Prédicats pour le diagramme de Voronoï de segments et CGAL

Christophe Delage

Juin & Juillet 2001

Résumé

Ce rapport rend compte d'un stage MIM1 effectué a L'INRIA Sophia Antipolis dans l'équipe PRISME entre le 11 juin et le 3 août 2001, sous la direction de Monique Teillaud[1]. Nous présenterons tout d'abord le sujet du stage, ensuite, après avoir introduit la notion de diagramme de Voronoï, les prédicats nécessaires seront décrits. Puis l'implémentation à l'intérieur de la bibliothèque CGAL sera décrite.

Mots-clé : Géométrie algorithmique, diagramme de Voronoï.

INRIA Sophia Antipolis, 2004 route des lucioles BP 93, 06902 Sophia Antipolis.
Tél : +33 (0)4 92 38 77 77, Fax : +33 (0)4 92 38 77 65

[1]Monique.Teillaud@sophia.inria.fr, `http ://www-sop.inria.fr/prisme/personnel/teillaud/`

Table des matières

1 Introduction **3**

2 Diagramme de Voronoï de segments **3**

3 Prédicats **4**
- 3.1 Région 4
- 3.2 Petit problème. 5
- 3.3 Solution 5
- 3.4 Déroulement du(des) prédicat(s) 7
  - 3.4.1 Cas fini 7
  - 3.4.2 Cas infini 8
  - 3.4.3 Le test *in circle* 8
  - 3.4.4 Tests supplémentaires 9
  - 3.4.5 Autres test supplémentaires 10

4 Implémentation dans CGAL **12**
- 4.1 Types de nombres 12
- 4.2 Exactitude *vs.* performance 13
- 4.3 Trait class 13
- 4.4 Description du code 14
  - 4.4.1 Parabola.h 14
  - 4.4.2 Conflict_type.h 14
  - 4.4.3 Site_2.h 15
  - 4.4.4 voronoi_kernel_2.h 15
  - 4.4.5 voronoi_check_vertices_2.h 15
  - 4.4.6 voronoi_compute_vertices_2.h 16
  - 4.4.7 voronoi_constructions_2.h 16
  - 4.4.8 voronoi_in_circle_2.h 16
  - 4.4.9 voronoi_middle_2.h 16
  - 4.4.10 voronoi_predicates_2.h 17
  - 4.4.11 voronoi_unbounded_conflict_2.h 17
  - 4.4.12 voronoi_Site_constructions_2.h et voronoi_Site_predicates_2.h 17
  - 4.4.13 Traits classes 17
  - 4.4.14 Jeu de tests 18

5 Conclusion **18**

1 Introduction

Les diagrammes de Voronoï sont des structures fondamentales de la géométrie algorithmique. On les retrouve d'ailleurs dans la nature (les écailles des tortues ou les taches sur le cou des girafes, par exemple, sont des diagrammes de Voronoï de points).

Le cas du diagramme de Voronoï de segments est un peu à part : Il est bien compris théoriquement depuis longtemps mais il est très difficile à implémenter (à cause en particulier du haut degré[2] des polynômes en jeu dans les prédicats).

On ne peut pas utiliser naïvement un calcul en virgule flottante, cela conduirait souvent, par des erreurs d'arrondi, à des résultats faux (pas seulement imprécis !) ou même à un plantage de l'algorithme sur des entrées pourtant correctes.

La bibliothèque CGAL (Computational Geometry Algorithms Library) n'implémente pas encore le diagramme de Voronoï de segment. Le but du stage était donc de programmer les prédicats géométriques nécessaires à l'algorithme (un autre stagiaire, arrivé trois semaines avant mon départ s'est occupé de la partie algorithmique).

2 Diagramme de Voronoï de segments

Soit $\mathcal{S}$ un ensemble de segments du plan euclidien[3] $\mathcal{P}$.

Definition 1 *La* cellule de Voronoï *de $s \in \mathcal{S}$ est :*

$$VR(s) = \{p \in \mathcal{P}\,;\, \forall t \in \mathcal{S},\ d(p,s) \leqslant d(p,t)\}$$

Definition 2 *Le* diagramme de Voronoï *de $\mathcal{S}$ est la réunion des bords des cellules des segments de $\mathcal{S}$.*

On partitionne donc le plan en cellules autour des segments. La figure 1 montre un exemple. En supposant que les segments ne s'intersectent pas sauf éventuellement à leurs extrémités, le diagramme de Voronoï est un graphe planaire.

