Rewriting-based derivation of efficient algorithms to build planar subdivisions [*]

David Cazier and Jean-François Dufourd

L.S.I.I.T. (URA CNRS 1871)
Université Louis Pasteur, Dép. Informatique
7, rue René Descartes, 67084 Strasbourg Cedex

E-mail: {cazier, dufourd}@dpt-info.u-strasbg.fr

Abstract. Algebraic specifications allied to rewriting are used more and more often in design and logical prototyping of programs. We show how these techniques can be applied to a basic problem in computational geometry, namely the construction of planar subdivisions. We build up a simple, complete and convergent system of rules to cope with this problem and show how it is transformed to describe concrete and efficient algorithms such as plane-sweep ones.

1 Introduction

Boolean (or set) operations in the plane is a crucial task in computational geometry, deserving a faultless and formal definition. Boolean operations amount to the refinement of superposed subdivisions, and thus, are an extension of line arrangements (see Bentley-Ottmann algorithm [1]). We generalize it to the self-refinement of embedded combinatorial maps. These problems have been approached incrementally [2] or more generally [3, 4, 5], the resulting algorithms beeing described by conventional methods.

Our works rely on the use of algebraic specifications [6, 7] allied to rewriting [8]. Formal objects are described with abstract data type generators and operations whose behaviour is modeled by equations. Specifications can be made operational in a term rewriting system [8], with a correct orientation of equations and sometimes completion. Then, techniques of logical prototyping can be used to point out possible design errors. These techniques have been fulfilling in computer graphics languages [9, 10], mechanical proof in geometry [11], and geometric modeling [12].

The method we propose is more abstract, general and productive than conventional ones for our geometric problem. It allows us to completely and logically define the self-refinement. Then, by appropriate choices of control structures and strategies, we are able to derive all self-refinement algorithms, from the most naive until the most efficient ones.

Finally, we know that the question of numeric approximation is essential in this kind of problems. But this paper is essentially turned into topological issues and eludes the difficulties due to real numbers which would need studies in themselves [3].

[*]This research is supported by the *GDR de Programmation* and the *GDR Algorithmique, modèles et infographie* (French CNRS).

In section 2, we define the self-refinement of subdivisions. In section 3, we specify maps in a formal and precise algebraic framework. In section 4 we describe self-refinement as a simple, convergent rewrite system. In sections 5 and 6, we describe, in terms of rewriting, a formal method to construct robust and efficient algorithms that result in successive changes on the simplest system. In section 7, we give some conclusions and present future works.

2 Self-refinement of subdivisions

To compute the union, intersection or difference between planar subdivisions, a convenient way is to construct a new subdivision where superposed edges or vertices are merged, intersecting edges are cut and edges are cut at existing incidence points. The result of the boolean operations can then be obtained from this *corefinement* by selecting the required parts. We generalize this idea through the notion of *self-refinement* of a set of vertices, edges and faces. This set can contain any number of subdivisions. The self-refinement consists in transforming it until it represents a (single) subdivision of the plane. In figure 1, two subdivisions (a) are superposed to form a set (b), which is self-refined (c).

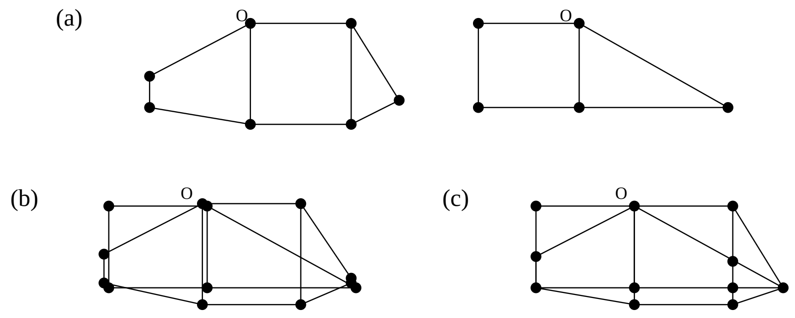

Figure 1: Example of self-refinement

It is usual now to distinguish topology from embedding. Topology defines vertices, edges and faces of objects, as well as theirs incidence relationships. Embedding concerns the position and shape of these cells. Combinatorial embedded maps supply an easy, precise and concise description of subdivisions [13]. Interest of topology was shown in [14] that presents a good alternative to describe subdivisions and topological operators.

