Unfortunately, brute-force algorithms are far from being optimal, and the optimal solutions that must be implemented are intricate, and often based on strong topological invariants, which double precision has no means to enforce in all circumstances.

There are many solutions to this problem, nearly all of them in the context of "rational" geometry. (The research concerning imprecision in the context of non-rational geometry is still an open field!). One popular "heuristic" consists in introducing an "epsilon", and in asking the program to declare any two quantities equal whenever their absolute values differ by less than ε, carefully chosen for the circumstance. Unfortunately, doing this implies that all events closer than ε turn out to be merged into one, and no deterministic algorithm relying on topological invariants may resist such bad treatment. An other method consists in performing all computations using an exact rational arithmetic. This has the dramatic effect of slowing runing times by a factor of up to 300. This means that if a double precision version processes a given set of data in about 12 seconds, its rational homologue will process it in an hour!

4.2 The lazy paradigm

However, we are forgetting the good news: only a few computations should be made exactly! Various solutions have been suggested in the literature. In [18] the two authors from *E.M.S.E.* present a survey of the problem and these solutions, and suggest a new way to fight imprecision, using the so-called *lazy arithmetic*. Basically, a lazy number is a record holding a double-precision interval, guaranteed to comprise the *real* number being represented, and a specialized structure to store its algebraic expression.

When a test is called for, it is first evaluated using intervals. If the interval associated to the test entirely lies on one side of zero, the answer may be given using double precision only.

Otherwise, its expression structure is evaluated with a specialized rational arithmetic, and the answer to the test is then given on the grounds of exact tests. The main advantages of such a technique is that programmers need no longer worry about precision issues in their programs, since it is the lazy library that is in charge of such problems. As the library is written in *C++*, programmers may use lazy numbers as if they were double precision numbers, and all the hard work is done behind the screen, leaving programmers to only worry about the exactness of their algorithms. Depending on the pathology of the data and the sensitivity of the algorithms to imprecision, the lazy version of an algorithm may be between 4 and 10 times slower than its double precision counterpart (when the latter does not simply refuse to work!)...

Unfortunately, it is not always possible to use the lazy arithmetic library. It was indeed not possible to do so in the present context, since *C++* was simply not available!

4.3 Reluctant programming

One of the earliest attempt to set up a hybrid solution, called *reluctant arithmetic*, appears in the PhD Thesis of one of the authors ([20]). Incidentally, this work, never published in other form, is at the very basis of the lazy arithmetic paradigm, although expressed in different terms. We shall sum up its principle here, as it is particularly well-adapted to the vast majority of problems in terrain simulation.

Suppose we know the size of the data. This may be rephrased by saying that we know the number of bits that will be needed to represent data exactly. If we also know in advance all the operations that will be performed on the data, it is possible to predict, at compile-time, an "epsilon" for each and every test appearing in the programs. In fact, contrary to the first epsilon mentioned earlier, this epsilon is a bound on the error committed in the double precision evaluation. In some cases, it might be preferable to use the greatest of all these epsilons, in some others using a specific epsilon for each test might be more appropriate. Should the magnitude of the input data vary a lot from one execution to the other, it would also be possible to "guess" the actual size of the data upon reading them, and to compute, at run-time, the magnitude-dependent value of these epsilons, and to use them as parameters, rather than constants, in the programs.

However, let $f(X)$ be a functional, the sign of which is required. In this function, X is a list of parameters $(x_1, x_2, \ldots, x_m)$, *i.e.* pieces of information manipulated by a program. A reluctant execution of this functional test will look as follows:

Algorithm 4.1 *Reluctant evaluation of function $f(X)$:*

`double_outcome` = `Double_Evaluation`($f(X)$);

if (`double_outcome` $> \epsilon_f$) return +1;

if (`double_outcome` $< \epsilon_f$) return -1;

return `Sign(Rational_Evaluation(`$f(X)$`))`;

In this informal presentation, `Double_Evaluation` stands for "use double precision numbers to perform the test relative to `f`", `Rational_Evaluation` stands for "use the rational library to perform the test relative to `f`", and `Sign` returns $-1, 0, +1$, depending on the sign of its rational argument. If ϵ_f is carefully chosen (with respect to the maximum magnitude of the data and the mathematical complexity of the functional), most executions of the code will only require the double precision evaluation, as intuition has it, while only "pathological" situations will call for a rational evaluation.

Now the main issue to solve is the following: is it always possible to retrieve enough information on the operands of `f` to allow an exact (rational) evaluation? In the case of a lazy arithmetic, the inner mechanisms of *C++* were used to construct the specialized data structures to store

the algebraic genesis of each number in a program automatically. Now we may only rely on a systematic *programming* approach, in which programmers must explicitly build up all necessary information themselves.

4.3.1 Reluctant vertices

When required, the floating point numbers for the co-ordinates of original polygon vertices are transformed into rational numbers. From a purely arithmetic point of view, this is equivalent to transform the original character strings, as read in a file, into a proper `integer/integer` entity. A specialized *rational* library is used to express `integer`s into chains of digits in a specified base.

But we have a new problem: The character strings for "decimals" read from the input have already been converted into double precision numbers. Such numbers have a limited capacity for their mantissa.

Consider reading the decimal number `0.2` in the input, using the standard *C* `fscanf` function. It is a well-known fact that the *theoretical* binary conversion of the corresponding real decimal number 0.2 is a never-ending process. This algorithm consists of a succession of multiplications by 2, such as

Result	Corresponding binary digit
$0.2 * 2 = 0.4$	0
$0.4 * 2 = 0.8$	0
$0.8 * 2 = 1.6$	1
$0.6 * 2 = 1.2$	1
$0.2 * 2 = 0.4$	0
...	...

and the initial sequence is resumed. Thus the *real* base-10 decimal number 0.2 is represented by the *real* infinite periodic binary number $0.\overline{0011}$.

As infinite expansions cannot be represented in memory, the corresponding (double precision) machine number is a "rounded", finite version, as allowed by the size of the mantissa. Converting such a number into rational form would yield very large numerators and denominators.

So, if we let data be read with `fscanf`-type functions, the corresponding `double`s in the program are machine numbers, close to, but not exactly equal to the decimal "strings" from the input. Reluctant programming will hence rule out such built-in input readings. However, we wish to ensure the following:

1. the same piece of data will be converted with the same controlled "internal accuracy" each time, and
2. the rational numbers we shall manipulate will evaluate to the exact decimal value conveyed by the associated double fields, and
3. the rational numbers in the programs will have reasonably sized integral components for reasonably sized data.

To allow for the rationals to evaluate exactly as their associated doubles, it is necessary to put the data in pseudo rational form, using floating-point numerators and denominators, and keep both the double precision number for the result, and the double precision numbers for the numerator and denominator, for a possible later rational conversion. In more prosaic terms, `"0.2"` must be read character by character, and written as $\frac{2.0}{10.0}$, and the double precision field to hold this information must be filled with the machine evaluation of 2.0/10.0.

This technique implies that the initial numbers must not be too large. If the mantissa of the double precision numbers is coded on 63 bits, it is possible to handle data with an average of 6 digits accuracy in base 10 and a magnitude of over 100 km, the unit being the millimiter.

However, there are other sorts of vertices than those read in the data. A vertex may be the result of two segments crossing. It is therefore necessary to distinguish those vertices from

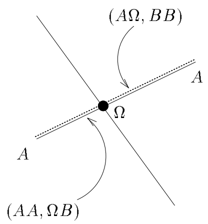

Figure 4.1: *Extremal and internal endpoints.*

"original" vertices, by keeping, together with their double co-ordinates (which are only reliable up to within a specifically computed epsilon), a record of the four endpoints of the two segments that generated this particular vertex. Such pointers are naturally set to `NULL` in the case of original vertices.

The main drawback of this technique is that every object manipulated by a program must have its own "history record", an obvious burden for the programmer, as opposed to the lazy case! But this is the price to pay for "reluctant" consistency!...

