Institut
National
Polytechnique
de Lorraine

Ecole Nationale Supérieure de Géologie de Nancy
(E.N.S.G. - L.I.A.D.)
Centre de Recherche Informatique de Nancy
(C.R.I.N.)

THESE

présentée devant l'Institut National Polytechnique de Lorraine
en vue de l'obtention du titre de

Docteur de l'I.N.P.L

Informatique

par

Pascal LE MELINAIRE

Modélisation de Relations Géométriques par la méthode DSI - Application à la Géologie -

Soutenue publiquement le 15 Janvier 1992, devant le Jury composé de :

Neil	CARMICHAEL	Président
Philippe	CINQUIN	Rapporteur
Pierre	LESCANNE	Rapporteur
Jean-Laurent	MALLET	Examinateur
André	HAAS	Examinateur
Abdel	BELAID	Examinateur

REMERCIEMENTS

C'est à tort que l'on qualifie une thèse de travail personnel, et je souhaite remercier ci-dessous l'ensemble des personnes qui m'ont aidé et permis d'aboutir dans ce travail.

En tout premier lieu, je tiens à exprimer ma gratitude au Professeur Jean-Laurent Mallet, Directeur du Département Informatique de l'Ecole de Géologie, pour la confiance qu'il a su me donner et le temps qu'il m'a consacré. Tout au long de ces deux années, il m'a orienté avec sa compétence, son enthousiasme, sa gentillesse et sa rigueur intellectuelle. Grâce au consortium GOCAD qu'il a mis en place, ses élèves jouissent d'un environnement exceptionnel, tant sur la qualité du matériel mis à leur disposition, que sur l'opportunité de parcourir le monde au gré des congrès internationaux.

C'est au cours de l'un de ces voyages, que j'ai eu la chance de rencontrer Monsieur Neil Carmichael, et je le remercie vivement d'avoir accepté la présidence de mon jury. Mes remerciements sont destinés également à Monsieur Pierre Lescanne, Directeur de recherche à l'INRIA, et à Monsieur Philippe Cinquin, Professeur d'Informatique Médicale au laboratoire TIMB-IMAG de Grenoble, qui ont gentiment acceptés d'être les rapporteurs de ce mémoire et je remercie tout autant, Monsieur André Haas, Ingénieur Elf-Aquitaine et Monsieur Abdel Belaïd, Chargé de recherche au CRIN, qui m'ont fait l'honneur de juger ce travail.

Mes remerciements s'adressent de même à l'Ecole Nationale Supérieure de Géologie, à l'Institut National Polytechnique de Lorraine et au Centre de Recherche en Informatique de Nancy, qui m'a récemment accueilli au sein de son équipe.

J'adresse également mes remerciements à tous les membres du LIAD, et à tous ceux qui m'ont soutenu dans ce travail, et je leur exprime mon amitié. Je remercie de même Mme Cucurgno pour sa compétence et son dévouement.

Enfin, je tiens à remercier l'ensemble des membres du consortium GOCAD,
qui supportent ce projet avec enthousiasme et intelligence :

Universités

Ecole des Mines de Paris
ENSG (Nancy)
Geologistes Institut da Universitat (Freiburg)
IPG (Paris)
Osservatorio Geofisico Trieste
Stanford University

Compagnies

Amoco
BRGM (France)
CFP TOTAL
Compagnie Générale de Géophysique
Chevron
DEC
ELF Aquitaine
Gaz de France
Hewlett-Packard
IFP
Phillips Petroleum
Shell Research
Silicon Graphics
Statoil
Sun
TNO Institut
Unocal

Résumé

La majorité des systèmes de *CAO* classiques supposent que les surfaces admettent une représentation paramétrique $B(u,v)$ et sont interpolées par des fonctions du type *Bezier*, *B_spline*, *β_spline* ou *NURBS*. Ces méthodes, généralement appliquées dans l'industrie automobile et mécanique, ne permettent pas de traiter correctement des surfaces géologiques complexes ou de tenir compte interactivement de changements topologiques.
Les travaux présentés dans cet ouvrage, furent menés au sein du projet GOCAD, dans lequel le Pr.Mallet propose une approche totalement différente, qui est basée sur un nouveau principe d'interpolation en trois dimensions, appelé D.S.I (Discrete Smooth Interpolation). Les surfaces considérées sont constituées d'un maillage triangulaire, et différents opérateurs linéaires, appelés "**contraintes**", gouvernent l'allure de ces surfaces.
Trois nouvelles contraintes vous sont présentées:
Les "**Fuzzy Control Point**" et les "**Fuzzy Control Line**" ont respectivement pour but d'ajuster une surface à un ensemble de points distribués dans l'espace et à un ensemble de courbes. Elles sont parfaitement adaptées à des jeux de données important (plusieurs milliers de points), et elles ne perturbent en aucune façon la régularité et l'équilatérité des triangles du maillage. La contrainte "**On-Tsurf**" est plus originale, car elle tente de modéliser des notions et des comportements spécifiquement géologiques, telles que la notion de contact entre deux surfaces et le glissement de l'une sur l'autre. Ces contraintes ont la particularité de mettre en jeu plusieurs objets, c'est pourquoi une grande partie de ce rapport est consacrée à un mécanisme de gestion de dépendances entre objets. Enfin, lorsque ces objets sont appelés à partager un ensemble de points communs, un mécanisme d'échange, basé sur des exportations et des importations vous est présenté.

Présentation de la thèse

Présentation

L'industrie pétrolière est un secteur qui regroupe de nombreux domaines d'activités tels que la géologie, la géophysique, et l'ensemble des techniques liées au réservoir. Les progrès réalisés grâce au développement de l'informatique, permettent d'envisager de regrouper l'ensemble de ses applications au sein d'un vaste système global. Cependant sa conception se heurte, à l'heure actuelle, à l'absence de support géométrique performant. En effet, lors de l'étude d'un bassin pétrolier, la modélisation géométrique des horizons représente le dénominateur commun, à travers lequel l'ensemble de ces techniques peuvent communiquer. Par exemple, le même modèle géométrique peut être utiliser afin de tester des simulations de sismique ou des calculs de volume. Par conséquent, le support géométrique représente à la fois le coeur et le chaînon manquant du problème. Pour deux raisons principales, l'ensemble de nos sponsors, espèrent déceler dans les travaux réalisés, au sein du projet GOCAD, une première solution:

- Le système n'est pas un logiciel, mais un ensemble de librairies de fonctions écrites en language C. Il constitue donc un univers ouvert qui peut être connecté à d'autres applications.
- Le système est conçu autour d'une nouvelle approche mathématique d'interpolation, appelée *DSI*, qui offre des possibilitées à la fois nouvelles et bien plus proches des problèmes rencontrés dans l'industrie pétrolière.

Ce bref exposé situe le cadre de mes travaux. L'objectif principal du projet consite à résoudre des problèmes d'ordre géométrique, et j'espère avoir contribué durant cette thèse à son enrichissement.

Organisation du manuscrit

Ce manuscrit est divisé en cinq chapitres:

1. Dans une première partie nous résumons succintement les notions et structures de la base de données, nécessaires à la compréhension de cet ouvrage.
2. Dans le second chapitre, nous abordons la description d'un mécanisme d'importations et d'exportations de points entre objets. Ce mécanisme

illustre la notion de soudure "hard" entre différents éléments.
(ex: soudure de deux surfaces, ou d'une surface et d'une ligne).

3. La troisième partie se subdivise en quatre sections. Tout d'abord nous présentons en détail le principe de la méthode *DSI*, et introduisons notamment le concept de "contrainte". Ensuite nous enchaînons sur l'implémentation de trois nouvelles contraintes. La première, appelée "Fuzzy Control Point", consiste à ajuster d'une manière originale une surface à un ensemble de points distribués dans l'espace. La seconde, appelée "Fuzzy Control Line", représente une extension de la première et permet de manipuler une surface à travers le déplacement d'un ensemble de lignes. Enfin, la troisième, dénommée "On-Tsurf", a pour but de prouver que la méthode *DSI* est parfaitement adaptée à la modélisation de comportements spécifiquement géologique. Elle illustre à la fois la notion de contact entre deux surfaces et le glissement de l'une sur l'autre (ex: l'horizon glisse le long de la faille). En fait, cette contrainte peut être considérée comme la représentation d'une soudure "soft" entre deux surfaces.

4. Les contraintes implémentées ont la particularité de mettre en jeu différents objets, et donc établissent des dépendances entre ceux-ci. Une des grandes difficultés de ces travaux consistait à imaginer un mécanisme capable de gérer ces relations en temps réel. Ce chapitre lui est consacré.

5. Enfin, la dernière partie pourra vous sembler superficielle, néanmoins elle est utilisée en permanence par le logiciel. Elle apporte une solution générale aux problèmes de la sauvegarde d'une base de données, qui gèrent des dépendances entre objets.

Ces travaux sont orientés selon deux thèmes:

Le premier est consacré aux aspect mathématiques. Il fait appel à des connaissances sur le principe de *DSI*, et il concerne la conception des contraintes. L'autre thème est beaucoup plus orientée sur l'informatique et concerne l'implémentation de ces contraintes au sein du système GOCAD. Cependant, les outils développés ont été conçus, si possible de façon générale, afin qu'ils puissent être adaptés à différentes bases de données.

Table des matières

Chapitre 1

Description de la base de données

Ce chapitre est un résumé des différentes notions et structures essentielles de la base de données du système GOCAD. Sa conception réalisée en language C, ainsi que l'ensemble du logiciel fut fortement influencée par les principes de la programmation objet. Elle est constituée d'objets et de sous-objets, dont les définitions sont très précises, et qui ont pour but de représenter au sein du système des notions courantes telles que, des surfaces, des lignes, ou des ensembles de points. Le lecteur est invité à s'y référer, en toute occasion.

1.1 Notion de point, vertex et vset

Une grande distinction est fait entre un point, en tant qu'élément d'un objet, que nous appellerons un vertex, et le point en tant que coordonnées dans l'espace.

```
typedef struct POINT3_t
        {
        float  x,y,z ;
        } POINT3_t;
```

- x,y,z sont les trois coordonnées du point dans le repère de travail.

Cette première structure, qui regroupe trois coordonnées dans l'espace, correspond à une structure de travail. Elle est utilisée pour définir des vecteurs, ou des coordonnées cartésiennes. Contrairement au point, le vertex est l'entité qui caractérise la position d'un objet. Le vertex est une sous-classe dérivée de celle d'un point, spécifiquement adaptée aux besoins de la base de données GOCAD.

```
typedef struct VERTEX_t
        {
        float    x,y,z   ;
        int      movable ;
        VSET_c   p_owner ;
        ...
        } VERTEX_t ;
```

- x,y,z sont les trois coordonnées du vertex.
- *movable* est un entier qui définit le degré de liberté du point selon les axes Ox,Oy,Oz.
- *p_owner* représente l'objet propriétaire du vertex. Chaque vertex est en effet contenu obligatoirement dans une structure spéciale de container appelée, vset, et décrite ci-dessous.

Chaque objet contient un unique **vset** et chaque vset regroupe l'ensemble des vertex qui définissent la position de cet objet dans l'espace. Les relations entre vset et vrtx, méritent une description minutieuse,et sont exposées dans le chapitre dédié aux importations et exportations de vertex (cf page 12).

1.2 Notion de surface, d'atome et de triangle

Une surface, au sens de la base de données, est constituée d'un maillage triangulaire. Chaque noeud du maillage, comme l'illustre la figure 1.1, est connecté à un certain nombre de premiers voisins, appelés "satellites". Etudions en détail la structure d'un atome.

```
typedef struct ATOM_t
        {
        ATOM_t    * next    ;
        VRTX_t    * p_vrtx  ;
        ATOM_t   ** p_sat   ;
        int         nb_sat  ;
        CNSTR_t   * p_const ;
        ...
        } ATOM_t ;
```

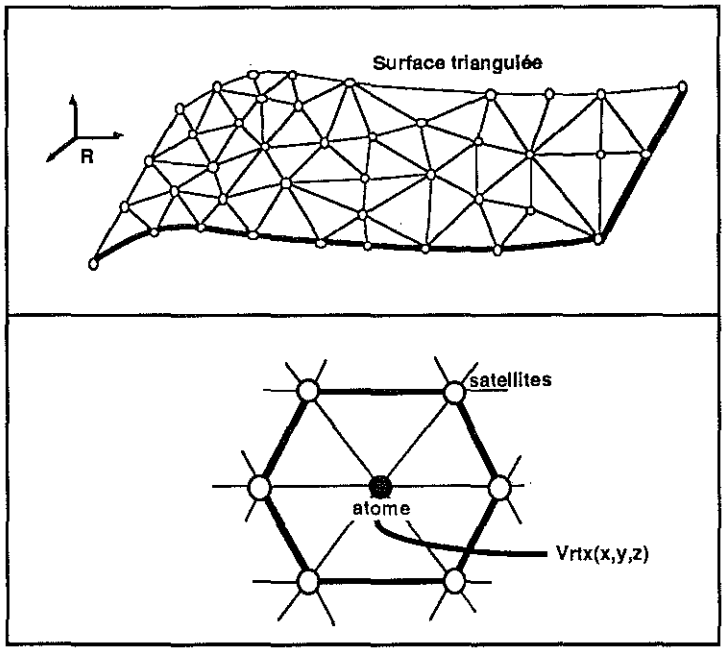

Figure 1.1 *shéma d'une surface triangulée.*

- *next* la structure d'atome est utilisée comme l'élément d'une liste chaînée, et le champ next procure l'accès à l'élément suivant de cette liste.
- *p_vrtx* est un pointeur vers un vertex. Ce champ caractérise la position d'un atome dans l'espace.
- *p_sat* est un tableau de pointeurs, où sont stockés pour chaque atome l'intégralité de ses premiers voisins, définis par le maillage de l'objet. Les atomes voisins d'un atome sont appelés ses satellites.
- *nb_sat* est le nombre des premiers voisins de l'atome considéré.
- *p_const* est un pointeur sur une liste de contraintes.

L'atome est l'entité élémentaire du maillage d'une surface. Chaque atome contient un vertex qui définit sa position dans l'espace et l'ensemble des ses satellites, qui caractérise sa connectivité avec les autres atomes de la surface. Cette connectivité induit la topologie même de cet objet. Le nombre de satellites d'un atome est en principe illimité [1], et chaque satellite est

[1] limité à la taille d'un int

évidemment lui-même un atome. La notion d'atome est la structure essentielle de la base de données car elle contient à la fois les caractéristiques topologiques et géométriques des objets.

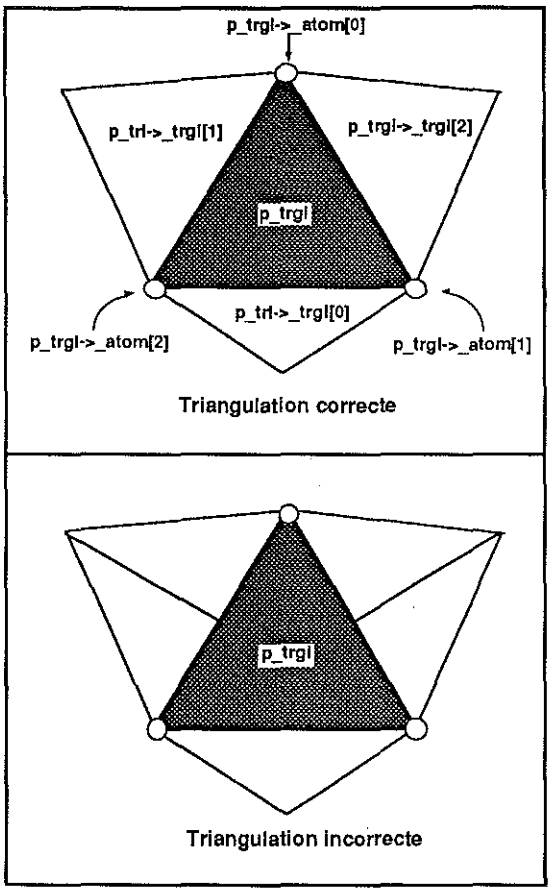

Figure 1.2 *schéma des conventions de la notion de triangle.*

Une surface est aussi un assemblage de triangles connectés les uns aux autres, par leurs arêtes. La structure *TRGL_t* définit la notion de triangle.

```
typedef struct TRGL_t
        {
        TRGL_t   * next      ;
        TRGL_t   * p_trgl[3] ;
        ATOM_t   * p_atom[3] ;
        ...
        } TRGL_t ;
```

- *next* la structure de triangle est une liste chaînée, et le champ next procure l'accès à l'élément suivant de cette liste.
- *p_trgl* est un pointeur vers un tableau de trois pointeurs, qui correspondent aux trois triangles adjacents d'un triangle.
- *p_atom* est un pointeur vers un tableau de trois pointeurs, qui correspondent aux trois atomes sommets du triangles.

Deux conventions régissent la notion de triangle. Elles sont illustrées par la figure 1.2.

1. Chaque triangle possède au plus trois voisins, et en aucun cas une de ses arêtes ne peut être partagée par plus d'un triangle. Si un triangle se situe sur le bord d'une surface, l'un de ses voisins sera nul.
2. La numérotation des indices dans les tableaux des atomes et des triangles n'est pas indépendante. Le triangle adjacent d'indice (i) ne possède jamais l'atome, d'indice (i), comme sommet. Il se situe donc sur le côté opposé au sommet d'indice (i).
   Exemple: p_atom[0] est le sommet opposé au triangle p_trgl[0].

La structure *TSURF_t* [2] définit la notion de surface.

```
typedef struct TSURF_t
        {
        int       type    ;
        char    * name    ;
        ...
        ATOM_t  * p_atom  ;
        long      nb_atom ;
        TRGL_t  * p_trgl  ;
        long      nb_trgl ;
        VSET_t  * p_vset  ;
        ...
        } TSURF_t ;
```

- *type* est un entier qui définit le type d'un objet.
- *name* est une chaîne de caractères qui caratérise l'objet.
- *p_atom* est un pointeur sur la liste des atomes que contient la surface.
- *nb_atom* est le nombre d'atomes de la précédente liste.

[2] TSURF_t signifie une surface triangulée

- *p_trgl* est un pointeur sur la liste de triangles qui constituent la surface.
- *nb_trgl* est le nombre de triangles de la précédente liste.
- *p_vset* est un pointeur sur le vset de l'objet.

Une surface est constituée à la fois d'une liste de triangles et d'atomes. Cette double représentation est très intéressante car suivant le type d'opérations appliquée sur la surface, l'une ou l'autre sera adoptée. Les calculs sont ainsi facilités et optimisés.
Exemple : Lors d'une translation, seule la position de chaque atome de la surface est modifiée, et à l'inverse, lors du calcul de l'intersection de deux surfaces la représentation triangulaire est plus judicieuse. Nous appellerons tsurf tout élément de structure *TSURF_t*.

1.3 Notion de ligne et de segment

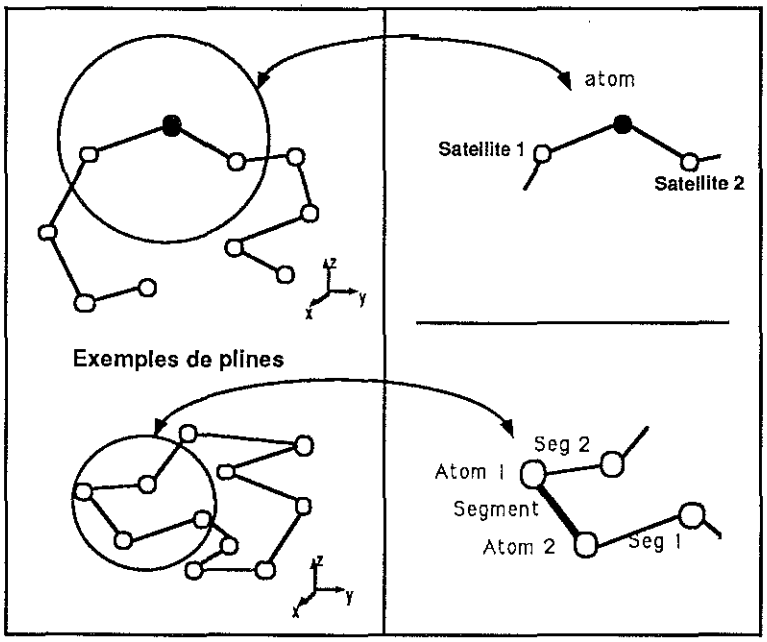

Figure 1.3 *schéma d'une ligne polygonale (pline).*

A l'image d'une surface, une ligne est un assemblage particulier de nœuds. Comme l'illustre la figure 1.3, ceux-ci correspondent à des atomes, et chaque

lien reliant deux atomes, est appelé un segment, dont la structure est définie comme suit:

```
typedef struct SEG_t
        {
         SEG_t   * next      ;
         SEG_t   * p_seg[2]  ;
         ATOM_t  * p_atom[2] ;
        } SEG_t ;
```

- *next* la structure de segment est utilisée comme l'élément d'une liste chaînée et le champ next procure un accès à l'élément suivant de cette liste.
- *p_seg* est un pointeur vers un tableau de deux pointeurs de type *SEG_t*, qui représentent les deux segments adjacents. Ces segments sont à l'occasion nuls.
- *p_atom* est un pointeur vers un tableau de deux pointeurs de type *ATOM_t*, qui représentent les deux extrémités d'un segment.

Un segment possède au plus deux segments adjacents, et le nombre de satellites d'un atome d'une ligne est limité à deux éléments. Une ligne s'apparente à un graphe unidimensionelle, tandis qu'une surface est un graphe bidimensionelle. La structure *PLINE_t* [3] caractérise la notion de ligne polygonale au sein du système.

```
typedef struct PLINE_t
        {
        int      type    ;
        char   * name    ;
        ...
        ATOM_t * p_atom  ;
        long     nb_atom ;
        SEG_t  * p_seg   ;
        long     nb_seg  ;
        VSET_t * p_vset  ;
        ...
        } PLINE_t;
```

- *type* est un entier qui définit le type d'un objet.
- *name* est une chaîne de caractères qui caratérise l'objet.

[3] PLINE_t signifie "polygonal line"

- *p_atom* est un pointeur sur la liste des atomes que contient la ligne.
- *nb_atom* est le nombre d'atomes de la précédente liste.
- *p_seg* est un pointeur sur la liste de segments que contient la ligne.
- *nb_seg* est le nombre de segments de la précédente liste.
- *p_vset* est un pointeur sur le vset de l'objet.

Nous appelerons **pline** tout élément de structure *PLINE_t*.

1.4 Notion de gocad-object

Toute structure dont les quatre premiers champs sont identiques à ceux de la structure *GOBJ_t* est considérée comme un gobj.

```
typedef struct GOBJ_t
        {
        int      type       ;
        char *   name       ;
        long     mailbox    ;
        SETK_t * blackboard ;
        } GOBJ_t ;
```

- *type* est un entier qui caractérise le type du gobj. Exemples : TSURF, PLINE ou VSET.
- *name* est une chaîne de caractères qui représente de façon univoque un gobj, puisque deux gobjs ne portent jamais le même nom.
- *mailbox* est un entier utilisé par un mécanisme interne de gestion des objets. Ce principe ne sera pas décrit dans cet ouvrage.
- *blackboard* est un "container" de type spécial. Il correspond à une zone d'extension variable, destinée au stockage d'informations. Cet espace permet d'étendre de manière artificielle la définition d'une structure. Cet outil complexe sera décrit dans le chapitre dédié aux relations entre objets (cf page 109) de manière plus complète.

