Remerciements

Je souhaite remercier David Ritchie et Bernard Maigret qui m'ont fait confiance pour mener à bien ce projet et m'ont suivi tout au long de mon stage. Je remercie également Olivier Devillers pour ses cours sur les triangulations de Delaunay et ses explications régulière concernant les librairies CGAL. Enfin, il me faut remercier toute l'équipe Capsid pour avoir facilité mon intégration dans l'institut ainsi que l'INRIA Nancy.

Table des matières

Introduction

The INRIA (Institut National de Recherche en Informatique et en Automaique) is a public French institute of research in computer science and mathematics. Founded in 1967, the INRIA accounts for 2600 collaborators gathered on various sites in France, one of which being Nancy. Within this center, the team CAPSID develops algorithms and softwares allowing to study biological phenomena and systems from a structural point of view, thanks to 3D modelling. I did my internship with this team under the supervision of Dave Ritchie who is its manager.

Overseen by Dave Ritchie and Bernard Maigret, this project's main goal is to model the interface of contact between two proteins. The properties of the interface can indeed give a lot of information about the interactions between proteins. This is particularly heplful in fields such as biology and medicinal search for the development of new medicine. The project also takes place in partnership with the research team Vegas, and especially with Olivier Devillers, who participated in the implementation of CGAL (Computational Geometry Algorithms Library).

This library allows, through to the numerous features it offers, to improve and to accelerate the development of the method chosen for the project. We will give more details in this report on the theoretical method adopted to approximate the interface between proteins : the Delaunay triangulation and the Voronoï Diagram. We will also explain how the structures supplied by CGAL are used during the development of the software by insisting on some of the most crucial parts of the implementation. Finally, we will see how the results (and the difficulties encountered) allow to elaborate future prospects for this project.

1 Theoretical Method

1.1 Structure of proteic complex

Proteins are biological molecules found in all the living cells. They are formed of a chain of amino acids. Proteins realize many functions within the living cells. They can have an enzymatic, structural role, they allow the mobility of molecules, the regulation of the genetic expression or to pass on cellular signals. The protein chains forming the proteins are synthesized in the cell. The genetic material of the cell determines the order of the amino acids.

The proteins have a structure in three dimensions which allows them to realize their biological function. Proteins can interact together to carry out some biological functions. These interactions form protein complexes. The structures of the proteins and their interactions are particularly used in medicinal chemistry. The study of surfaces and available spaces can guide the research for new medicine. Crystallography is used to study the structure of proteins at the atomic scale. It is based on the physical phenomenon of diffraction of the electromagnetic waves (X-rays).

The data from a protein that we can us is stored in *.pdb* files (see figure**??**). These files, the reading and the interpretation of the data they contain, are essential to the observation of proteins. Indeed, every line, excepted the first one, corresponds to an atom of protein being studied. These lines contain information such as the chain to which the atom belongs, its amino acid or its address and coordinates in space (Å).

Dans la suite du projet, nous considérerons que les atomes sont des points de l'espace de masses identiques. Dans un premier temps, les coordonnées des atomes seront extraites. Le nuage de point obtenu constitue la base sur laquelle la méthode de détermination de l'interface va s'appuyer. Il est donc crucial d'obtenir du fichier *.pdb* un jeu de données valide et utilisable par le programme implémenté ensuite. Nous verrons également que d'autres informations, telles que la chaîne (A ou B, colonne 5 de la figure 1.1), jouent un rôle prépondérant dans le bon fonctionnement du programme. L'objectif est donc d'extraire un nuage de points cohérent (modélisant le complexe), pour lui appliquer la triangulation de Delaunay.

```
CRYST1    0.000    0.000    0.000  90.00  90.00  90.00 P 1           1
ATOM      1  CA  MET A  76      11.682 -15.962 -22.500  1.00  0.00           C
ATOM      2  C   MET A  76      10.913 -14.834 -21.818  1.00  0.00           C
ATOM      3  O   MET A  76      11.505 -13.842 -21.393  1.00  0.00           O
ATOM      4  CB  MET A  76      10.786 -16.678 -23.515  1.00  0.00           C
ATOM      5  CG  MET A  76      10.101 -17.879 -22.854  1.00  0.00           C
ATOM      6  SD  MET A  76       8.947 -18.633 -24.027  1.00  0.00           S
ATOM      7  CE  MET A  76       8.478 -20.050 -23.004  1.00  0.00           C
ATOM      8  N   LYS A  77       9.594 -14.989 -21.723  1.00  0.00           N
ATOM      9  CA  LYS A  77       8.754 -13.971 -21.096  1.00  0.00           C
ATOM     10  C   LYS A  77       8.140 -14.495 -19.799  1.00  0.00           C
ATOM     11  O   LYS A  77       7.465 -15.525 -19.790  1.00  0.00           O
ATOM     12  CB  LYS A  77       7.641 -13.555 -22.064  1.00  0.00           C
ATOM     13  CG  LYS A  77       6.806 -12.424 -21.443  1.00  0.00           C
ATOM     14  CD  LYS A  77       5.692 -11.990 -22.407  1.00  0.00           C
ATOM     15  CE  LYS A  77       6.244 -11.023 -23.461  1.00  0.00           C
ATOM     16  NZ  LYS A  77       6.791  -9.814 -22.783  1.00  0.00           N
ATOM     17  N   ASP A  78       8.370 -13.768 -18.710  1.00  0.00           N
ATOM     18  CA  ASP A  78       7.831 -14.148 -17.408  1.00  0.00           C
ATOM     19  C   ASP A  78       6.674 -13.227 -17.034  1.00  0.00           C
ATOM     20  O   ASP A  78       6.415 -12.239 -17.721  1.00  0.00           O
ATOM     21  CB  ASP A  78       8.925 -14.062 -16.342  1.00  0.00           C
ATOM     22  CG  ASP A  78       9.934 -15.188 -16.539  1.00  0.00           C
ATOM     23  OD1 ASP A  78       9.634 -16.101 -17.290  1.00  0.00           O
ATOM     24  OD2 ASP A  78      10.992 -15.121 -15.935  1.00  0.00           O
ATOM     25  N   THR A  79       5.975 -13.552 -15.951  1.00  0.00           N
ATOM     26  CA  THR A  79       4.844 -12.740 -15.514  1.00  0.00           C
ATOM     27  C   THR A  79       5.309 -11.353 -15.082  1.00  0.00           C
ATOM     28  O   THR A  79       6.434 -10.947 -15.373  1.00  0.00           O
ATOM     29  CB  THR A  79       4.127 -13.427 -14.350  1.00  0.00           C
ATOM     30  OG1 THR A  79       5.009 -13.517 -13.240  1.00  0.00           O
```