3 Formal specification of subdivisions

Let us recall that a *combinatorial map* is a triplet (B, α_0, α_1) where B is a finite set of *darts*, α_0 is an involution on B without fixed points, i.e. a permutation such that $\alpha_0(\alpha_0(x)) = x$ and $\alpha_0(x) \neq x$ for every x, and α_1 is a permutation on B. Two darts are said to be *linked* with respect to α_0 (resp. α_1), or 0-linked (resp. 1-linked), if they belong to the same orbit with respect to α_0 (resp. α_1). Darts are usually interpreted as half-edges. Thus, two 0-linked darts form a *topological edge* and an orbit with respect to α_1 defines a *topological vertex* of the map. Figure 2 (a) presents the drawing conventions.

Example 3.1 Figure 2 (b) shows a map with B = {1, ..., 7, -1, ..., -7} and, in cyclic notation, α_0 = (-1, 1) (-2, 2) (-3, 3) (-4, 4) (-5, 5) (-6, 6) (-7, 7) and α_1 = (1, 2) (-2, 3) (-3, -4, 7) (4, -6) (-7, 6, 5, -1) (-5). Thus α_0(1) = -1, α_0(-1) = 1, and the orbit <α_0>(1) = {1,

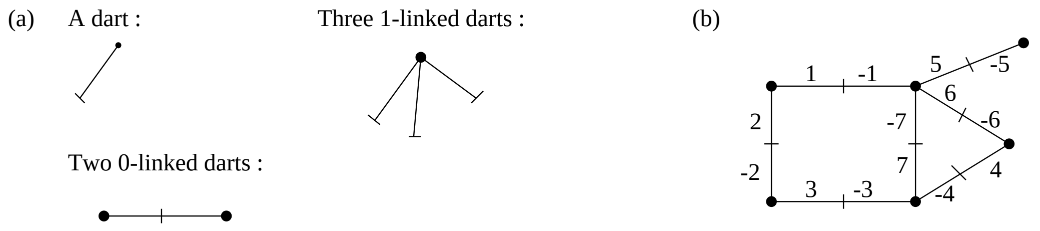

Figure 2: Conventions and example of combinatorial map

-1} defines an edge. Similarly, $\alpha_1(6)$ = 5, $\alpha_1(5)$ = -1, $\alpha_1(-1)$ = -7, $\alpha_1(-7)$ = 6, <α_1>(6) = {5, -1, -7, 6}, and the vertex dart 6 belongs to contains darts -7, 6, 5 and -1. □

The *geometry* of subdivisions is described by the embedding of maps. Each topological part is associated with a geometrical object of the same dimension. Points are associated with vertices (0-embedding) and *Jordan arcs* with edges (1-embedding). Figure 3 presents the graphical conventions for the embedding. In the following, only 0-embeddings are considered. Thus, edges are implicitely 1-embedded on line segments.

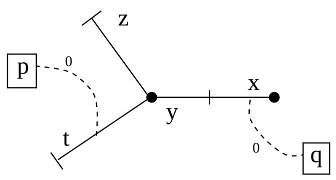

Figure 3: Graphical convention and example for embeddings

To formalize the notion of subdivision and to avoid problems due to data structures and programming languages, we define maps and operations handling them through an algebraic specification [6]. Maps are defined from four basic functional *generators*: $v, l0, l1, em0$ [12]. Generator v creates the empty map, $l0(M, x, y)$ and $l1(M, x, y)$ link in map M dart x to y with respect to α_0 or α_1. Generator $em0(M, x, p)$ embeds dart x on point p. Note that the order these operations are applied is indifferent. A map is described by first order equivalent terms where applications of $l0$, $l1$ and $em0$ are the same but permutated. Thus, the map of figure 3 can be written $l1(em0(l1(l1(l0(em0(v, x, q), x, y), y, z), z, t), t, p), t, y)$.