4.3.2 Reluctant edges

Reluctant edges require reluctant endpoints. However, as soon as an initial segment is divided into smaller parts (through intersections with others), it is next to impossible that segments will have the same floating-point polar angle, due to truncature.

So we define a segment as having four endpoints and a generic polar angle (as computed using the initial endpoints). The four endpoints are further divided into two sorts: two at the (lexicographical) leftmost end of the segment, and two at the other. In the original version of a reluctant segment, the two left vertices are identical, and so are the right ones.

Referring to Figure 4.1, suppose these vertices are labelled A and B so the original vertices are, respectively, (A, A, B, B). We shall refer to the first and last vertices in this quadruple as the *extremal* endpoints of the segment. They constantly refer to the original endpoints of the segment. The other two vertices , called its *internal* endpoints, always refer to the actual endpoints of the underlying subsegment.

If this segment meets another in Ω, it is now to be split into two collinear segments with vertices (A, A, Ω, B), and (A, Ω, B, B). What really happens is that the original segment is shortened to become the first (lexicographically leftmost) of the two new segments, and another segment is allocated to represent the other, with its polar angle field copied out (bit-by-bit) using the polar angle of the original segment.

This time, redundancy is playing for us: each section of the divided segment has enough information to reconstruct all its history. A rational polar angle may even be computed, if need be, using the original extremal endpoints of the relevant section! The use, in each constituants of an original segment, of an extra pointer to the leftmost subsegment ensures that only one rational version of the polar angle is ever computed, if it is at all...

4.3.3 Performance issues

In general, it is important to refrain from computing the same information (more than) twice. This is usually ensured by referring to "master" pointers for each reluctant item. For instance, when two abscissæ are found out to be equal, their rational fields are unified into a single one, and the second is tagged as a logical copy of the first. Thus, when an algorithm next encounters the

two values, finding out equality no longer requires a rational comparison: the tag will indicate it directly.

Redundancy is the worse enemy of the computational geometer: Nothing is harder than to prove, at a later stage in the program, that a vertex, which was earlier found to be at the intersection of two segments, actually lies aligned with the endpoints of either its genitors! Attempting to do so invariably ends up with an exact evaluation, in either reluctant or lazy context. One would have to record all topological assertions made by the algorithms, an exponential task. It is up to the programmer to be fully aware that careless programming entails prohibitive exact computations.

With this in mind, in most applications, the rule of thumb is that the (running-time) *overhead* one has to pay for consistency in this setting is almost negligible in the vast majority of cases, and should never exceed the running time of the pure double precision version, in the most pathological situations. A carefully coded version of the algorithms presented in this paper will process about $10,000$ real-scale (extremely heterogeneous) data in under 10 seconds on a standard Sun Sparc station, while, on the same machine, the double precision version would run in about 5 seconds (but rarely does, due to imprecision problems!).

Depending on the magnitude and pathology of data, the sentivity of algorithms to imprecision, and largely on the programmer's skill to avoid redundant computations, such a bound is rather conservative: Most of the time, the overhead only adds up to recording a little more information, and next to no exact evaluation needs be performed! Our own experience shows that most of the overhead lies in the initial conversion of (decimal *ASCII*) doubles into a pseudo-rational form, itself highly dependent on the number of digits after the decimal point in the data...

Of course, a great amount of time must also be accounted for in the *reluctant programming* stage, which is not easily quantified but should not be overlooked.

A frequent question asked by specialists outside the field of practical geometry programming is: "Do precision problems derive from the geometric methods or/and the precarity of the data structures employed?" This would be fine, since finding better methods and more robust data structures would be the answer. Unfortunately, this is not so, for a very simple reason: machine numbers are finite, and physical reality is a continuum. So modeling the Euclidian plane with finite precision numbers is asking for trouble. Specialists of Discrete Mathematics don't experience the same problems because they do not try to model continuous phenomena. In Computational Geometry, we use strong results from classical geometry, and try to apply them on finite precision machines, on which detecting cases is next to impossible with floating-point tools. In such a setting, only "random" data will not produce special cases. For applications to run on *any* data, it is necessary to use an exact arithmetic, one way or another...A complete survey of the usual problems one faces in Computational Geometry, and a catalogue of solutions may be found in [18].

4.4 Re-entrance

If one wishes to allow a given application to either use original data or the results of other computations, including its own outputs as recorded with the help of a planar map structure in our case, some sort of standard must be agreed upon.

A lot of effort has been put into normalization in terrain processing applications, but the question of precision issues often remains eluded...

As the reader might already have guessed, the major problem is not *which syntax* to use for input/output, but *which semantics* must be adopted to exploit all previous computations as cheaply as possible!

Suppose we have read the data, and found out that points A, B, C are aligned. No double precision mechanism will be capable of writing this information so as to be fully retrieved later, in the general case: The co-ordinates of the middle point, say B, will be non-decimal, and both truncature and `fprintf` will make sure no one will ever be able to know that vectors $\vec{AB}$, and $\vec{BC}$ are collinear!...

Redundancy must be inforced by other means. Here is a general framework to solve this important problem. Suppose we are dealing with vertices, segments, and polygons, using the planar-map-array scheme described earlier. We shall assume that original data files as well as output files share the same structure. In any such file, graphs will be described by listing vertices first, the edges, then faces, according to the following scheme:

1. In the case of an original data file:
   - Such a file carries an "original data" tag.
   - Vertices are listed first, sequentially, the first for index 0, and so on. Each vertex is accompanied with its 3-D co-ordinates, plus possibly some extra fields for attributes.

     There is no guarantee that one vertex does not appear twice in the list.
   - Edges are not listed.
   - Faces (or polygonal chains) are then listed, using the ranks of their boundary vertices in the previous sequential list.

     No face appears twice in the list, but the underlying graph is *a priori* not planar.
2. In the case of a result file:
   - Such a file carries a "result data" tag.
   - Vertices are listed first, sequentially, the first for index 0, and so on. Each vertex is accompanied with its 3-D co-ordinates, plus possibly some extra fields for attributes, and then a list of vertex references (up to 4 references).

     A 4-vertex reference list symbolizes an intersection point. A 2-vertex reference list enforces an alignment of three points, and so on. This list is then followed by two (possibly empty) lists of integer "digits", corresponding to the numerator and denominator (expressed in a global constant base) of the rational form of each of its x, y co-ordinates, if they happen to have been computed in the program. It is also wise to end this description of each vertex by a double precision number indicating the last epsilon associated with its co-ordinates.

     No vertex appears twice in the list.
   - Edges refer to 4 vertices, and to as many faces as necessary on each side, using the same "planar map" conventions as expressed earlier.

     Face references (objects and their attributes) are given using ranks in the sequential list at the end of the file, and may be sorted by decreasing priority (as extracted from priority queues).

     No edge appears twice in the list, and the graph is indeed planar.
   - Faces (or polygonal chains) are then listed, together with their general attributes, and, optionally, together with the list for the ranks of their boundary edges in the previous sequential list if desired.

     No face appears twice in the list.

This scheme has the advantage of "leaving room" for all situations, and of avoiding recomputation of already computed data, by simply allowing storage of all available information, for possible later reluctant *use*. In the "original input" situation, the graph must be made planar, while it is taken for granted in the second case, and all its information is readily available.

Another solution would be to record all topological events (modifications, results of tests, requests) that were witnessed during the execution of the program, and to list them, somehow, at the end of the input file (for next sessions), but this might be a much more impractical, space- and time-consuming task. . .

5

Establishing priority among objects

The problem we now have to solve is this: Considering priorities are highly dependent on time and context, how do we design methods to enforce dynamic priorities among objects?

The first step is to enumerate the different situations one may encounter. For this we shall consider an ideal application with a few typical situations:

Altimetric priority: during the expansion phase, only constraint lines and linear objects interact. Constraint lines impose their altitude to linear objects. When two constraint lines meet, or when two linear objects meet, the one with highest serial number wins.