L'algorithme qui sera programmé est incrémental, c'est-à-dire que l'on construit le diagramme en rajoutant les segments au fur et à mesure. L'idée

[2] jusqu'à 40, ou plus suivant la façon de les formuler

[3] On peut très bien définir des diagrammes de Voronoï dans un cadre beaucoup plus général. Mais, à ma connaissance, personne n'a osé tenter de calculer un diagramme de Voronoï de segments dans autre chose qu'un plan euclidien.

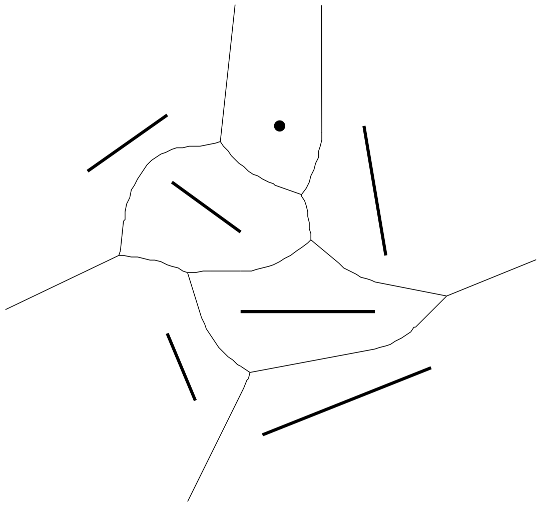

FIG. 1 – Exemple de diagramme de Voronoï.

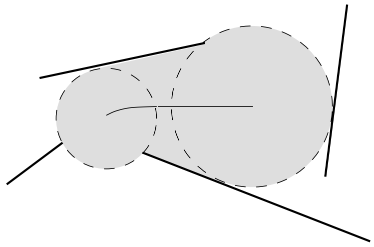

FIG. 2 – Une région de Voronoï (en grisé).

initiale était de suivre ce qui est décrit dans [5] On doit donc savoir quelles sont les arêtes du diagramme qui vont disparaître, être raccourcies, coupées en deux, *etc.* après l'insertion du nouveau segment.

3 Prédicats

3.1 Région

Soient p, q, r et s quatre segments. On définit leur région de Voronoï comme l'union des cercles tangents à p et q qui ne contiennent ni r, ni s (*cf.* figure 2).

Pour savoir comment un nouveau segment affecte une arête, on fait un test d'intersection avec :

– les cercles extrêmes, cela permet de savoir si l'une des (ou les deux) extrémités sont mangées par le nouveau segment ;

– le reste de la région (cela se ramène à des tests d'orientation et des comparaisons de distance), on sait comme cela si l'intérieur de l'arête disparaît.

3.2 Petit problème...

Étant donnés trois segments, il est difficile de calculer le centre du cercle tri-tangent.

En effet, on peut calculer le cercle tangent à trois points, à deux points et une droite, *etc.* mais comment savoir à l'avance, pour chaque segment, s'il est tangent au cercle en une extrémité ou à l'intérieur ?

Il y a plusieurs années, une stagiaire dans le projet avait implémenté le diagramme de Voronoï de segment. A chaque fois que son programme devait calculer un tel cercle, il essayait tous les cas (il y en a 27 [4]), et regardait pour chacun d'eux si c'était le bon (si les segments ne coupent pas le cercle).

Ce n'est pas trop coûteux si on considère qu'elle ne faisait cette construction qu'une seule fois (donc, en tout, seulement une fois par sommet du diagramme). Malheureusement, pour des raisons qui seront claires plus tard (*cf.* section 4.2), il n'est pas raisonnable de se servir de données calculées précédemment et stockées si l'on souhaite un prédicat efficace et correct.

3.3 Solution

On considère séparément les segments ouverts et leurs extrémités (comme dans Burnikel [1]), décider si on identifie un segment à une de ses extrémités ou à la droite qui le porte n'est donc plus nécessaire. De plus on insère toujours un segment après ses extrémités, cela permet de simplifier énormément les tests géométriques.

On rajoute donc au diagramme des arêtes entre un segment et chacune de ses extrémités : (un morceau de) la droite perpendiculaire au segment passant par l'extrémité. En cas de segments adjacents, une arête disparaît (*cf.* figures 3 et 4).