A set of functional *selectors* and *constructors* on maps is then defined through a first order equational theory. For instance, we build topological selectors to compare in map M the vertices or edges x and y belong to: $eqv(M, x, y)$ and $eqe(M, x, y)$. We also define geometrical selectors to compare embeddedings of vertices or edges: $eqev(M, x, y)$ and $eqee(M, x, y)$. Among the constructors that modify the topology and the geometry of a map, we define $cutee(M, x, p)$ that cuts at point p the edge x belongs to and $merge(M, x, y)$ that merges two distinct vertices reordering the 1-links of these two vertices. Finally, the destructor $dv(M, x)$ deletes dart x from the vertex it belongs to. Other easily understandable selectors appear in the rules of section 4.

A map is *planar* if it can be embedded without any self-intersection or overlapping. That way, self-refinement of any embedded map have to transforms it into a planar map that, in fact, correctly models a planar subdivision. Thus, the result must satisfy the following conditions: (i) all darts of a vertex are embedded on a same point; (ii) distinct vertices are embedded on distinct points; (iii) vertices do not overlap any edge; (iv) distinct edges do not intersect themselves; (iv) vertices are *arranged*, i.e. consistently embedded w.r.t. α_1.

The self-refinement we define is a generic description of all refinement problems. Particular algorithms can be described by restricting the kinds of used starting sets. Maps are used in order to carry boolean operations. If a set of segments is used, self-refinement corresponds to intersections finding algorithms as discussed in [4, 5] or polygones clipping.

4 Rewrite system for the self-refinement of maps

We define the self-refinement of maps through a set of elementary and independent operations that are nicely described as rules of a *conditional modulo rewrite system* [8]. In the rules of table 1, numerators represent a map and denominators represent it after one rewrite step. Rules can be applied only when the conditions are satisfied. Rules are graphically depicted in figure 4.

$$\mathrm{R}_1 : \frac{M}{dee(M,x)} \ \textbf{if} \begin{cases} x \in M \\ nullee(M,x) \end{cases} \qquad \mathrm{R}_4 : \frac{M}{cutee(cutee(M,x,i),z,i)} \ \textbf{if} \begin{cases} x \in M \wedge z \in M \\ secant(M,x,z) \end{cases}$$

$$\textbf{with} \quad i = intersection(x,z)$$

$$\mathrm{R}_2 : \frac{M}{dee(M,z)} \ \textbf{if} \begin{cases} x \in M \wedge z \in M \\ \neg\ eqe(M,x,z) \\ eqee(M,x,z) \end{cases} \qquad \mathrm{R}_5 : \frac{M}{merge(M,x,z)} \ \textbf{if} \begin{cases} x \in M \wedge z \in M \\ \neg\ eqv(M,x,z) \\ eqev(M,x,z) \\ mergeable(M,x,z) \end{cases}$$

$$\mathrm{R}_3 : \frac{M}{cutee(M,x,p)} \ \textbf{if} \begin{cases} x \in M \wedge z \in M \\ \neg\ nullee(M,z) \\ incident(M,z,x) \end{cases} \qquad \mathrm{R}_6 : \frac{M}{dv(M,x)} \ \textbf{if} \begin{cases} x \in M \\ \neg\ arranged(M,x) \end{cases}$$

Table 1: Rewrite system for embedded map self-refinement

Rule R1 deletes a null edge (represented as a loop in figure 4). If a dart x in map M ($x \in M$) belongs to a null edge ($nullee(M, x)$), this edge is deleted ($dee(M, x)$). If two edges are superposed, rule R2 deletes the second one. Thus, if x and z belong to distinct edges ($\neg eqe(M, x, z)$) and are 1-embedded on equal segments ($eqee(M, x, z)$), the edge z belongs to is deleted from M. Rule R3 performs the incidence cutting. It cuts in two parts an edge incident to a vertex. If z is 0-embedded on a point p incident to the edge x belongs to, then this edge is cut at p (obtained by the $gem0(M, z)$ selector). Rule R4 realizes the intersection cutting. If x and z belong to secant edges, these edges are cut at the intersection point i.