After expansion, constraint lines impose their altitudes to any other object except linear objects that they are not supposed to cross over. During this phase, altimetric priority between intersecting objects of different classes is edicted by "common-sense regulations", such as: A field is given its altitude by the expanded road or a building it hits upon. Altimetric priority between objects of the same class is ruled by some extra information, such as the serial number. The most important thing is never to form cycles in this partial directed graph. Thus, if an object is self-intersecting, it is either rejected or the lexicographically leftmost of the two edges meeting wins. If an object is rejected, it is tagged as invalid from then on, and all its constituants are subsequently made "invisible" in due time.

The lowest rank in altimetric priority is reached by the so-called "Domain" (when it exists). All other objects impose their altitude to it.

Intersection priority: Very much akin to altimetric priority, it is used to issue warnings or error messages to the user as to which illegitimate situations the current database is having to deal with. In some ways, it extends the range of the previous priority rules by possibly invalidating trouble-making objects, or some of their sections. For instance, although not illegal, the crossing of the Domain boundary by an object signals that some portion of it is "inside" (valid) and another is "outside" (invalid). This information will be useful for the subsequent clipping module.

Inclusion/clipping priority: Again akin and complementary to intersection priority, although differring from it by its having to also consider priority between truly (non-intersecting) nested objects. In such cases, priority depends upon another set of "common-sense" regulations, such as: Houses usually win over fields, roads over fields, and so forth. But there are exceptions, such as houses right in the middle of a motorway (these things happen when dealing with completely heterogeneous data!), *etc.*. Again, inclusion priority between objects of the same class may be derived from some extra information, such as the serial number.

A special object, the Universe, plays a fundamental role for the inclusion priority. By definition all objects are included in the Universe. In fact, this object has two different ranks in the priority ordering. Within the Domain, the Universe has the lowest rank, so that if object o lies in the Domain, the list of object by decreasing priority order is (o, Domain, Universe). Outside the Domain, the Universe has maximal priority, so that if object o is present outside the Domain, the corresponding list is (Universe, o). This allows to absorb all elements outside the range of the current database.

It is the (pointers to) comparison functions applied to the priority queues which allow so many different situations, while the implementation of the priority queues remain the same!

That a database may include objects outside its range is not due to an error: The whole database may be too large to be processed in one time, and is therefore divided into smaller units, represented by the notion of current Domain. This partition is induced by computer limitations or characteristics, and is artificial from the point of view of geography. Suppose an object, such as a river, spans the equivalent of several domains; Since it is defined by straight-line sections, practical considerations suggest that all sections interacting with the current Domain be listed, although some might have to be clipped against the Domain. This ensures that corresponding sections of the same objects must match exactly, regardless of precision issues.

From this rapid description, the reader may infer that some priority management rules may be decided upon at run-time only, for each different execution of the application. To allow all the flexibility required, priority rules must be enforced, using the double mechanisms of dynamically filled tables, and of function pointers. Let us try to give a short overview of both, in pure *C* context.

Suppose we know the maximum number C of classes present in any database, and that each class is indexed by an integer in $[0, \ldots C-1]$. Each specific priority-based module (priority manager) in a program may be applied in any phase $p \in [0, \ldots P-1]$ of the program, and will consult the contents of as many $C \times C$ arrays, containing the default priority rules for each of the $C * C$ potential priority conflicts. Informally speaking, a phase correspond to a stage in the algorithm when a certain number of object classes are selected and processed together.

Default priority rules are the basic common-sense regulations described in the documentation of the application. Consider a hypothetic ideal application, with, say 5 object classes, 3 modules and 2 phases. There would be 25 priority cases, and 6 specialized priority rules tables, that would be "collected" in a two-dimensional array of pointers, `P_Tables`, to allow constant-time access.

Basically, each specific priority manager (a function dedicated to comparing ranks of objects with regard to a specific priority situation –using "semantic values" in the associated priority table–, and returning codes for action) might be called in any phase of the process. Since all phases entail potentially different actions for the same couple of objects, each priority manager is in fact a pointer to a specialized function. In our description, there would be 6 specialized fonctions. These functions would be declared as part of a 3×2 array of function pointers, say

```
int (*P_Manager(obj *, obj *))[3][2] =
 { {sf00, sf01},
   {sf10, sf11},
   {sf10, sf11}
 }
```

each `int sfij(obj *, obj *)` being itself one of the specialized functions described above. Each function has a specific body whose generic form might be:

```
int sfij(obj *o1, obj *o2)
{
  int diagnosis =
    P_Tables[i][j][o1->class][o2->class];

  switch (diagnosis)
```



```
   {
     case FORBIDDEN: /* illegal situation */
       ...
       break;
     case 1stWINS: /* acknowledge */
       ...
       break;
     case 2ndWINS: /* acknowledge */
       ...
       break;
     case DRAW: /* same priority: use extra */
       ...      /* information to discriminate */
       break;
     case USERDEFINED:
       ...             /* user-code */
       break;
     default: /* unplanned situations,
 error or other actions */
       ...
   }
}
```

The diagnoses stored in the tables have the following generic meaning:

`FORBIDDEN:` Such situation is illegal.

`WARNING1stWINS:` This should not happen, but let first argument win over second.

`WARNING2ndWINS:` This should not happen, but let second argument win over first.

`1stWINS:` First argument wins.

`2ndWINS:` Second argument wins.

`DRAW:` Arguments have the same priority. Extra information might be needed to break ties. Depending on the specialized fonction, this will be followed by further refinement, such as: are we in the presence of the same classes, of the same object, *etc.*

`USERDEFINED:` This sort of situation calls for user-defined decisions. Appropriate code must be written to take action.

`NO-IDEA:` Unclear situation, with no definite solution. So use the `default` section of the `switch` instruction to reach a decision, or reject with an error message, accordingly.

All these values are either `#define`d, or rather, enumerated in the compiled code. It is possible to let the user fill the tables as desired at run-time, with the help of an initialization routine and a parameter file with the appropriate syntax. The `USERDEFINED` section is a simple entry point into the code to allow maximum flexibility. In particular, it must be relatively easy to add new classes or subclasses.

Using function pointers arrays is a fast, compact and efficient way to switch on the appropriate section of code. When *C++* is available, this might be made in an even more elegant fashion, through the use of the overloading and heritage mechanisms available in the language!...

6

Road networks and river systems

The work presented in this section is a more detailed account of a previous publication ([4]), written in collaboration with Philippe Biscondi (ENSM.SE).

Rivers and roads have been said to be modeled by linear (dynamic) objects in the input. Their description includes the list of vertices for the medial axis, and the sequence of edges defining their profile. Profiles may be simple (as simple as one horizontal segments on either side of the medial axis), or reflect the complexity of a real-world object (in particular one having variable width at different sections). The methods described in this paper may be extended to the most complex cases. However, arguing with the simplest case is difficult enough, and carries all the information for the reader to be able to generalize. Hence we shall only describe the case of simple linear objects (constant width, profile defined by one segment on either side of the medial axis), and conclude with hints on how to handle the more sophisticated ones.

Basically, the expanded version of a linear object is obtained by sweeping its profile along the entire span of its medial axis. This operation produces two "generatrices", parallel to the current section of the medial axis, and at a distance from it specified by the profile. Expanding the whole network then consist in constructing *expansion* vertices at each node in the original linear objects graph, and then creating generatrices between them, and special (orthogonal) *connectors* to close objects at each end. This ideal situation is illustrated on Figure 6.1.

Although the previous description seems to imply elementary actions, such is not the case, as the following argument will prove: Consider a typical U-shaped road with constant width, as illustrated on Figure 6.2.