La notion de gobj permet de manipuler l'ensemble des macro-objets de la base de données de façon unique. Par exemple, une tsurf, une pline, un vset sont tous des gobjs.

Chapitre 2

Exportation et Importation de Vertex

2.1 Exposé du problème

Un des objectifs majeurs du système GOCAD consiste à modéliser des ensembles géologiques. Cependant, un bassin n'est jamais une simple disposition d'horizons dans l'espace, et le comportement de chaque couche dépend des couches voisines. La simple représentation géométrique des surfaces est insuffisante et il est indispensable de modéliser des comportements entre objets. Ces comportements se divisent en deux catégories. La première, appelée "hard", se caractérise par le fait que deux objets se partagent physiquement un certain nombre de points dans l'espace. A l'inverse la deuxième, appelée "soft", regroupe l'ensemble des relations qui lient deux objets qui sont à priori totalement indépendants. La figure 2.1, qui représente une soudure entre deux surfaces, illustre ce propos. Cette soudure peut être considérée de deux manières distinctes:

- *La soudure hard*
  La ligne de soudure est un ensemble de points partagés par deux surfaces. Elle est parfaite, et correspond désormais à une liaison figée, qui induit le comportement respectif des deux objets. Par exemple, si la surface *S2* est translatée, la géométrie de la surface *S1* est automa-

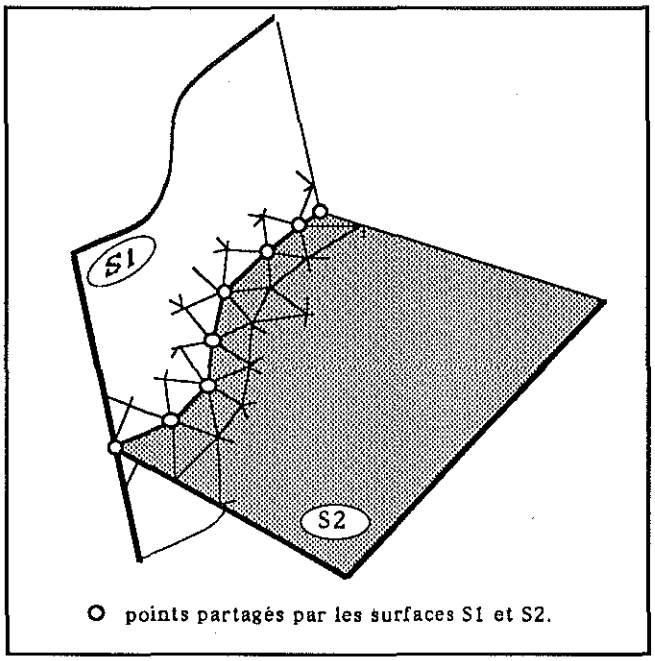

Figure 2.1 *soudure entre deux surfaces.*

tiquement modifiée puisque l'ensemble des points qu'elle partage s'est déplacé.

- *La soudure soft*
  La ligne de points est désormais composée de points de *S2* qui reposent à tout moment sur la surface *S1*, mais aucun point n'est réellement partagé par les deux objets. Ce type de soudure, permet à la surface *S2* de glisser le long de la surface *S1*, puisque la ligne de soudure, est constituée de points de *S2* qui sont astreints à se déplacer uniquement le long des facettes triangulaires de la surface *S1*.

Ces deux types de comportements sont donc bien différents. Dans le chapitre présent, nous ne traiterons que le cas "hard", et nous reviendrons au cas "soft" dans le chapitre consacré à la contrainte "On-Tsurf" (cf page 78).

Le partage de points dans l'espace n'est nullement réservé aux domaines des surfaces. La figure 2.2 représente un exemple où une surface et une ligne se partagent un ensemble de points. Dès lors le déplacement de la ligne permet de modifier de façon conviviale et interactive l'allure générale de la

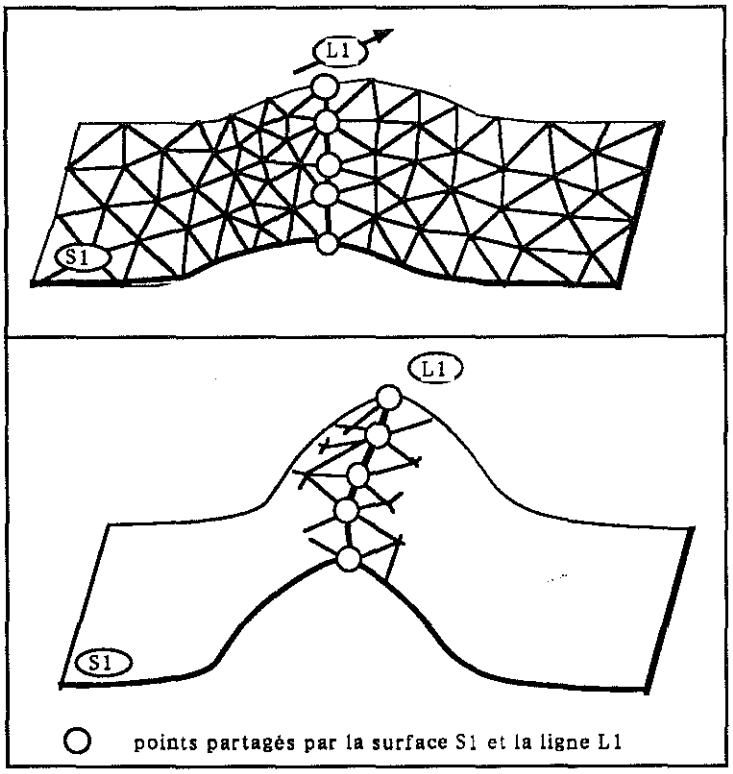

Figure 2.2 *la translation de la ligne L1 modifie automatiquement la géométrie de la surface S1.*

surface. Cet outil est très intéressant, en CAO, pour remodeler des objets.

Le but de ce chapitre est de proposer un mécanisme, qui puisse résoudre de manière cohérente le partage de points entre objets. Son efficacité se jugera par sa capacité à résoudre, les problèmes suivants:

1. *la destruction d'un point*
   Supposons que deux objets A et B se partagent un certain nombre de points. Si au cours de la session, A est détruit, alors tous ses points doivent être détruits, exceptés ceux qu'il partage. Par conséquent, la destruction d'un point est dépendante de son utilisation.

2. *la transparence du mécanisme*
   L'utilisateur ne doit pas se soucier de l'état d'un point. Selon le nombre d'objets qui le partagent, le système doit être à même de déclencher

les actions appropriées. A charge au système et non à l'utilisateur de résoudre ces problèmes.
Exemple: Quelque soit ld degré d'utilisation d'un point, l'utilisateur ne doit avoir qu'une unique fonction pour le détruire.

3. *la dépendance d'un objet*
Lorsque deux objets se partagent au moins un point, leurs comportements ne sont plus indépendants. Le mécanisme installé doit donc permettre de sonder pour tout objet et à tout moment, son état de dépendance.

2.2 Le mécanisme adopté

La base de données GOCAD sépare distinctement les notions de topologie et de géométrie de ses objets. Une surface et une ligne peuvent être considérées comme des ensembles de noeuds [1] connectés dans l'espace, qui contiennent chacun deux types d'informations:

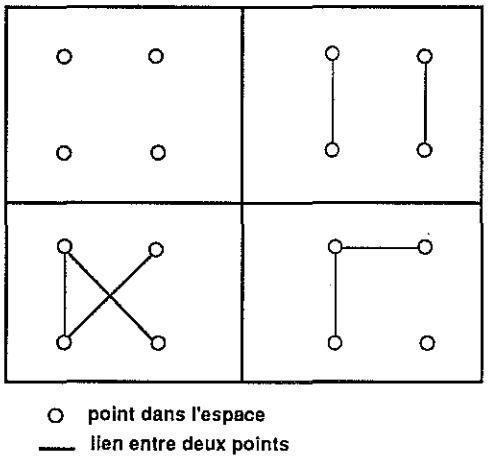

Figure 2.3 *exemples de graphes différents basés sur quatre points identiques.*

1. **sa connectivité avec les autres noeuds.**
Les points d'un même ensemble peuvent être reliés les uns aux autres de multiples façons, comme vous l'indique la figure 2.3. L'ensemble de ces liens correspond à un graphe qui définit la topologie de l'objet. La

[1] au sens de la terminologie des graphes

structure d'atome (cf page 5) constitue la structure élémentaire d'un tel graphe. Elle stocke pour chaque noeud, l'ensemble de ses premiers voisins, appelés satellites.
La notion d'**atome** définit la **topologie** d'un objet.

2. **sa position géométrique dans l'espace**
La connectivité d'un noeud est absolument indépendante de sa position dans l'espace. Les coordonnées (x,y,z) d'un noeud sont stockées dans une structure appelée vertex (cf page 5). Chaque atome contient un vertex qui détermine sa position dans l'espace. La notion de **vertex** définit la **géométrie** d'un objet.

Tout noeud de l'espace, est donc la combinaison d'un atome et d'un vertex. Si un atome appartient toujours à un seul et unique objet, deux atomes, d'objets différents, peuvent partager un même vertex, afin d'acquérir une position commune.
Exemple: La ligne de soudure des objets L1 et S1 représentée à la figure 2.2, est constituée d'un ensemble de couples (*atome de* $S1$, *atome de* $L1$) qui partagent les mêmes vertex.

Lorsque deux objets partagent un point, ils partagent en fait un même vertex. L'unicité de ce vertex est un élément essentiel de la cohérence du système, et nous l'avons caractérisé, en attribuant à chaque vertex un unique propriétaire. Ainsi, la relation entre les deux objets se conçoit plus comme un prêt que comme un partage. Plus précisemment l'objet "propriétaire" se charge d'exporter un de ses vertex vers un autre objet. Le système adopté correspond donc à un mécanisme d'exportations et d'importations de vertex entre différents objets.

Avant de rentrer dans de plus amples détails, attardons nous sur les notions de "vset" et de "vport", qui régissent ce mécanisme.

2.2.1 notion de vset

Nous appelerons **vset** [2](cf page 5), une structure qui regroupe un ensemble de vertex. Chaque objet du système contient un unique vset et, lors d'une exportation ou importation de vertex, la communication est réalisée via les deux vsets respectifs des objets.

Exemple: les objets "tsurf" (surface triangulée) et "pline" (ligne polygonale), qui représentent respectivement une surface et une ligne au sein de

[2]vset signi e \set of vertices".

la base de données, possèdent chacun leur vset.

Le vertex est un sous-objet qui ne peut exister en lui-même. Il doit obligatoirement appartenir à un vset. Lors de sa création tout vertex est rattaché à un vset, appelé son **propriétaire**[3] et, désormais, seul celui-ci est habilité à l'exporter vers d'autres vsets. En aucun cas le propriétaire d'un vertex ne peut refuser son exportation, et le nombre de vertex susceptibles d'être exportés est illimité. Ce principe assure la cohérence du système ainsi que l'unicité du vertex. En effet, il est impossible de générer ou d'utiliser des vertex qui n'appartiendraient à aucun vset. Notez dès à présent, que lorsqu'un vset A exporte l'un de ses vertex vers le vset B, ce dernier réalise une importation. Cette réciprocité doit bien entendu être gérée et mis à jour dans les deux vsets concernés. De même, la notion de propriété n'est en aucun cas définitive. En effet ces droits de propriété sont transmis, si besoin est, lors de la destruction du vset propriétaire. Nous reviendrons sur ce point plus en détail.

Afin de répondre aux besoins d'exportations et d'importations, chaque vset est décomposé en trois parties :

1. **la partie domestique et privée:**
   Elle contient tous les vertex, appartenant à ce vset.

2. **la partie domestique et exportée:**
   Elle contient tous les vertex, que ce vset exporte vers d'autres vsets. Le vset est donc propriétaire de tous ces vertex.

3. **la partie importée:**
   Elle contient tous les vertex importés par ce vset. Aucun de ces vertex ne lui appartient, puisqu'ils sont la propriété du vset qui les exporte.

Lors de tout échange de vertex, ces trois sous-ensembles doivent impérativement être mis à jour de façon cohérente.

Exemple: soit *vrtx* un vertex appartenant à la partie domestique et privée du vset *vset1*. Si *vrtx* est exporté vers un deuxième vset *vset2*, alors il doit être installé dans la partie domestique exportée de *vset1* et dans la partie importée de *vset2*.

Concrètement, la mise à jour est plus complexe. En effet, supposons qu'un vset exporte deux cents vertex vers un vset et cinq cents vers un autre. Il semble astucieux de regrouper chaque vertex de même destinataire

[3] dorénavant un vertex appartient à un vset U, signifie que U est son vset propriétaire.

dans le même "colis". La notion de vport, évoquée ci-dessous, correspond à ce besoin.

2.2.2 notion de vport

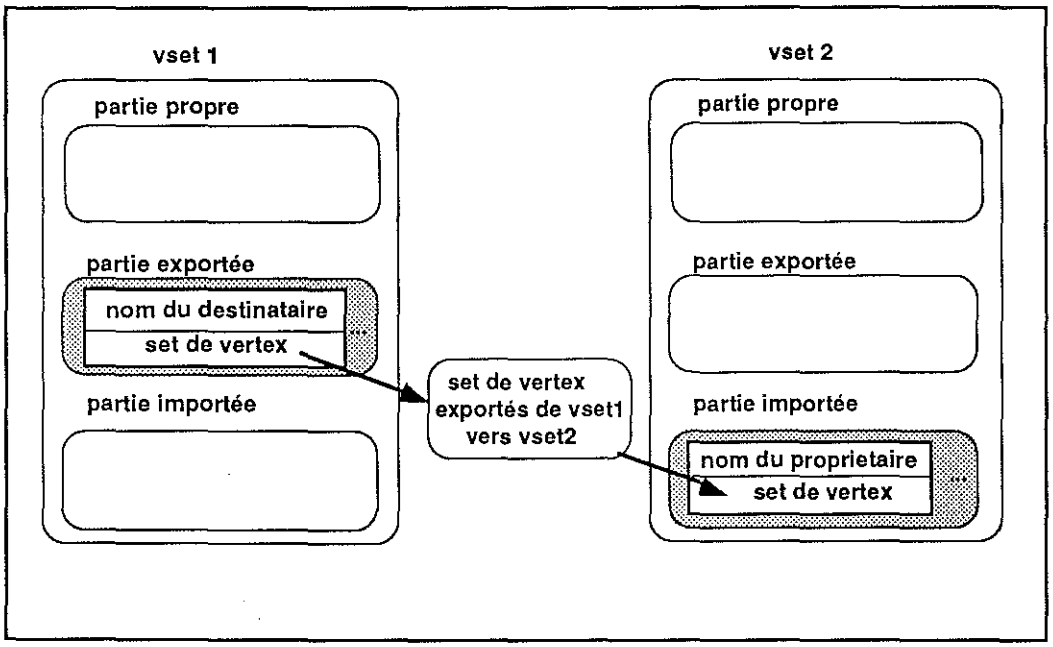

Figure 2.4 *lors de tout échange de vertex, chacun des vsets concernés utilise un vport pour communiquer.*

Les communications de vset à vset sont réalisées à travers des **vports**. Le vport est une structure qui regroupe tous les vertex mis en jeu entre deux vsets.
La figure 2.4 illustre l'exemple suivant: soit *vset1* un vset qui exporte plusieurs vrtx vers un second vset appelé *vset2*. Remarquez tout de suite que *vset2* importe exactement les mêmes vertex de *vset1*. Cet ensemble de vertex est regroupé dans un container commun appelé **set**, de telle sorte que chaque vport comporte un set et un destinataire. Dans la partie exportée de *vset1*, un vport destiné à *vset2* et contenant le set des vertex exportés, sera créé. Réciproquement, dans la partie importée de *vset2* un vport contenant le même set mais issu cette fois de *vset1* sera initialisé.

La gestion des exportations et importations de vertex entre objets se résume à l'existence et à la cohérence de ces vports.

2.3 Les structures implantées

La conception purement informatique de ce mécanisme nécessite la création de nouvelles structures. En effet, l'efficacité du mécanisme dépend de sa rapidité d'exécution et de sa facilité de gestion. L'information contenue doit être le moins redondante possible, et occuper le minimum d'espace mémoire. De même, les accès aux informations doivent être optimisés. Ce paragraphe présente dans le détail les structures adoptées. Je tiens à préciser que l'ensemble de ces structures furent mises au point par Monsieur Philippe Nobili, puisqu'il réalisa au cours de son passage au laboratoire de Nancy, une toute première version de ce mécanisme.

2.3.1 le vertex

La structure **vertex**, décrite ci-dessous, est une classe dérivée de celle d'un point.

```
typedef struct VRTX_t
  {
   float    x,y,z   ;
   long     movable ;
   VSET_t * p_owner ;
  } VRTX_t ;

 typedef struct VRTX_t * VRTX_c;
```

- x,y,z sont les coordonnées du vertex, dans l'espace.
- *movable* est un attribut du vertex, qui exprime son degré de liberté suivant les axes ox, oy, oz.
- *p_owner* est un pointeur sur le vset propriétaire du vertex.

Le vertex est la structure élémentaire qui définit la géométrie d'un objet.

2.3.2 le set

Le set est une structure spéciale de container. Elle offre les capacités réunies d'une hash-table et d'une simple liste, comme le décrit le schéma figure 2.5.

```
typedef struct SET_t
  {
   long       nb_item    ;
   long       size       ;
   SNODE_t ** pp_node    ;
  } SET_t ;

 typedef struct SET_t * SET_c;
```

- *nb_item* représente le nombre d'items stockés .
- *size* est égale aux nombre de cases de la htable.
- *pp_node* est un double pointeur sur une structure SNODE_t, conçue comme un tableau à deux dimensions ayant un nombre de colonnes variables; elle est décrite ci-dessous.

```
typedef struct SNODE_t
  {
   SNODE_t *  next_c    ;
   SNODE_t *  next_r    ;
   PNTR_t     item      ;
  } SET_t ;

 typedef struct SNODE_t * SNODE_c;
```

- *next_c* est un pointeur qui chaîne sous forme d'une liste simple les éléments contenus sur une même ligne. Il procure en fait l'accès à la colonne suivante.
- *next_r* est un pointeur qui chaîne tous les éléments sous forme d'une liste simple.

L'index d'un élément de la hash-table est calculé par la formule de hashing suivante:

$$\boxed{\text{index(p\_item)} = \text{(long)(p\_item)} \ \% \ \text{Size\_of\_Set}}$$

Le set est une structure adaptée pour le classement des vertex, puisque le nombre de vertex contenus dans un objet peut être très important(en général plusieurs milliers). Sa qualité d'hash-table permet des accès quasi-directs. Elle est efficace lors de l'ajout ou du retrait de vertex, puisqu'il n'est pas utile de parcourir l'ensemble des éléments du vset, pour trouver leurs positions. Sa qualité de liste est précieuse, lorsqu'il est nécessaire de parcourir rapidement

l'ensemble de tous les vertex d'un vset. Cette opération est fréquente lors de nombreux calculs, tel que celui des axes principaux d'un ensemble de points. Cependant, le set reste un outil général, à la disposition de nombreuses autres applications. Il représente un certain type de container, au même titre qu'un tableau ou une liste.

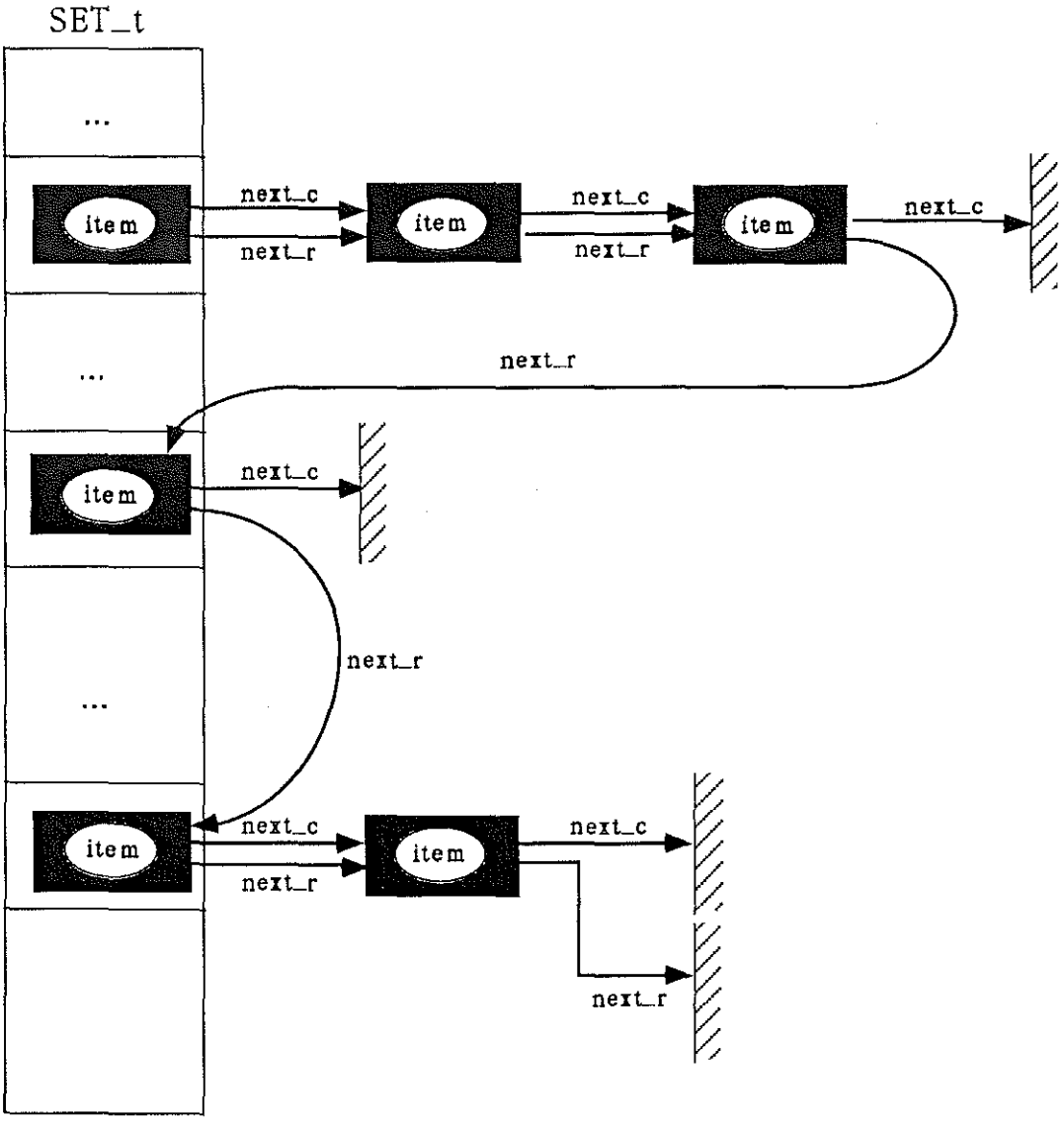

Figure 2.5 *schéma de la structure SET_t.*

2.3.3 le vport

Le **vport** est une structure destinée à contenir des sets, et elle est organisée sous forme d'une liste. Chaque élément de cette liste est identifiable par une chaîne de caractères, appelée *VportName*.

```
typedef struct VPORT_t
  {
   VPORT_t  * next      ;
   STRING_c   VportName ;
   SET_t    * p_set     ;
  } VPORT_t ;

 typedef struct VPORT_t * VPORT_c;
```

- *next* est un pointeur qui chaîne les vports sous forme de simple liste.
- *VportName* est la chaîne de caractères associée à chaque élément du vport.
- *p_set* est un pointeur sur une structure set.