FIGURE 1.1 – Example of a .pdb file

1.2 Triangulation de Delaunay

Une protéine est donc représentée comme un nuage de points où chacun de ces points représente un atome de la protéine. Dans le but d'optimiser les temps de parcours dans ce nuage de points et de modéliser l'interface entre les deux protéines, on utilise la triangulation de Delaunay.

Cette triangulation est unique et peut être expliquée de la manière suivante : chaque cercle circonscrit à un triangle du nuage de point ne contient que les points dudit triangle (voir figure 1.2). Nous expliquerons dans un premier temps la méthode pour déterminer l'interface en deux dimensions.

Dimension 2

Il faut appliquer cette triangulation au complexe dont on veut déterminer l'interface, c'est-à dire aux points donnés par les coordonnées des atomes qui composent le complexe (voir figure 1.3a). Les protéines du complexe sont différenciées par leurs couleurs (rouge et bleu) sur le schéma. Nous sélectionnons alors la partie utile de cette triangulation (voir figure 1.3b) : nous ne gardons que les triangles qui contiennent au moins un point à l'interface. Un point est à l'interface s'il appartient à un triangle contenant au moins un point de chaque protéine. Le fait de réduire la taille de la triangulation sera utile par la suite pour accélérer les temps de traitement et de récupération de données concernant l'interface.

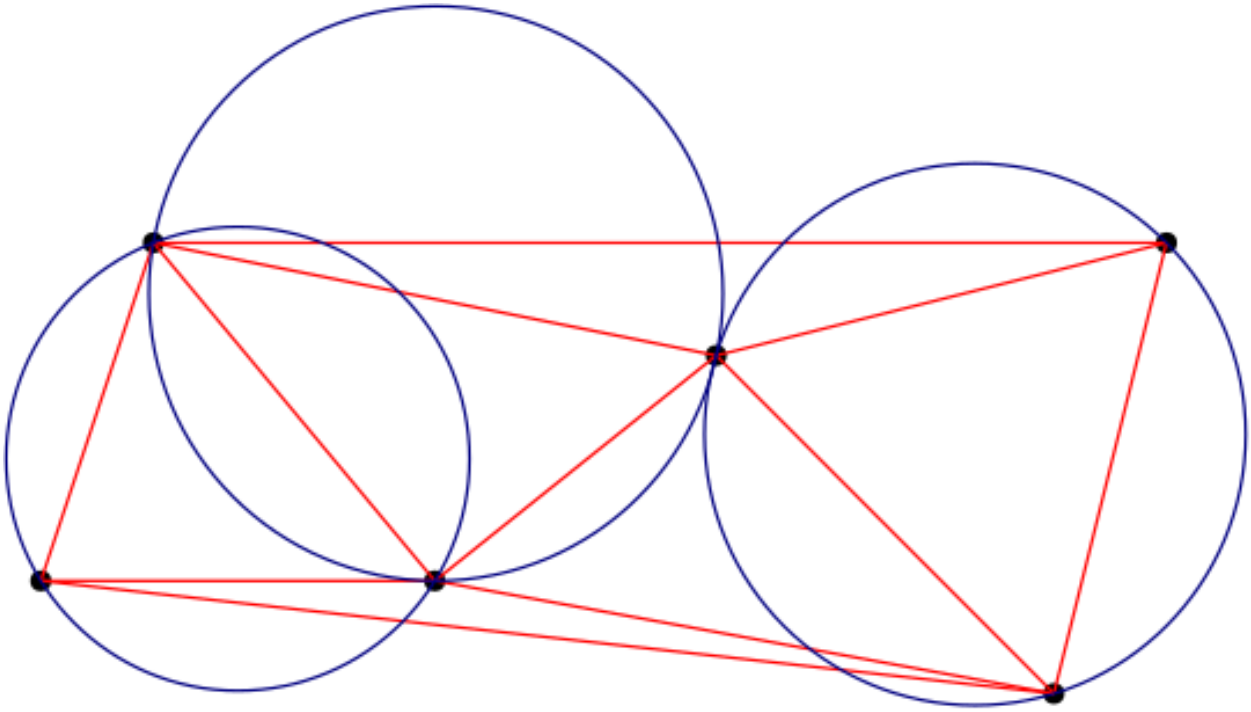
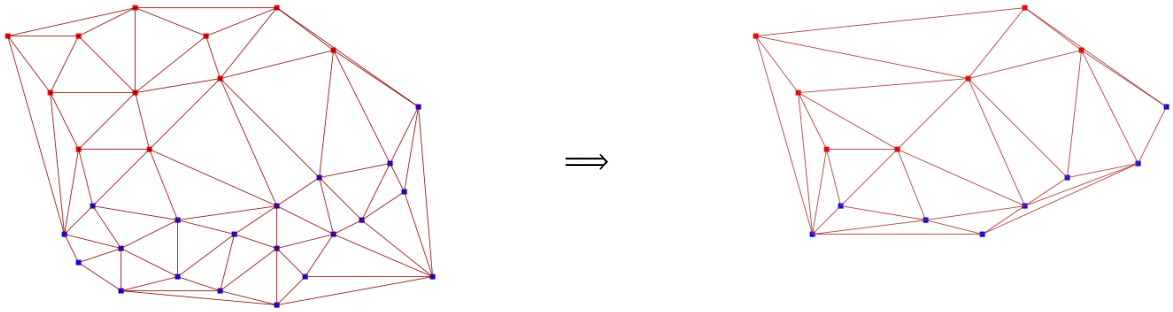