Clearly, in a "normal situation" (recall that normal has no meaning when dealing with heterogeneous data!), succcessive generatrices meet in a new expansion vertex (for instance A_l and B_l on the figure). Considering section $A \rightarrow B$ of this road, it is quite normal to expect that the endpoints of its expanded version (*i.e.* (A_l, B_l) and (A_r, B_r)) lie in the same order as A and B themselves. This will no longer be true if the half-witdh of the road becomes strictly larger than half the distance from A to B: in such a case, the order of (A_l, B_l) is reversed, and the construction is no longer possible.

In such a simple situation, problems seem to have simple solutions, but this is not the case. We must allow asymmetric profiles, and right angles is not such a common situation in natural databases. furthermore, there might be *any* number of medial axes around one given vertex, and so on!

It might be suggested that the solution described below is far too complex, and that the construction of linear objects might be performed much more easily by expanding each such object independently, and letting the "clipping" module do all the work. This is true, except for one, major problem: Networks constructed this way (which was the first solution we had elected) contain an incredible number of artefacts (micro-edges or faces), due to the independent handling of highly interacting complex objects. To prevent such dramatic increase in the complexity of the output, and, in particular, to simulate realistic junctions, one must retain complex, but much more efficient, solutions, such as the one presented now.

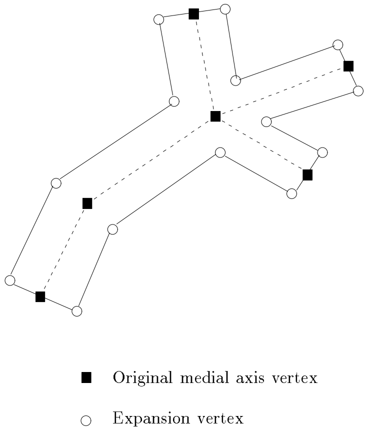

Figure 6.1: *Expansion of standard road network.*

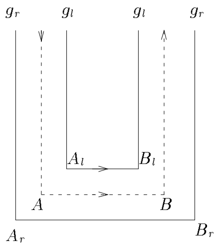

Figure 6.2: *A typical U-shape road section.*

Postponing all explanations relative to planarity (and thus to phase 1 of the algorithm), here is the sketch of a general solution to the problem.

Algorithm 6.1 *Expansion of linear objects:*

1. Determine the set V of all vertices on medial axes, and the associated set E of all edges of the corresponding planar graph.
2. For each vertex v in V do
   (a) Organize all edges around v in a circular list $L(v)$
   (b) For each successive couple (e_i, e_i^+) of edges in $L(v)$ do
      i. construct *expansion vertex* v_i produced by generatrices γ_i and γ_i^+ (as defined by edges e_i and e_i^+, respectively).
3. For each edge e in E, inspect relative positions of **both** couples of expansion vertices:
   if (either couple of positions is reversed) mark endpoints of e as "defective".
4. For each vertex v of V in lexicographical order, do
   if (v is defective)
      (a) For each successive directed edge e originating from v do:
      Initiate two generatrices, as stipulated by the profile, regardless of any other object; attach these generatrices to e, for future completion. "Close" these generatrices with a perpendicular connecting segment through v and store it in the output list[1].
      (b) For each successive directed edge e ending in v do:
      Finalize the generatrices initiated when leftmost endpoint was considered, create perpendicular connector through v, and store all three in output list.
      Dispose of e.
   else /* (v is not defective) */
      (a) For each successive directed edge e originating from v do:
      If degree of v is one, initiate two generatrices and attach them to e, then output the perpendicular connector through v and store it in the output list. Else initiate two generatrices using information about local expansion vertices (on either side of e), as computed in phase 2.b, and attach these generatrices to e for future completion.
      (b) For each successive directed edge e ending in v do:
      Terminate the two generatrices initiated when leftmost endpoint was previously encountered and store them in output list, together with perpendicular connector if degree of v is one.
      Dispose of e.

Another problem that must be solved by this algorithm is the attribution of priorities around each central "junction" vertex. This is done by inspecting the linear object ranking highest in priority around the vertex, and then adding possible connectors between successive expansion points of the junction to "close" objects with lower ranks. This constitutes what shall be called a closed *round-about* polygon.

An example of a typical situation is shown on Figure 6.3. Each of such connectors is assigned one face with the attributes of the "main" road, and another with the attributes of the linear object it is closing. One has to make sure no added connector creates self-intersections in the round-about polygon. Such pathological situations, which may happen with highly perturbed data, are quite easily predictible (maintain the star-shapedness of the round-about polygon around its central vertex), and treated by forcing "bending" connectors through the central vertex.

[1]In a more refined version, it would possible to let the "well-behaved" edges around v be expanded object by object, a much more elegant, though significantly harder to implement, solution with the same optimal time complexity.

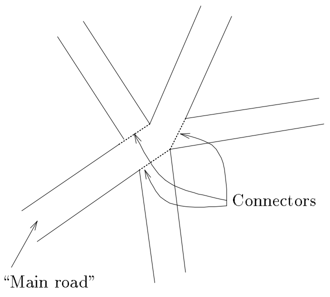

Figure 6.3: *Handling priority in a typical junction.*

In Algorithm 6.1, the original edges put in the priority queues are not output in the final list, and are even disposed of. It is because they are segments from medial axes, and are only used to initiate, transport and finalize the final side generatrices.

If the number of edges to be considered in the previous algorithm is K, it is easily seen that the overall complexity of the algorithm in $O(K \log K)$, since all operations on the queues have logarithmic complexity, and all other operations around vertices have a global complexity proportional to the size of the graph, K.

The single-segment profile model we have used as support for this presentation may be made more complex. Expansion vertices are still needed to create generatrices, although the "bodies" of the linear objects require more care at expansion time. The details of the corresponding algorithms are also more complex. In particular, sections with different profiles meeting in the same vertex will cause serious problems. However, it is readily seen that the main structure of our algorithm still holds, in the more sophisticated situations.

6.1 Retracting static objects

When designing databases as those described in this paper, one faces the problem of producing objects that do not exactly correspond to what one thought they would look like. For instance, imagine the fields around a given junction. Since the construction of the junction is such a complex problem, the designer has no idea where to anchor the vertices of the neighbouring fields, except maybe at junction vertices, where he or she has decided to make medial axes meet.

Now, for obvious reasons, the field boundary edges emanating from junction vertices are bound to cross generatrices at unpredictable places. Just consider the rather simple "bend" of Figure 6.4.

Field edge AB crosses generatrix g_r in S, and most likely, edge ST is very tiny, resulting in an undesirable artefact. What the designer really hoped for was that edge ST simply would not be created!

There is a way to achieve this, by *retracting* the edges from static objects converging to a junction vertex appropriately.

Just add an extra loop in Algorithm 6.1, and organize all static edges of objects susceptible to be retracted (an information that must be made available in the input, or deduced from the class of the objects: For instance no building should be thus retracted!) into yet another couple of priority queues (along the same birth-death scheme as before). Now, for each vertex v in V, reconstruct on-the-fly the (possibly empty) list of all static edges converging to v. Using a linear-time algorithm to synchronize the list of medial axis edges around v and that of static edges,

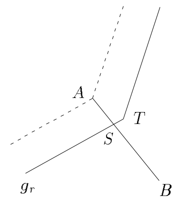

Figure 6.4: *Where bends and fields start making trouble.*

each element of the latter may be located within a unique (open-closed) wedge of the former list, and thus, the corresponding static edge may be updated by being assigned the unique expansion vertex associated with this wedge (T on Figure 6.4). The overall cost for this new process is again proportional to the size of the graph.

This description is unfortunately too optimistic, for two reasons:

- Junction vertices may be defective (as defined earlier: some objects attached to them cannot be expanded in a natural fashion), and

- Data may contain situations where the retraction of static objects fails to yield topologically sound solutions.

In the first case, it is quite reasonable to refrain from adding to confusion, and thus to decide not to perform retraction in such vertices. The consequences of this might very well be that we end up with a topological aberration, just as in the second case!