Nous avions préalablement associée l'idée de vport à celle de "colis". La chaîne de caractères "VportName" représente son adresse tandis que le contenu du colis correspond à un set, c'est-à-dire à un ensemble d'entités regroupées les unes avec les autres. Le vport combiné au set, constituent des outils généraux de stockage. Leur utilisation déborde du simple cadre des échanges de vertex.

2.3.4 le vset

Le **vset** est un objet très important, car seuls les objets qui possèdent un vset sont susceptibles de s'échanger des vertex.

```
typedef struct VSET_t
  {
   int        type       ;
   STRING_c   name       ;
   long       mailbox    ;
   SETK_c     blackboard ;

   ...

   int        nb_vrtx    ;
   int        nb_importer ;
```

```
  int         nb_exporter ;
  VPORT_t   * dom_prv     ;
  VPORT_t   * dom_exp     ;
  VPORT_t   * import      ;
} VSET_t ;

typedef struct VSET_t * VSET_c;
```

Les quatre premiers champs de la structure correspondent aux champs obligatoires d'un objet du système, appelé "gobj". Nous reviendrons plus en détail sur cette nouvelle notion dans le chapitre dédié aux relations entre objets (cf page 109).

- *type* est un entier qui représente la classe du gobj.
- *name* est une chaîne de caractères, caractérisant le vset.
- *mailbox* est un entier dont l'usage ne sera pas évoqué..
- *blackboard* est un container, dont l'utilisation sera décrite ultérieurement.

Les autres champs sont spécifiques à la structure vset:

- *nb_vrtx* représente le nombre de vertex stockés.
- *nb_importer* est le nombre d'objets vers lesquel ce vset exporte des vertex.
- *nb_exporter* est le nombre de vset, qui lui exporte des vertex.
- *dom_prv* représente le vport de la partie privée du vset.
- *dom_exp* représente le vport de la partie exportée du vset.
- *import* représente le vport de la partie importée du vset.

2.4 Réflexions sur ces structures

2.4.1 les liaisons dangereuses ...

Monsieur le Marquis de Valmont, grand ami des lettres, était contrarié. Sa correspondance était si conséquente, qu'il ne savait commment la classer. Permuter deux billets pouvaient être très fâcheux !!
Son amie, Madame de Merteuil, lui vint au secours:
" Mon cher ami, ma longue expérience en ce domaine, m'apprit à classer mon courrier de la façon suivante :
Tout d'abord, procurez vous un joli secrétaire à trois tiroirs. Dans le premier, rangez toutes les lettres que vous écrivez, même celles qui n'ont pas trouvé acquéreuses !! Dans le second, répartissez dans des coffrets au nom de vos amies, une copie de toutes les lettres que vous leur avez adressées. Enfin, dans le troisième, classez de la même façon toutes leurs réponses.
Croyez-moi, cher Vicomte, il est impossible d'imaginer rangement plus agréable."
Celui-ci, en fut ravi. Les conseils de Madame de Merteuil, suivis à la lettre, étaient réellement judicieux. Pour remerciements, il se divertit, en lui communiquant les suppliques de ses soupirantes.
Ce petit jeu le perdit.
A l'aube de la mort, il eut le souci de léguer l'ensemble de sa correspondance au chevalier d'Anceny. Madame de Merteuil aussitôt avertie, voulu effacer toute trace de cet encombrant courrier. Néanmoins, elle ne parvint qu'à vider son secrétaire, et non celui du sieur d'Anceny.

Les liaisons sont parfois dangereuses ...

Maintenant, redistribuons les rôles:

acteur	objet
secrétaire	vset
tiroir	partie du vset
coffret	vport
l'adresse du destinataire	VportName du vport
le contenu de chaque coffret	le set de chaque vport
le contenu de chaque lettre	vertex

Les acteurs sont des objets qui possèdent chacun un secrétaire. Chaque secrétaire constitue un vset. Les trois tiroirs correspondent aux trois parties distinctes du vset. A l'intérieur de ceux-ci, chaque coffret est l'équivalent d'un vport dont l'attribut VportName représente l'adresse du destinataire. Le set regroupe l'ensemble des lettres contenues dans un coffret, c'est-à-dire celles qui sont adressées à la même personne. Enfin, un vertex est assimilé au contenu d'une lettre.

Ce "remake" (cf:[12]) résume les relations entre les objets. Les lettres sont créées, échangées, léguées et enfin détruites, selon des règles bien précises.

2.4.2 conventions d'utilisation

Examinons, à présent, les conventions qui régissent ces structures, en nous référant au schéma de la figure 2.6.

- Seuls les objets qui possèdent un vset peuvent importer ou exporter des vertex.
- En aucun cas un vertex ne peut être la propriété de plusieurs vsets différents. La qualité de propriétaire est unique.
- La partie propre et privée d'un vset est constituée d'un unique vport. Son "VportName" porte le même nom que le vset.
  Exemple: le champ *dom_prv* du vset *exporter* est dénommé *exportername*.
  Ce vport contient sans exception tous les vertex dont il est le propriétaire.
- La partie exportée d'un vset est constituée d'une liste de vports. Chacun de ces vports est désigné par le nom de son vset "destinataire".

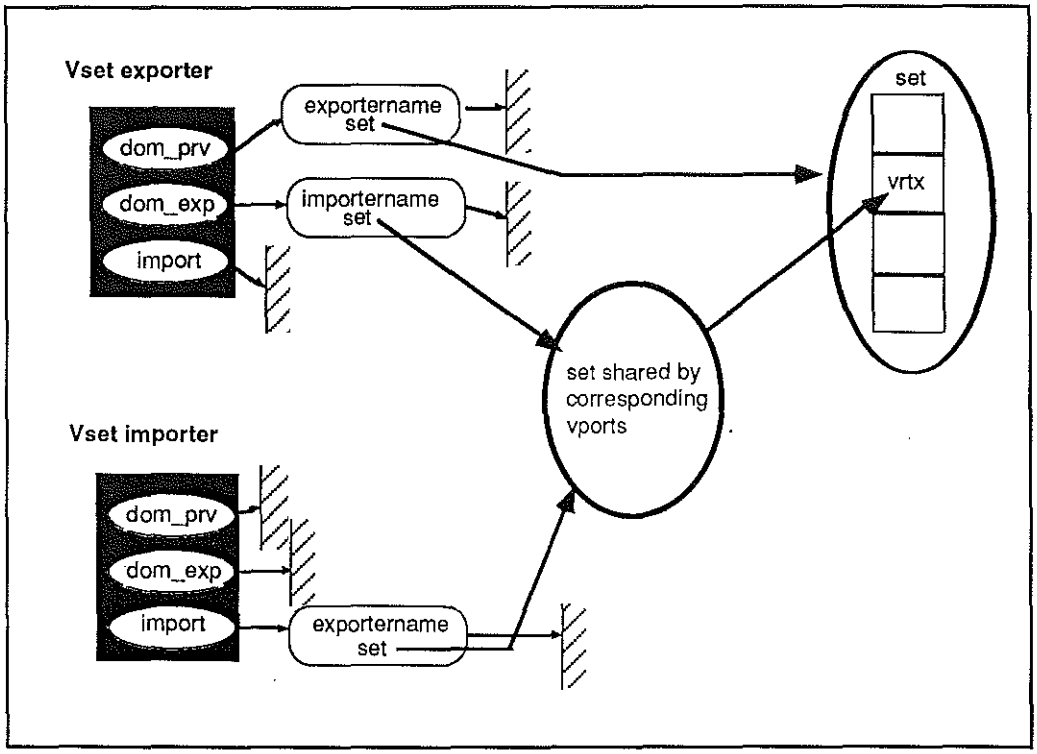

Figure 2.6 *schéma des relations entre vports; le vset "exporter"exporte le vertex "vrtx" vers le vset "importer".*

De même, dans sa partie importée, chacun des vports porte le nom du vset "origine", c'est-à-dire le propriétaire des vertex qui y sont contenus.
Exemple:
le vport de la partie exportée de *exporter*, s'appelle *importername*, et le vport de la partie importée de *importer*, s'appelle *exportername*.
Le nombre de vports contenu dans les parties exportées et importées de chaque vset, dépend évidemment du nombre de vsets différents vers lesquel il exporte ou importe des vertex.

- Lors d'un échange de vertex d'un vset A vers un vset B, les vports correspondant pointent sur la même structure de set.
  Exemple : le vport exporté de *exporter* et le vport importé de *importer*

contiennent le même set.
Ce partage correspond au fait qu'il ne peut exister d'importations sans exportations et vice et versa.

Ne confondez pas : Le set des exportations de A vers B contient la totalité des vertex exportés par A en direction de B. Il est égal au set des importations de B issus de A, mais différent de celui des importations de A issus de B.

- La structure de set interdit l'exportation d'un vertex deux fois vers le même objet. En effet, son adresse en mémoire centrale définit de façon univoque sa position dans le set. Par conséquent tout vertex ne peut en aucun cas être dupliqué dans le même set.

- Si un vertex peut être exporté plusieurs fois, un vertex importé provient d'une et unique origine, celle de son vset propriétaire. Ce vertex est stocké dans l'unique vport de la partie importée qui porte le nom de son propriétaire.

- Lorsqu'un vertex est exporté, il est contenu à la fois dans la partie privée et la partie exportée de son propriétaire. Cette solution a été adoptée en toute conscience de la redondance de l'information, afin d'optimiser le parcours de l'ensemble des vertex contenus dans un vset. Imaginons un instant, que tout vertex exporté soit stocké dans la partie exportée sans l'être dans la partie privée. Le parcours de l'ensemble des vrtx du vset nécessiterait d'inspecter, les trois sous-ensembles du vset. Cependant, le même vertex exporté vers deux objets différents serait compté deux fois, puisqu'il serait contenu dans deux vports. La solution retenue lève toute ambiguité. Désormais, le parcours des parties privées et importées suffit, puisque dans chacun de ces sets chaque vertex n'est stocké qu'une et une seule fois.

- La propriété d'un vertex est transmissible.
Exemple: A la mort du vicomte de Valmont, le Chevalier d'Anceny devint le nouveau propriétaire de sa correspondance.
Lors de la destruction d'un objet, seuls les vertex qui lui appartiennent et qui ne sont pas exportés sont libérés de mémoire centrale. Les vertex importés demeurent chez leur propriétaire. Enfin pour tout vertex exporté, le premier des vsets qui l'importe devient son nouveau propriétaire et se charge aussitôt de le distribuer aux éventuels acquéreurs.

2.5 Les fonctions utilisateurs

Les fonctions décrites concernent la création, l'exportation et la destruction d'un vertex. Elles constituent le coeur du mécanisme. Les fonctions chargées d'évaluer le degré de dépendance d'un vset ne sont pas décrites puisque le simple parcours des vports répond à la question. Néanmoins, je désire attirer l'attention du lecteur sur les choix des trois fonctions essentielles, et de leurs paramètres. Ceux-ci ôtent toute ambiguité, sur la nature et le but des fonctions. La transparence du système découle de leur simplicité et de leur unicité.

2.5.1 création d'un vertex

La fonction, nommée *VSET_Add_Vrtx()*, est l'unique "créateur" de vertex.

VRTX_t * VSET_Add_Vrtx(p_vset,x,y,z,movable)

VSET_T	**p_vset*;
float	*x, y, z*;
long	*movable*;

- *p_vset* est un pointeur sur le vset propriétaire du futur vertex.
- *x,y,z* sont les coordonnées du vertex dans l'espace.
- *movable* est un entier qui exprime le degré de liberté du vertex.

Les paramètres de cette fonction, impose à l'utilisateur de définir la position du futur vertex, ainsi que son propriétaire. La création d'un vertex isolé et sans container est donc impossible.

algorithme: *VSET_Add_Vrtx()* alloue la mémoire nécessaire pour une structure de type VRTX_t. Le pointeur ainsi créé initialise ses champs aux valeurs x, y, z, movable, transmises comme paramètres de la fonction. Ce pointeur est ensuite inséré dans le vport de la partie privée du vset *p_vset*. Si ce vport n'existe pas, il est automatiquement créé. Enfin, la fonction retourne l'adresse du pointeur alloué.

2.5.2 destruction d'un vertex

La fonction *VSET_Rem_Vrtx()* est l'unique "destructeur" de vertex.

BOOLEAN_t VSET_Rem_Vrtx(p_vset,p_vrtx)

```
VSET_T   *p_vset;
VRTX_T   *p_vrtx;
```

Le mot "destructeur" porte une signification bien précise. En effet, cette fonction ôte un vertex d'un vset plus qu'elle ne le détruit physiquement, et l'utilisateur doit impérativement spécifier de quel vset il désire supprimer le vertex.

algorithme: Suivant la nature du vset, trois cas se présentent:

1. **p_vrtx est importé par p_vset.**
   Soit *p_exporter* le vset propriétaire de *p_vrtx*. *exvport* est le vport de la partie exportée de *p_exporter* qui correspond à *imvport*, le vport de la partie importée de *p_vset*. *exvport* et *imvport* partagent le même set, *set*, qui contient *p_vrtx*. *p_vrtx* est retiré de *set*. Si *set* devient vide, alors plus aucun vertex n'est exporté par *p_exporter* en direction de *p_vset*. Dans ce cas, les vports *imvport* et *exvport* sont détruits. *p_vrtx* n'est en aucun cas libéré de mémoire centrale, puisqu'il est toujours la propriété de *p_exporter*.

2. **p_vrtx est exporté vers un ou plusieurs vsets.**
   p_vrtx est retiré du vport de la partie privée et de tous les vports de la partie exportée de p_vset. Le premier vset qui importait *p_vrtx* devient le nouveau propriétaire du vertex, et se charge de remettre à jour l'ensemble des relations avec les autres vsets importateurs. Cette opération consite à mettre à jour des vports entre ceux-ci et le nouveau propriétaire. Le vertex n'est en aucun cas détruit de mémoire centrale, puisqu'il a été transféré vers un nouveau propriétaire. La qualité de propriétaire est donc transmissible.

3. **p_vrtx est un élément strictement privé du vset.**
   le vertex est ôté du set de la partie privée de *p_vset*, puis il est détruit de mémoire centrale. Ce cas correspond à la destruction physique du vertex, puisqu'il était la propriété unique de *p_vset*. La place mémoire allouée au vertex est libérée.

2.5.3 importation & exportation de vertex

Les échanges entre vsets se résument à une unique fonction.

BOOLEAN_t VSET_Import_Vrtx(p_importer,p_vrtx)

```
VSET_T  *p_importer;
VRTX_T  *p_vrtx;
```

algorithme: *p_importer* représente le vset qui désire importer le vertex *p_vrtx*. Soit *exporter* le propriétaire de *p_vrtx*. Si *p_importer* importe déjà un vertex de *p_exporter*, alors *p_vrtx* est ajouté au set commun, contenu dans les vports respectifs des deux objets. Si *p_vrtx* est le premier vertex importé par *p_importer*, alors les vports qui établissent la communication entre les deux vsets sont préalablement créés.

2.5.4 informations concernant un vset

Les fonctions décrites succintement dans cette partie, ont pour objectif de consulter le contenu et les dépendances d'un vset. Elles ont un caractère strictement consultatif et ne jouent pas un rôle actif dans le mécanisme lui-même. Elles sont cependant, à la disposition de l'utilisateur.

```
BOOLEAN_t  VSET_Contains_Vrtx(p_vset,p_vrtx)
BOOLEAN_t  VSET_Contains_Vrtx_in_Private_Part(p_vset,p_vrtx)
BOOLEAN_t  VSET_Contains_Vrtx_in_Exported_Part(p_vset,p_vrtx)
BOOLEAN_t  VSET_Contains_Vrtx_in_Imported_Part(p_vset,p_vrtx)
```

Ces fonctions consultent les relations entre un vset et un vrtx.

```
ITERATOR_t VSET_Init_Next_VRTX(p_vset)
ITERATOR_t VSET_Init_Next_Private_Vrtx(p_vset)
ITERATOR_t VSET_Init_Next_Exported_Vrtx(p_vset)
ITERATOR_t VSET_Init_Next_Imported_Vrtx(p_vset)
```

Ces fonctions retournent la liste des vrtx répartis dans les différentes parties d'un vset.

```
ITERATOR_t VSET_Init_Next_Exporter_Vset(p_vset)
ITERATOR_t VSET_Init_Next_Importer_Vset(p_vset)
ITERATOR_t VSET_Init_Next_Associated_Vset(p_vset)
```

Ces fonctions retournent la liste des dépendances d'un vset, c'est-à-dire l'ensemble des vsets avec lesquels il importe ou exporte des vertex.

2.6 Tests de rapidité

Les tests répertoriés ci-dessous ont été réalisés sur une station de travail DEC série 5000/200.

Importation de vertex :

Les importations mettent en jeu deux vsets possédant chacun 10000 vertex. La taille de la htable du set des importations est égale à 250.

nombre de vertex	100	1000	2000	5000	8000	10000
durée en secondes	1.5	3	6.5	30	88	140

Destruction de vertex :

Les destructions ne concernent que les vertex de la partie propre d'un vset qui possède à l'origine 10000 éléments.

nombre de vertex	1000	2000	5000	8000	10000
durée en secondes	0.3	1	7	17	26

L'ordre de grandeur des temps de réponses, prouve l'interactivité du mécanisme.

2.7 Autres solutions envisagées

La conception et l'implémentation du mécanisme ne fut pas simple, et nombreuses furent les solutions envisagées. Je vous propose d'étudier deux d'entre elles, afin de comprendre pourquoi nous avons dû les abandonner.

- **la méthode du compteur**

  Le principe consiste à associer un compteur à chaque vertex. Toutes les notions de vset, vport, set disparaissent. Dès qu'un vrtx est utilisé par un nouvel objet, son compteur est incrémenté. Ainsi, le problème de la destruction d'un vertex est résolu de façon très simple. En effet, seul un vertex dont le compteur sera égale à 1, sera susceptible d'être détruit physiquement de mémoire centrale, puisqu'il n'engage dans ce cas qu'un seul et unique objet. Malheureusement, ce principe ne permet en aucun cas de définir, l'état de dépendance d'un objet. La valeur du compteur indique le nombre d'utilisateurs du vertex, mais nullement l'adresse de ceux-ci. Cette solution fut abandonnée car elle interdit toute gestion des dépendances entre objets.

- **la méthode du distributeur universel**

Cette méthode consiste à tenir à jour dans le système une table qui pour chaque vertex nous indiquera, le nombre de ses utilisateurs ainsi que leurs adresses. Cette table est maintenue au niveau d'un projet, et se charge de distribuer les vertex et de stocker leur degré d'utilisation. Cependant cette gestion soulève un premier problème lorsqu'un projet est sauvé sur disque magnétique. En effet, lors du chargement d'un projet en mémoire centrale, cette table doit être inévitablement mise automatiquement à jour et en totalité. Il est donc nécessaire de charger tous les vertex ainsi que tous les objets du projet, afin d'assurer la cohérence de la table. Cette contrainte n'est pas acceptable car elle induit un encombrement mémoire bien trop important et le plus souvent inutile. A l'inverse le mécanisme adopté propose une solution plus efficace, puisque lors du chargement d'un objet en mémoire centrale, seuls les vsets avec lesquels il partage des vertex sont chargés. Le deuxième problème concerne la gestion des dépendances des objets. Bien que la table enregistre pour chaque vertex l'ensemble de ses vsets utilisateurs, ce mécanisme n'offre pas un accès direct des dépendances entre vsets. Si dorénavant, vous désirez obtenir la liste des vsets associés à un vset donné, il est nécessaire de parcourir tous ses vertex, de stocker les vsets qui les utilisent et ensuite d'en faire le tri. La connaissance de la relation du vertex vers ses vsets utilisateurs n'est donc pas suffisante. Dans le système installé, le simple parcours des vports des parties exportées et importées suffit.

2.8 conclusion

La solution adoptée répond aux objectifs fixés. Le nombre peu élevé, le choix des paramètres et la transparence des fonctions procurent au mécanisme simplicité et robustesse. De même, l'implémentation choisie confère une grande efficacité aux opérations de parcours, d'ajout et de retrait de vertex. Seul l'encombrement mémoire des structures de set, tout en restant très raisonnable sur les stations de travail couramment utilisées, peut prêter à caution. Néanmoins ce système évite le chargement en unité centrale de la totalité des objets existant au sein d'un projet, ce qui est un gain énorme quant à l'encombrement mémoire. Il est aussi très important de souligner que le mécanisme gère durant l'exécution les dépendances entre les objets. En effet, les relations entre vsets sont mises à jour en temps réel, et en aucun cas elles ne sont fixées préalablement, ni figées. Au cours d'une session, les

échanges de vertex entres deux objets, peuvent apparaîtrent, disparaîtrent puis réapparaîtrent. En revanche, la chronologie des exportations ou importations n'est jamais prise en compte, mais cet aspect, volontairement omis, n'a pas été préjudiciable aux applications de GOCAD.

Chapitre 3

Modélisation des contraintes FCP et OTS

3.1 Introduction

L'avènement, dans le monde informatique, des outils de CAO [1] suscita dès leur apparition un formidable engouement, car pour la première fois, ces techniques permettaient de modéliser interactivement sur un écran des objets en trois dimensions. La rencontre du monde de l'image et de celui de la modélisation s'avérait enfin possible. Les applications dans des domaines aussi variés que l'ingénierie mécanique, l'industrie automobile, le secteur du batiment, la médecine et bien sûr l'industrie pétrolière furent très nombreuses. L'innovation technologique reposait sur un type particulier de fonctions mathématiques, mis au point par un ingénieur de la firme automobile Renault, puisqu'il donna son nom aux désormais célèbres "courbes ou surfaces de Bézier".