The two last rules handle vertices. Rule R5 merges two vertices embedded on equal points, if possible. Thus, if x and z belong to distinct vertices ($\neg eqv(M, x, z)$) and are 0-embedded on equal points ($eqev(M, x, z)$) then their two vertices are merged. This fusion can be done if the two vertices are mergeable, i.e. if each one is arranged. Finally, rule R6 deletes a dart from a non-arranged vertex. Thus if x belongs to a non-arranged vertex, it is deleted from the vertex it belongs to. Dart x and its previous vertex can be remerged later by rule 5, to give a well arranged vertex.

Our rewrite system is *convergent* [8] as explained in a previous paper [15]. This implies that, for each map, the rewriting is terminating, what is proved using [16], and leads to a unique normal form modulo map equality. Therefore, the rewrite system can be seen as a function of map normalization which projects any map into its self-refinement. Thus, the use of rewriting techniques allows us to express a complex problem as a set of elementary transformations. Let us see now how this can be used to derive concrete algorithms.

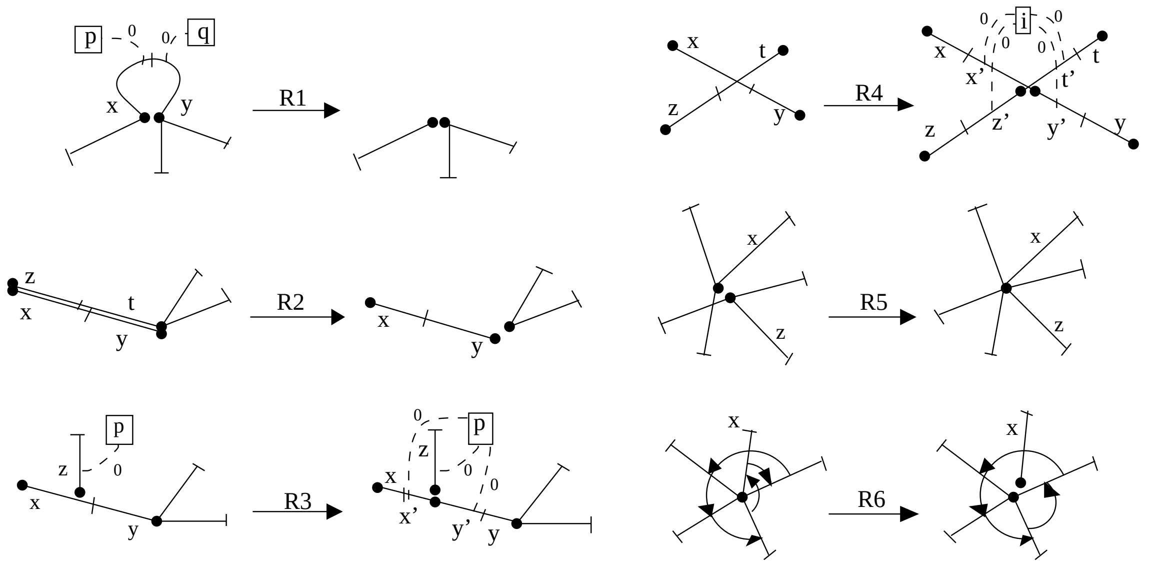

Figure 4: Graphical illustration of the rewrite rules

5 Deriving simple concrete algorithms

A naïve use of the rewrite system described above is to test, for each dart or couple of darts, if a rule can be executed. The corresponding algorithm has the following shape:

```
Repeat
    Choose $x$ in $M$
    Try to execute rule 1 and 6 with $x$
    Repeat
        Choose $z$ in $M$
        Try to execute rule 2 to 5 with $x$ and $z$
    Until no rule can be executed with $x$
Until no rule can be executed
```

Such an abstract algorithm is not deterministic, because darts are randomly chosen. To describe a concrete (i.e. real and efficient) algorithm, we have to hold a strategy to choose darts. We achieve this goal adding to the rewrite system *control structures* that yield the dart or couple of darts that is going to be examined. A rewrite rule then describes the transformations of the map *and* those of the control structures.

The simplest control structures we can use are lists of darts. In this case, the algorithm begin with a list L of all darts. A dart is removed from L when no rule can be executed with it. The algorithm terminates when the list is empty. As we need to choose couples of darts, we have to handle another list L' for each chosen dart of L. The chosen darts x and z are always at the head of the two current lists.