FIGURE 1.2 – Triangulation de Delaunay



(a) Triangulation de Delaunay (b) Triangulation réduite

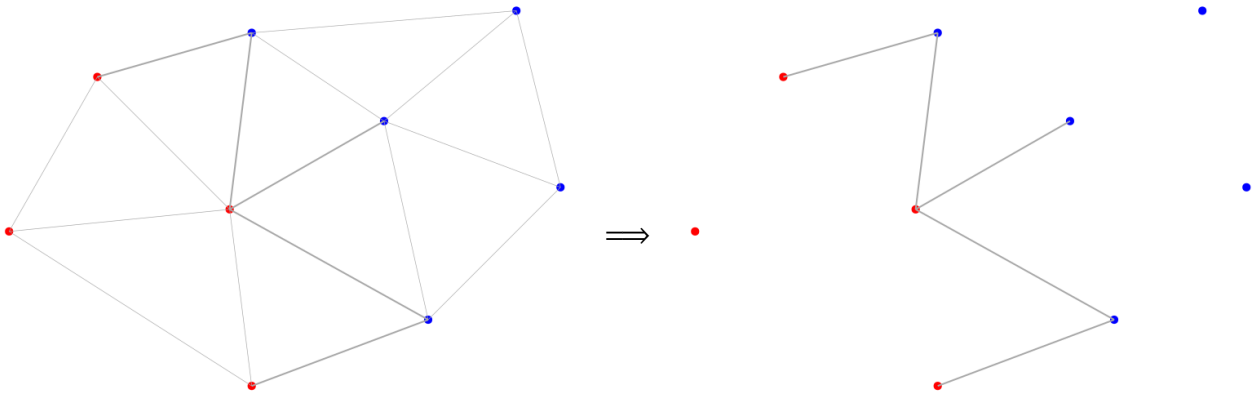
FIGURE 1.3 – Réduction d'une triangulation

Nous nous intéressons maintenant à la détermination de l'interface, proprement dite. En ne gardant que les arêtes utiles (voir figure 1.4), c'est-à-dire celles reliant deux points appartenant à deux protéines différentes, nous pouvons approximer l'interface de contact grâce au diagramme de Voronoï.

Ce diagramme est le dual de la triangulation de Delaunay et représente les points à égale distance des points de la triangulation (voir figure 1.5a). En ne gardant que les parties du diagramme de Voronoï qui correspondent aux arêtes sélectionnées précédemment (voir figure 1.5b), nous obtenons une droite par morceaux qui approxime l'interface de contact en dimension 2.

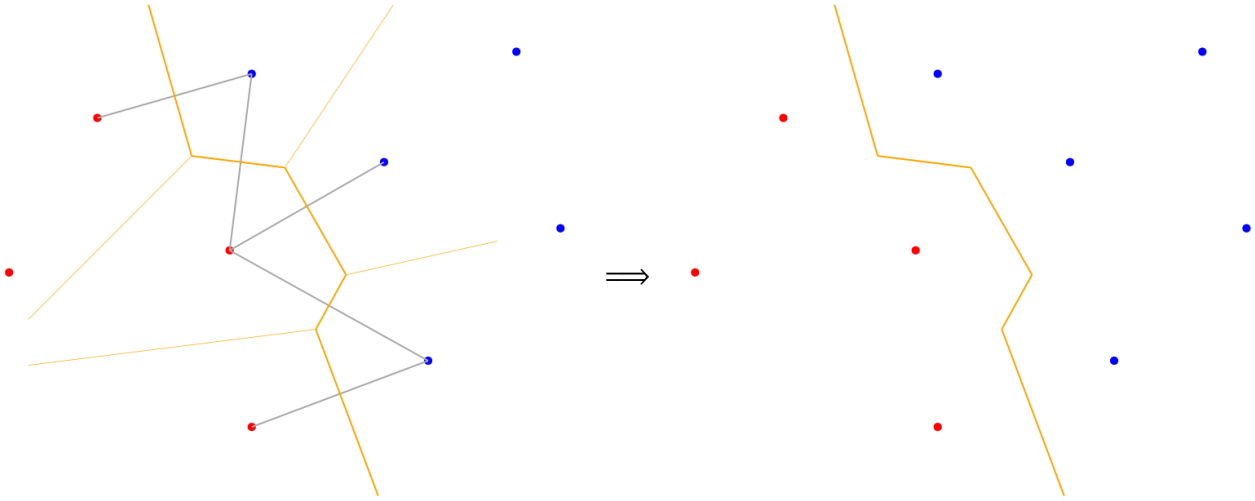
Dimension 3

Si nous transposons la méthode vue ci-dessus en dimension 3, les triangles formés par les points deviennent des tetrahèdres sur lesquels nous travaillerons pour vérifier les zones utiles au calcul de l'interface. De la même manière, un tétrahèdre sera considéré à l'interface s'il contient



(a) Triangulation de Delaunay (b) Arêtes à l'interface

FIGURE 1.4 – Triangulations et zone utile



(a) Diagramme de Voronoï (b) Interface

FIGURE 1.5 – Recherche de la surface de contact

au moins un atome de chaque protéine. De plus, le dual d'une arête devient une surface entourant cette arête (voir figure 1.6). En rassemblant ces morceaux de surface, nous obtenons une surface en trois dimensions qui modélise la surface de contact entre les deux protéines du complexe étudié.

La triangulation en trois dimensions d'un nuage de points correspondants aux atomes d'un complexe donne la figure 1.7.