Cela a tout de même quelques inconvénients :
- le diagramme est légèrement différent (mais c'est mineur) ;
- comme il y a nettement plus d'arêtes dans le diagramme, une insertion coûtera probablement plus de tests de conflit ;
- il y a trois fois plus d'insertion à faire.

Pourtant, il semblerait ([1] et [2]) que la simplicité (et donc la rapidité) des tests soit suffisante pour qu'un algorithme utilisant ces prédicats soit

[4] 3 pour chaque segment (les deux extrémités plus l'intérieur) et 3 segments, donc $3 \cdot 3 \cdot 3$

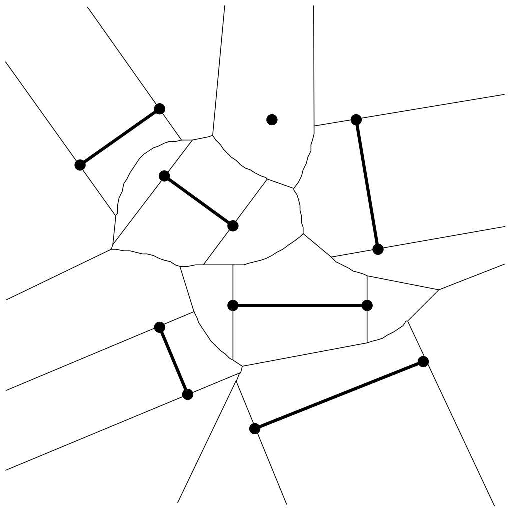

Fig. 3 – Diagramme de Voronoï, version 2.

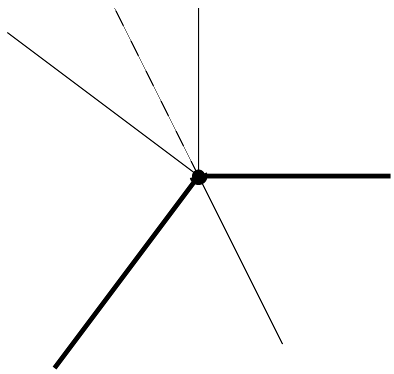

Fig. 4 – Apparition et disparition d'arête...

plus rapide.

Dans la suite, j'appellerai *site* un objet qui est un point ou un segment ouvert.

3.4 Déroulement du(des) prédicat(s)

Nous avons en effet plusieurs prédicats à faire (pour chaque site entrant en jeu, est-ce un point ou un segment ?). De plus, dans un diagramme de Voronoï il y a des arêtes finies et des arêtes infinies, le prédicat est différent pour ces deux cas.

On note $l_t^{\perp}(p)$ la droite perpendiculaire à t passant par p, (si a est un segment) l_a la droite traversant a, v_{prq} et v_{pqs} les extrémités de l'arête[5].

3.4.1 Cas fini

L'arête (et donc la région) est ici définie par quatre sites. On les note p, q, r et s (figure 5, l'ordre est important) et le site que l'on souhaite ajouter est noté t. Il y a 5 cas de conflit possibles :

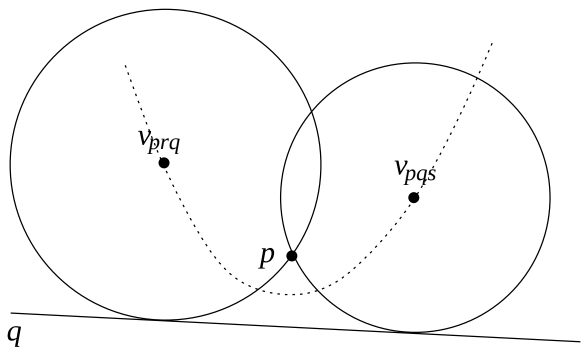

FIG. 5 – Une région finie

1. pas de conflit, l'arête reste à l'identique dans le nouveau diagramme ;
2. une extrémité de l'arête disparaît ;
3. les deux extrémités de l'arête disparaissent, mais une partie au milieu reste ;
4. les deux extrémités restent, mais une partie au milieu disparaît ;
5. la totalité de l'arête disparaît.

[5]L'ordre des sites p, q, r et s est celui obtenu en parcourant le cercle dans le sens positif.