To illustrate this second case, consider Figure 6.5. The two static edges converging to central vertex v are retracted to vertex T, while vertex w remains at the same place. Since, in the "normal" cases, the face-orientation of retracted edges should not be modified, the new egdes keep facing their associated static object, which is now wrong: Triangle uwv is paritally turned inside-out. Retraction has obviously caused a topological aberration.

Both problems make retraction hazardous in an unpredictable number of cases. The solution is to check the topological validity of the resulting retracted version of static objects. If a given object passes this test (which may only be performed after the expansion process is completed), its retracted version is substituted to its former initial definition. Otherwise, its retracted version is abandoned and its original version takes over.

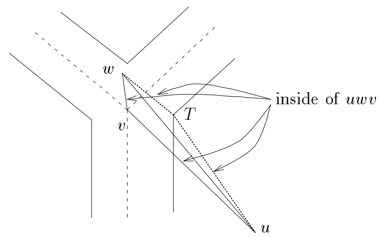

Figure 6.5: *Retraction may yield topological impossibilities.*

The question now is: How do we check the topological validity of retracted objects? Another question is: What risks do we incur by altering the position of endpoints in some segments from the original database? Yet another question remains to be settled: In the first phase of Algorithm 6.1, we are supposed to enumerate all the vertices and edges of the graph for linear objects. How do we manage this?

All these questions are strongly related to the process of making non-planar graphs planar, the subject of the next section.

7

Making graphs planar

There are quite a few algorithms available to make graphs planar ([3], [24], [23]). The one that we advocate for in the context of off-line terrain processing in the presence of semantic hierarchies is the algorithm by Bentley and Ottmann ([3]). This method strongly relies on the *sweep-line* paradigm.

In our setting, it is desirable to filter out the input of the algorithm so as to select which subgraphs are to be processed at any one time. The basic outline of the overall application suggest the following stages:

1. Make the subgraph of constraint lines and medial axes for linear objects planar.
2. Expand linear objects. Collect resulting generatrices and connectors, discard constraint line section that cross medial axes (to avoid further interactions with expanded linear objects). Retract static objects as needed.
3. Check the topological validity of the retracted objects.
4. Merge all final objects into one graph, and make it planar.

Performing all these actions require some flexibility in the "main" method. Priority queues, constructed on edge nodes, allow this flexibility: Insert all selected edges into an initially empty priority queue, and let the internal mechanisms of the birth- and death-priority queues do all the work for the sweep-line process.

Two major modifications that must be brought to the original version of Bentley-Ottmann's algorithm are related to its input:

- It does not readily accept special cases (namely vertical segments, nested segments, or more than two segments meeting in a single point)
- It does not readily accept mixing isolated vertices and segments.

Obviously, real-world databases almost always include special cases, and by essence, isolated vertices (altimetric data) are numerous, and edges often run over them. So the algorithm must be "robust" in that sense. Finally, Bentley-Ottmann's algorithm is known to be very sensitive to imprecision. We shall see that the reluctant version of Bentley-Ottmann's may be made robust in both senses.

7.1 Mixing vertices and segments

Unifying segments and vertices (an endpoint may be regarded as a degenerate segments with identical endpoints) would not be a good idea: it would only increase the number of special cases. A much better idea is to let the primitives of the associated data structures do all the work for us.

By virtue of the internal priority-queue mechanisms, all edges originating from the same vertex are extracted in increasing (counter-clockwise) polar angle.

As soon as a segment is removed from PQ_b, its set of leftist-tree pointers is available, but will be used by the mechanisms of PQ_d. We shall need another (edge-based) AVL structure, Y, to reflect the y-order at the vertical of the current vertex v. Another two AVL pointers to edges will be used for this purpose. Edges are inserted in the y-order AVL after they are extracted from PQ_b, and they are also inserted in PQ_d.

The y-order AVL also yields (through a cross-sectional doubly linked list structure) the edges above and below any of its members, in constant time. Another y-order AVL primitive provided is `Floor(Y, v, &(v->floor))`, which stores in its second argument the address of the unique y-order segment immediately below v in all cases (using a dummy edge when v is lower than all active segments), and returning the address of the unique y-order segment *through* v if there is one, and `NULL` otherwise. This function performs a search in Y, by using the signed distance between v and the current segment-node at each level visited.

If two edges are identical (recall that the input may have many adjacent, or nested and edge-sharing 3-D polygons), we must merge them into a single one, and merge the faces on either side. This process is facilitated by the priority queue structure used for the faces themselves.

If two distinct edges have the same polar angle, the edges are extracted from PQ_b by increasing size. Now, consider two distinct edges with same origin v, and same polar angle around it. After the first, shorter segment, e, is removed from PQ_b, inserted in Y and PQ_d, the second one, f, is extracted from PQ_b. The function for inserting f in the y-order tree tries to locate the position of f in Y, and naturally detects that the nearest element "below" f is collinear e. f is subsequently split into two subsegments: $e \cap f \equiv e$ itself, and that portion of f extending beyond e. Another merge between e and f is performed, and f is simply "shortened" (its origin is assigned the position of e's right endpoint) and then inserted in PQ_b, since it won't be born before reaching e's right endpoint!

This scheme ensures that no edge is present more than once in the y-order tree, and hence in PQ_d. The outline of the algorithm may now be given, as follows:

Algorithm 7.1 *Reluctant and generalized version of Bentley-Ottmann's algorithm:*

1. Insert all edges in initially empty PQ_b. Set PQ_d to null.
2. For each vertex v in AVL do
   (a) $\vec{AB}$ = `Floor(Y, v,&(v->floor));`
   (b) if no edge ends in v
      i. if ($\vec{AB}$≠`NULL`) /* $\vec{AB}$ goes through v */ split $\vec{AB}$ into $\vec{Av}$ and $\vec{vB}$. Update the position of $\vec{Av}$, the shorter version of e, in PQ_d, since its death has just been set forward. (Its position in Y remains of course unchanged.)
   (c) For each edge e_v ending in v, (output in decreasing polar angle by PQ_d) do
      i. Remove e_v from PQ_d.
      ii. Let e_v^+ and e_v^- be the edges in Y immediately above and below e_v, respectively.
      iii. Remove e_v from Y.
      iv. If e_v^+ meets e_v^- in Ω, split both segments into four, all sharing Ω. Those originating from Ω are not born yet, and hence must be inserted in PQ_b. The other two, ending in Ω, are shortened versions of e_v^+ and e_v^-, and their positions in PQ_d must be updated.
   (d) For each edge e_v originating from v, (output in increasing polar angle by PQ_b) do
      i. Remove e_v from PQ_b. Extract all duplicates of e_v still in PQ_b, and merge them with e_v.
      ii. Insert e_v in Y and PQ_d.
      iii. Let e_v^+ and e_v^- be the edges in Y immediately above and below e_v, respectively.
      iv. If e_v and e_v^+ meet in Ω^+, split both segments into four, sharing Ω^+. Those originating from Ω^+ are not born yet, and hence must be inserted in PQ_b. The other two, ending in Ω^+, are shortened versions of e_v and e_v^+, and their positions in PQ_d must be updated.
      v. If (possibly shorter) e_v and e_v^- meet in Ω^-, split both segments into four, sharing Ω^-. Those originating from Ω^- are not born yet, and hence must be inserted in PQ_b. The other two, ending in Ω^-, are shortened versions of e_v and e_v^-, and their positions in PQ_d must be updated.