Table 2 shows the rewrite system R_L where the rules of R have been transformed to take into account the two lists used as control structures. To simplify, only rule 1 and 4 are shown. The other ones are transformed the same way. The lists are generated by the head-constructor $[x|L]$ that adds dart x ahead of list L. The destructor $d(L,x)$ deletes x from L. Thus, in rule 1, the darts of the deleted edges are removed from the two lists. In rule 4, darts x', y', z' and t', that are created when edges are cut, are simply added ahead of L, because we are not concerned about their order. As obviously these new darts cannot interfer with x, they are not added to L'.

$$\mathrm{R}_{L1} : \frac{L,\ L',\ M}{d(d(L,x),y),\ d(d(L',x),y),\ dee(M,x)} \ \textbf{if} \begin{cases} x = first(L) \wedge y = \alpha_0(M,x) \\ nullee(M,x) \end{cases}$$

$$\mathrm{R}_{L4} : \frac{L,\ L',\ M}{[x',y',z',t'|L],\ L',\ cutee(cutee(M,x,i),z,i)} \ \textbf{if} \begin{cases} x = first(L) \wedge z = first(L') \\ secant(M,x,z) \end{cases}$$

$$\mathrm{R}_{L7} : \frac{L,\ L',\ M}{L,\ d(L',z),\ M} \ \textbf{if} \begin{cases} z = first(L') \\ \neg(R_{L2} \vee \ldots \vee R_{L5}) \end{cases}$$

$$\mathrm{R}_{L8} : \frac{L,\ [\,],\ M}{d(L,x),\ d(L,x),\ M} \ \textbf{if} \begin{cases} x = first(L) \\ \neg(R_{L1} \vee R_{L6}) \end{cases}$$

Table 2: Lists structured rewrite system

Rules 7 and 8 work on control structures when no previous rule can be executed. If z cannot be used to execute a rule with a given x, its successor in L' is taken. Thus, when rules 2 to 5 cannot be executed, what we denote $\neg(R_{L2} \vee \ldots \vee R_{L5})$, the head of L' is removed. Thus, the system goes on with x and the next dart in L'. When L' is empty, x has been checked against all other darts. Rule 8 then removes dart x from L and initializes the second list. This way, all couples of darts are examined only once.

An algorithm can be easily derived from R_L. The rewrite rules share the common conditions $x = first(L)$ and $z = first(L')$. To construct our algorithm, we factorize them and handle directly the control rules R_{L7} and R_{L8} and the two lists. The produced algorithm has the following structure:

```
$L$ = list-of-dart($M$)
While $L \neq [\,]$
    $x$ = first($L$)
    Try to execute rule 1 with $x$
    If rule 1 fails Then
        Try to execute rule 6 with $x$
        $L'$ = $L$
        While $L' \neq [\,]$
            $z$ = first($L'$)
            Try to execute rule 2 with $x$ and $z$
            If rule 2 fails Then
                Try to execute rule 3 to 5 with $x$ and $z$
                $L'$ = $d(L', z)$
        End {while $L'$}
        $L$ = $d(L, x)$
End {while $L$}
```

If we count the number of attemps of rule executions, this simple algorithm has a complexity in $O((n+i)^2)$, n being the number of darts and i the number of interacting darts.

The rewrite system R_L we obtain is a formal, precise and complete description of a strategy to compute maps self-refinement. This strategy is not guessed by a global approach. But, as the rewrite rules are independant, it is constructed by a step by step study of the different cases and the changes on the rewrite system are simply made rule by rule.

6 Deriving efficient algorithms

6.1 Use of geometrical properties to choose darts

When we use lists as control structures, we avoid choosing darts and couples of darts more than once. However all couples of darts are examined. Geometrical properties can be exploited to avoid examining couples of darts that cannot interact. If D and D' are the vertical lines that pass through the vertices of an edge $\{x, y\}$, then the darts that can interact with dart x or y are those whose vertices stand in the plane region lying between D and D'.