En *CAO* classique, les surfaces modélisées sont supposées admettre une représentation paramétrique $B(u,v)$ et sont interpolées par des interpolations du type *Bezier*, *B_spline*, *β_spline* ou *NURBS*. Comme l'indique la Figure 3.1, une surface de *Bezier* $B(u,v)$ est définie par une série de points de

[1] Conception Assistée par Ordinateur

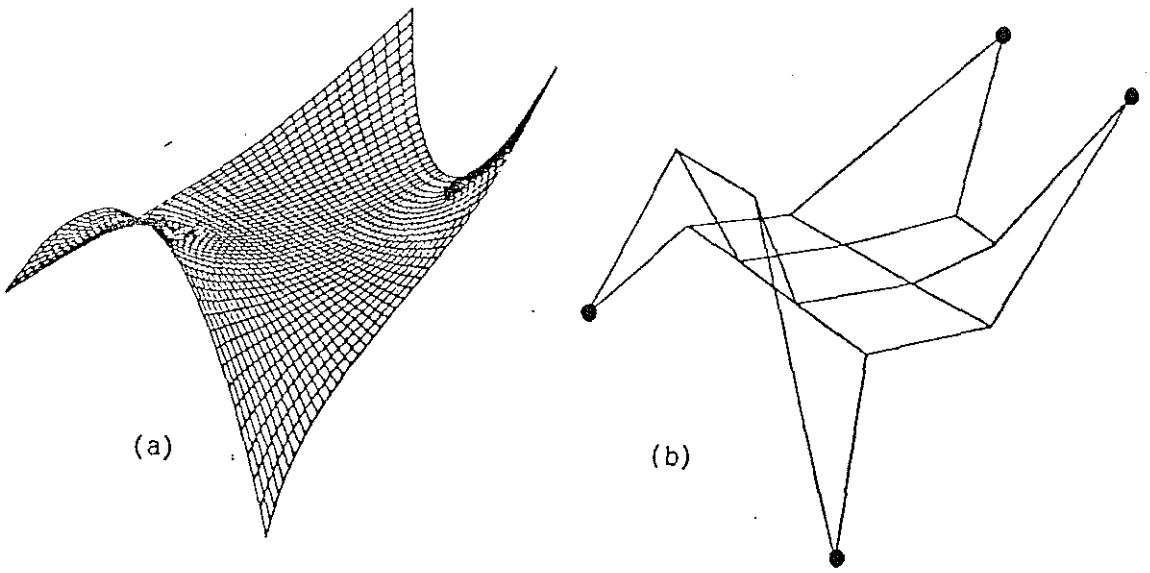

Figure 3.1 *(a) Les points de contrôle $\{p_{ij}\}$; (b) la surface de Bezier associée.*

contrôle $\{p_{ij}\}$ formant un réseau régulier rectangulaire $\{p_{11}, p_{12}, \ldots, p_{mn}\}$:

$$B(u,v) = \sum_{i=1}^{m}\sum_{j=1}^{n} p_{ij} \cdot B_{i,m}(u) \cdot B_{j,n}(v) \qquad u,\ v\ \in [0,1]$$

avec

$$\left| \begin{array}{l} B_{i,m}(u) = \frac{m!}{i!(m-i)!} u^i (1-u)^{m-1} \\ \\ B_{j,n}(v) = \frac{n!}{j!(n-j)!} v^i (1-v)^{n-1} \end{array} \right.$$

Dans cette formule, les points de contrôles $\{p_{ij}\ (0 \leq i \leq m$ et $0 \leq j \leq n\}$ sont approximés et non pas interpolés par $B(u,v)$ (sauf les points situés aux 4 coins).

Les bases mathématiques des *B_splines*, des *β_splines* ou des *NURBS* sont proches de celle de la méthode de *Bezier* si bien que les interpolateurs correspondant ont des propriétés similaires (cf:[9]). Les surfaces modélisées par ces méthodes sont généralement régulières et lisses et possèdent des propriétés très séduisantes pour les applications mécaniques (par exemple, ces surfaces sont souvent plusieurs fois dérivables et ceci est important en *CFAO*).

La figure 3.2 résume succintement le principe de manipulation de ces surfaces.

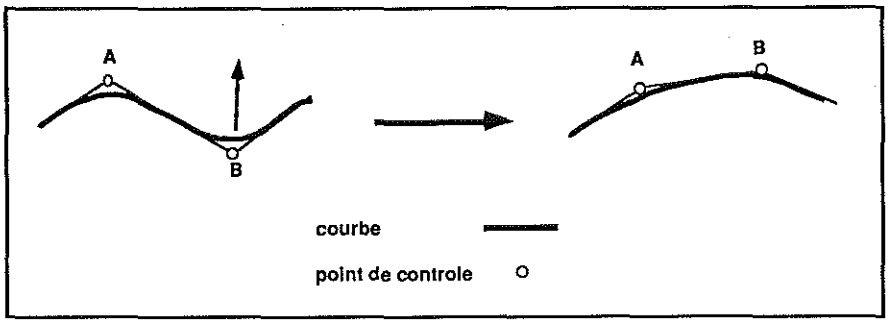

Figure 3.2 *manipulation d'une courbe de Bézier.*

Pour des raisons de simplification, nous décrivons dans le schéma ci-dessous la manipulation d'une ligne dans un espace à trois dimensions. Ce principe est néanmoins identique pour la manipulation d'une surface. Le déplacement de l'un des points de contrôle, tel B (cf figure 3.2) permet de modifier interactivement la géométrie d'une ligne ou d'une surface. L'industrie pétrolière a souhaitée aussitôt utiliser cette nouvelle technique afin de modéliser le sous-sol de bassins géologiques mais malheureusement les résultats ne furent jamais à la hauteur de son espérance. En effet, la CAO fut conçue dès l'origine pour et par l'industrie automobile, or les problèmes rencontrés dans le domaine pétrolier ne sont pas tout à fait les mêmes, ni les objectifs d'ailleurs. Citons trois différences majeures:

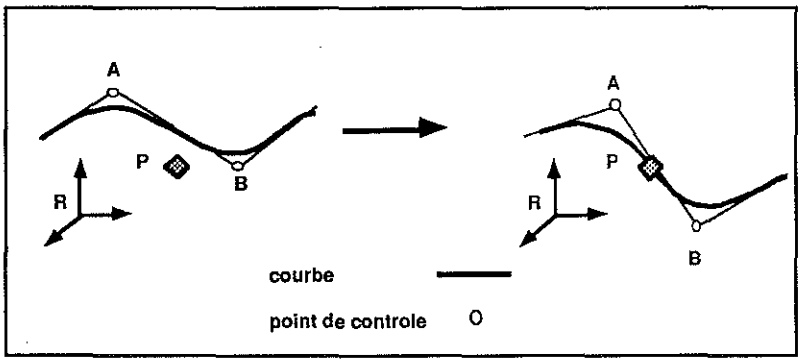

Figure 3.3 *ajustement d'une courbe de Bézier à un point.*

1. **l'ajustement aux données**
   Le principe des courbes de Bézier permet de modifier interactivement

l'allure générale d'une surface. Cependant, le problème du géologue est beaucoup plus précis puisqu'il consiste à ajuster une surface à un ensemble de données. Concrètement, comme le montre la figure 3.3, lorsque un utilisateur désire ajuster une courbe C à un point P, il doit calculer les positions adéquates des points de contrôle afin que C passe parfaitement par P. Or, aucun lien direct n'existe entre la position du point P et la position des points de contrôle. L'ajustement aux données soulève trois problèmes essentiels:

- lors de l'ajustement d'une surface à des données, le calcul de la position des points de contrôle n'est ni direct ni simple.
  En général, la surface à modéliser est connue aux points échantillonnés $\{q_1, q_2, \ldots, q_m\}$ qui sont répartis irrégulièrement sur la couche. Ce fait implique que les points de contrôle doivent être considérés comme une fonction des points connus $\{p(i,j) = D(q_1, \ldots, q_m)\}$.
- il n'y a pas en général d'unicité de la solution. En effet, dans l'exemple de la fig 3.3, plusieurs combinaisons des positions des points de contrôle sont possibles afin de respecter l'ajustement au point P.
- lors d'une étude d'un cas réel, le problème se révèle rapidement inextricable, puisque le géologue doit tenir compte de milliers de données.

2. **l'hétérogénité des données**
   Les sources des données pétrolières sont très hétérogènes, puisque le géologue doit tenir compte à la fois d'informations sismiques, de relevés de forages, ou de données de terrains. La précision de chacune de ces données est très variable et dépend des techniques et des appareils de mesure. La CAO ne peut tenir compte que de données exactes, et en aucun cas il n'est possible de mêler au sein d'une même modélisation, des données plus ou moins précises, c'est pourquoi elle ne se prête pas à la particularité des données mises à la disposition du géologue.

3. **la modification topologique**

   Lors de la modélisation d'un bassin pétrolier, le géologue tente à partir des données disponibles et de son savoir, de restituer l'ensemble des horizons existants, dans un modèle cohérent. Une des grandes difficultés rencontrées, provient du fait que la topologie des surfaces est inconnue. Par exemple, l'utilisateur doit par son expérience décider

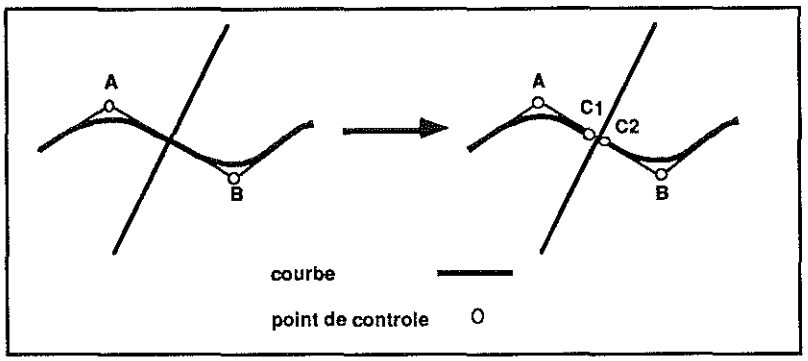

Figure 3.4 *rupture de continuité d'une courbe de Bézier.*

si tel horizon est constitué d'une couche continue, ou bien s'il s'agit de la même surface découpée en plusieurs morceaux. Les méthodes classiques s'adaptent difficilement aux modifications interactives de la topologie des surfaces modélisées. Par exemple, pour mettre à jour une faille (ajouter ou supprimer une faille, etc..), il est indispensable de reconstruire complètement le réseau rectangulaire correspondant aux "points de contrôle" p_{ij}, et ceci est incompatible avec une utilisation interactive puisque dans ce cas l'utilisateur doit reconstruire son modèle depuis le début. Comme l'illustre la figure 3.4, une rupture de continuité dans une courbe nécessite de redéfinir deux courbes indépendantes.

Je désire insister sur le fait que cette introduction n'est pas une critique des méthodes de CAO classique. En effet, le principe des courbes de Bézier correspond parfaitement aux besoins de l'industrie automobile et mécanique, il fut conçu spécialement pour résoudre leurs problèmes, et les résultats obtenus dans ces domaines furent remarquables. Néanmoins, je pense que son utilisation dans le domaine des applications pétrolières n'est pas souhaitable. Les interfaces utilisateurs des logiciels de CAO classique ne sont pas seules en cause, et il faut admettre que le principe mathématique lui-même n'est pas adapté aux besoins et caractéristiques spécifiques de la modélisation géologique.

3.2 La méthode DSI

3.2.1 présentation

Le défi du projet GOCAD consiste à proposer une nouvelle méthode de modélisation, différente des méthodes classiques, et spécialement adaptée aux problèmes de l'industrie pétrolière ou minière. Le coeur du système réside dans un nouveau principe d'interpolation, appelée DSI^2, et mis au point par le professeur Mallet (cf [15]). La fin de ce paragraphe est consacrée à sa présentation. Cette méthode est à rapprocher de la conception mathématique d'une surface et, il est important de comprendre que les surfaces que nous allons évoquer par la suite ne sont en aucun cas des grilles, des polynômes ou des fonctions. L'originalité de l'approche en découle (cf: [16]).

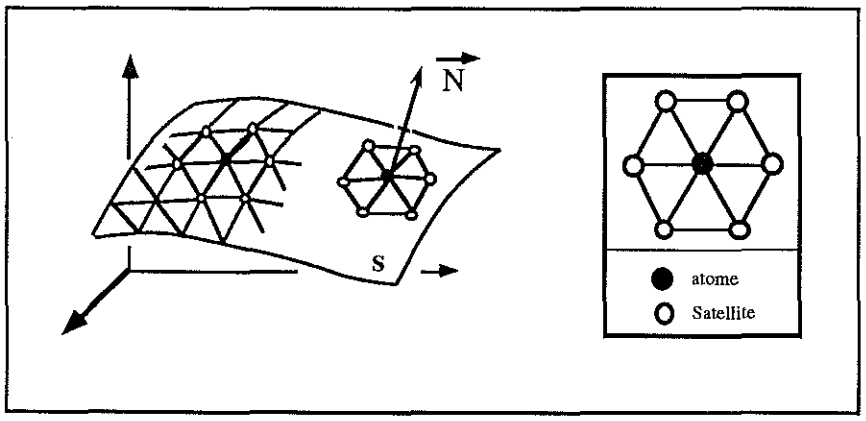

Figure 3.5 *shéma d'une surface triangulée.*

Une surface au sein du système GOCAD est un simple maillage triangulaire. La figure 3.5 illustre mon propos et nous montre qu'une surface peut être interprétée de deux façons.

1. La surface S est représentée par un assemblage de triangles connectés les uns aux autres par leurs arêtes.

2. La surface S est un réseau de points connectés les uns aux autres par des liens, c'est-à-dire que toute surface de la base de données est comparable à un réseau atomique, modélisé par un graphe planaire.

[2] Discrete Smooth Interpolation

Ces deux visions ne sont en aucune façon contradictoires, et suivant les applications que l'on désire construire, l'une ou l'autre peut être adoptée. Puisque la géométrie d'une surface est induite par la position de chacun des noeuds de son maillage dans l'espace, la seconde interprétation apparait, pour les problèmes d'ajustement, la plus judicieuse. Le principe de *DSI* consiste à minimiser sur l'ensemble d'une surface, une liste de critères qui gouvernent sa géométrie, c'est-à-dire son allure et sa position dans l'espace. Cette liste est décomposée en deux parties. La première est constituée d'un critère caractérisant la rugosité totale de la surface tandis que la seconde est une liste de critères qui représente l'ensemble des contraintes qui lui sont appliquées. Cette notion de "contraintes" correspond aux paramètres qui contrôlent la géométrie de la surface. Le résultat obtenu, est un compromis entre une surface la plus lisse possible, et une surface qui respecte les contraintes qui lui sont imposées. Toutes ces notions sont exprimées localement au niveau des atomes de la surface. Un critère de rugosité local est défini autour de chaque atome ainsi qu'un ensemble de contraintes locales. Afin de minimiser la rugosité locale, *DSI* tente par une méthode itérative de réduire globalement les écarts de distance entre un atome et le centre de son voisinage. Rappelons que le voisinage d'un atome est défini par l'ensemble de ses premiers voisins, appelés ses satellites (cf page 5). Ce comportement peut être comparé à celui d'un point relié à chacun de ses voisins par un ressort, et l'illustration "physique" de la méthode *DSI*, correspond à la minimisation de l'énergie potentielle totale de la surface. *DSI* doit de plus respecter l'ensemble des contraintes locales, appliquées sur cet atome. La contrainte la plus forte et la plus simple, consiste en l'immobilisation d'un point. Ce type de contrainte est appelée "control node" car elle fixe définitivement la position de l'atome. Cependant, l'éventail des types de contraintes installées est bien plus vaste et s'étoffe de jour en jour, car la conception et la réalisation de nouvelles contraintes, est une des orientations principales de la recherche sur *DSI*, et prouve l'étendu de son potentiel. Elles traduisent des comportements spécifiques de chaque surface, et le but de ce chapitre, est de vous présenter deux nouveaux types de contraintes appelés *FCP* (Fuzzy Control Point) et *OTS* (On Tsurf).

Revenons à présent sur la structure d'atome[3], qui est reportée à la page 5. Elle contient toute l'information nécessaire à l'interpolateur *DSI*: la position géométrique (p_vrtx), la connectivité avec les autres atomes (p_sat) et enfin la liste des contraintes (p_const), exprimées au niveau local, qu'il doit

[3] cette structure atomique fait d'ailleurs l'objet d'un brevet international dans le cadre du projet GOCAD

respecter. Le principe, l'efficacité et la rapidité de DSI sont indissociables de cette structure puisque l'accès aux informations utiles y est direct. L'exposé mathématique qui suit en apporte la preuve.

3.2.2 principe mathématique

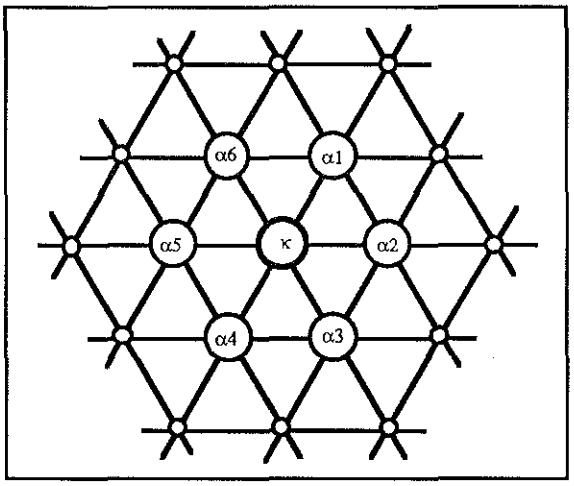

Figure 3.6 *réseau atomique d'une surface.*

Supposons que les sommets de la surface S soient numérotés de 1 à N et soit Ω l'ensemble de ces N sommets. Dans la suite, nous identifierons le sommet numéro "k" avec le "sommet k" :

$$\Omega = \{1, 2, \ldots, N\}$$

Pour un sommet quelconque "k" appartenant à Ω, nous définissons le voisinage $N(k)$ comme le sous-ensemble de Ω :

$$\alpha \in N(k) \Longleftrightarrow (\alpha, k) \text{ est le côté d'un triangle de } S \text{ ou } \alpha = k$$

le vecteur joignant l'origine de l'espace 3D et un sommet donné k est noté $\vec{\varphi_k}$ et φ est la collection de tous ces sommets :

$$\varphi = \{\vec{\varphi_1}, \vec{\varphi_2}, \ldots, \vec{\varphi_N}\}$$

le critère de rugosité: La méthode DSI est basée sur un critère de rugosité locale $R(\varphi|k)$ définie en chaque noeud $k \in \Omega$:

$$R(\varphi|k) = || \sum_{\alpha \in N(k)} v^{\alpha}(k) \cdot \vec{\varphi_{\alpha}} ||^2$$

les coefficients $\{v^{\alpha}(k)\}$ sont des pondérateurs donnés, et sont définis de la façon suivante :

$$\left[\begin{array}{lll} v^{\alpha}(k) & = & \left\{ \begin{array}{lll} -|\Lambda(k)| & \text{si} & \alpha = k \\ 1 & \text{si} & \alpha \in \Lambda(k) \end{array} \right. \\ \\ avec \quad : & & \left\{ \begin{array}{lll} \Lambda(k) & = & N(k) - \{k\} \\ |\Lambda(k)| & = & \text{le nombre d'éléments de } \Lambda(k) \end{array} \right. \end{array} \right.$$

La somme des critères locaux $R(\varphi|k)$, représente une rugosité globale notée $R(\varphi)$:

$$R(\varphi) = \sum_{k \in \Omega} R(\varphi|k)$$

les critères de contraintes: DSI suppose que chacune de ces contraintes peut être exprimée de la manière suivante :

$$A_i(\varphi) = B_i$$

Le degré de violation de la contrainte $n^o i$ peut alors être mesuré par :

$$|A_i(\varphi) - B_i|^2$$

Ce nouveau critère modifie par conséquent l'ensemble φ des solutions admissibles. Le but de la méthode DSI est désormais de rechercher l'ensemble φ des sommets qui minimise $R^*(\varphi)$ tel que :

$$R^*(\varphi) = \sum_{k \in \Omega} R(\varphi|k) + \sum_i \varpi_i^2 \cdot |A_i(\varphi) - B_i|^2$$

Les coefficients ϖ_i^2 sont appelés "facteurs de certitude", et ils sont donnés afin de moduler l'importance relative de chaque contrainte. Le facteur de certitude correspond en quelque sorte à un poids de la contrainte qui est

relatif à la rugosité totale de la surface. Puisque le résultat obtenu est un compromis entre la rugosité et les contraintes affectées, ces poids permettent d'ajuster l'importance relative de chacune.
La solution $\varphi(\alpha)$ correspond à la solution de l'équation $\partial R^{\star}(\varphi)/\partial\varphi(\alpha) = 0$, qui est satisfaite lorsque chaque composante $\varphi(\alpha)$ de φ vérifie l'équation $DSI(\alpha)$ suivante:

$$
\mathrm{DSI}(\alpha) : \left[\begin{array}{l}
\varphi(\alpha) = -\dfrac{1}{M(\alpha)} \cdot \left\{ \displaystyle\sum_{k \in N(\alpha)} \{\mu(k)\, v^{\alpha}(k) \cdot \sum_{\substack{\beta \in N(k) \\ \beta \neq \alpha}} v^{\beta}(k) \cdot \varphi(\beta)\} \right. \\
\qquad\qquad \left. + \displaystyle\sum_{i} \Gamma_i(\alpha) \right\} \\
\text{avec :} \left\{ \begin{array}{lcl}
M(\alpha) & = & \displaystyle\sum_{k \in N(\alpha)} \mu(k) \cdot (v^{\alpha}(k))^2 + \sum_{i} \gamma_i(\alpha) \\
\gamma_i(\alpha) & = & \varpi_i^2 \cdot (A_i(\alpha))^2 \\
\Gamma_i(\alpha) & = & \varpi_i^2 \cdot A_i(\alpha) \cdot \left\{ \displaystyle\sum_{\beta \neq \alpha} A_i(\beta) \cdot \varphi(\beta) - B_i \right\}
\end{array} \right.
\end{array} \right.
$$

Cette équation est appelée *la forme locale de DSI* au noeud α.

3.2.3 conclusion

La méthode *DSI* consiste donc à calculer la position des nœuds du maillage d'une surface en minimisant une somme de critères mis sous forme quadratique qui régissent sa position dans l'espace. Le premier de ces critères $R(\varphi)$ est une évaluation au sens des moindres carrés de la rugosité locale autour d'un atome. Puis, en pratique chaque information concernant la surface est traduite sous forme d'une contrainte locale que la surface doit respecter. Ces contraintes sont exprimées par une équation linéaire, et chacune est affectée d'un facteur de certitude, qui permet de gérer l'incertitude des informations. Par conséquent la méthode est parfaitement adaptée à l'hétérogénéité des données pétrolières et permet de mêler des données "exactes" et "floues". Ces contraintes sont toutes prises en compte au cours du processus d'interpolation et leur nombre est à priori illimité.

Remarquez que *DSI* sous-entend l'existence d'une solution initiale qui définit la connectivité des noeuds de la surface à approximer. On peut toutefois noter qu'il est possible de démontrer que le résultat final dépend de la connectivité de ces noeuds et non de leur position initiale; en d'autres

termes la méthode converge et procure une solution unique. Lorsqu'une faille intersecte une surface, la topologie de cette surface est modifiée, ce qui signifie que la connectivité des atomes situés le long de la ligne d'intersection change. La figure 3.7 illustre ce propos:

Après le calcul de l'intersection, l'atome A n'est plus lié directement à l'atome B, donc les voisinages de A et de B sont modifiés. Puisque A ne fait plus parti de l'ensemble des satellites de B, le critère de rugosité locale appliqué en B ne tiendra plus compte de la position de l'atome A. Ainsi toute modification topologique des surfaces se traduit uniquement par la création ou la disparition de liens entre atomes, et en aucun cas elle n'impose de redéfinir globalement le modèle.

Je désire aussi insister sur le fait que la méthode DSI est une méthode générique, et que la version utilisée n'est qu'une des interprétations possibles. En effet, un grand nombre de paramètres ont été fixés. Je citerais deux exemples :

1. *le voisinage d'un atome*
   Le voisinage d'un atome n'est pas obligatoirement fixé par l'orbite de ses premiers voisins du maillage. Il est possible de définir de différentes manières le voisinage d'un point, ceci, même en dehors de toute considération géométrique.