Pour décider le cas de conflit, il faut tout d'abord faire deux tests *in circle*[6] (*cf.* section 3.4.3), pour savoir comment se comportent les extrémités, puis suivant le résultat de ces tests, il y a plusieurs cas :

- une extrémité disparaît et l'autre reste inchangée, le conflit est de type 2 ;
- les deux extrémités disparaissent, type 3 ou type 5, il faut des tests supplémentaires (*cf.* section 3.4.4) pour décider ;
- les deux extrémités restent inchangées, type 1 ou type 4, il faut aussi d'autre tests (*cf.* section 3.4.5).

Si l'une des extrémités est affectée, et que l'autre :

- reste inchangée, pas de conflit, type 1 ;
- disparaît, type 5 ;
- est aussi affectée, type 1 ou type 5 (*cf.* section 3.4.4).

3.4.2 Cas infini

Le cas infini est plus facile que le cas fini (une fois n'est pas coutume). Tout d'abord, si l'arête n'est infinie que d'un côté : le conflit ne peut être que de type 1, 2 ou 5 (la preuve se trouve dans Burnikel [1], section 2.2.6) après avoir fait le test *in circle* sur l'extrémité, il suffit de savoir de quel côté de la droite formée par p et q se trouve t, avec les mêmes notations que précédemment.

Si l'arête est infinie des deux côtés, le type du conflit ne peut être que 1 ou 2, suivant que t est colinéaire à la droite (pq) ou non (respectivement).

3.4.3 Le test *in circle*

J'ai repris les tests de Burnikel (Les formules sont détaillées et prouvées dans sa thèse), si t est un point :

- pour un cercle déterminé par trois points, c'est le signe d'un déterminant ;
- sinon, on calcule le centre du cercle (*i.e.* : les coordonnées de l'extrémité), puis on fait une comparaison de distance.
-

si t est un segment, c'est un peu plus compliqué :

- on fait le test avec les extrémités, si l'une des deux est dedans, t aussi ;
- sinon on fait le test *in circle* avec l_t, si l_t est à l'extérieur, t aussi ;

[6] le résultat d'un test *in circle* peut être :
- extérieur (l'extrémité reste inchangée) ;
- tangent (l'extrémité est affectée, mais reste dans le diagramme) ;
- intérieur (l'extrémité disparaît).

– sinon on regarde de quel côté sont les extrémités par rapport à $l_t^\perp(v)$ si v est le centre du cercle.

3.4.4 Tests supplémentaires lorsque les deux extrémités disparaissent

Si p est un point et t un segment, on défini $r(p,t)$ comme la demi-droite partant de p, perpendiculaire à l_t et s'éloignant de t. L'arête disparaît totalement si et seulement si $r(p,t)$ ne l'intersecte pas (*cf.* figures 6 et 7), c'est cela que l'on cherche à savoir dans cette section[7].

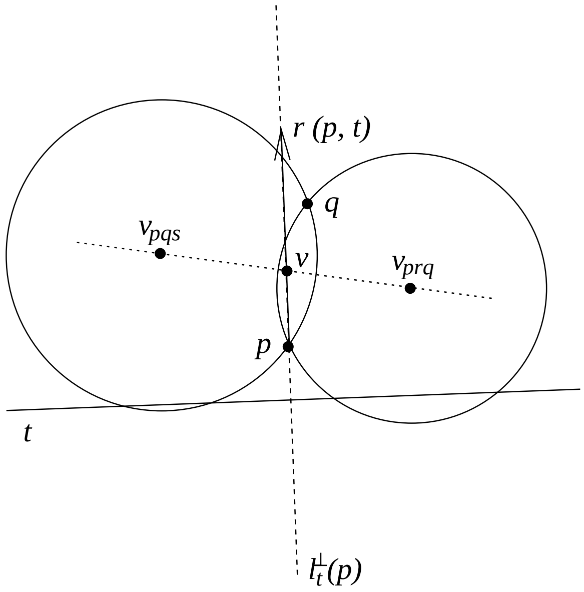

FIG. 6 – $r(p,t)$ croise l'arête : un bout reste.

Le type de conflit est déterminé comme suit.
Si p et q sont des points :

- si t est un point, type 5 (facile) ;
- sinon, si p et q sont équidistants à l_t, type 5 ;
- sinon, on peut supposer que p est plus proche que q de l_t, et alors le conflit est de type 3 si et seulement si $l_t^\perp(p)$ sépare les extrémités de l'arête.

Si p et q sont des segments, type 5.
Si p est un point et q un segment :

- si p est une extrémité de q, type 3 ;
- sinon, si t est un point, type 3 si et seulement si :

[7] Les preuves sont dans la thèse de Burnikel (comme d'habitude).