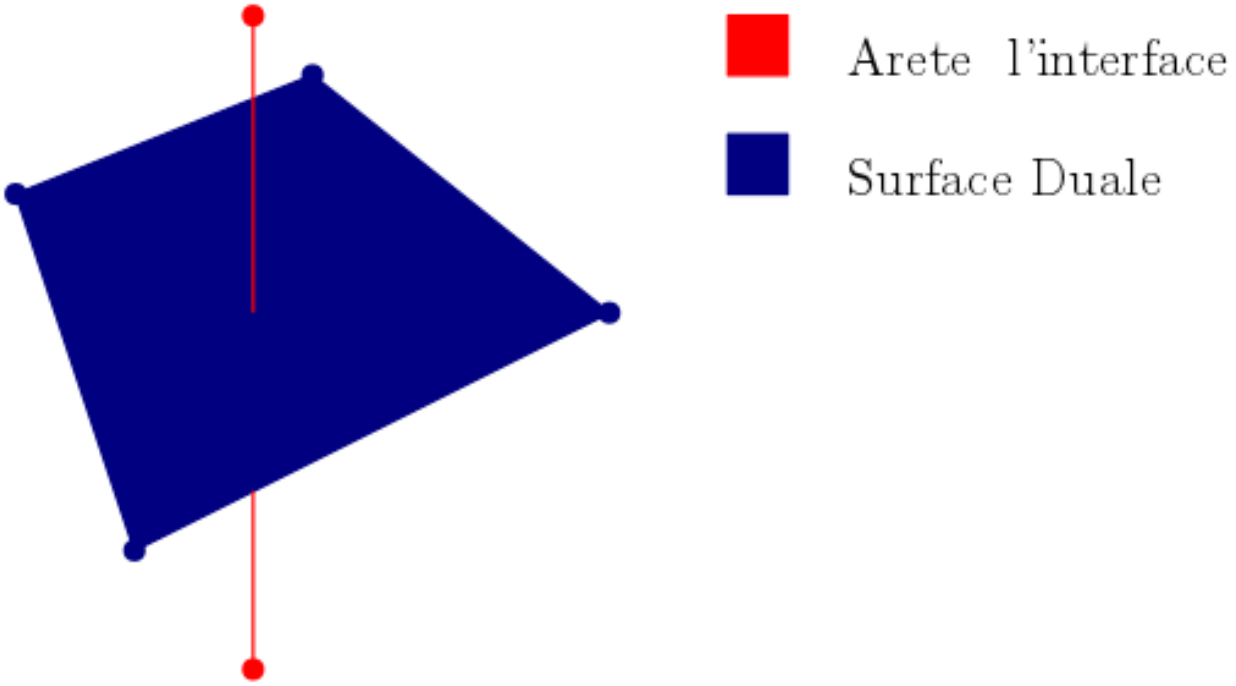
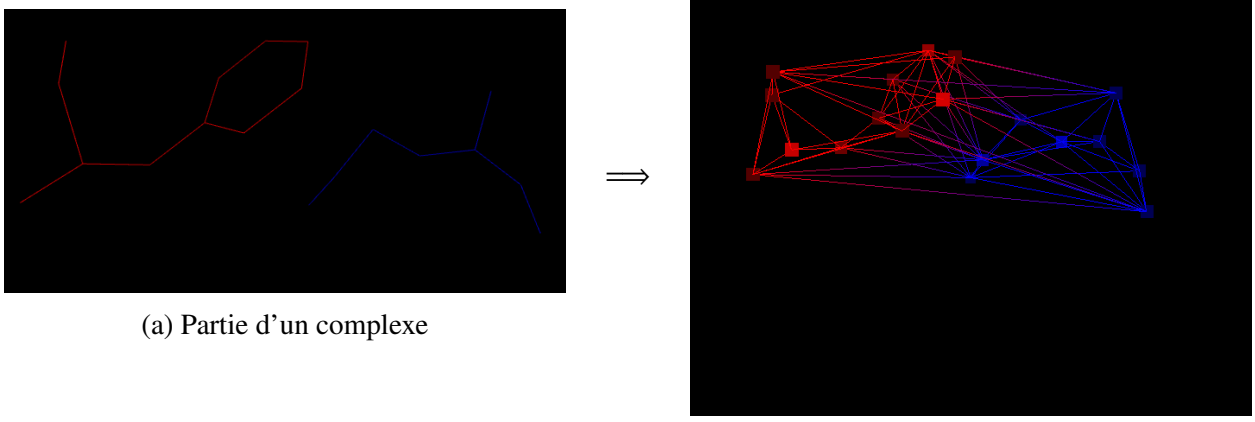


FIGURE 1.6 – Dual d'une arête en Dimension 3



(a) Partie d'un complexe

(b) Triangulation

FIGURE 1.7 – Triangulation 3D d'un complexe

1.3 CGAL

— Structures
— Iterateurs
— Compilation -> Cmake

Pour développer la méthode vue précédemment, nous avons choisi d'utiliser CGAL (Computational Geometry Algorithms Library). CGAL est un projet logiciel qui fournit un accès libre à de nombreux algorithmes géométriques efficaces et fiables sous forme d'une bibliothèque C++. CGAL est utilisé dans des domaines diverses ayant besoin de calcul géométrique, tels que des systèmes d'information géographiques, la conception assistée par ordinateur, la biologie moléculaire, l'imagerie médicale, l'infographie et la robotique.

Nous nous sommes particulièrement intéressés à une partie de CGAL qui permet le stockage de nuages de points sous forme de triangulations de Delaunay. L'avantage de cette bibliothèque réside dans les structure et les méthodes accélérant les différentes étapes du calcul de l'interface entre deux protéines.