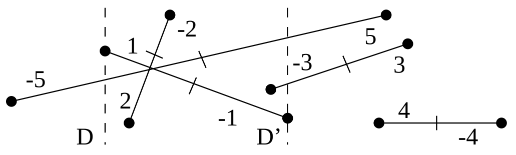

Figure 5: Plane region defined by edge {1,-1}: 2, -2 and -3 are relevant darts

The rewrite system is transformed to use this property. Darts x are examined from left to right. Darts z are taken in the plane region defined by the edge of x. For this reason, we use sorted lists of darts or *priority queues* (implemented with heaps) as control structures. In fact, rules 1 to 6 do not change, x and z are the first elements of the corresponding priority queues. The main changes appear in rules 7 and 8.

We define an order $<_M$ on darts in a map M. If x and y are 0-embedded on points p and q in M, then $x <_M y$ means that p is lower than q in the lexicographic order on points' coordinates. As darts are examined from left to right, we have $x \leq_M z$. Therefore, darts of the second priority queue are examined as long as they stand in the region of x, what setting $y = \alpha_0(M, x)$, we write:

$$\begin{cases} z \leq_M y & \text{if } x <_M y \\ z \leq_M x & \text{if } x \geq_M y \end{cases} \Rightarrow \begin{cases} z \leq_M y & \text{if } x <_M y \\ eqev(M,x,z) & \text{if } x \geq_M y \end{cases} \Rightarrow [x <_M y \wedge z \leq_M y] \vee eqev(M,x,z)$$

as $x \leq_M z \wedge z \leq_M x$ implies that x and z are 0-embedded on equal points. Finally the third condition is used in rules 7 and 8 to check if z interacts with the current x, as shown in table 3.

$$\mathrm{R}_{L7} : \frac{L,\ L',\ M}{L,\ d(L',z),\ M} \ \textbf{if} \begin{cases} x = first(L) \wedge z = first(L') \\ \neg(R_{L2} \ldots R_{L5}) \\ [x <_M y \wedge z \leq_M y] \vee eqev(M,x,z) \end{cases} \qquad \textbf{with } y = \alpha_0(M,x)$$

$$\mathrm{R}_{L8} : \frac{L,\ L',\ M}{d(L,x),\ d(L,x),\ M} \ \textbf{if} \begin{cases} x = first(L) \wedge z = first(L') \\ \neg(R_{L1} \vee R_{L6}) \\ x <_M y \wedge y <_M z \wedge \neg\, eqev(M,x,z) \end{cases} \qquad \textbf{with } y = \alpha_0(M,x)$$

Table 3: Rule 7 and 8 for a rewrite system structured with priority queues

6.2 A plane-sweep algorithm

This simple plane-sweep algorithm is not optimal since all darts of the region defined by an edge are examined. To avoid this, we sort the edges of this region from bottom to top.

Thus, there are only two relevant darts to examine, namely the dart whose edge is just below edge $\{x, y\}$ and the one whose edge is just above. To avoid computing for each x the sorted *set of active edges*, this set is maintained by each rewrite rule as a *dictionary.*

The rewrite system begins with all darts ordered from left to right and put in a priority queue X and with an empty dictionary A. Dart x is the first element of X and z is chosen by searching in A the two darts that are just below and above x. Moreover, as equal vertices are behind each others in X, the merging can only occur for two consecutive vertices. So, we add a structure S that only contains the last used vertex.

$$\mathrm{R}_{S1} : \frac{X,\ A,\ S,\ M}{d(d(X,x),\ \alpha_0(M,x)),\ A,\ S,\ dee(M,x)} \ \textbf{if} \begin{cases} x = first(X) \\ nullee(M,x) \end{cases}$$

$$\mathrm{R}_{S4} : \frac{X,\ A,\ S,\ M}{i(i(i(i(X,x'),y'),z'),t'),\ A,\ S,\ cutee(cutee(M,x,i),z,i)} \ \textbf{if} \begin{cases} x = first(X) \\ z = gsecant(A,M,x) \neq nil \end{cases}$$

$$\mathrm{R}_{S5} : \frac{X,\ A,\ \{z\},\ M}{X,\ A,\ S,\ merge(M,x,z)} \ \textbf{if} \begin{cases} x = first(X) \\ \neg\ eqv(M,x,z) \wedge eqev(M,x,z) \wedge mergeable(M,x,z) \end{cases}$$