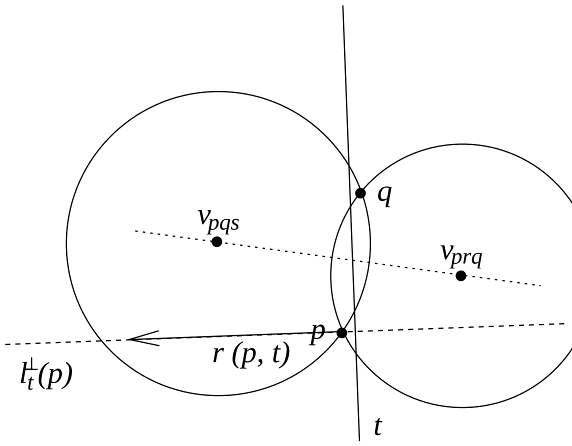

FIG. 7 – $r(p,t)$ ne croise pas l'arête : elle disparaît.

1. p est plus proche de l_q que t, et
2. $l_q^{\perp}(p)$ sépare les extrémités de l'arête ;

– sinon, t est un segment, soit $l = l_t^{\perp}(p)$ orientée comme $r(p,t)$:
  – si l_t et l_q sont parallèles et que p n'est pas entre les deux droites, type 5 ;
  – sinon, le conflit est de type 3 si et seulement si :
    1. l sépare les extrémités de l'arête et v_{prq} est à gauche de l et v_{pqs} est à droite de l ou
    2. v_{prq} et v_{pqs} sont séparés par $l_q^{\perp}(p)$ et l'angle entre l et l_q est supérieur à $\frac{\pi}{2}$, pour l_q orientée telle que p soit à sa gauche.

De plus, si les deux extrémités sont affectées :
– si t est un segment, type 3 ;
– sinon, si il y a un segment (a) et un point (b) parmi p et q, tel que b est plus proche de l_a que t, alors type 3 ;
– sinon, type 5.

3.4.5 Tests supplémentaires lorsqu'aucune des extrémités ne disparaît

On reprend les mêmes notations que dans la section précédente.

Burnikel a prouvé qu'un segment t ne pouvait entrer en conflit avec une arête du diagramme que si il entrait en conflit avec une extrémité de l'arête.
On ne va donc regarder que le cas où t est un point.
Si p et q sont des points, aucun conflit.
Si p est un point et q un segment, type 4 si et seulement si :

1. p et t sont du même côté de l_q et

2. t est plus proche de l_q que p et
3. $l_q^{\perp}(t)$ sépare les extrémités de l'arête.

Si p et q sont des segments, type 4 si et seulement si :

1. t est du même côté de l_p et l_q que les extrémités de l'arête et
2. $l_q^{\perp}(t)$ sépare les extrémités de l'arête.

4 Implémentation dans CGAL

CGAL est une bibliothèque d'algorithmes géométriques (écrite en C++) développée en partie à l'INRIA de Sophia Antipolis (*cf.* sa documentation [3]). Elle fournit :

- plusieurs "noyaux" (2D, 3D, ..., coordonnées cartésiennes ou homogènes) contenant :
  - des classes de base (points, segments, *etc.*) avec les méthodes élémentaires ;
  - des fonctions (`squared_distance`, `midpoint`, ...) ;
- une classe par algorithme implémenté : par exemple la classe `Convex_hull` qui calcule l'enveloppe convexe d'un ensemble de points dynamiquement (méthodes `add` et `remove`).

4.1 Types de nombres

CGAL utilise massivement une fonctionnalité de C++ appelée *template* (voir [4] pour plus de détails), il s'agit de paramétrer une fonction (ou une classe, *etc.*) par une classe, l'instanciation se faisant à la compilation, exemple :

```
template <class T>
void exchange (T &x, T &y)
{
  T tmp;
  tmp = x;
  x = y;
  y = tmp;
}

...

  MyClass obj1, obj2;

  exchange<MyClass> (obj1, obj2);
```

La fonction `exchange` pourra être appelée avec n'importe quel type de donnée fournissant les opérations utilisées par elle. Par exemple, ici les opérations sont :

- un constructeur sans argument (`T tmp;`) ;
- et l'opérateur d'affectation (`x = y;`).