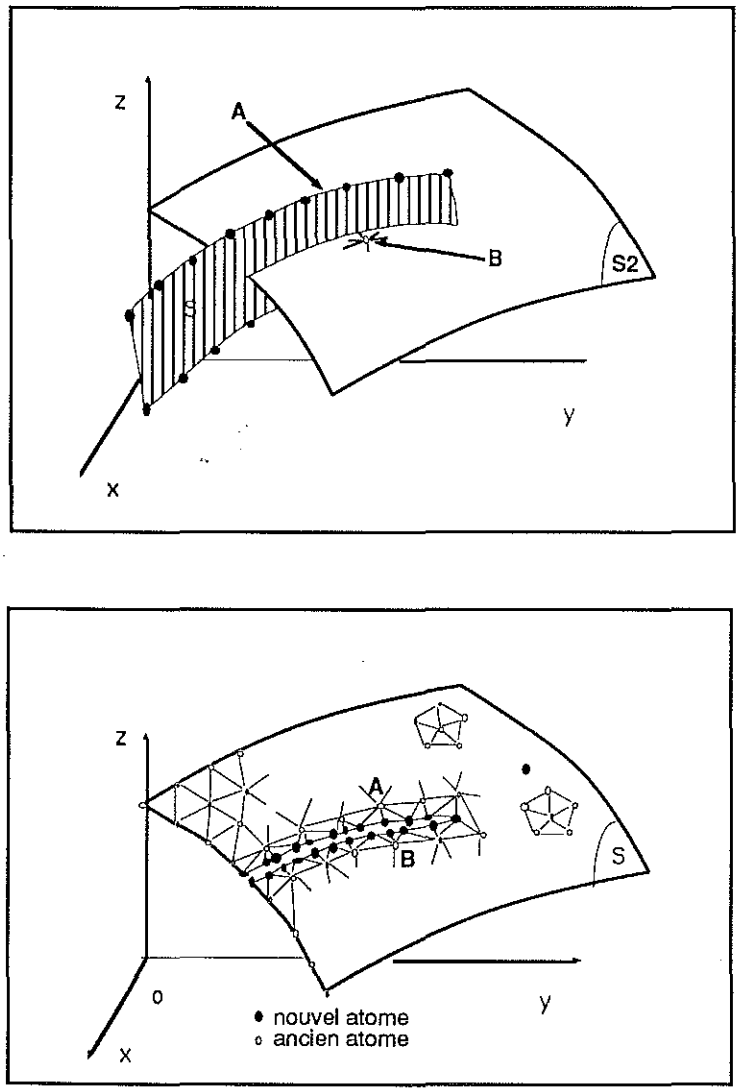

Figure 3.7 *modification de la topologie d'une surface.*

2. *les coefficients* $v^{\alpha}(k)$
   La définition présentée de ces paramètres correspond à une interprétation "harmonique" de l'orbite d'un atome. L'utilisateur peut à loisir les redéfinir. Par exemple, la méthode de krigeage, bien connu des géostatisticiens, peut être simulée par la méthode DSI, en ajustant certains paramètres.

Enfin, un des traits les plus intéressant de la méthode DSI, réside dans le fait qu'elle est transposable au problème d'interpolation d'une ligne, ou d'un volume tétrahèdrique. En fait, DSI peut être appliquée sans changement à tout graphe, qu'il soit unidimensionnel, bidimensionnel ou de dimension "n". Puisque une ligne est une succession de noeuds dont le nombre de voisins est limité à deux, DSI est à même de la lisser, dans un espace à trois dimensions.

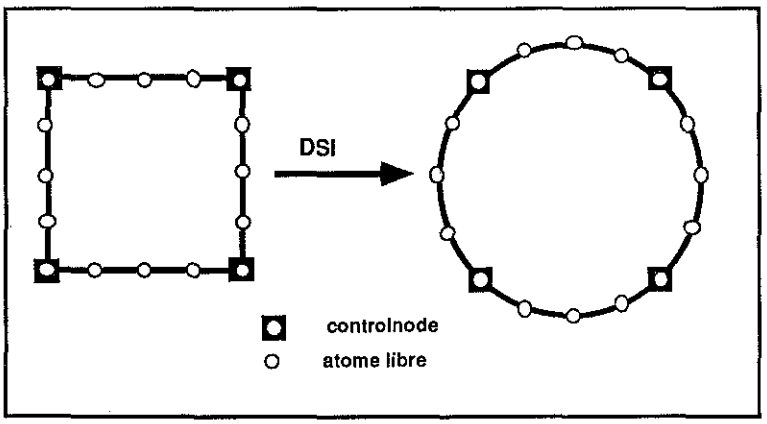

Figure 3.8 *effet de l'interpolateur DSIsur une ligne polygonale (pline).*

L'exemple de la figure 3.8 représente une ligne initialement fermée et de forme carrée, dont les quatre sommets ont été fixés comme "control nodes", c'est-à-dire qu'ils sont parfaitement immobiles. Le résultat de trente itérations fait apparaître la convergence de la ligne vers un cercle. Cet exemple très simple, démontre clairement l'effet de lissage de DSI.

3.3 La contrainte FCP

Cette section est consacrée à la présentation d'un nouveau type de contrainte appellée **fuzzy control point**. Elle répond de façon très efficace au délicat problème d'ajustement d'une surface à un ensemble de données réelles.

3.3.1 exposé du problème

Les premiers tests de la méthode DSI sur des cas réels révélèrent de graves problèmes d'ajustement. Le principe mathématique n'était pas mis en cause, mais plutôt la manière dont nous l'utilisions à l'époque. Ces tests consistaient à ajuster une surface à un ensemble de points répartis dans l'espace or, seuls deux types de contraintes étaient alors disponibles:

- *les control nodes* : un control node représente un atome parfaitement fixe. L'interpolateur ne modifie jamais ses coordonnées. C'est par exemple, le cas des quatre coins du carré de la figure 3.8.

- *les fuzzy control nodes*: un fuzzy control node est une contrainte qui fixe la position d'un atome à une incertitude près. Le point P joue un rôle d'aimant sur l'atome, c'est-à-dire que l'atome est attiré vers P sans avoir toutefois l'obligation de passer parfaitement par celui-ci. Cette contrainte est exprimée sous la forme suivante et illustrée par la figure 3.9:

  $$A_i(\varphi) = B_i \Longleftrightarrow \varphi(\alpha) = P$$

  Plusieurs contraintes de ce type, peuvent être appliquées sur le même atome α, afin que celui-ci adopte une position moyenne entre tous les points qui l'attirent.

A l'origine la méthode employée, combinait les deux contraintes de la façon suivante: Pour chaque point P du nuage des points de données, l'atome de la surface A le plus proche était sélectionné. Trois cas alors se présentaient:

1. *A est parfaitement libre*
   il est alors déplacé sur la position de P et fixé comme control node.

2. *A est un control node*
   A perd son statut de control node, mais il subit désormais l'attraction de deux fuzzy control nodes, qui sont le point P et le point correspondant à l'ancienne position de A. Ce faisant, dès que l'on applique DSI, A adopte une position moyenne entre ces deux points.

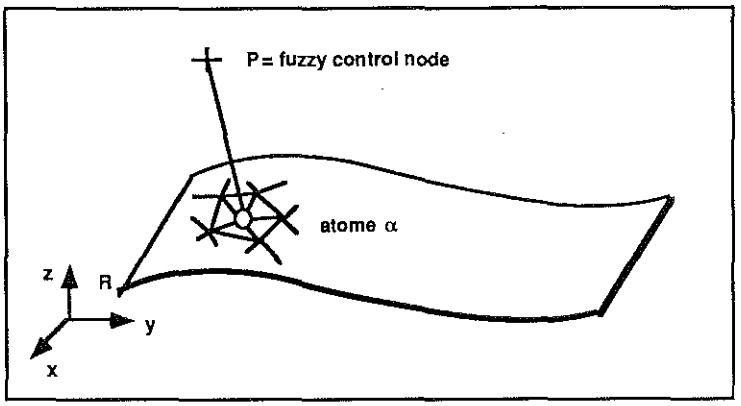

Figure 3.9 *shéma d'un fuzzy control node.*

3. *A est déjà contraint par un ou plusieurs fuzzy control nodes*
   P est transformé en un point de type fuzzy control node, et est rattaché en supplément à la liste des contraintes de l'atome A.

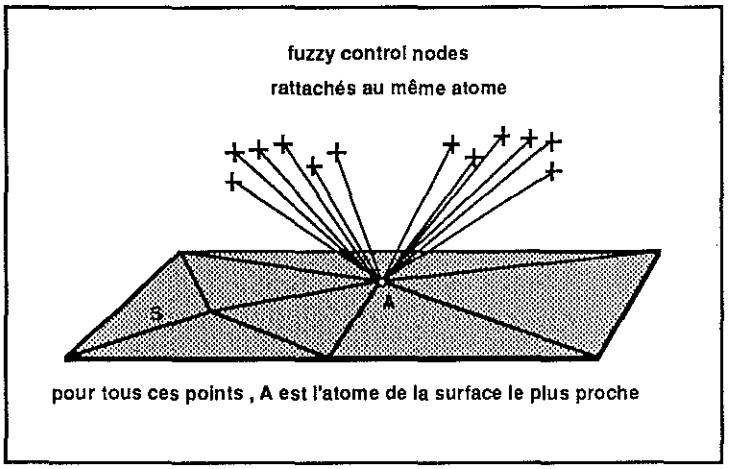

Figure 3.10 *distribution incohérente de fuzzy control nodes.*

Cette méthode s'avéra parfaitement inefficace, pour deux raisons principales. Tout d'abord, il n'y a en général aucune adéquation entre la distribution des atomes du maillage et celle des points de données. Par conséquent, comme l'illustre la figure 3.10, le regroupement de fuzzy control nodes sur un même atome peut n'avoir aucune signification. D'autre part, la liaison

entre un atome et un fuzzy control node perdure tout au long du processus d'itérations. Elle n'est jamais reévaluée en fonction de l'évolution de la surface, or il eut été parfois judicieux de supprimer un fuzzy control node d'un atome pour l'affecter à un autre. Le résultat, comme l'illustre l' image 3.11, produit des irrégularités et des enchevêtrements de triangles tout à fait inaceptable.

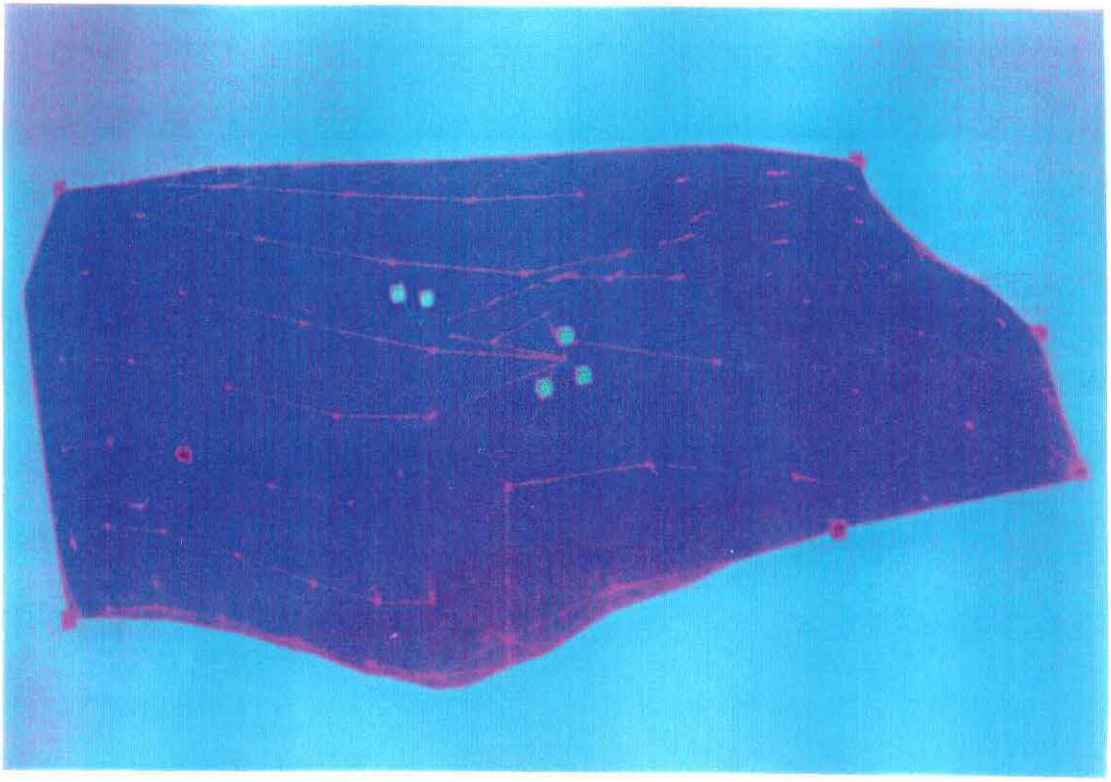

Figure 3.11 *ajustement d'une surface par l'utilisation combinée de control nodes représentés par des cubes roses et de fuzzy control nodes en vert.*

Cette voie n'était donc pas la bonne. Suivant l'intuition de André.Haas, de la société Elf-Aquitaine, il fallait reporter ce problème local à un problème plus général, puisque le but de l'opération est d'ajuster une surface dans son ensemble et non sa simple liste d'atomes. Plus précisément, chaque point de données doit attirer globalement la surface et non obligatoirement seul l'un de ses atomes. L'idée sous-jacente consiste à déconnecter les contraintes du maillage.

3.3.2 principe adopté

L'inefficacité de la solution présentée dans le paragraphe précédent provenait de la focalisation du problème sur les atomes. Elle correspondait, à l'interprétation d'une surface en un réseau atomique, et à l'importance de la notion d'atome dans la méthode *DSI*. L'idée, comme vous vous en doutez, consiste désormais à explorer la seconde interprétation d'une surface. En effet, pourquoi ne pas tenter d'attirer la surface directement par l'intermédiaire de ses triangles ?

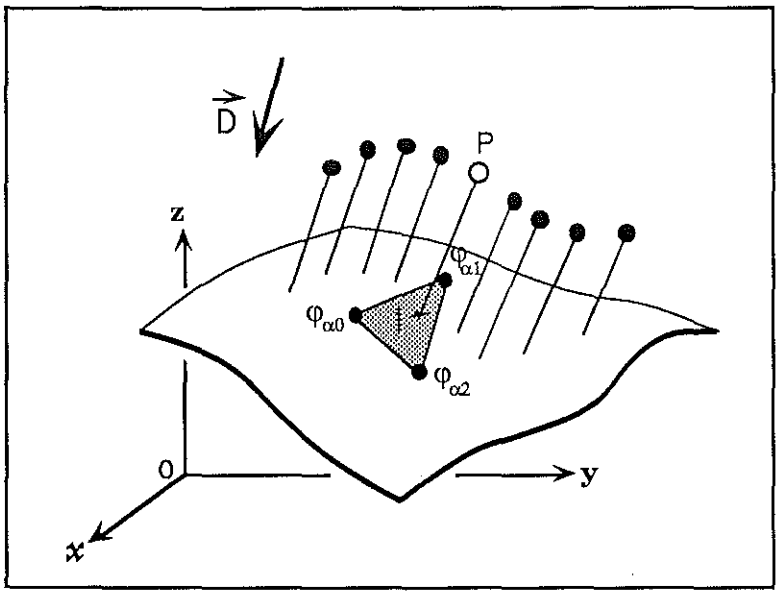

Figure 3.12 *illustration de fuzzy control points.*

Le principe consiste à associer à chaque point de données P une direction de tir, définie par un vecteur $\vec{D}$. Suivant cette direction, comme l'illustre la figure 3.12, chaque point P intercepte un triangle de la surface à ajuster, en un point d'impact I. Si aucun triangle n'est intercepté, le système considère que le point n'attire pas la surface. Notre souhait est de minimiser la distance $d(P,I)$ qui sépare P de I. Désormais P représente une nouvelle contrainte appelée **fuzzy control point**. Soient (u, v, w) les coordonnées barycentriques du point d'impact I relativement à $(\varphi(\alpha_1), \varphi(\alpha_2), \varphi(\alpha_0))$, coordonnées des trois atomes sommets du triangle. et soient (P^x, P^y, P^z) les coordonnées euclidiennes du point P. La contrainte de type fuzzy control point s'exprime de la manière suivante:

$$\forall \quad \nu=(x,y,z) \quad : \quad w\cdot\varphi^{\nu}(\alpha_0) \quad + \quad u\cdot\varphi^{\nu}(\alpha_1) \quad + \quad v\cdot\varphi^{\nu}(\alpha_2) \quad = \quad P^{\nu}$$
$$avec \quad u \quad + \quad v \quad + \quad w \quad = \quad 1$$

L'expression locale de l'équation $DSI(\ \alpha\ )$ d'une telle contrainte s'exprime alors par la formule :

$$\forall \quad \nu=(x,y,z) \quad : \quad \left[\begin{array}{l} \displaystyle\sum_{\beta\in\{\alpha_0,\alpha_1,\alpha_2\}} A_i^{\nu}(\beta)\cdot\varphi^{\nu}(\beta) \quad = \quad B_i^{\nu} \\ \text{avec :} \left|\begin{array}{l} A_i^{\nu}(\beta) \quad = \quad \left\{\begin{array}{ll} w & \text{si } \beta=\alpha_0 \\ u & \text{si } \beta=\alpha_1 \\ v & \text{si } \beta=\alpha_2 \\ 0 & \text{sinon} \end{array}\right. \\ \\ B_i^{\nu} \quad = \quad \mathbf{P}^{\nu} \end{array}\right. \end{array}\right.$$

Grâce à l'indépendance des trois expressions ci-dessus, les coeficients $\Gamma_i^{\nu}(\alpha)$ et $\gamma_i^{\nu}(\alpha)$ (cf page 44) se calculent comme suit:

$$\Gamma_i^{\nu}(\alpha) \quad = \quad \left\{\begin{array}{lll} \varpi_i^2\cdot w\cdot & \{u\cdot\varphi^{\nu}(\alpha_1)+v\cdot\varphi^{\nu}(\alpha_2)-\mathbf{P}^{\nu}\} & \text{si } \alpha=\alpha_0 \\ \varpi_i^2\cdot u\cdot & \{v\cdot\varphi^{\nu}(\alpha_1)+w\cdot\varphi^{\nu}(\alpha_0)-\mathbf{P}^{\nu}\} & \text{si } \alpha=\alpha_1 \\ \varpi_i^2\cdot v\cdot & \{w\cdot\varphi^{\nu}(\alpha_0)+u\cdot\varphi^{\nu}(\alpha_1)-\mathbf{P}^{\nu}\} & \text{si } \alpha=\alpha_2 \\ 0 & & \text{sinon} \end{array}\right.$$

$$\gamma_i^{\nu}(\alpha) \quad = \quad \left\{\begin{array}{ll} \varpi_i^2\cdot(w)^2 & \text{si } \alpha=\alpha_0 \\ \varpi_i^2\cdot(u)^2 & \text{si } \alpha=\alpha_1 \\ \varpi_i^2\cdot(v)^2 & \text{si } \alpha=\alpha_2 \\ 0 & \text{sinon} \end{array}\right.$$

La contrainte qui consiste à minimiser la distance *d(P,I)* se répartit sur les trois atomes sommets du triangle, et à chaque itération de DSI, les sommets du triangle se déplaceront de façon à réduire l'écart entre I et P. Par conséquent, la position du point d'impact sera mobile au cours du processus. En effet, au cours d'une itération, DSI tente aussi de minimiser la rugosité de la surface, donc il est tout à fait possible que le triangle intercepté à un instant donné se trouve décalé de la direction de tir à l'instant suivant. Il est donc nécessaire de s'assurer en premier lieu que le point d'impact continue d'appartenir au triangle. Ce test est simple, puisque si I appartient au triangle, alors les coordonnées (u,v,w) sont toutes les trois comprises entre 0 et 1.

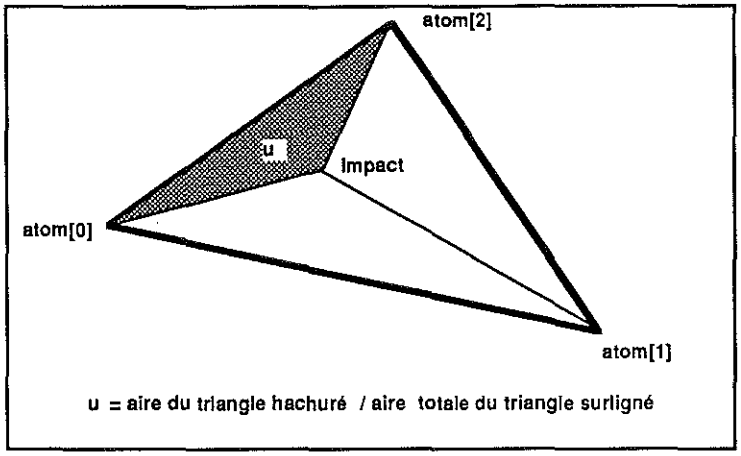

Figure 3.13 *signification des coordonnées (u,v,w).*

Ces trois coordonnées ont une interprétation physique bien précise, décrite sur la figure 3.13. Soit T le triangle intercepté sur la surface et t le triangle inscrit dans T ayant le point d'impact I pour sommet. La valeur de la coordonnée barycentrique correspond au rapport de l'aire de t sur T. Si l'une des valeurs (u,v,w) est négative, alors le point d'impact ne se situe plus à l'intérieur du triangle, mais se trouve décalé sur un triangle adjacent. Cependant la définition et les conventions qui régissent la structure *TRGL_t*, utilisée pour modéliser la notion de triangle, permettent de façon très simple, de détecter le trajet du point I sur la surface. Cette structure est reportée à la page 7.

Afin de simplifier l'écriture des programmes la notion de triangle fut dès l'origine soumise à la convention suivante, illustrée à la figure 3.14:

la position du triangle adjacent p_trgl[i] se situe du côté de l'arête opposée au sommet p_atom[i] du triangle.

Puisque les valeurs des coordonnées *(u,v,w)* dépendent de l'indiçage des sommets du triangle, il est très facile de suivre l'évolution du point d'impact *I*. Par exemple, si u est négative, alors le point *I* se situe du côté du triangle adjacent p_trgl[1]. Cette information, permet grâce à un mécanisme de boucles de retrouver le triangle réllement intercepté. Si le mécanisme boucle, alors le mécanisme tente de recalculer directement le triangle intercepté. Cette option, beaucoup plus coûteuse en temps d'exécution est décrite plus en détail dans la suite de cet exposé. Néanmoins, dans la majorité des cas,

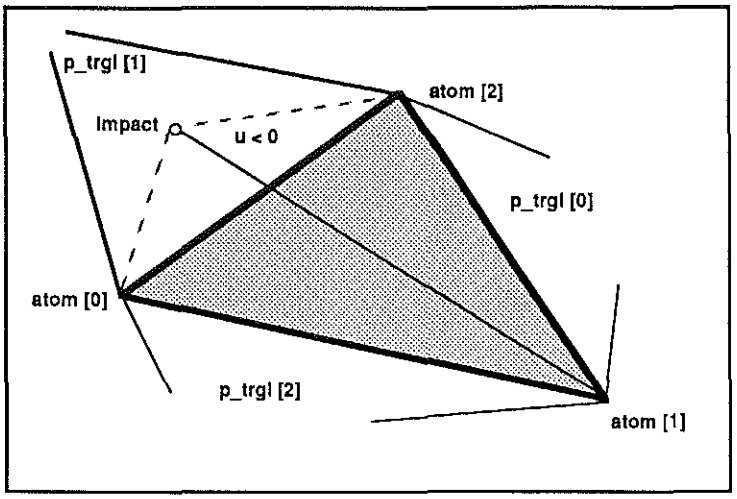

Figure 3.14 *interprétation d'une cordonnée (u,v,w) négative.*

le système est capable d'une façon très simple, de détecter le parcours du point d'impact sur la surface, et donc de décider à tout moment quel triangle doit être attiré par le fuzzy control point. Cette faculté est très importante, puisqu'elle annule de fait tout lien direct entre les données et le maillage. Une contrainte de type *FCP* [4] n'est jamais liée de façon définitive à un triangle, mais au contraire cherche toujours à attirer le triangle qui se trouve à tout instant intersecté par la droite issue du point P et parallèlle au vecteur $\vec{D}$. La rupture entre le maillage et les données est donc réalisée.