$$\mathrm{R}_{S7} : \frac{X,\ A,\ S,\ M}{d(X,x),\ i(A,x),\ \{x\},\ M} \ \textbf{if} \begin{cases} x = first(X) \wedge leftdart(M,x) \\ \neg\ (R2 \vee \ldots \vee R5) \end{cases}$$

$$\mathrm{R}_{S7'} : \frac{X,\ A,\ S,\ M}{i(d(X,x),z),\ d(d(A,\alpha_0(M,x)),z),\ \{x\},\ M} \ \textbf{if} \begin{cases} x = first(X) \wedge rightdart(M,x) \wedge \neg\ (R2 \ldots R5) \\ z = gbelow(A,M,x) \wedge z' = gabove(A,M,x) \\ interact(M,z,z') \end{cases}$$

$$\mathrm{R}_{S7''} : \frac{X,\ A,\ S,\ M}{d(X,x),\ d(A,\alpha_0(M,x)),\ \{x\},\ M} \ \textbf{if} \begin{cases} x = first(X) \wedge rightdart(M,x) \wedge \neg\ (R2 \ldots R5) \\ z = gbelow(A,M,x) \wedge z' = gabove(A,M,x) \\ \neg\ interact(M,z,z') \vee z = nil \vee z' = nil \end{cases}$$

Table 4: Rewrite system with plane sweep strategy

In the rewrite system R_S displayed in table 4, only the relevant rules 1, 4 and 5 are described. The other ones are similar. The i and d operators insert and delete darts. As before, in rule 1, the darts of the deleted edge are removed from X. In rule 4, the $gsecant(A, M, x)$ operator searches within A if either the edge just below or just above x is secant to the edge x belongs to and returns it or *nil* if it cannot be found. Then the new darts are inserted in X. The last changes appear in rule 5. As we said before, the only dart that can be merged with x is the last examined dart that has been placed in S. So z is taken in S.

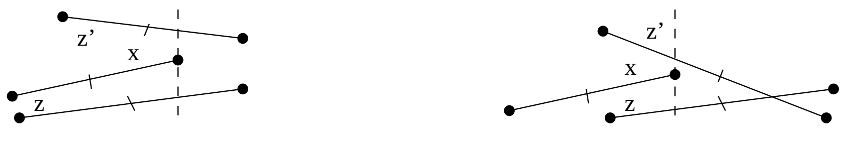

Figure 6: The two cases of right dart inactivation

Rules 7, $7'$ and $7''$ are used to maintain X, A and S when the rewrite system sweeps from left to right, what occurs when rules 1 to 6 cannot be executed. In these three rules dart x is placed in S. If x is a left dart, i.e. x is the left vertex of its edge, it is deleted from X and becomes active as rule 7 inserts it in A. On the other hand, if x is a right dart, it is removed from X and its edge is removed from A. In the second case, the edges that are just below and above x have never been checked together and can interact as shown in figure 6. Thus, if these edges are *nil* or do not interact there is nothing more to do (rule 7"). Else, dart z, whose edge is just below x, is removed from A and reinserted in X to be examined once more (rule 7').

As before, a classical algorithm can be derived from the rewrite system. We obtain a classical plane sweep algorithm [1, 4], the complexity of which is in $O((n+i)\ln(n))$. We think that more complex algorithms, like those of [5] that adds new edges to make the subdivision's faces convexe and have a complexity in $O(n\ln(n)+i)$ thanks to this improvement, can be rigorously designed that way.

```
$X$ = set of darts of $M$ sorted by $x$-coordinates
$A$ = $\emptyset$
$S$ = $\emptyset$
While $X \neq \emptyset$
    $x$ = first($X$)
    Try to execute rule 1 and 6 with $x$
    get $z$ from $A$ and $x$
    Try to execute rule 2 to 4 with $x$ and $z$
    get $z$ from $S$
    Try to execute rule 5 with $x$ and $z$
    remove $x$ from $X$
    $S = \{x\}$
    If $x$ is a leftdart
        Then add $x$ to $A$
    Else $\{x$ is a rightdart$\}$
        remove $\alpha_0(M,x)$ from $A$
        $z$ is the edge below $x$ in $A$
        $z'$ is the edge above $x$ in $A$
        If $z$ and $z'$ interact and $z \neq nil$ and $z' \neq nil$
            Then remove $z$ from $A$, insert $z$ in $X$
        Endif
    Endif
End $\{$while $X\}$
```

We have proposed a general mechanism to describe any concrete self-refinement algorithm. Different control structures lead to different algorithms. A classification of refinement algorithms can thus be done. It is based upon the kind of structures and the kind of research functions that are used. The interest of this kind of classification is the clear separation between data structures used to handle maps and data structures used to improve control and thus complexity of the algorithms.