Ce mécanisme permet de programmer des fonctions (et des classes) génériques. On peut aussi utiliser une hiérarchie de classe pour réaliser cela (toute fonction prévue pour une classe peut utiliser ses descendantes), l'avantage des *templates* est que l'instanciation se fait à la compilation : cela ne coûte donc rien à l'exécution.

Dans CGAL, les noyaux (cartésien ou homogène) sont paramétrés par un type de nombre, et toutes les fonctions et algorithmes sont parametrés par un noyau. Cela permet d'utiliser les algorithmes déjà programmés si (d'aventure) un nouveau noyau était écrit ou d'utiliser les noyaux existants avec de nouveau types de nombres.

Les prédicats que j'ai écrit sont bien entendus paramétrés comme il faut.

4.2 Exactitude *vs.* performance

Ainsi il est possible de choisir un type de nombre rapide mais pouvant amener à des résultats faux (*e.g.* : `double`), ou d'utiliser des type qui garantissent des comparaisons correctes, au prix d'une grande lenteur (*e.g.* : `leda_real`[8]).

Il existe un compromis : filtrer les prédicats. On essaye de calculer le prédicat avec une arithmétique d'intervalle, lors d'une comparaison si les intervalles sont disjoints le résultat est sûr, sinon une exception est levée et le calcul est recommencé avec un type exact.

C'est ce mécanisme qui ne peut fonctionner si on utilise pour un prédicat des valeurs calculées (si on les calcule exactes on perd l'intérêt de l'outil et si on les calcule approchées comment les recalculer ?), d'où le problème à la section 3.2.

Ce mécanisme est présent dans CGAL[9], il suffit donc de programmer un prédicat et le reste se fait tout seul (par l'intermédiaire d'une *trait class*).

4.3 Trait class

Une *trait class* est une façon de décrire les objets et opérations géométriques de base dont a besoin un algorithme. une trait class doit contenir comme membre les types de donnée nécessaire à l'algorithme. De plus pour chaque fonction (prédicat ou constructeur) utilisée, la trait class doit contenir une classe correspondante possédant un opérateur `()` qui se comporte comme la fonction. Les algorithmes sont paramétrés par une trait class, c'est ainsi qu'ils accèdent à ce qu'il leur est nécessaire.

Par exemple, la triangulation de Delaunay est définie par le prédicat `in_circle` (en dehors des choses basiques telles que `Point`, `Triangle`, ...). Une trait class pour cette triangulation doit donc contenir une class `In_circle`

[8] Les types de nombres implémentés par LEDA (Library for Efficient Datatype Algorithms) peuvent être utilisés avec CGAL, voir [3] et [6]

[9] enfin...ce n'est pas encore documenté, ça l'est peut-être la version 2.3

avec un opérateur `()`. Si l'on souhaite, par exemple, faire une triangulation de Delaunay dans un espace non-euclidien il suffit de récrire comme il faut la fonction `in_circle` ainsi qu'une autre trait class, et le tour est joué.

Le travail de mon stage était donc finalement d'écrire une trait class pour un algorithme de diagramme de Voronoï de segments.

4.4 Description du code

Du fait de l'utilisation de templates, le code se retrouve dans les fichiers d'en-têtes (`.h`). En effet, on ne peut pas compiler une fonction template séparément du code qui l'appelle, car elle doit être instanciée (*i.e.* compilée) une fois pour chaque paramètre différent pour lequel elle est appelée.

Avec l'exemple précédent, la fonction `exchange` devra être compilé une fois avec la classe `MyClass`, mais si elle est appelée avec la classe `MyOtherClass`, il faudra compiler une deuxième version[10] (surcharge des noms de fonction).

Les fichiers dont le nom commence par `voronoi_` contiennent les fonctions et classes qui font effectivement le travail pour les prédicats et les constructions. Ces fichiers sont souvent accompagnés d'homonymes à des suffixes (`C2`, `_ftH2`, *etc.*) près. Explications, si le suffixe est :

_2 le fichier contient des fonctions qui travaillent sur des objets géométriques sans se soucier de leur représentation ;

C2 ou H2 ces fonctions ont besoin de connaître la représentation des objets : **C**artésien ou **H**omogène, elles se contentent souvent d'appeler les fonctions des fichiers **ftC2** ou **ftH2** ;

ftC2 ou ftH2 ces fonctions travaillent directement sur les coordonnées des objets (ce sont elles qui sont paramétrées par un type de nombre).