Comments on Algorithm 7.1:

- Intersections are detected by reluctant functions, using signed distances and information from the algorithm (for instance in case `2.(d).iii` e_v is *below* e_v^+, while in case `2.(d).iv`, e_v is *above* e_v^-, so why not use this previous information?).
- Intersection-detecting functions operate on *internal* vertices, but may use external vertices to prevent accumulating errors.
- The updation of the position of an element in PQ_d, subsequently to its right endpoint being moved to the left, is made through a specialized priority queue primitive, and only consists in promoting the element along the path to the root of the leftist-tree, until it does not violate the "heap" propriety any longer. This is made possible in logarithmic time by the fact that all data structures uniformly use edges as their nodes, (so the address returned by function `Floor` or that returned by the functions for the immediate neighbours of a given edge are adresses in the priority queues, too!), and edges have extra pointers to their father-nodes, to allow backtracking in the tree...
- The reluctant version of Bentley-Ottman's algorithm uses the 4-vertex edges described earlier. In our setting, all the precision problems (and their reluctant solutions) are "encapsulated" in the internal mechanisms of the various data structures. This is a great improvement over classical versions of Bentley-Ottmann's method, where imprecision-solving instructions are scattered randomly throughout the code. In this fashion, we have integrated the special cases of the initial algorithm into the reluctant mechanism of the sweep-line process.
- This algorithm has one more advantage, no other intersection algorithm shares with it: It allows to assign to every vertex in the graph its "floor", *i.e.* the graph segment immediately below it. This information will be crucial in the clipping phase.
- Its running time is, of course, identical to that of the original version, $O((N+k)\log(N+k))$ if N is the number of initial edges, and k the number of intersections detected.

7.2 Checking topological validity, and other unanswered questions

Checking the topological validity of input polygons and of retracted objects only requires to specialize the previous algorithm so it detects situations where the famous em Jordan curve theorem is violated: In simple terms, this theorem states that any line crossing a simple polygon alternates between "in" and "out" regions, and that boundary edges go by pairs. Using the y-order vertical line, and also checking no intersection between edges is detected outside vertices, and that the degree of any vertex is two, yields a solution to the problem at hand, in time proportional to $e \log e$ if the current polygon has e edges.

Altering the topology of static objects through retraction is therefore less harmful than it first seemed, since the final objects of the graph are all processed through Bentley-Ottmann's algorithm, and the resulting graph is hence planar. The only side-effects are possible changes in the outcome of location procedures, due to the alteration in the equations of some edges. But one must recall that, in "normal" conditions, only micro-alterations will be performed, a minor risk for major benefits.

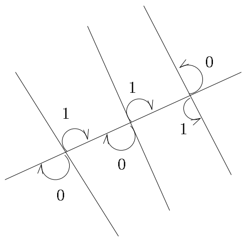

Figure 7.1: *Concatenation in `zlinks` chains.*

7.3 Altimetric considerations

After the final graph has been made planar, the first action to be taken is to compute the altitudes of all its vertices. There are three basic sorts of vertices:

- Static vertex: Defined in the data. This could be a pure altimetric point (from an underlying altimetric grid), or a vertex from the boundary of an input polygon or the segments of a polygonal chain.
- Dynamic (or expansion) vertex: As computed during the expansion of linear objects
- Intersection vertex: As discovered while making graphs planar.

The genesis of vertices is crucial in determining their altitudes. Thus, unless it coincides with a vertex of another sort with higher rank in the priority order, a static vertex has its altitude interpolated on the altimetric grid. The altitude of a dynamic vertex is computed using that of the medial axis vertex that it originates from: each medial axis gives birth to a planar section, and hence the altitude of either its endpoints (in the absence of any other object) is simply copied out to both its expanded offsprings.

As for intersection vertices, the rules are more complex. If two segments cross in a static vertex, the latter keeps its definition and its altitude stays as computed on the grid. Otherwise, the intersection vertex is made to "point" to the two endpoints belonging to its genitor edge ranking higher in priority. These pointers (later designated as `zlinks[0]` and `zlinks[1]`) are added to the `Vertex` *AVL* structure, point to `NULL` in the case of a static vertex, and are updated by the intersection-detecting functions mentioned in Algorithm 7.1.

Of course, `zlinks` may concatenate when one single segment is split by several transversal segments with lower priorities. Note that two successive such vertices v and w (with, say, v to the left of w) on the segment have `zlinks` mutually pointing to each other (`v->zlinks[1] == w` and `w->zlinks[0] == v`, 0 for "left", 1 for "right"). This situation, illustrated on Figure 7.1, necessarily ends with either a static or dynamic endpoint, or an intersection vertex whose `zlinks` point in a direction orthogonal to the segment. Finally recall that no cycle may appear in the `zlinks` chaining system, thanks to the conventions we have adopted for settling priority conflicts.

The computation of altitudes for the vertices of a database requires applying the following algorithm to every vertex in the database:

Algorithm 7.2 *Compute the altitude of a vertex in the database:*

ComputeAltitude(Vertex *v)

if vertex v has not already been visited

1. Mark v as visited
2. if v is a static point, interpolate its altitude on the altimetric grid
3. elsif v is an expansion vertex, recursively call `ComputeAltitude` to compute the altitude of the medial axis vertex it has been attached to during expansion (a straightforward operation kept silent in our description), and use it for v.
4. else /* v is an intersection vertex */:
   (a) Follow the `zlinks` in direction 0, until reaching a vertex v_0 with links to null or heading another direction. Recursively call `ComputeAltitude` on v_0.
   (b) Follow the `zlinks` in direction 1, until reaching a vertex v_1 with links to null or heading another direction. Recursively call `ComputeAltitude` on v_1.
   (c) Interpolate the altitude of all vertices between v_0 and v_1, including v, using their "abscissæ" on the segment, and the newly computed altitudes of v_0 and v_1.

8

Clipping: From segments to regions

Now that all intersection points have been computed, the graph is planar and it is possible to decompose it into *regions*, on the principle that any edge has exactly two regions on either side, with one region possibly occupied by the Universe.

This stage is achieved by computing a *planar map* of the graph: this is detailed in section 8.1. Each region is occupied by one or several faces, and a second stage computes the faces present in each region, taking their priority into account: see Section 8.2. Finally, some edges appear to be useless, since their both regions carry exactly the same higher priority face: they are removed in a the third stage, which also removes useless vertices: see Section 8.3.

8.1 Computing the planar map

8.1.1 Definition of the data structure

A *planar map* is a data structure, with already defined edges, and *regions*. In fact, it is the completion of the planar map array we have mentioned in subsection refdspm. Each half-edge will refer to its *upper* region, carried by its left side, and its *lower* region carried by its right side: due to the lexicographic ordering, the upper region of a vertical half-edge lies left of it. A region will know at least one of its half-edges (an edge and one of its side), called its *initial half-edge*.

From this half-edge, it is easy to recover the next half-edge of the same region: Take next counter-clockwise half-edge around the origin of the current half-edge, and then its dual half-edge. Repeating, we traverse the boundary of the current cyclic region, and eventually return to the initial half-edge. Note that this procedure is the dual of the one enumerating all edges incident to a given vertex.

There are two types of region: The boundary of an *inner* region is to be traversed in clockwise order; that of an *outer* region counter-clockwise. Moreover some regions may be included in others. Inner regions are contained in outer ones, and outer regions in inner ones, except the outermost ones, which, by convention and for convenience, are contained in some virtual inner region (corresponding to the Universe, and dummy floor edges, already mentioned). These inclusion relations are represented in the planar map data structure by a specific binary rooted tree, in each node of which is stored a pointer to the first *son* of a region, and a pointer to its first *brother*. The brother field allows to represent the list of all the regions contained in the same *father* region. The virtual inner region is in the root of this tree.

8.1.2 Construction method

Algorithm 8.1 *Build the planar map:*

1. For each vertex v in lexicographical order do:
   (a) For each edge e in R_b-order (*cf.* Section 3.4) do:
      i. If the e's lower region has already been marked, move on to the next initial edge, or to the next vertex.
      ii. Otherwise:
         A. Create a new region r;
         B. follow the half-edges of r starting from the right side of e, mark their region pointer with r and set the initial half-edge of r with the last marked one.
         It is easily seen (and it can be proved) that r is an outer region iff e has only initial edges and e is the first one. Bentley-Ottman's method has given the edge b just below the vertex v, and this information has been recorded in due time. b is v's *floor* edge. If b does not exist, let r' be the virtual inner region, otherwise r' is the above region of b. When r' is an inner region, r' is r's father (for the inclusion tree), *i.e.* r is contained in r'.
         When r' is an outer region, r' and r are brothers, so the r's father is set to the father of r', already known by construction. The inclusion tree must be updated.
         C. If r is an inner region, let r' be the lower region of r's initial edge.
         If r' is an outer region, it is the father of r, otherwise r and r' are brothers, thus r's father is set to the father of r', already known by construction.
         Again, the inclusion tree must be updated.