7 Conclusions

We have defined topological and geometrical operations to construct and handle planar subdivisions. To achieve this, we have based our approach on the combinatorial map mathematical model and on rewriting techniques, which led us to express a complex problem at different levels as a set of elementary and independent transformations. The result is a complete formal specification and a derivation of efficient algorithms. Finally, we proove that specification formalism and rewriting expressiveness produce a safe and rigorous algorithm design, even in computational geometry.

In a first stage, operations on maps and rewrite rules were implemented in Prolog. Using this logical prototyping, we were able to check quickly, in a practical way, the validity of our specifications. In a second stage the different algorithms we defined have been implemented in C, what allowed us a practical study of the different strategies and control structures for rules application.

Our prospects are to complete the rewrite systems to handle more complex structures, like 2- and 3-generalized maps which enable to describe topological varieties [13], to go about the problems of 3D boolean operations that involve a lot of difficulties. Another project is to deal with the numeric approximations. We feel that our approach, which separates on one hand topology and embedding and on the other hand logic and control, can help to locate the sensible points were these questions have to be treated.

References

[1] J.L. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, 28:643–647, 1979.

[2] J.F. Dufourd, C. Gross, and J.C. Spehner. A digitization algorithm for the entry of planar maps. In *Proc. Computer Graphics International Conf.*, pages 649–661, Leeds, U.K., 1989. Springer-Verlag.

[3] M. Gangnet, J.C. Hervé, T. Pudet, and J.M. van Thong. Incremental computation of planar maps. In *Proc. of ACM Siggraph Conf., Boston, Computer Graphics*, volume 23, pages 345–354, July 1989.

[4] J. Nievergelt and F.P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Com. of ACM*, 25(10):739–747, 1982.

[5] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of ACM*, 39(1):1–54, 1992.

[6] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 1. Equations and initial semantics*, volume 6 of *EATCS Monograph on Theoretical Computer Science.* Springer, 1985.

[7] M. Wirsing. Algebraic specifications. In *Formal models and semantics*, Handbook of Theoretical Computer Science, chapter 13, pages 675–788. Elsevier, 1990.

[8] N. Dershowitz and J.P. Jouannaud. Rewrite systems. In *Formal models and semantics*, Handbook of Theoretical Computer Science, chapter 6, pages 243–320. Elsevier, 1990.

[9] W.R. Mallgren. *Formal specification of interactive graphic programming languages.* ACM Dist. Dissertation. MIT Press, USA, 1982.

[10] D.A. Duce, E.V. Fielding, and L.S. Marshall. Formal specification of a small example based on GKS. *ACM Trans. on Graphics*, 7(3):180–197, 1988.

[11] B. Brüderlin. Using geometric rewrite rules for solving geometric problems symbolically. *Theoretical Computer Science*, 116:291–303, 1993.

[12] Y. Bertrand and J.F. Dufourd. Algebraic specification of a 3D-modeler based on hypermaps. *CVGIP : Graphical Models and Image Processing*, 56(1):29–60, 1994.

[13] P. Lienhardt. Topological models for boundary representation : a comparison with n-dimensional generalized maps. *Computer Aided Design*, 23(1):59–82, 1991.

[14] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoï diagrams. *ACM Trans. on Graphics*, 4(2):74–123, April 1985.

[15] D. Cazier and J.F. Dufourd. A rewrite system to build planar subdivisions. In *Proc. Canadian Conf. on Computational Geometry*, pages 235–240, Québec, 1995.

[16] E. Bevers and J. Lewi. Proving termination of (conditional) rewrite systems. A semantic approach. *Acta Informatica*, 30:537–568, 1993.