Structures

En effet, CGAL comprend notamment une classe (que l'on peut voir comme une structure) *Delaunay_Triangulation_3*, permettant de calculer et de stocker une triangulation de Delaunay depuis de simples tableaux (*C++ Arrays*) listant des points dans l'espace. Pour mieux, comprendre l'implémentation réalisée durant ce projet, il est important de préciser la structure des tétrahèdres composant une trianguation de Delaunay (voir figure 1.8).

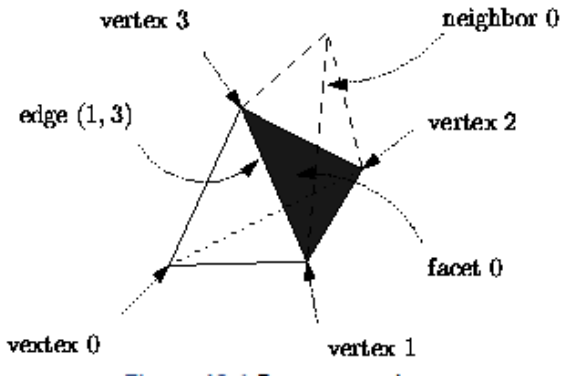


FIGURE 1.8 – Structure d'un tetrahèdre dans CGAL

Un tetrahèdre est représenté par quatre entités :

— Vertex (vertice) : contient un point (coordonnées 3D)
— Edge (arête) : une arête contenant deux vertex ordonnés et une cellule
— Facet (face) : une face stockée grâce à une cellule et le vertex qui lui est opposé dans cette cellule
— Cell (cellule) : un tetrahèdre qui donne accès à quatre vertex et quatre cellules adjacentes

Il est important de comprendre la structure mise en place dans CGAL car il sera nécessaire d'accéder au différentes parties d'un tétrahèdre. Par exemple, lorsque nous travaillons sur les arêtes pour rechercher l'interface, nous avons besoin de connaître les vertices qui la compose.

Il existe également une seconde strucure de données, *Polyhedron*, qui sera utilisée pour stockée la surface. Cette classe permet de travailler sur des surfaces en trois dimensions c'est-à-dire que les morceaux des cette surface sont en deux dimensions. Le fonctionnement est sensiblement identique à la classe *Delaunay_Triangulation_3*.

La compréhension des structures fournies par CGAL fut donc une partie fondamentale du projet. En effet la maîtrise des accesseurs est cruciale afin d'optimiser les parcours dans les nuages de points via les itérateurs.

Itérateurs

Les itérateurs sont une généralisation de pointeur qui permettent de travailler sur différentes structures de données. Il existe des itérateurs permettant de parcourir une triangulation de Delaunay selon chacune des entités formant celle-ci : vertice, arête, face, cellule. Quelle que soit l'entité utilisée, il est possible d'accéder au trois autres entités et aux données qu'elles contiennent. Les itérateurs fonctionnent dans CGAL grâce aux *Handles*. Celle-ci sont des références indirectes qui fonctionnent sensiblement comme des pointeurs. Lorsqu'on itère sur un des types composant les tétrahèdre, une *Handle* vers le type concerné est renvoyée. Par exemple, pour une itération sur des vertices, on peut accéder à l'indice stocké dans *info()* par la commande fournie en figure 1.8.

```
// Iterate on the vertices of the triangulation
for(Delaunay::Finite_vertices_iterator vit=m_dt.finite_vertices_begin(); vit!=m_dt.finite_vertices_end(); ++vit) {
    vit->info();
}
```

FIGURE 1.9 – Utilisation d'un itérateur

Il existe également des circulateurs qui, bien que semblables au itérateurs par le fait qu'ils parcourent des structures de données, fonctionne différemment. Commme leur nom l'indique , ils permettent de "tourner" autour d'une entité. Par exemple ils peuvent servir à parcourir toutes les cellules adjacentes d'une cellule données. Ils s'utilisent sous la forme d'une boucle *do...while* (voir figure 1.10). Le circulateur est incrémenté à chaque passage dans la boucle jusqu'à ce qu'il soit

```
Delaunay::Cell_circulator circ; // Circulator for cells around a given edge
Delaunay::Cell_circulator circ_copy; // Copy to check when all cells have been tested
circ = interface_tr.incident_cells(*eit); //circulateur of celles adjacent to an edge
do {
    //whatever on the incident cells
} while(circ != circ_copy); // Circle around the edge until the first cell is reached again
```

FIGURE 1.10 – Utilisation d'un circulateur

revenu à son point de départ. Il aura alors effectué un "tour" complet autour de l'entité de notre choix.

Nous verrons, lors de l'implémentation, qu'un bon usage des circulateurs, et plus encore des itérateurs, est absolument crucial pour l'optimisation des temps de parcours et la validité des résultats obtenus. Ils permettent également de rendre le code plus générique.

2 Développement Logiciel

Dans cette partie, expliquerons le développement du programme en différentes étapes : l'architecture du programme, la récupération des données, la construction de la surface et l'affichage (voir figure 2.1).

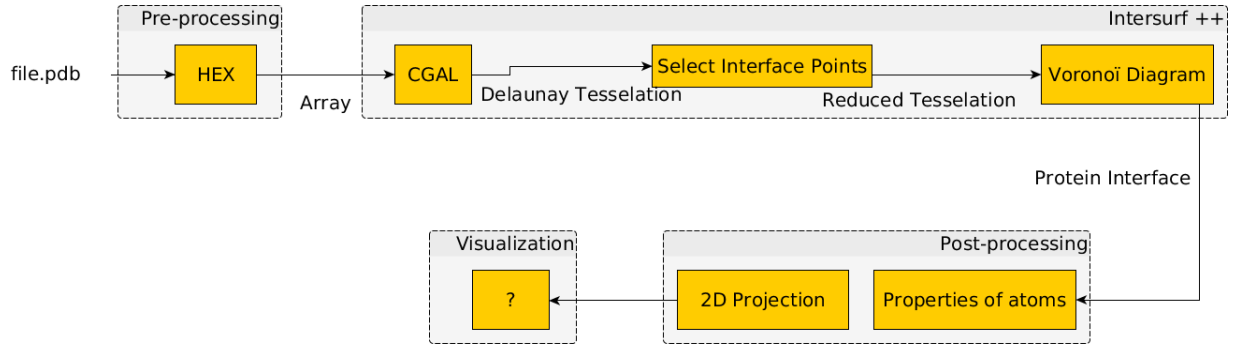


FIGURE 2.1 – Flow chart du programme

2.1 Architecture Logicielle

— C / C++ / Libraries
— Compilation Cmake

Les différentes étapes

2.2 Récupération de données + stockage

— .pdb files + pdb reader
— Tableau et méthode de stockage (C++ structures)
— CGAL structures : Delaunay + Polyhedron