Remarque : Il ne faut pas confondre les notion de "fuzzy control node" et de "fuzzy control point": La contrainte de type fuzzy control node attire un atome d'une surface alors que la contrainte de type fuzzy control point attire globalement l'un de ses triangles.

3.3.3 description du mécanisme

L'ajustement d'une surface à un ensemble ou nuage de points, consiste, au sein du système GOCAD, en une relation entre une tsurf et un vset, de telle sorte que l'ensemble des données doit être obligatoirement un objet de type vset. Par conséquent, chacun des points de données, c'est-à-dire chaque fuzzy control point, est un vertex. Lorsque l'un de ses points attire un triangle de la surface, la contrainte est en fait répartie sur ses trois atomes sommets et les

[4] Fuzzy Control Point

trois contraintes ainsi générées ne sont pas totalement indépendantes. Elles partagent des données communes, telles que la surface attirée, la direction de tir, le facteur de certitude. Comme le suggère la figure 3.15, la contrainte fuzzy control point est donc conçue autour de deux structures.

1. La structure, dénommée *FCP_INFO_t* regroupe les informations communes.

2. la structure *FCP_t* est spécifique à chacun des atomes.

structures adoptées

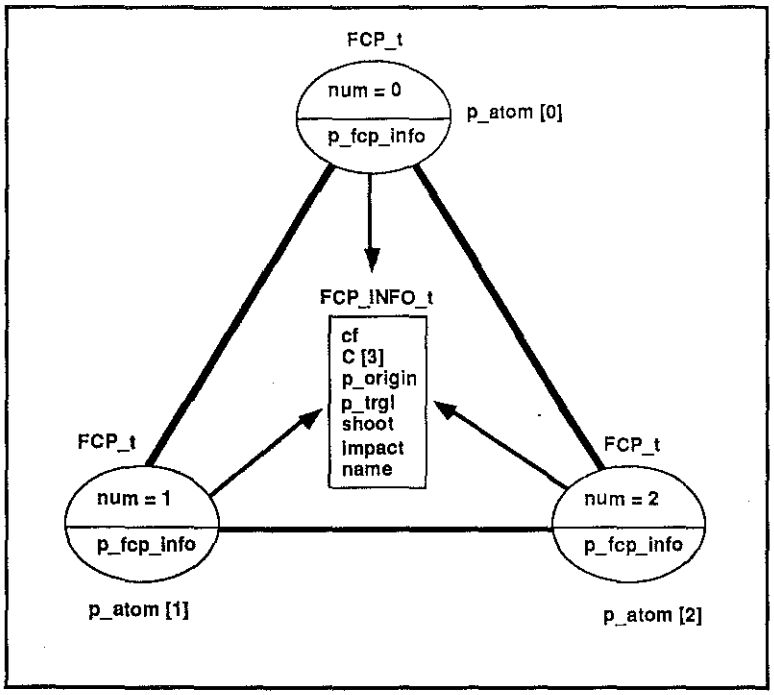

Figure 3.15 *shéma des structures d'un fuzzy control point.*

```
typedef struct  FCP_INFO_t
                {
                float       cf       ;
                float       C[3]     ;
                VRTX_t    * p_origin ;
                TRGL_t    * p_trgl   ;
                POINT3_t    shoot    ;
```

```
POINT3_t    impact   ;
} FCP_INFO_t ;
```

- *cf* est le facteur de certitude de la contrainte.
- *C* est un tableau de trois réels, qui représentent les valeurs *(u,v,w)* évoquées auparavant. Dans la suite les valeurs u,v,w seront respectivement égales à *C[1], C[2], C[0].*
- *p_origin* est un pointeur sur le vertex du vset des données, c'est-à-dire le point *P*.
- *p_trgl* est un pointeur sur le triangle intercepté sur la surface.
- *shoot* représente la direction de tir associée au vertex *p_origin*.
- *impact* représente le point d'impact sur le triangle.

```
typedef struct FCP_t
        {
        int          num        ;
        FCP_INFO_t * p_fcp_info ;
        } FCP_T ;
```

- *num* est un entier compris entre 0 et 2. Cet indice associe la valeur de de l'indice des atomes *p_atom[i]* du triangle avec les valeurs respectives de *C[i]*.
- *p_fcp_info* est un pointeur sur la structure commune aux contraintes FCP des trois atomes du triangle.

Remarque : Le nombre et le type des contraintes liées à un atome sont illimités. Par conséquent, la liste des contraintes d'un atome peut contenir plusieurs contraintes de type *FCP*.

La figure 3.16 illustre cette possibilité. L'atome *A*, sommet des triangles *T1*, *T2* et *T3*, est contraint par les fuzzy control points *P1,P2,P3* et *P4*. De même, notez que le nombre d'impacts par triangle est illimité.

initialisation de la contrainte

Lors de l'installation d'une relation de type FCP entre une surface et un vset, l'utilisateur doit définir une direction de tir et un facteur de certitude, soit globaux, soit spécifiques à chacun des vertex données. Ensuite la fonction *TSURF_Shoot()* se charge de calculer le triangle intercepté sur la surface.

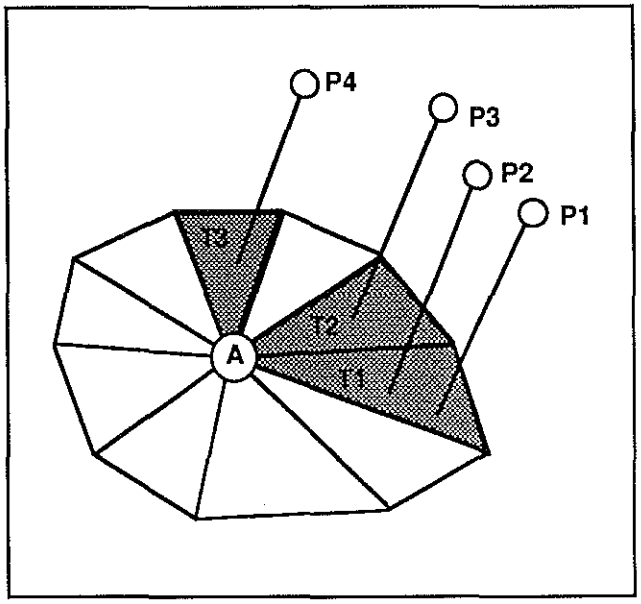

Figure 3.16 *un triangle peut être contraint par de multiples fuzzy control points.*

*TRGL_t * TSURF_Shoot(tsurf,origin,direction,impact)*

TSURF_t	**tsurf*
VRTX_t	**origin*
POINT3_t	**direction*
POINT3_t	**impact*

Cette fonction, crée par Yungao Huang lors de sa thèse au sein du groupe GOCAD(cf: [13]), calcule la liste des triangles de la surface *tsurf*, qui sont interceptés par la droite issue du vertex *origin* et parallèle au vecteur *direction*. Comme l'indique la figure 3.17, cette liste peut être constituée de plusieurs éléments. Cependant, la fonction ne retourne que le triangle situé le plus près du vertex *origin* et le pointeur de sortie *impact* permet de récupérer les coordonnées x,y,z du point d'impact. Si aucun triangle n'est intersecté, la fonction retourne un pointeur nul.

La fonction décrite ci-dessous se charge d'initialiser les contraintes sur les atomes sommets des triangles interceptés.

FCP_INFO_t TRGL__Add_FCP_on_ATOMs(trgl, origin,direction,impact,cf,vset)

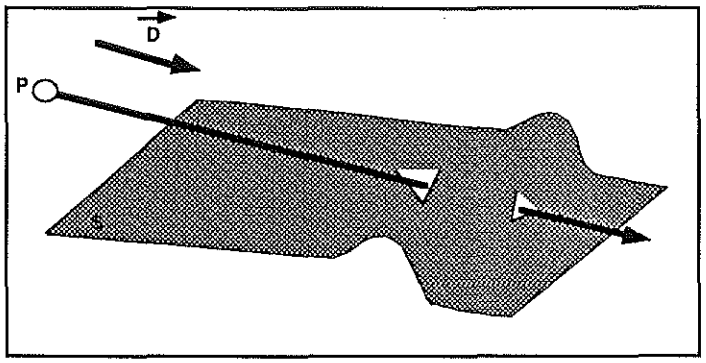

Figure 3.17 *intersection d'une droite et d'une surface.*

TRGL_t	**trgl*
VRTX_t	**origin*
POINT3_t	**direction,*
POINT3_t	**impact*
float	*cf*
VSET_t	**vset*

Cette fonction nécessite en arguments d'entrée, la quasi totalité des paramètres d'une structure de type *FCP_INFO_t*. Seules les trois valeurs des coordonnées barycentriques du point d'impact dans le repère lié au triangle sont calculées. Une instance *fcp_info* de type *FCP_INFO_t* est allouée dynamiquement et les champs de cette structure sont mis à jour. Puis, trois instances de type *FCP_t* sont allouées. L'instance dont le champ *num* est égal à *i*, est inséré dans la liste des contraintes de l'atome *p_atom[i]* du triangle. Le champs *p_fcp_info* de ses trois structures est initialisé à la même instance commune *fcp_info*. Cette dernière valeur est l'instance retournée par la fonction. En cas d'erreur un pointeur nul est retourné.

mise à jour des coordonnées barycentriques

void FCP_Compute_Values(fcp_info)

FCP_INFO_t	**fcp_info*

Cette fonction se charge de recalculer les valeurs *C[3]*, suivant l'évolution des sommets du triangle intercepté. S'il s'avère que le triangle est décalé par rapport à la direction de tir, et que de ce fait le point

d'impact enregistré ne lui appartienne plus, alors un transfert de contrainte sur le triangle adéquat est réalisé, en appelant la fonction *FCP_Shift_to_Trgl_i()*.
Le détail de la fonction est explicité ci-dessous sous forme d'un programme écrit en pseudo-langage C.

FCP_Compute_C_Values(fcp_info)

{

soient

C	= fcp_info ->C
impact	= fcp_info ->impact
trgl	= fcp_info ->p_trgl
D	= fcp_info ->shoot
et	
A0	= trgl ->p_atom[0]
A1	= trgl ->p_atom[1]
A2	= trgl ->p_atom[2]
p0	= A0 ->vrtx
p1	= A1 ->vrtx
p2	= A2 ->vrtx

$\vec{U} = \overrightarrow{p0p1}$
$\vec{V} = \overrightarrow{p0p2}$

det = produit mixte$(\vec{U}, \vec{V}, \vec{D})$

si det == 0 /* alors $\vec{D}$ est parallèle au plan $(\vec{U}, \vec{V})$ */
{
/* le point d'impact est ramené au centre du triangle */

impact.x = 0.333 * (p0.x + p1.x + p2.x)
impact.y = 0.333 * (p0.y + p1.y + p2.y)
impact.z = 0.333 * (p0.z + p1.z + p2.z)
C[0] = C[1] = C[2] = 0.333
fin
}
/* Le calcul de u , v , w, coordonnées barycentrique est effectué */

...
si (u < 0. ou v <0. ou w < 0.0)
{

```
/* les coordonnées u,v,w sont normalisées.*/

si u = minimum( u,v,w )
    {
     les coordonnées barycentriques sont
    recalculées en fonction du triangle
    trgl ->p_trgl[1]
    *
    }
si v = minimum( u,v,w )
    {
     les coordonnées barycentriques sont
    recalculées en fonction du triangle
    trgl ->p_trgl[2]
    *
    }
si w = minimum( u,v,w )
    {
     les coordonnées barycentriques sont
    recalculées en fonction du triangle
    trgl ->p_trgl[0]
    *
    }
/* ainsi les coordonnées barycentriques correspondent à l'arete
commune aux triangles trgl et trgl->p_trgl[i] */

C[0] = w ; C[1] = u ; C[2] = v ;
new_trgl = trgl->p_trgl[i] ;

si new_trgl est nul
    {
    /* le point d'impact est repositionné sur l'arete commune */

    si i = 1 alors
        {
        impact.x = w * p0.x + v * p2.x ;
        impact.y = w * p0.y + v * p2.y ;
        impact.y = w * p0.y + v * p2.y ;
        fin ;
        }

    si i == 2 alors ···
    si i == 0 alors ···
    }
```

```
        sinon
            {
            /* transfert de la contrainte sur le triangle voisin */

            FCP_Shift_to_Trgl_i(fcp_info,i);
            fin;
            }

    C[0] = w ; C[1] = u ; C[2] = v ;
    impact.x = w * p0.x + u * p1.x + v * p2.x ;
    impact.y = w * p0.y + u * p1.y + v * p2.y ;
    impact.z = w * p0.z + u * p1.z + v * p2.z ;
    fin ;
    }
```

transfert de contrainte

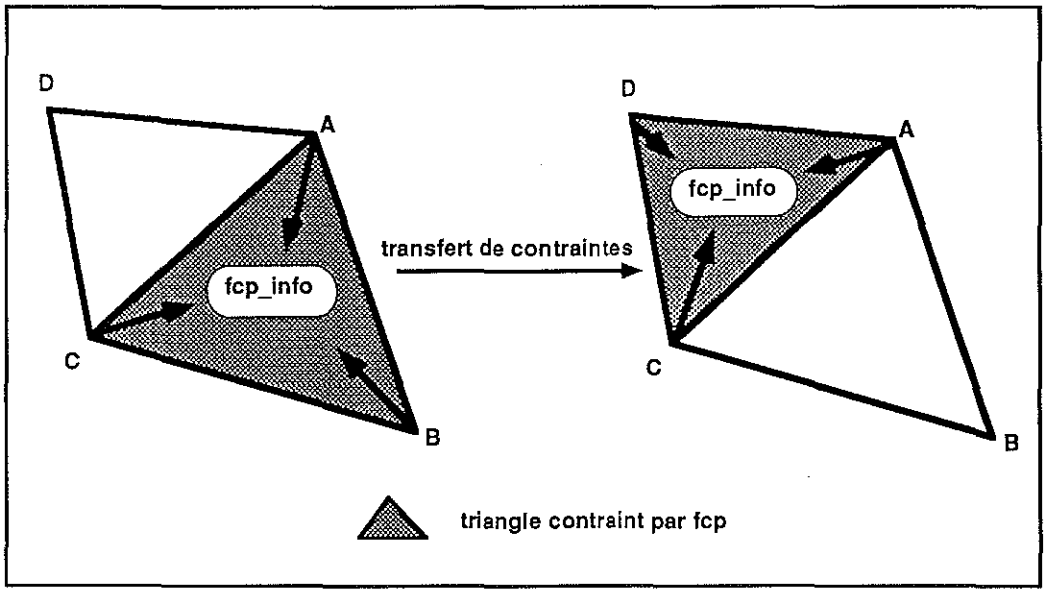

Figure 3.18 *transfert d'une contrainte FCP sur un nouveau triangle.*

void FCP_Shift_to_Trgl_i (fcp_info,i)

FCP_INFO_t	**fcp_info*
int	*i*

Soit *trgl* le pointeur sur le triangle *fcp_info->p_trgl* . Cette fonction transfère l'ensemble des contraintes de type *FCP* de ce triangle sur son triangle adjacent *trgl->p_trgl[i]*. Elle consiste d'une part à reinitialiser les champs *p_trgl* et *C* de la structure *fcp_info*. D'autre part, comme l'illustre la figure 3.18, la contrainte appliquée sur l'atome *B* doit être transférée sur l'atome *D*. Ce transfert, engendre donc une mise à jour complète des contraintes de type *FCP_t* associées aux atomes, dont notamment la valeur du paramètre *num*. Le détail de la fonction est explicité ci-dessous sous forme d'un programme écrit en pseudo-langage C.

```
void FCP_Shift_to_Trgl_i( fcp_info , i )
    {
          | old_trgl    = fcp_info ->p_trgl
          | new_trgl    = old_trgl ->p_trgl[i]
    soient| old_atom    = old_trgl ->p_atom[i]
          | C           = fcp_info ->C
          | impact      = fcp_info ->impact

    /* calcul de l'atom de new_trgl qui auquel est transféré la contrainte */

    old_atom1 = old_trgl ->p_atom[ (i+1) % 3 ];
    old_atom2 = old_trgl ->p_atom[ (i+2) % 3 ];
    pour num de 0 à 3
       {

       new_atom = new_trgl ->p_atom[num] ;
       si ( new_atom ≠ old_atom1 et new_atom ≠ old_atom2 )
       alors stop;
       num = num +1

       }

    /* ainsi l'indice "num" de new_atom dans new_trgl est calculé. */

    /* mise à jour des coordonnées barycentriques */

    C_new[0] = C[i] ;
```

```
C_new[1] = C[(i+1)%3] ;
C_new[2] = C[(i+2)%3] ;

C[ num ] = C_new[0] ;
si ( new_trgl ->p_atom[(num+1)%3] == old_atom1 )
   {

C[(num+1)%3] = C_new[1] ;
C[(num+2)%3] = C_new[2] ;

   }
sinon
   {

C[(num+1)%3] = C_new[2] ;
C[(num+2)%3] = C_new[1] ;

   }

/* calcul du point d'impact */

soient | atom_tab = new_trgl ->p_atom
       | p0       = atom_tab[0] ->p_vrtx
       | p1       = atom_tab[1] ->p_vrtx
       | p2       = atom_tab[2] ->p_vrtx

impact.x = C[0] * p0->x + C[1] * p1->x + C[2] * p2->x
impact.y = C[0] * p0->y + C[1] * p1->y + C[2] * p2->y
impact.x = C[0] * p0->z + C[1] * p1->z + C[2] * p2->z

/* la contrainte de type FCP_t est détruite de la liste de
contrainte de old_atom et rajoutée à celle de new_atom */

Translate_FCP_from_Old_to_New (old_atom,fcp_info);

/* enfin mise à jour des index num dans ces contraintes.
En effet num doit correspondre à l'indice de l'atome dans new_trgl*/

Update_num_Value( atom_tab , fcp_info );
fin;
}
```

3.3.4 notion de fuzzy control line

Le type de contrainte "Fuzzy Control Point" permet de définir une relation entre une surface et un ensemble de points de données distribués dans l'espace. Cependant chacune des relations $(point, surface)$ est traitée de façon individuelle et indépendante, par la méthode DSI. Supposons, comme l'illustre la figure 3.12 que l'ensemble des données correspond à la trace d'une ligne dans l'espace. Lors de l'installation d'une contrainte de type fuzzy control point, il est nécessaire de définir une direction de tir pour chaque point, ou d'accorder la même direction de tir pour l'ensemble de ces points. Afin d'améliorer la convivialité du système, nous avons élaboré une solution qui permette de définir interactivement chacune de ces directions de tir. En effet, si les points de données correspondent aux vertex des atomes d'une pline, alors il est possible d'utiliser la connectivité de ses atomes afin d'interpoler au sens de DSI les directions de tir. Ce type de comportement qui relie une pline à une tsurf est appelé **Fuzzy Control Line**.

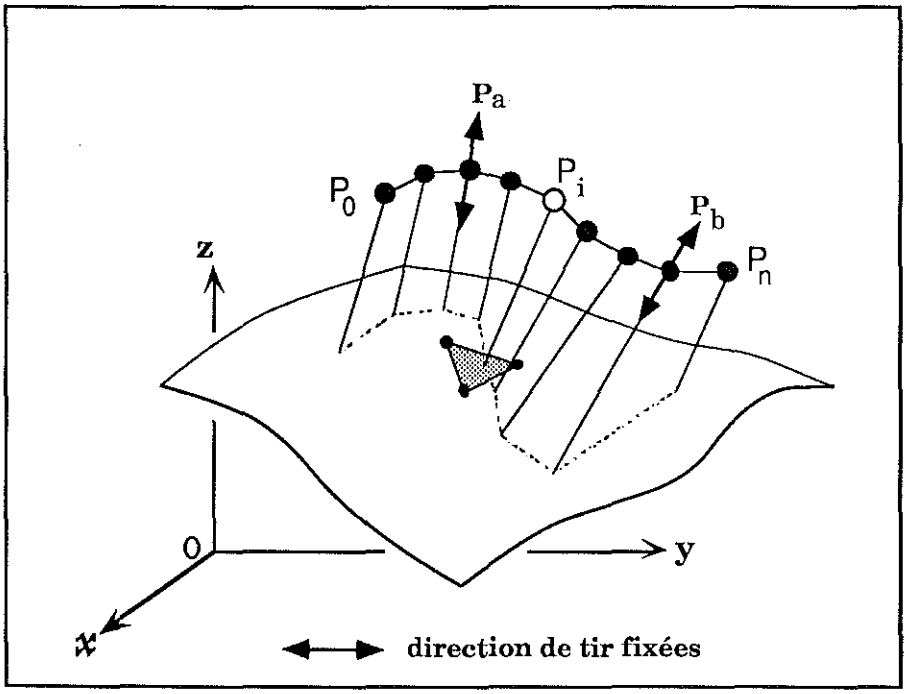

Figure 3.19 *schéma d'une d'un fuzzy control line.*

Reportons nous à la figure 3.19, afin de suivre en détail le principe de

cette interpolation. Souvenez vous que *DSI* peut être appliqué au lissage d'une pline (cf page 47). Le critère de rugosité locale tente de minimiser les écarts entre un atome de la pline et ses voisins, tout en respectant les points définitivement fixés en tant que control node. Supposons désormais que les directions de tir des points *Pa* et *Pb*, atomes de la ligne polygonale *pline* soient fixées. De la même façon, la méthode *DSI* peut, en s'appuyant sur le voisinage de chaque atome de la pline, calculer la position des directions de tirs qui leur sont associées. Ce calcul correspond à la minimisation des écarts relatifs entre des directions de tirs voisines. Les directions de tir fixées jouent le rôle de control nodes, tandis que les directions de tir calculées correspondent à la position des atomes libres de la pline.

Remarque : Créer une fuzzy control line ne consiste pas à créer une pline, mais consiste à initialiser un comportement de type *FCP* entre les atomes d'une pline et une surface. De même, détruire une fuzzy control line, correspond à la rupture de ce comportement entre les deux objets, et non en la destruction physique de la pline.

Je souligne qu'une contrainte de type fuzzy control point et une contrainte de type fuzzy control line sont parfaitement identiques du point de vue de l'ajustement par la méthode *DSI*. Cependant, l'apparition des fuzzy control lines améliore, pour deux raisons essentielles, la convivialité et l'efficacité du mécanisme.

1. Une fuzzy control line est par nature une pline, qui est très facilement modifiable de façon interactive. Ses possibilités sont nombreuses: déplacement d'un point de la ligne, translation globale de la ligne, lissage de la ligne par *DSI*$\cdots$. Une fuzzy control line acquiert par conséquent toute les facilités de modélisation d'une pline, qui permettent de redéfinir la position des contraintes de type FCP qui gouvernent la géométrie de la surface.

2. Désormais, la possibilité d'ajuster interactivement les directions de tir des contraintes de type fuzzy control line, en les interpolant au sens de *DSI*, procure une grande souplesse d'utilisation.

L'extension de la notion de fuzzy control point à celle de fuzzy control line permet à l'utilisateur de tenir compte de l'une de ses sources de données les plus importantes. En effet, l'interprétation sismique de cross-sections définit des lignes d'horizons dans l'espace, qui peuvent être ainsi intégrées

dans le processus de modélisation des couches géologiques.