This algorithm works in time linear in the number of edges.

8.2 Computing the faces in each region

This stage computes the faces present in every region of the planar map, and their priority.

Starting from the outermost inner region, the regions of the planar map are successively visited, by stepping over boundary edges. The faces present in the current region are stored in a *global* priority queue, that must be updated each time we step over an edge.

8.2.1 Crossing an edge

We are standing in some region r, and we know the faces present, as well as their priority, thanks to a *global* priority queue. We store the face ranking highest in some specific field of r, called the (dominant) *region-face*.

We want to step over an edge e of r to enter region r'. Recall that we have stored all faces piling up on either side of any edge, with the help of a (local) priority queue. This structure only seems to reveal the object ranking highest in priority, but all the underlying information is still available.

To traverse e, we have to update our priority queue: Crossing over edge e simply means removing from the global priority queue those faces piled up on the side of e facing us: Copies of these faces are necessarily present in our global queue as the construction will ensure, although they might be quite low in the global order. This deletion is possible in logarithmic time (without affecting the state of e's own private priority queue, of course) if we store, in the object records, the address of the element inserted in the global priority queue, and a counter indicating how many times a face of one specific object has been inserted into the global queue. This counter is logically decremented for each deletion, and there is a physical deletion (as described in the subsection dedicated to data strutures) only when there is no face for that object left in the global queue.

Now, we must insert all the faces piled up on the side of border edge e facing away from us into the global queue. After doing this, we may proceed into region r', and the global priority queue now indicates the (possibly identical) face ranking highest in global priority.

8.2.2 Traversing the planar map

The traversal of the planar map starts from the virtual infinite inner region, and the global priority queue of faces is initialized with its faces (they are trivially known). All the sons of this region –in fact, all sons of an inner region– have the same hierarchy of faces as their father. The problem boils down to *visiting* all (inner) regions of an outer region, the hierarchy of faces of which is available, and the region-face of which already has been set. This may be done as follows:

Algorithm 8.2 *Visit (inner or outer) region r:*

1. Start from r's initial half-edge.
2. For each of its half-edges, in turn, do:
   (a) If the region r' on the other side has already been visited (its region-face field is set), then continue.
   (b) Otherwise cross the edge, say e, while updating the priority queue of faces present by the procedure previously presented in Section 8.2.1.
   Set the region-face field of r' to the dominant face, and recursively visit r'.
   Then *step back* over edge e, updating the priority queue to recover the present faces in r, and consider the next half-edge of r, after e.
3. When all half-edges of r have been considered, and if r is an inner region, consider its sons: they are outer regions, thus they have the same hierarchy of faces than r: their region-face field is set and they are visited.

Note that the global priority is in exactly the same state, before and after one region is visited. The number of crossings is 2 times the number of regions, since we enter 1 time and exit 1 time in each region. Each half-edge is "followed" one time.

8.3 Removing useless edges and vertices

Discarding vertices and edges made invalid in the previous process is now easy: Scan all edges and check whether they have been assigned two distinct region-faces.

If yes, move on to next edge, else invalidate edge and decrement the degree of either its endpoints by one, if it is part of a polygonal object.

If its two region-faces are identical and it is a constraint line edge, just check, using a specialized priority function, whether such a constraint is legitimate inside the region it lies in.

If not, invalidate edge and decrement the degree of its endpoints.

When this loop is over, remove all invalidated edges and vertices from the global database. A little care must be taken with respect to vertices: A vertex may be isolated (null degree), or not, valid or not. An isolated vertex will only be valid if it is legal for it to lie in the region its floor edge is associated to. A non-isolated vertex will be invalid if its degree becomes null, through invalidation of all the edges converging to it in the previous process.

8.4 Performances of the clipping stage

The algorithm presented in this section takes the same time as does a straighforward sweep-line algorithm. Hence, if the size of the planar graph is $M = k + N$, it takes $O(M \log M)$ time to run, and is therefore optimal. It is worth noting that the clipping of edges and faces may *not* be achieved during the process of making the global graph planar, since one needs to mark connected components with (planarity) information only available when Bentley-Ottmann's algorithm is over!... Also notice that randomized algorithms ([27], [29], [23]) or Chazelle-Edelsbrunner's $O(k + \log N)$ algorithm ([6]) $-N$ being the size of the original non-planar graph– would require some extra work to provide for the floor information readily available through Bentley-Ottman's algorithm, for the same $O(M \log M)$ time-complexity!

Our motivation for favouring sweep-line algorithms over others should now be quite obvious: all the operations so far benefit greatly from the sweep-line paradigm.

9

Triangulation: From faces to simplices

This section presents a few notes recapitulating two previous publications ([22], [21]) by the first author from *E.M.S.E.*, to be written in more detail and submitted shortly to the same journal. We shall only give a flavour of the algorithms here.

The Delaunay triangulation of a set of points is a well-acknowledged and understood problem. The main survey, containing all references, on the subject is F. Aurenhammer's authoritative paper ([1]). Interested readers are referred to it and to [24] for more details on Voronoi Diagrams and Delaunay Triangulations.

A lot of work has also gone into the harder problem related to the construction of the so-called *Constrained* Delaunay Triangulation of a planar graph. The first optimal solution was presented by P. Chew ([7], [8]) in 1989. The paper referred to in the beginning of this section brings improvements over Chew's algorithm by allowing special cases the latter did not easily support, and by providing an extension that made it capable of being applied to local sections of the same graph, while retaining its optimal running-time characteristics.

Delaunay triangulations on sets of points in plane are famous for requiring that the triangles they are composed of be such that the circumcircle through their vertices do not contain any other point of the (point) set in their interior.

9.1 Divide-and-conquer *versus* sweep-line

It is an intriguing fact that there are optimal sweep-line as well as divide-and-conquer solutions to most major problems in Computational Geometry, while the problem of segment intersection has no known divide-and-conquer solution!

There are sweep-line solutions to the Delaunay Triangulation problem, both in unconstrained and constrained cases ([11], [12]), [26]). Unfortunately, all sweep-line solutions are by essence impossible to implement in a reluctant (rational) way, since they all require that non-rational tests (including square-roots of rational elements) be performed. Although it would be possible to adapt the underlying arithmetic to accept such tests by allowing representation of square-roots for specific numbers, it is much more appealing to resort to divide-and-conquer techniques for constructing the Delaunay Triangulations (constrained or not), since these methods exclusively rely on rational tests! In particular, it may be shown ([14]) that testing whether a circle around three given points contains a fourth one or not may be performed using a 4×4 rational determinant.