Le point de départ du projet est de lire un fichier *.pdb* et de le stocker de manière à créer les structures fournies par CGAL. Nous avons utilisé une partie du programme **Hex** développé en C par David Ritchie qui permet de *parser* le fichier *.pdb* pour récupérer les données utiles du complexe étudié. Ces informations constituent la référence utilisée lors de la construction de l'interface lorsque des détails sur les atomes (chaîne, etc.) doivent êre connus. La fonction utilisée

```
/// Read a .pdb file into an array
PdbImage *pdb = hex_readPdb("../data/2n77_reduced.pdb", "new_protein");  // Reading a .pdb file
```

FIGURE 2.2 – Récupération des données du fichier *.pdb*

pour cela renvoie un pointeur vers un tableau recensant toutes les informations utiles, c'est-à-dire une image du fichier *.pdb* que nous appellerons *Image pdb* (voir figure 2.2).

Pour créer une structure de Delaunay, CGAL prend normalement une liste de points stockée sous forme de tableau. Cependant, utiliser cette méthode simple ne premettrait pas d'accéder aux informations contenues dans le fichier *.pdb*. En effet, chaque vertice (point), de la triangulation de Delaunay doit être indexé de manière à retrouver les informations voulues dans la liste d'atome fournie par le fichier de départ. Les index doivent donc être choisis au moment de la lecture de l'*Image pdb* pour correspondre à la place de l'atome dans le tableau. Il existe deux méthodes pour insérer un index à une variable de type Vertex dans CGAL :

— on modifie la classe Vertex en ajoutant un attribut

— on utilise une classe différente fournie par CGAL

Nous avons choisi d'utiliser la seconde méthode (voir figure 2.3) pour sa simplicité. Lors de l'instanciation, cette classe prend notamment en paramètre *template* un type qui de variable qui sera accessible comme l'attribut *info()*.

```
typedef CGAL::Triangulation_vertex_base_with_info_3<unsigned int, K>    Vb;
```

FIGURE 2.3 – Déclaration de la classe *Vertex_base_with_info*

Le type de l'index est donc un entier positif qui fera partie de la structure Delaunay et sera accessible lors du parcours de ces structures. Cependant, la structure de Delaunay requiert l'ajout d'un index lors de son instanciation. Un vecteur de paires contenant un point 3D et une variable du type de l'attribut *info()* (voir figure 2.4) convient.

```
std::vector<std::pair< Point, unsigned> > atoms_coordinates;  // Vector for the atoms of the protein
```

FIGURE 2.4 – Déclaration du vecteur contenant les atomes et leurs index

La fonction complète permettant d'écrire un fichier.*pdb* dans une structure *CGAL : :Delaunay* est disponible en annexe (voir figure 2.2). On remarque la présence de la *map* C++ *is_interface* (voir figure 2.5) qui sera utilisée par la suite pour sélectionner les tétrahèdres présents à l'interface (voir figure 1.3).

```
std::map<unsigned int, bool> is_interface
```

FIGURE 2.5 – Déclaration de la map is_interface

2.3 Construction de la surface

— Tesselation → Arêtes à l'interface → Dual (Surface)

— Index + Informations : Lien entre le .pdb (Chaîne, Atome, etc.) et les points de la surface

— smoothing

Lorsque les strucures de Delaunay ont été créées, il devient possible de calculer l'interface entre les deux protéines du complexe, ce qui constitue la partie centrale du projet. En implémentant la méthode vue dans la partie Triangulation de Delaunay, nous pouvons générer une surface qui pourra être affichée. Cette surface est un regroupement de polyhèdres de nombre de côtés variable.

Dans la première partie de l'implémentation, nous stockons la surface dans *vecteur* (*all_faces_indexes*) et une *map* (*surf_points*) (voir figure 2.6) qui permettent de garder en mémoire les coordonnées des points et les indices permettant d'afficher les faces. La *map* recense tous les points de la surface une fois, c'est-à-dire que les points utilisés dans différentes face de la surface ne sont présent qu'une seule fois dans la map. Cela permet de réduire considérablement la quantité de données et d'accélerer l'affichage. A chaque point de la map correspond un indice initialisé à 0 et qui sera mis à jour lors de l'écriture des fichiers utilisés pour l'affichage. Ensuite, le *vecteur* regroupe toutes les faces de la surface sous forme de vecteurs de point 3D. Le vecteur *all_faces_indexes* contiendra donc autant de vecteurs que le nombre de faces de la surface.

```
std::map<K::Point_3, unsigned int > surf_points; // link between point and index for faces
std::vector<std::vector<K::Point_3> > all_faces_indexes; // vector of faces (each line stores n points)
```

FIGURE 2.6 – Déclaration des vecteurs stockant la surface

Nous allons maintenant détailler les différentes parties de la fonction écrite pour le calcul de l'interface (*calculateInterface()*) qui est disponible en annexe (voir figure **??**).