Remarque : La tentation d'établir un parallèle entre les contraintes de type FCP et les points de contrôle définis par Bézier est grande. Bien que, le déplacement de ces points permette dans les deux cas de contrôler la géométrie de la surface, de grandes différences existent:

1. Les points supportant les contraintes de type FCP n'ont aucune relation directe avec les surfaces, car, contrairement aux points de Bézier, la topologie des surfaces ne repose en aucune façon sur leur existence. Il est possible, de créer ou de détruire interactivement autant de fuzzy control lines désirées sur une même surface, sans pour cela remettre en cause son existence.

2. Les contraintes de type FCP sont associées à un facteur de certitude. Cette possiblité qui n'est pas offerte par les méthodes classiques de type Bézier, offre à l'utilisateur la possibilité de gérer l'incertitude relative de ses données dans le processus de modélisation d'une surface.

3.3.5 résultats obtenus

ajustement d'une surface à un nuage de points

Les données représentées à la figure 3.20, représentent un ensemble de 3000 points réparties dans l'espace. Cet exemple est un cas réel qui nous a été fourni par la société Elf-Aquitaine, afin de tester la robutesse et l'efficacité de la nouvelle méthode d'ajustement liée aux contraintes fuzzy control point. Elles correspondent à des courbes de niveau. Le but de cet exercice consiste à reconstituer une surface initiale, à l'ajuster aux données, puis recalculer les courbes de niveau sur cette surface, afin de les comparer aux données initiales.

initialisations : Dans un premier temps, l'enveloppe convexe de tous les points de données, projetés sur le plan (OX,OY) est calculée. Ensuite, l'utilisation de l'un des algorithmes de triangulation, développé par Yveline Chipot durant sa thèse au sein du groupe GOCAD(cf :[5]), permet de reconstituer une surface triangulée, dont la frontière correspond à cette enveloppe convexe. Cette opération nous restitue une surface plane, parallèle aux axes OX et OY. Avant d'installer les contraintes, il est utile d'ajuster

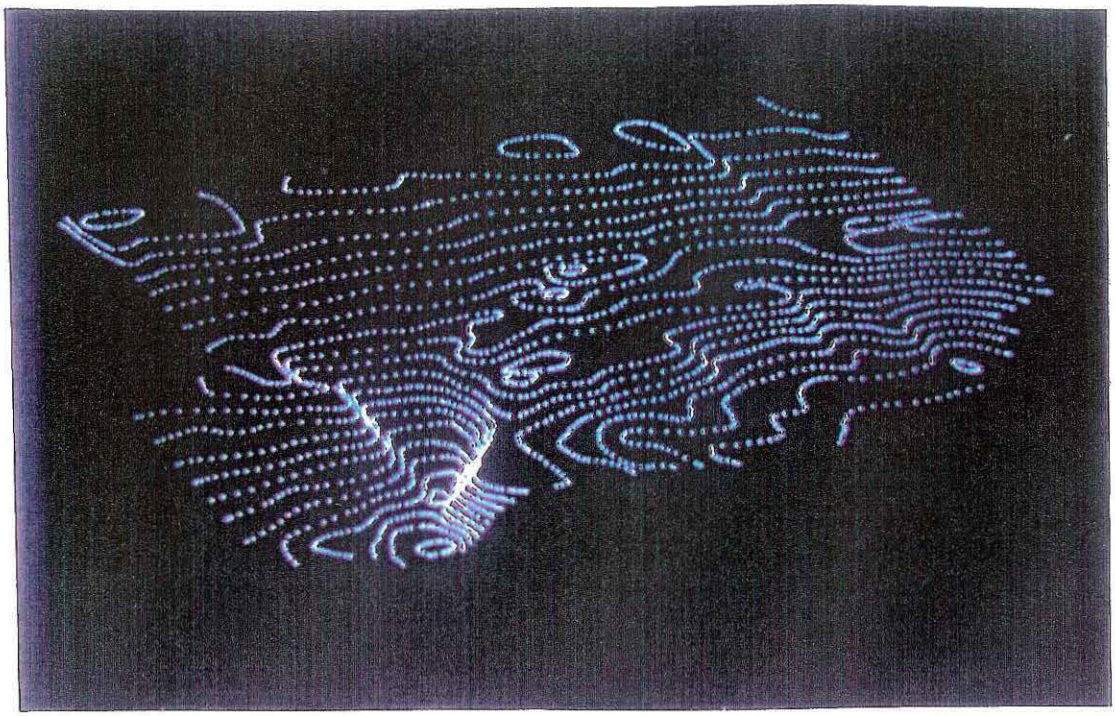

Figure 3.20 *Visualisation des données initiales, qui correspondent à des courbes de niveaux.*

le nombre de triangles de la surface en fonction du nombre de données. En effet, une surface plane consituée de deux triangles ne présente pas la même flexibilité, ou degré de liberté qu'une surface de deux mille triangles. L'opérateur "split" conçu toujours par Yveline Chipot, subdivise chaque triangle en quatre triangles internes, et il peut être appelé itérativement afin d'ajuster le nombre des triangles de la surface. Ensuite, chaque point de données est transformé en fuzzy control point. Toute ces contraintes, ont une direction de tir paralèlle à l'axe OZ et un facteur de certitude égale à 1.

résultat de l'interpolation : Ce test n'est pas du tout trivial. En effet, d'une part les points de données ne sont pas distribués aléatoirement dans l'espace, mais sont alignés le long de courbes de niveau, d'autre part l'exemple présente de forte ruptures de pente. Les sources d'instabilité numérique sont donc nombreuses.

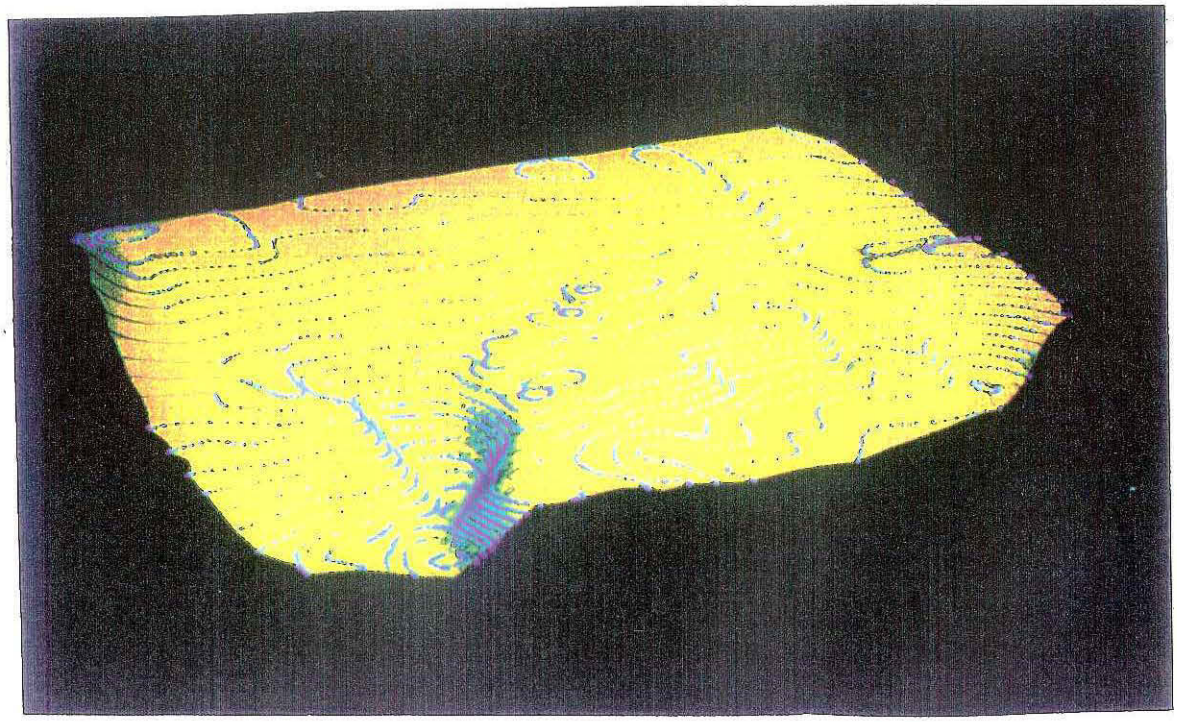

Figure 3.21 *Résultat de l'interpolation: Les courbes de niveaux recalculées sur la surface sont dessinées en noire et elle s'ajustent parfaitement aux données initiales représentées en bleu.*

Le résultat nous apparaît très satisfaisant pour trois raisons:

1. *l'ajustement aux données.*
   La comparaison des lignes de niveaux calculées sur la surface obtenue, par rapport aux données initiales est quasi parfaite, même dans la région de forte pente. Seuls de petits décalages sont observés dans les régions presque planes, ou les régions où le nombre de triangles est insuffisant par rapport au nombre de points de données. Cet exemple prouve que la méthode est robuste et parfaitement adaptée aux simulations réelles, puisque l'ensemble des données est constituée d'une somme non négligeable de trois mille points.

2. *la stabilité numérique*
   Habituellement, les méthodes classiques d'ajustement (par ex : Krigeage) deviennent numériquement instables lorsque les données sont fortement "clustérisées". On notera que ce n'est pas le cas de DSI .

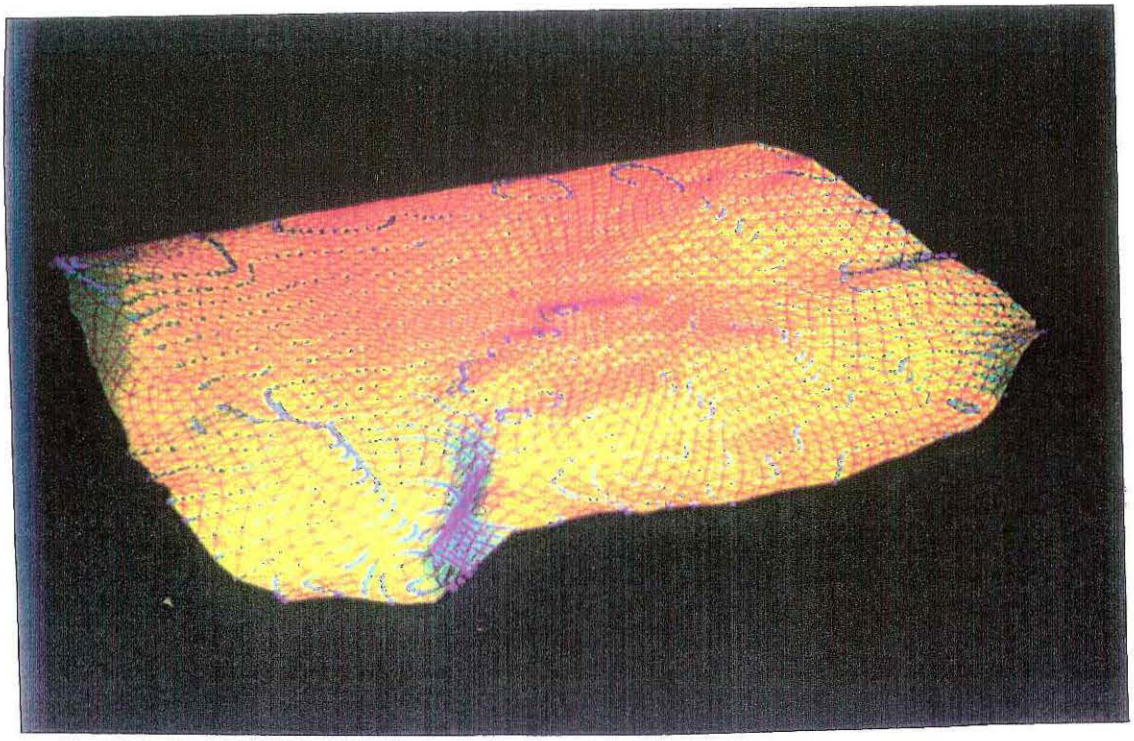

Figure 3.22 *Le maillage de la surface est représenté en rouge, et on peut constater qu'il conserve après l'interpolation une étonnante régularité.*

3. *la rapidité de l'interpolation*
   La convergence de la surface est obtenue au bout d'une vingtaine d'itérations de l'interpolateur *DSI* . L'opération nécessite moins de 5 secondes sur une station de travail DEC série 5000/200. La méthode est donc tout à fait rapide.

4. *la qualité du maillage obtenu*
   La figure 3.22 représente le maillage de la surface aprés l'ajustement. Sa régularité est essentiellement due au principe de transfert de la contrainte *FCP* d'un triangle à un autre au cours des itérations. En effet, ce principe correspond en quelque sorte à un mécanisme de relaxation automatique des contraintes. Cette relaxation est une des raisons essentielles de l'efficacité de la méthode et de la qualité des résultats.Il est aussi très important de noter que ce type de contraintes est tout à fait adapter aux discontinuités topologiques, puisque que la surface proposée peut être faillée ou découpée plusieurs fois, sans que celà

n'affecte le mécanisme d'interpolation. Cette notion est représentée à la figure 3.47.

contrôle de la géométrie d'une surface par une ligne

Cet exemple illustre les capacités d'une fuzzy control line pour gouverner l'allure d'une surface. La surface représentée est un dôme de sel. Quatre

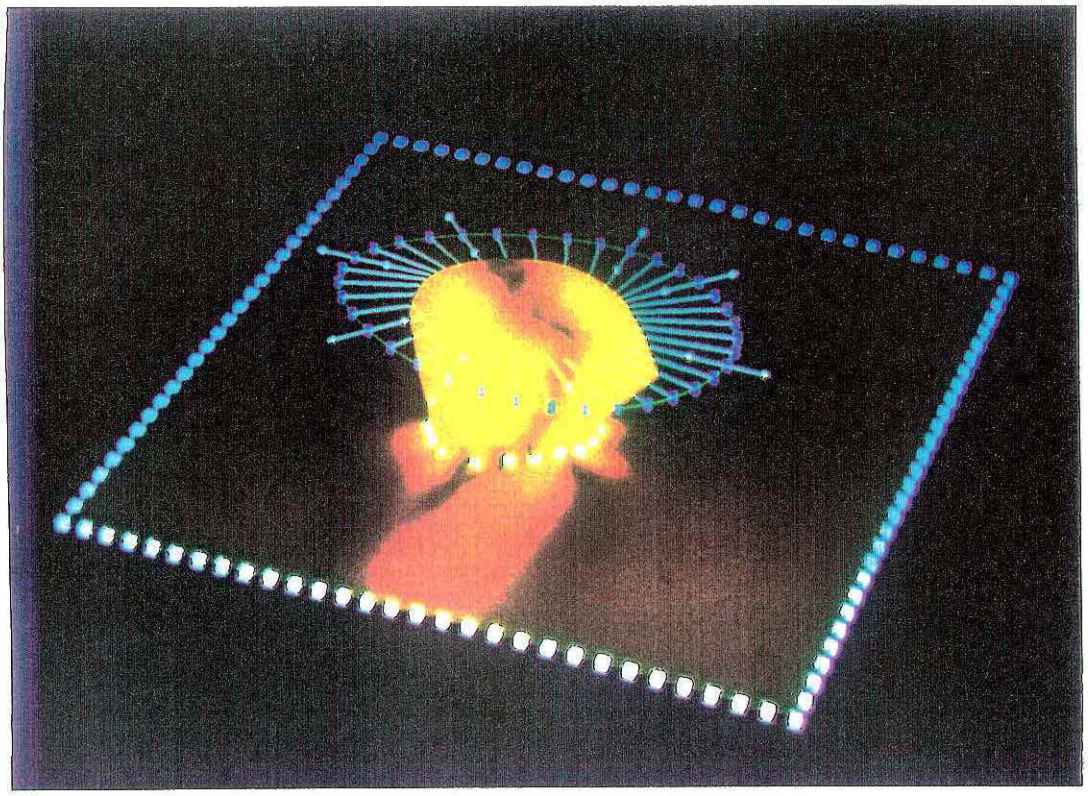

Figure 3.23 *Représentation d'une fuzzy control line: Les flèches dessinées en blanc représentent les directions de tirs fixées, qui ont permis d'interpoler l'ensemble des autres directions de tirs. Chaque atome de la ligne, dessinées par un sablier bleu, est ainsi associé à une contrainte. Les cubes gris clair constituent l'ensemble des controle nodes de la surface.*

directions de tirs ont été fixées, et l'ensemble des autres directions de tirs

sont interpolées, de telle façon qu'elles soient toutes orientées vers le centre du sommet du dôme. Chaque contrainte est affectée du même facteur de certitude. Le résultat de dix itérations est représenté à la figure 3.24.

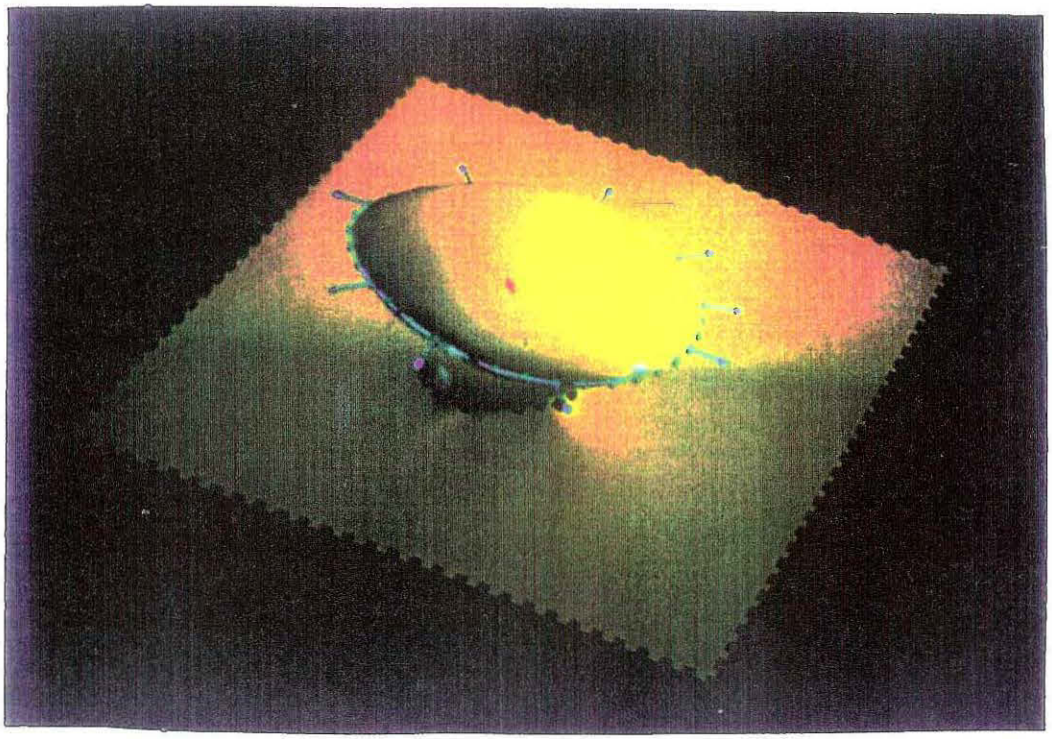

Figure 3.24 *Résultat après l'interpolation: La partie supérieure du dôme de sel vient s'ajuster sur la ligne de contraintes.*

La surface vient s'ajuster au contour de la fuzzy control line. La notion de fuzzy control line joue donc parfaitement son rôle, d'ajustement. Comme l'illustre la figure 3.25, l'utilisateur peut définir autant de fuzzy control line qui le désire sur une même surface. Leur orientation est tout à fait libre, et elles ont la possibilité de s'intersecter. D'autre part, une observation plus détaillée, du pourtour inférieur du dôme de sel, ne laisse apparaitre aucune instabilité numérique de type "pic de Gibbs" (cf figure 3.27).

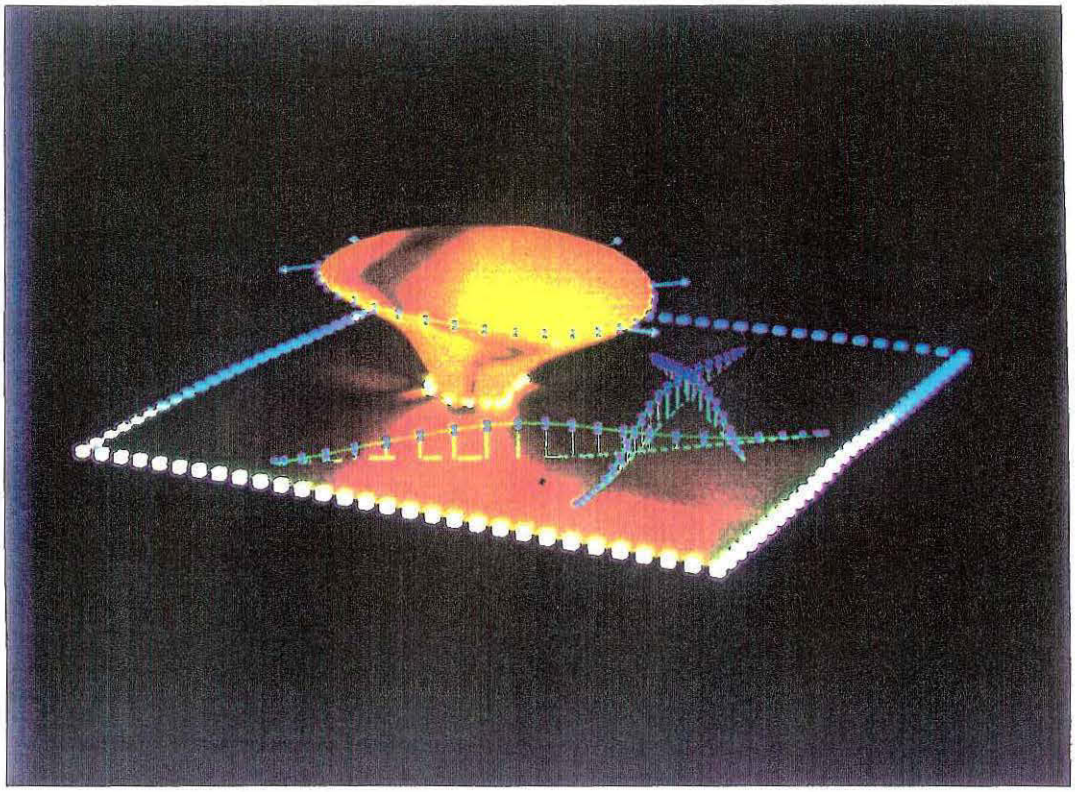

Figure 3.25 *La surface est désormais contraintes par trois fuzzy control lines supplémentaires, qui ont la particularité de se croiser. Toutes les directions de tirs sont parallèles à la direction z.*