9.2 Outline of the solution

Basically, what the algorithm does is to divide the *vertices* of the graph into two equal-size subsets, to process them recursively, and to merge the resulting sub-triangulations into one. Of course, the real problem is the merge phase. As this phase is much too involved to be explained here, let us state a few comments on it:

1. A triangle formed on three graph vertices is said to be of the *(constrained) Delaunay type* if and only if the circumcircle across its vertices contains no other vertex in its interior, or only contains vertices that are invisible from the third vertex due to a constraint graph edge between the other two, in the original graph.
2. A typical subtriangulation has three types of edges: *Free* edges only have unprocessed endpoints. *Captive* edges have both endpoints processed (*i.e.* part of the final planar map). *Semi-captive* edges, for which only one endpoint has been processed.
3. A given subtriangulation is complete when all its edges are captive. Otherwise, it is incomplete. The final triangulation is, of course, complete (unless infinite edges are allowed in the initial graph).
4. Merging two subtriangulations may be regarded as synchronizing corresponding semi-captive edges in both subtriangulations, and filling the sectors between two such successive constraint edges (called "bridges") with Delaunay-type triangles. This phase is reminiscent of Lee and Schachter's original algorithm to construct the Delaunay triangulation of a set of points in the plane ([17]).
5. The leaves of the recursion tree are traversed by the divide-and-conquer process in the lexicographical order.
6. This allows to synchronize the divide-and-conquer mechanism and a sweep-line algorithm.
7. The sweep-line algorithm is implemented using the same priority queue structures as presented in this paper.
8. Using the sweep-line algorithm allows to introduce the notion of constraints in the progress of Lee and Schachter's original algorithm quite naturally.
9. Thanks to the mechanism of the priority queues/*AVL* structures, the algorithm is unsensitive to special cases. Symmetrically, a reluctant version is readily available, since all the history information collected by the previous phases are still available at this point, and only rational operations are required.
10. The same algorithm may be improved so it may be applied to local sections of the graph, provided they are properly *constrained*. Such a property is crucial when the need for some *interaction* is called for. It allows updations of the database as it is being designed, and is therefore some sort of half-way solution between a truly on-line application and a truly off-line one.

10

Summary and Conclusion

This paper has presented basic concepts for the manipulation of terrains structured by semantic hierarchies. Methods were shown to implement the complex notion of context-dependent hierarchy, and we have shown how the fundamental sweep-line paradigm was well-suited for a large variety of fundamentaloperations.

It was also shown how imprecision problems could be settled by adopting a "reluctant" philosophy in the programming of applications, and that the overhead one had to pay for this style of programming was much more noticeable when programming than at run-time.

Another problem we have mentioned is that of making heterogeneous data homogeneous. In a sense, this problem is simple: If two objects have elements *nearly* identical, make them identical. Thus, two vertices closer than a given tolerance should be only one, one vertex too close to a segment should lie on it, *etc.* Of course, the problem is much more intricate than it just seems, and it is necessary to use complex structures, like the point-and-segment Voronoi diagram of the graph to try and cure the simplest forms of degeneracies. This very interesting problem is unfortunately beyond the scope of this paper.

There are a lot of other operations that might be useful for *G.I.S.* applications, one of them being the location of a point in a database. We have not mentioned algorithms for such operations because problems of this sort fell outside the scope of this paper. The interested reader is referred to the seminal books [24], [23].

We have explained how off-line processing was possible, and also that on-line solutions might prove efficient in the future. We need say a little more on that now.

One basic requirement to allow full on-line processing of hierarchical terrains is that the data structures may uniformly convey the different aspects of the same objects throughout their life-span. This also suggests that more topology be known at the time when data is read.

One of the major open problems is to find well-adapted and adaptive data structures to support both randomization and all the operations to be performed. It is not clear at the time of writing whether such a structure may be used that would equally support the algorithm for intersecting plane segments and construct the Constrained Delaunay Triangulation of the graph. There is no obvious reason why such a unified structure should not be found. Generally speaking, it takes time before a theoretically optimal structure becomes practical. Let us hope that the research going on in this direction will help bridge this gap. Two particularly informative books on the subject on randomization are [23] and [5].

Another complex problem related to the algorithms presented in this paper is to adapt the density of points in a region to the requirements of the target machine. It is often necessary to limit the number of points in a region not to overflow the processor in charge of displaying the output. Such topics are being investigated by the authors of the paper (and others), and are the subject of future research.

Acknowledgements

The authors would like to thank Jean-Louis Beau, Sylvie Rouillard, and Frank Rousee, from SOGITEC, as well as Philippe Biscondi and Pr. Bernard Péroche from LISSE-ENSM.SE for having taking part in many discussions or code sessions, and helped make the right decisions at the right time.

Bibliography

[1] F. Aurenhammer. Voronoi diagrams – A survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.

[2] B.G. Baumgart. A polyhedron representation for Computer Vision. In *National Computer Conference*, pages 589–596, 1975.

[3] J.L. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comp.*, C-28(9):643–647, 09 1979.

[4] P. Biscondi and J-M. Moreau. Computer-aided road network design. In *Proceedings of the 7th Canadian Conference on Computational Geometry*, pages 229–234, Québec (Québec, Canada), August 10-13, 1995.

[5] J-D. Boissonnat and M. Yvinec. *Géométrie Algorithmique*. Édiscience International, Paris, France, 1995.

[6] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. Technical Report CS-TR 148-88, Dept of Computer Science, Princeton University, 1988.

[7] P. Chew. Constrained Delaunay triangulations. In *Proceedings of the 3rd ACM Symposium on Computational Geometry*, pages 215–222, 1987.

[8] P. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4:97–108, 1989.

[9] R. Cori. Un code pour les graphes planaires et ses applications. *Astérisque*, 27, 1975.

[10] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. M.I.T. Press, McGraw-Hill Book Company, Cambridge, Mass., New York, N.Y., 1989.

[11] S. Fortune. A sweep-line algorithm for Voronoi diagrams. In *Proceedings of the 2nd ACM Symposium on Computational Geometry*, pages 313–322, 1986.

[12] S. Fortune. A sweep-line algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[13] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. In *Proceedings of the 9th ACM Symposium on Computational Geometry*, pages 163–172, San Diego, May 1993.

[14] L.J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphic*, 4:74–123, 1985.

[15] C. Hoffman, J. Hopcroft, and M. Karasick. Robust set operations on polyhedral solids. *IEEE Comp. Graphics and Appli.*, 9(6):50–59, Nov. 1989.

[16] E. Horowitz, S. Sahni, and S. Anderson-Freed. *Fundamentals of Data Structures in C*. Computer Science Press, New York, 1993.

[17] D.T. Lee and B.J. Schachter. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computing Information Sciences*, 9:219–242, 1980.

[18] D. Michelucci and J-M. Moreau. Lazy arithmetic. *IEEE Transactions on Computers*, 46(9):961–975, September 1997.

[19] V.J. Milenkovic. *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic*. PhD thesis, Carnegie-Mellon, 1988.

[20] J-M. Moreau. *Facétisation et hiérarchisation de la représentation par segments d'un graphe planaire dans le cadre d'une arithmétique mixte*. PhD thesis, ENSM.SE, 1990.

[21] J-M. Moreau. Hierarchical Delaunay triangulation. In *Proceedings of the 6th Canadian Conference on Computational Geometry*, pages 165–170, Saskatchewan, Canada, August 2-6, 1994.

[22] J-M. Moreau and P. Volino. Delaunay triangulation revisited. In *Proceedings of the 5th Canadian Conference on Computational Geometry*, pages 340–345, Waterloo, Canada, August 5-9, 1993.

[23] K. Mulmuley. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice-Hall, Englewood Cliffs, N.J., 1994.

[24] F.P. Preparata and M.I. Shamos. *Computational Geometry – An Introduction*. Springer-Verlag, New York, N.Y., 1985.

[25] R. Sedgewick. *Algorithms in C*. Addison-Wesley, Reading, Mass., 1990.

[26] R. Seidel. Constrained Delaunay triangulation and Voronoi diagram with obstacles. Technical Report 260, IIG-TU Graz, Austria, 1988.

[27] R. Seidel. Backwards analysis of randomized geometric algorithms. In *New Trends in Discrete and Computational Geometry, J. Pach, ed., Springer-Verlag*, pages 37–67, Berlin, 1991.

[28] K. Sugihara and M. Iri. Two design principles of geometric algorithms in finite-precision arithmetic. *Appl. Math. Letter*, 2(2):203–206, 1989.

[29] M. Teillaud. Towards dynamic randomized algorithms in computaional geometry. Technical Report 1727, INRIA, 1992.

[30] C.K. Yap. Towards exact geometric computation. In *Proceedings of the 5th Canadian Conference on Computational Geometry*, pages 405–419, Watreloo, Ontario, Aug. 5-9, 1993.