Dans un premier temps, nous utilisons un itérateur fourni par CGAL pour parcourir les arêtes de la triangulation (voir figure 2.7). Dans cette itération, on vérifie pour chaque arête si elle est à

```
for (Delaunay::Finite_edges_iterator eit = interface_tr.finite_edges_begin(); eit != interface_tr.finite_edges_end(); ++eit)
```

FIGURE 2.7 – Itération sur les arêtes

l'interface, c'est-à-dire si elle possède un atome de chaque chaîne. Pour cela, nous nous référons à l'*Image pdb* grâce à l'indice stocké dans l'attribut *info()* de chaque vertice. Il est possible d'accéder à la valeur de la chaîne (A ou B) avec la commande en figure 2.6. Dans cette commande, *first* correspond à la cellule en cours d'accès et *second* à l'indice de la première vertice de l'arête (*third* permettra d'accéder à la seconde vertice de l'arête).

Après cette vérification, nous utilisons un circulateur pour "tourner" autour de l'arête concernée. En effet, le dual de Voronoï d'une arête dans un espace en trois dimension est un polyèdre. Celui-ci est composé des points qui sont les duaux des cellules (tétrahèdres) adjacentes à l'arête étudiée. Le circulateur permet donc de parcourir les cellules adjacentes des arêtes à l'interface. Chacun de ces point est alors stockés dans le *vecteur* et la *map* de la manière expliquée plus haut.

```
pdb->atom[eit->first->vertex(eit->second)->info()].chain
```

FIGURE 2.8 – Accès à la chaîne à laquelle appartient une vertice

Il est important d'expliquer d'autres vérifications qui sont efféctuées avant de stocker chaque point. Par exemple, il existe dans la structure *Delaunay* de CGAL un point "infini" permettant de "fermer" la triangulation dans l'espace. Ce point, bien qu'utile aux calculs effectués par CGAL, ne doit pas être pris en compte car il n'est pas réel. Nous vérifions donc qu'aucun des tetrahèdres adjacents à l'arête parcourue ne contient ce point spécifique. De plus, si l'on récupère tous les points générés par le calcul du diagramme de Voronoï, certains d'entre eux se trouveront en dehors de la zone utile à l'analyse de l'interface. Pour éliminer ces points, nous avons procédé en amont au calcul du barycentre de la protéine puis nous avons recensé la distance maximale entre un point du complexe et son barycentre. Si la distance entre un point d'une face et le barycentre calculé est supérieure à un certain pourcentage (ici 130%) de la distance maximale, alors la face ne sera pas gardée (voir figure 2.9).

```
for (int j = 0; j < face_indexes.size(); j++) { // Go through the points of the current face
    dist = barycenter - face_indexes[j]; // calculate the distance between the barycenter and the current point
    if ( dist.squared_length() > size_lim) { // If the distance of one point is over the limit
        is_used = false; // we will not store the face
    }
}
```

FIGURE 2.9 – Vérification de la distance au barycentre

Nous utilisons ensuite une autre classe CGAL (*Polyhedron*) qui permet de stocker une surface en trois dimension. L'avantage de cette classe est qu'elle possède une méthode permettant d'appliquer une subdivision de Catmull-Clark (voir figure 2.10). Cette subdivision permet de lisser la surface par morceaux pour l'affichage.

```
// Apply Catmull/Clark subdivision method to the polyhedron
int degree = 2; // Degree of the subdivision
CGAL::Subdivision_method_3::CatmullClark_subdivision(poly_surf,degree);
```

FIGURE 2.10 – Subdivision de Catmull-Clark

2.4 Affichage

— .off files → écriture avec indexation

Pour l'affichage des protéines et de l'interface, nous avons choisi les fichiers *.off* qui permettent de stocker une liste de points (colorés ou non) et d'indiquer les liens entre chacun de ces points (voir figure 2.11).

La première ligne indique le type de fichier (OFF) et la présence ou non de coloration : [C] si oui un espace vide sinon. La seconde ligne donne, dans cet ordre, le nombre de points, le nombre de cellules (polyhèdres) et le nombre d'arêtes ce dernier n'étant pas nécessaire à la lecture du fichier. Dans notre exemple, les lignes 4 à 22 listent les coordonnées des points et la couleur associée à chacun. La couleur est stockée en RGB (Rouge, Vert, Bleu) avec des entiers entre 0 et 255 ou des flottants entre 0.0 et 1.0.

```
1  [C]OFF
2  19 99 158
3
4  0.239 4.621 1.992 0 0 255
5  0.39 6.049 0.891 0 0 255
6  -0.066 7.299 2.11 0 0 255
7  -1.585 7.305 2.288 0 0 255
8  -3.73 8.162 1.335 0 0 255
9  -1.813 9.651 1.714 0 0 255
10 -2.207 8.305 1.312 0 0 255
11 -4.314 7.545 0.447 0 0 255
12 2.283 7.777 -0.568 255 0 0
13 2.794 6.49 -0.772 255 0 0
14 5.982 6.709 -1.786 255 0 0
15 2.589 8.474 0.606 255 0 0
16 3.915 6.595 1.378 255 0 0
17 3.405 7.883 1.578 255 0 0
18 4.174 4.513 -0.034 255 0 0
19 3.608 5.896 0.201 255 0 0
20 6.103 5.676 -1.129 255 0 0
21 5.091 4.543 -1.262 255 0 0
22 5.769 3.262 -1.424 255 0 0
23
24 3 0 2 3
25 3 1 2 3
26 3 1 0 3
27 3 1 0 2
28 3 7 5 4
29 3 6 5 4
30 3 7 6 4
31 3 7 6 5
32 3 9 10 7
33 3 18 10 7
34 3 18 9 7
35 3 18 9 10
36 3 15 1 8
37 3 13 1 8
38 3 13 15 8
39 3 13 15 1
40 3 1 8 7
41 3 9 8 7
42 3 9 1 7
43 3 9 1 8
44 3 6 2 3
45 3 6 1 3
```

FIGURE 2.11 – Exemple de fichier *.off*

Au delà de la ligne 23, les cellules sont listées, avec comme premier entier à chaque ligne, le nombre de points de la cellule. Comme notre exemple représente une triangulation de Delaunay, ce nombre vaut 3 car chaque face de la triangulation correspond à un triangle. A la suite de cet entier viennent les indices des points composant la cellule (ou la face). Cet indice correspond à la place des coordonnées listées plus haut.