4.4.1 Parabola.h

Ce fichier contient la définition d'une classe parabole pour dessiner les arêtes du diagramme.

4.4.2 Conflict_type.h

Définition du type de retour des prédicats, c'est une structure à trois champs :

v1 et v2 le conflit avec les extrémités de l'arête, de type `Oriented_side`, un type enum de CGAL à trois valeurs ;

[10] Rem. : une fonction template qui n'est jamais appelée n'est pas compilée, elle est juste vérifiée syntaxiquement. Cela peut faire apparaître des "bugs à retardement".

cut_midlle un booléen vrai si et seulement si une partie à l'intérieur de l'arête disparaît (n'a pas toujours de sens suivant les valeurs de `v1` et `v2`).

4.4.3 Site_2.h

Dans ce qui précède, j'ai parlé de *sites* comme étant des points ou des segments, il s'agit ici de son implémentation, la classe `Site_2` possède les méthodes :

is_defined() qui dit si le site est défini ;

is_point() et is_segment() si le site est défini, disent ce que c'est ;

point() et segment() qui renvoient le site en tant que `Point_2` ou `Segment_2`.

4.4.4 voronoi_kernel_2.h

Ce fichier contient des tests de base utilisés par les prédicats et constructeurs, qui ne sont pas spécifiques au diagramme de Voronoï de segments, mais qui ne sont pas non plus suffisamment généraux et utiles pour être dans le noyau de CGAL.

cmp_dist_to_line(l,p,q) Si l est une droite et p et q des points, compare $d(p,l)$ et $d(q,l)$. Il existe dans CGAL `cmp_signed_dist_to_line`, une droite sépare le plan en un demi-plan positif et un demi-plan négatif.

are_on_different_side_perp(d,v,p,q) C'est le test "est-ce que $l_d^{\perp}(v)$ sépare p et q" qui est utilisé presque partout.

cmp_angle_with_half_pi(l$_1$,l$_2$) utilisé pour le cas où p est un point et q et t des segments.

4.4.5 voronoi_check_vertices_2.h

Étant donné trois sites, les fonctions `pre_check` déterminent si ils définissent un sommet de Voronoï (elles renvoient 0 si ce n'est pas le cas), et si oui, elles renvoient un nombre non nul qui informe les fonctions `compute_vertex` sur la configuration des trois sites.

En effet, les formules qui calculent les coordonnées du centre du cercle circonscrit à trois points ou segments font des suppositions sur les orientations des segments, sont différentes si deux des segments sont parallèles, *etc.*

Les fonctions `post_check` vérifient que le point calculé est correct, en effet pour le calcul on identifie un segment à la droite le traversant. Or un cercle peut parfaitement être tangent à une droite mais pas à un de ses segments.

4.4.6 voronoi_compute_vertices_2.h

Ce fichier contient les fonctions `compute_vertex` qui calculent le centre du cercle circonscrit à trois points ou droites. Elles prennent aussi en argument le résultat des fonctions `pre_check`.

4.4.7 voronoi_constructions_2.h

Ce fichier contient les fonctions `vertex` qui regroupent les appels à `pre_check`, `compute_vertex` et `post_check`.

Comme les prédicats savent que les sommets sont définis, il y a aussi les fonctions `vertex_sure` qui ne font pas de post-vérification (elles sont donc plus rapides).

4.4.8 voronoi_in_circle_2.h

Ce fichier contient toutes les variantes du test *in circle* qui sont utilisées par les prédicats.

- Test d'un point dans un cercle défini par trois points (déjà dans CGAL : fonction `side_of_oriented_circle`) ;
- Test d'un point dans un cercle si un des sites qui le définit est un point (aussi déjà dans CGAL : fonction `cmp_dist_to_point`) ;
- Test d'un point dans un cercle si un des sites qui le définit est un segment (identifié ici a une droite)[11] ;
- Test d'un segment dans un cercle si un des sites qui le définit est une de ses extrémités ;
- Test d'une droite dans un cercle si un des sites qui le définit est une droite ;
- Test d'un segment dans un cercle ;
- Tests pour des "cercles infinis" (*i.e.* : des demi plans).

4.4.9 voronoi_middle_2.h

Ce fichier implémente les tests additionnels décrits dans les sections 3.4.4 et 3.4.5 avec les fonctions (surchargées) `cut_middle_both`, `cut_middle_none` et `cut_middle_touch` dont les noms sont parlants.