3 Résultats et perspectives

3.1 Résultats

— surface

Nous sommes donc capables, étant donné un complexe stocké dans un fichier *.pdb*, d'afficher ce complexe (voir figure 3.1) et de visualiser la surface de contact dans Meshlab (voir figure 3.2).

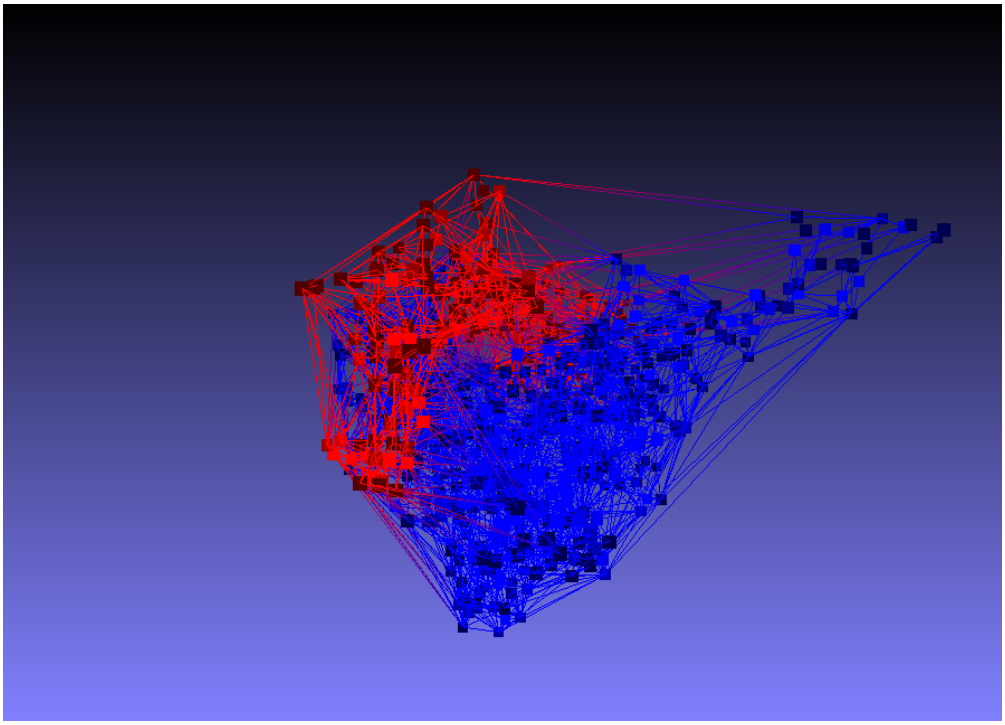


FIGURE 3.1 – Complexe

La méthode d'indexation permet de connaître à tout moment les particularités de chaque atome et de lier ces informations à la surface. Les deux atomes formant une arête de l'interface correspondent à un morceau de la surface.

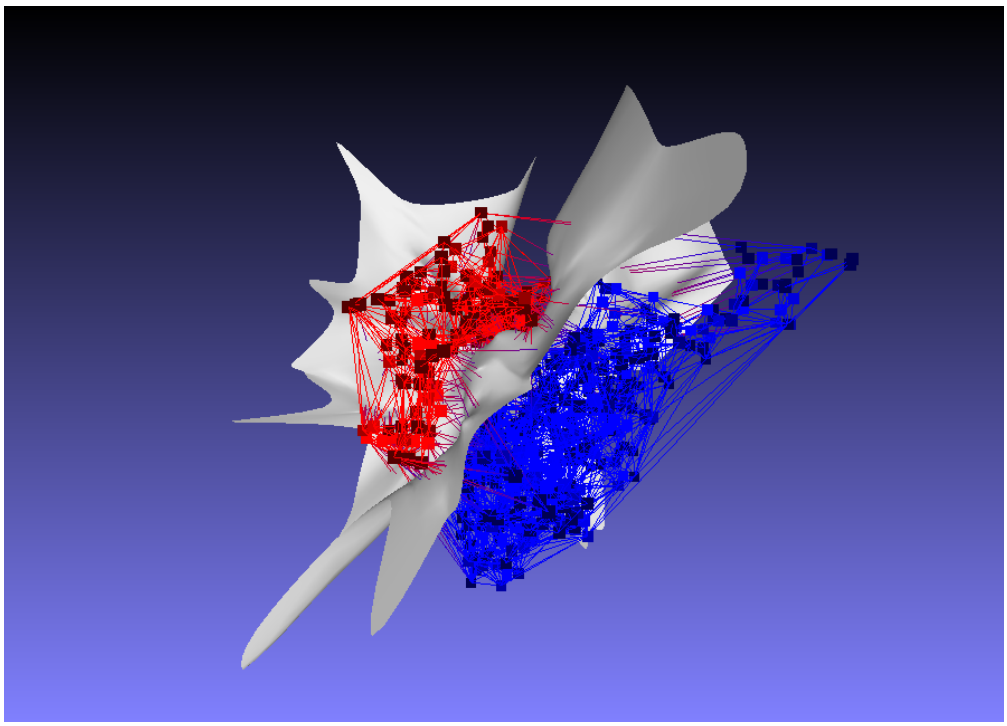


FIGURE 3.2 – Affichage de l'interface entre deux protéines

3.2 Perspectives

— Fonctionnalités

La méthode d'affichage utilisée (indexation des points et affichage par triangles) peut permettre de passer facilement en OpenGL.

Conclusion

Liste des illustrations

Bibliographie

Annexes