[11] le cas d'une droite dans un cercle défini par un point est symétrique

4.4.10 voronoi_predicates_2.h

Maintenant que nous avons de quoi calculer un sommet du diagramme, faire des tests *in circle* et connaître le devenir de l'intérieur d'une arête, nous pouvons effectivement programmer les prédicats. Ce fichier contient beaucoup de versions de la fonction `in_conflict` (pour les arêtes finies). En effet un prédicat prend en argument 4 sites qui définissent l'arête plus le nouveau site à tester, un site peut être un point ou un segment, il y a donc *a priori* $2^5 = 32$ fonctions à écrire. En fait, il y a beaucoup de symétries (par exemple, les fonctions qui prennent (point, segment, ...) sont les mêmes que celles qui prennent (segment, point, ...) en argument) et donc moins de fonctions à écrire finalement (et puis les fonctions non symétriques se ressemblent quand même beaucoup).

4.4.11 voronoi_unbounded_conflict_2.h

Il ne faut pas oublier le cas des arêtes infinies, dans ce fichier se trouve les fonctions `in_conflict` pour les arêtes infinies et bi-infinies.

4.4.12 voronoi_Site_constructions_2.h et voronoi_Site_predicates_2.h

Les fonctions définies jusqu'à présent prennent toutes des points ou des segments en argument, or j'ai défini une classe `Site_2` (*cf.* section 4.4.3), il est temps de l'utiliser. Ce fichier contient des fonctions qui réalisent un "aiguillage" qui regarde de quel type sont les sites qu'on lui passe en argument et appelle la bonne fonction.

4.4.13 Traits classes

Le fichier `Voronoi_segment_euclidean_trait_2.h` contient une trait class normale (paramétrée par un noyau).

Le fichier `Voronoi_filtered_trait_2.h` contient une trait class filtrée qui utilise des `double` pour les constructions, des intervalles pour les prédicats, avec `leda_real` comme type exact si les intervalles échouent.

Une trait class pour le diagramme de Voronoï de segment doit fournir :
- les types géométriques de base : point, segment, droite, cercle, site ;
- un type parabole (pour dessiner les arêtes entre un point et un segment) ;
- le prédicat `in_conflict` avec 5, 4 ou 3 sites en arguments (arêtes finies, infinies ou bi-infinies) ;
- le constructeur (3 sites $\longrightarrow$ site).

4.4.14 Jeu de tests

J'ai écrit une classe `Voronoi_test` (dans le fichier `Voronoi_test.h`) paramétrée par une trait class qui se comporte comme un algorithme (en fait elle se contente de savoir afficher une région, de tester des conflits avec elle et de voir le résultat en couleur).

Le fichier `prog.C` contient le programme principal.

5 Conclusion

Ce stage m'a permis de découvrir la géométrie algorithmique, un domaine que je ne connaissais pas du tout auparavant. Il a été intéressant de se plonger dans un problème et rencontrer au fur et à mesure une large variété de difficultés.

De plus, j'ai dû écrire du code qui devait s'intégrer dans un gros projet : CGAL est développé conjointement par plusieurs dizaines de personnes réparties dans différents pays et compte environ 300 000 lignes de C++. Il a donc fallut respecter scrupuleusement les règles de programmation imposées (convention de nommage, *etc.*).

Je voudrais enfin remercier toute l'équipe PRISME pour m'avoir accueilli pendant ces 8 semaines, repondu à mes questions, *etc.*

Références

[1] C. Burnikel : *Exact Computation of Voronoï Diagrams and Line Segment Intersection*, Phd. Thesis, 1996.

[2] M. Held : *VRONI : An Engineering Approach to the Reliable and Efficient Computation of Voronoï Diagrams of Points and Line Segments*, Computational Geometry : Theory and Application 18(2) : p. 95–123, 2001.

[3] *The CGAL Reference Manual*, disponible sur `http://www.cgal.org`.

[4] Bjarne Stroustrup : *The C++ Programming Language.*

[5] M. Teillaud : *Toward Dynamic Randomized Algorithms in Computational Geometry*, Rapport de Recherche n° 1727, INRIA Sophia Antipolis, Juillet 1992[12].

[6] K. Mehlhorn, S. Näher, M. Seel et C. Uhrig : *The LEDA User Manual*, `http ://www.mpi-sb.mpg.de/LEDA/leda.html`.

[12] disponible sur le site internet de l'INRIA : `http ://www.inria.fr`