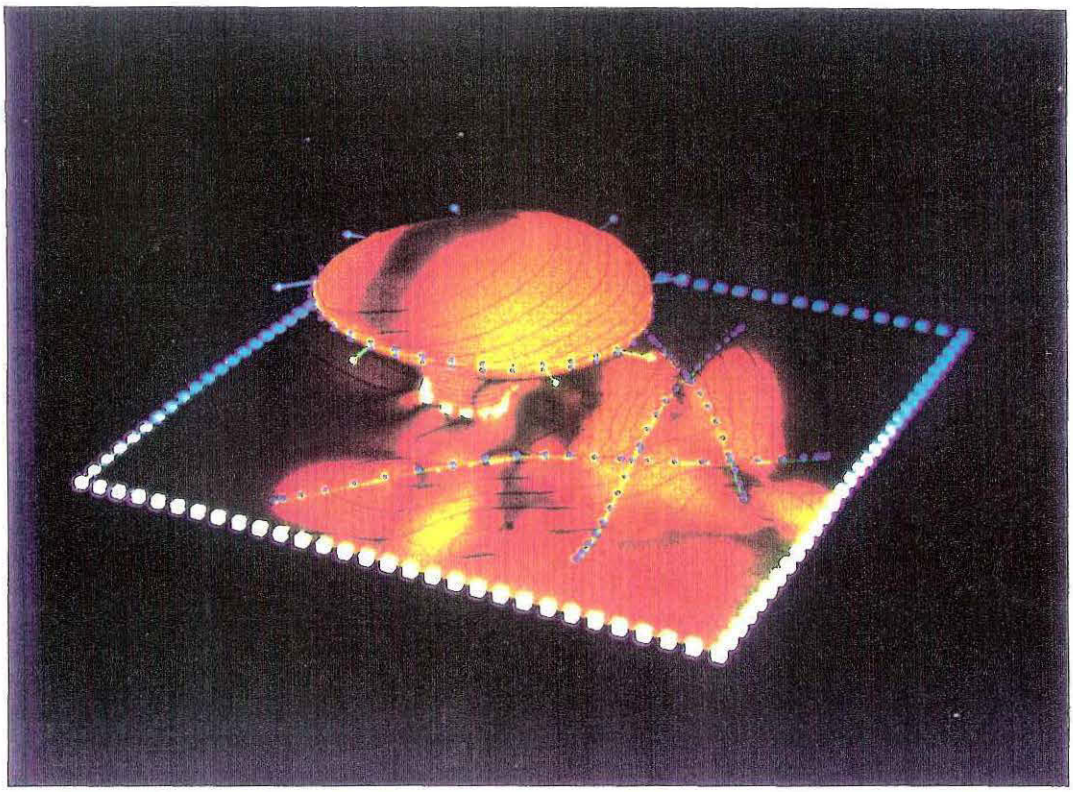

Figure 3.26 *Résultat de l'interpolation: Le croisement des lignes ne présente aucune difficulté, comme le prouve les courbes de niveaux qui ont été calculées et dessinées en noires.*

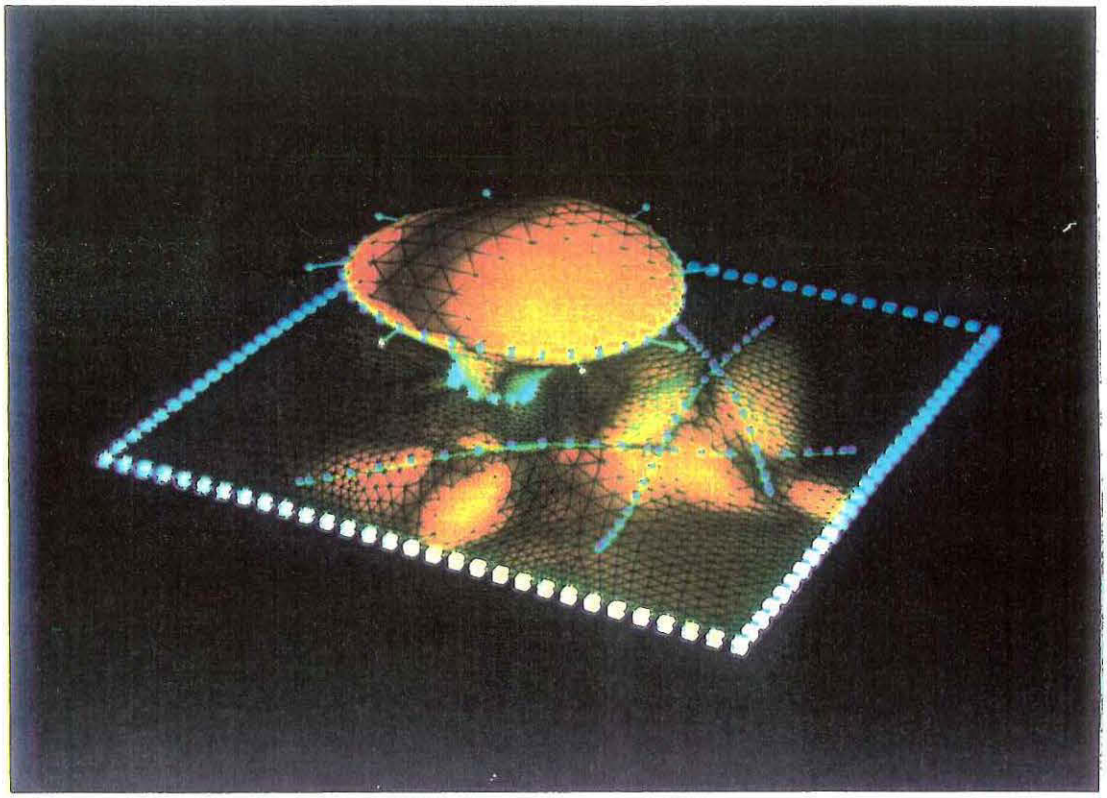

Figure 3.27 *Cette figure illustre la faculté du mécanisme à préserver la régularité des maillages.*

test de rapidité

Les test de rapidité, présentés ci-dessous ont été réalisés sur une station de travail DEC série 5000/200. Les temps d'exécution de l'installation et de l'interpolation de contraintes de type FCP ont été calculés pour différents jeu de données.

installation :

test $n°1$: surface de 200 triangles							
nombre de données	10	100	500	1000	2000	5000	10000
durée en secondes	0.4	0.7	2	3.6	7.25	18	41

test $n°2$: surface de 1000 triangles							
nombre de données	10	100	500	1000	2000	5000	10000
durée en secondes	2	3	8.5	19	55	310	1200

interpolation :

test $n°3$: surface de 200 triangles							
nombre de données	10	100	500	1000	2000	5000	10000
durée en secondes	0.2	0.57	1.2	2.3	4.6	12	28

test $n°4$: surface de 1000 triangles							
nombre de données	10	100	500	1000	2000	5000	10000
durée en secondes	1.1	1.5	2.2	3.7	5.5	15.7	13

3.3.6 conclusion

La qualité des résultats obtenus ne doit pas dissimuler les limites de la méthode.

1. *définition d'une direction de tir*
   La seule connaissance d'un ensemble de points et d'une surface initiale ne suffit pas à caractériser une solution unique, car il est nécessaire de définir un paramètre supplémentaire. Celui-ci correspond pour les contraintes *FCP* à la définition d'une direction de tir. Le résultat dépend donc de la manière dont l'utilisateur distribue les directions de tir, qui sont associées aux contraintes. Ces choix déterminent la qualité et la cohérence du résultat.

2. *importance de la surface initiale*
   La possibilité de gérer plusieurs fuzzy control points sur le même triangle ainsi que le transfert automatique de contrainte, permet à la méthode des contraintes *FCP* de résoudre le problème de l'adéquation entre la distribution des points de données et la répartition des atomes de la surface. Néanmoins, une relation directe existe entre le résultat et le nombre d'atomes de la surface initiale. Je rappelle que la méthode *DSI* implémentée converge pour un maillage donné vers une solution unique. Par conséquent, deux surfaces dont les connectivités des atomes sont différentes mais soumises aux mêmes contraintes, procurent deux résultats différents. La qualité du résultat dépend donc de la solution initiale, et en toute logique le nombre adéquat des atomes de la surface doit être supérieur à l'ordre du volume des données.

3. *la distribution des données*

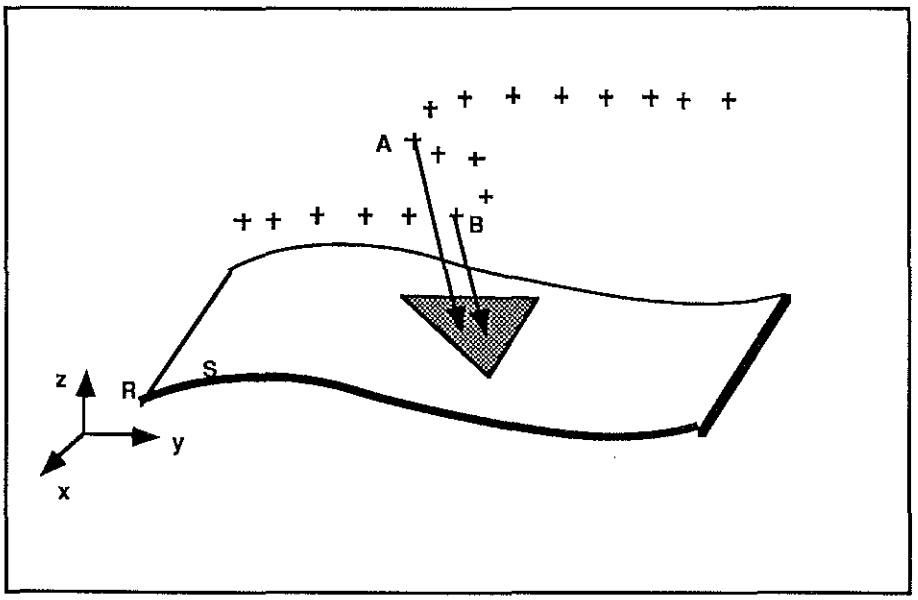

Figure 3.28 *distribution non-univoque de données.*

L'implémentation actuelle des contraintes de type *FCP* ne convient pas à toutes les distributions de données. La figure 3.28 illustre un de ces exemples. En effet, la méthode ne permet pas l'ajustement

automatique de la surface plane S aux données qui représente un pli couché. Les fuzzy control points A et B sont manifestement en conflit. L'implémentation proposée doit donc être affinée, afin de résoudre le problème des modélisation de formes géométriques non-univoques.

Néanmoins, l'utilisation des contraintes de type *FCP* marque une étape importante du développement de *DSI*. En effet, pour la première fois une contrainte est déconnectée du maillage de la surface, puisqu'elle ne repose pas sur l'attraction de tel atome ou de tel triangle. Cette faculté de relaxation automatique est très importante car elle préserve la régularité des maillages. De plus ce type de contraintes s'est révélé parfaitement adapté aux exemples réels qui nécessitent la gestion de milliers de points de données. La stabilité numérique et la robustesse de la méthode ont été depuis maintes fois vérifiées. La contrainte *FCP* est par conséquent promise à un bel avenir.

3.4 La contrainte OTS

Lors du chapitre consacré aux exportations et importations de vertex (cf page 12), nous avions distingué deux types de soudure, qui simulent chacune un contact entre deux surfaces. Ces relations sont de natures très différentes, puisque contrairement à la version "hard", une soudure "soft" autorise le glissement d'une surface le long de l'autre. Ce type de soudure représente un intérêt particulier, car elle simule parfaitement le comportement géologique d'un horizon traversé par une faille, et nous présentons ci-dessous cette notion, exprimée sous la forme d'une nouvelle contrainte appelée **On-Tsurf**.

3.4.1 exposé du problème

Lors de nos premiers travaux concernant la modélisation d'une faille, nous avons recherché en vain un système de CAO classique, susceptible de résoudre ce type de problème. Deux raisons principales expliquent cette difficulté.

1. La majorité des logiciels étant conçus pour l'industrie automobile ou mécanique, la modélisation d'une notion aussi spécifique que la faille géologique apportait peu d'intérêt et elle ne fut jamais réellement abordée. Seul un logiciel, tel GOCAD, spécialement conçu pour des applications pétrolières ou minières, peut investir dans ce domaine.

2. Le principe des courbes de Bézier n'est pas adapté à ce problème, car comme nous l'avons expliqué lors de la présentation de *DSI*, cette méthode mathématique se prête difficilement, aux ruptures de discontinuités, et donc à la prise en compte interactive de failles. La représentation d'une faille s'avérait techniquement difficile à réaliser.

Forts de la nouvelle approche suivie dans le projet GOCAD, nous espérons concevoir une solution efficace, néanmoins, à l'heure de cette rédaction, l'ensemble de tous les problèmes ne sont pas résolus. Nous ne présenterons pas en globalité la modélisation d'une faille, mais plus modestement nous nous sommes attachés, dans ce qui suit, à simuler la nature d'un contact qui lie un horizon à une faille. Nous considérerons notamment dans l'avenir que la faille existe et qu'elle est représentée par une surface triangulée de type "tsurf"; l'initialisation et la construction d'une faille ne seront pas abordées.

La présence de failles au sein d'un ensemble géologique augmente considérablement la complexité d'un modèle, car elles conditionnent directement

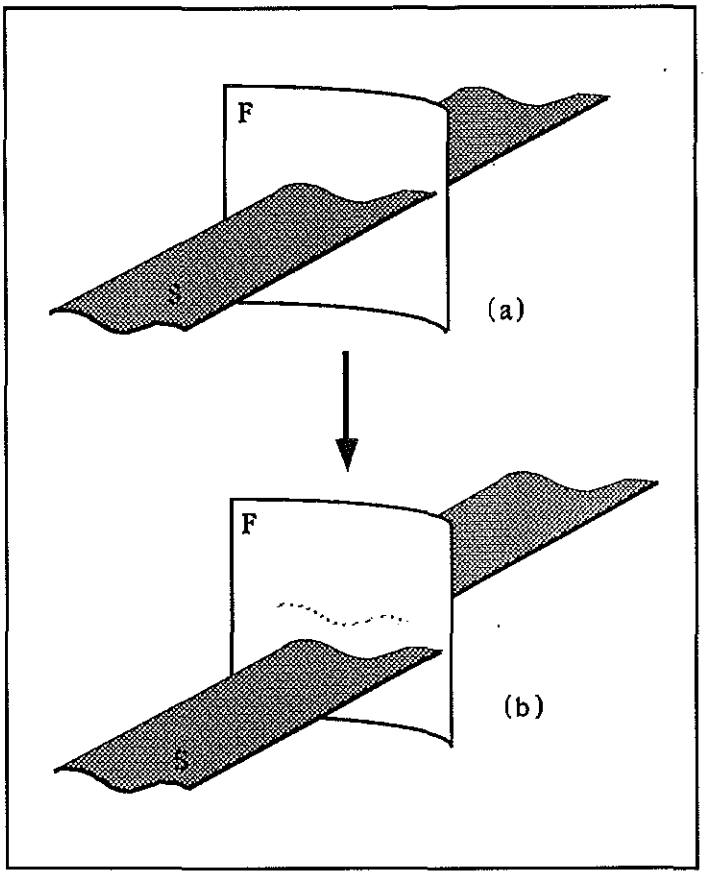

Figure 3.29 *un horizon à la possibilité de glisser le long d'une surface de faille.*

le comportement des horizons traversés. La figure 3.29 illustre mon propos, et représente un exemple où l'horizon S glisse le long de la faille F, mais en aucun cas il ne peut s'en décoller. Plus généralement, cet exemple démontre que la géométrie d'un objet est fonction du nombre et du type de relations qu'il entretient avec les autres objets, et à l'image de notre société, le sous-sol terrestre apparaît comme un univers organisé où le comportement de chacun est soumis à des règles précises. Aussi, la conception d'un système performant ne peut s'envisager sans la modélisation de tels comportements.

Nous avons interprété la relation $(horizon, faille)$ comme une soudure "soft", qui se caractérise par l'existence d'un contact parfait mais mobile entre les deux surfaces. Contrairement à la soudure "hard", les deux objets ne partagent aucun point, mais ils ne sont pas pour autant indépendants. Ce

contact est défini par le fait que le bord de l'horizon, jointif de la faille, est astreint à se mouvoir uniquement le long de celle-ci. La modélisation de cette relation représente un grand intérêt car elle assure la cohérence géologique du système. Citons deux raisons principales:

1. **lors de la construction**
   Lors de la construction d'un modèle, l'utilisateur reconstitue chaque horizon séparément, et la précision des données disponibles ne permet jamais de définir automatiquement les contacts entre surfaces. La figure 3.30 illustre ce problème:

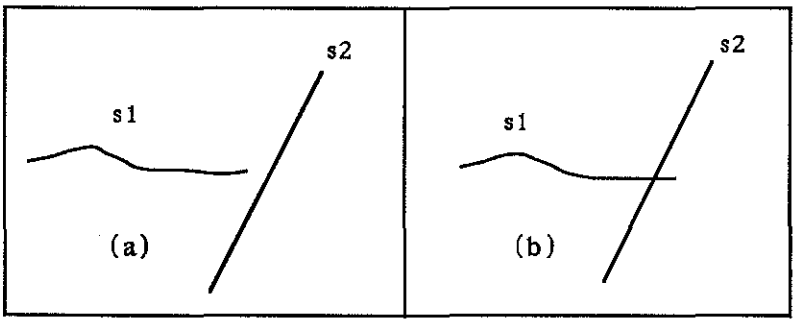

Figure 3.30 *les contacts entre surfaces ne sont jamais automatiquement définis:*
(a) les deux surfaces sont disjointes.
(b) les deux surfaces s'intersectent.

   Dans le cas *(a)*, la surface *S1* est disjointe de la surface *S2*, et dans le cas *(b)* les deux surfaces s'intersectent. Or nombre d'applications basées sur notre modèle ne peuvent se satisfaire d'une modélisation aussi imparfaite.
   Exemple : le tracé de rayons sismiques.
   Si un seul contact n'est pas parfaitement représenté, notre modèle est inconsistant puisque dans le cas *(a)* des rayons pourront traverser l'interstice laissé entre les deux surfaces, au lieu de s'y réfléchir.

2. **lors d'une simulation**
   Le principe de toute simulation consiste à modifier la forme et la position des surfaces qui constituent le modèle. Par exemple, la tomographie cherche à adapter l'allure générale d'un horizon afin de minimiser une fonction de coût, or si une surface est translatée ou déformée, ses contacts avec les autres objets sont définitivements rompus.

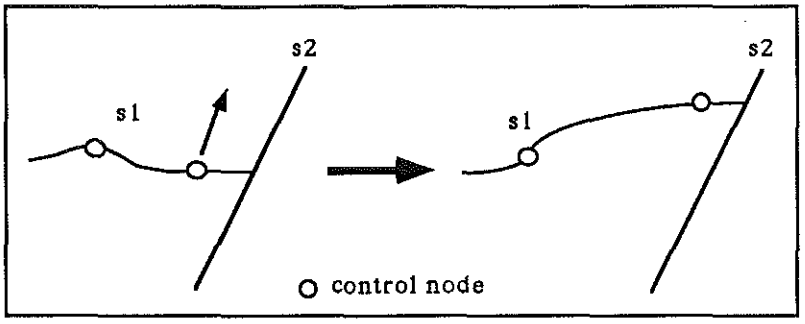

Figure 3.31 *le déplacement d'un control node déforme la surface S1, mais le contact doit glisser le long de la surface S2 .*

> Un système performant ne peut envisager l'évolution d'une surface de façon isolée, mais doit impérativement la replacer dans le contexte général du modèle, où il est nécessaire de tenir compte de ses relations avec les autres objets. La modélisation d'une soudure soft nous permet d'assurer la cohérence des contacts, car elle contrôle leurs évolutions, puisque que tel l'illustre la figure 3.31, le déplacement d'un contact se restreint à un glissement.

La soudure soft soulève de nombreux problèmes d'ordre informatique, puisque le système doit être capable de gérer de façon automatique des relations entre objets, mais ce point sera évoqué ultérieurement dans le chapitre dédié aux relations entre objets (cf page 118). La suite de cet exposé est désormais consacrée à l'interprétation mathématique de la notion de glissement, qui s'exprime par une nouvelle contrainte appelée **On-Tsurf**.

3.4.2 principe adopté

Le principe adopté, et illustré par la figure 3.32 consiste à appliquer une contrainte sur chacun des atomes situé sur la bordure de l'horizon S, qui constitue la ligne de soudure avec la faille F. Pour chacun de ces atomes, une direction de tir, $\overrightarrow{Tir}$, et une cible sont définies. Dans l'exemple proposé, la droite issue de l'atome A et parallèle au vecteur $\overrightarrow{Tir}$ intercepte le triangle T de la surface F en un point d'impact I. Si désormais l'atome A est déplacé sur la position I, le contact entre les deux surfaces est réalisé. Les difficultés rencontrées ne concernent pas directement cette simple projection, mais l'implémentation et la gestion de ces contraintes. Les questions

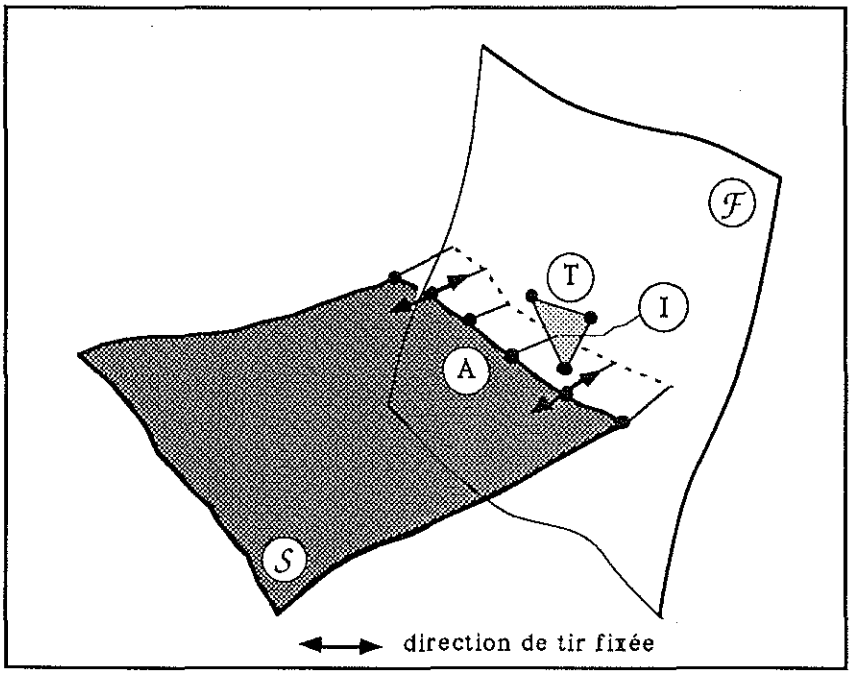

Figure 3.32 *shéma d'une d'une contrainte On-Tsurf.*

suivantes mettent en évidence l'ensemble de ces problèmes:

• **Comment définir la ligne de contact ?**
Il n'existe malheureusement pas de critères universels, qui permettent automatiquement et dans tous les cas de figures de décider que tel atome plutôt que tel autre fait partie d'une ligne de soudure. La définition de l'ensemble de ces points est difficile car elle dépend principalement de la forme géométrique des deux surfaces concernées et l'exemple de la figure 3.33 décrit cette ambiguité: Dans le cas *(b)* l'atome B doit être projeté sur la surface F alors que dans le cas *(a)* cette projection n'a plus de sens.

Une première solution consiste à faire appel au bon sens de l'utilisateur afin qu'il sélectionne lui-même l'ensemble des atomes qu'il désire rassembler dans une ligne de soudure, sinon, la définition automatique de la ligne de contact n'est possible que dans le cas particulier où deux surfaces s'interpénètrent. La fonction *TSURF_Cut_by_TSurf()* conçue par Yungao Huang (cf :[13]), permet comme l'illustre la figure 3.7 de découper une surface par une autre, le long de leur intersection. Le principe de cette opération, consiste tout

d'abord à calculer la ligne d'intersection puis à reévaluer la triangulation, afin de déconnecter les triangles d'un bord et de l'autre de cette coupure. Lorsque ces nouveaux triangles sont créés, une ligne d'atomes dédoublés est constituée le long de l'intersection, or ceux-ci représentent exactement l'ensemble des atomes qui forment la ligne de soudure entre ces deux surfaces. Notez qu'elle est constituée de deux morceaux qui sont répartis sur les deux faces de la surface de coupe.

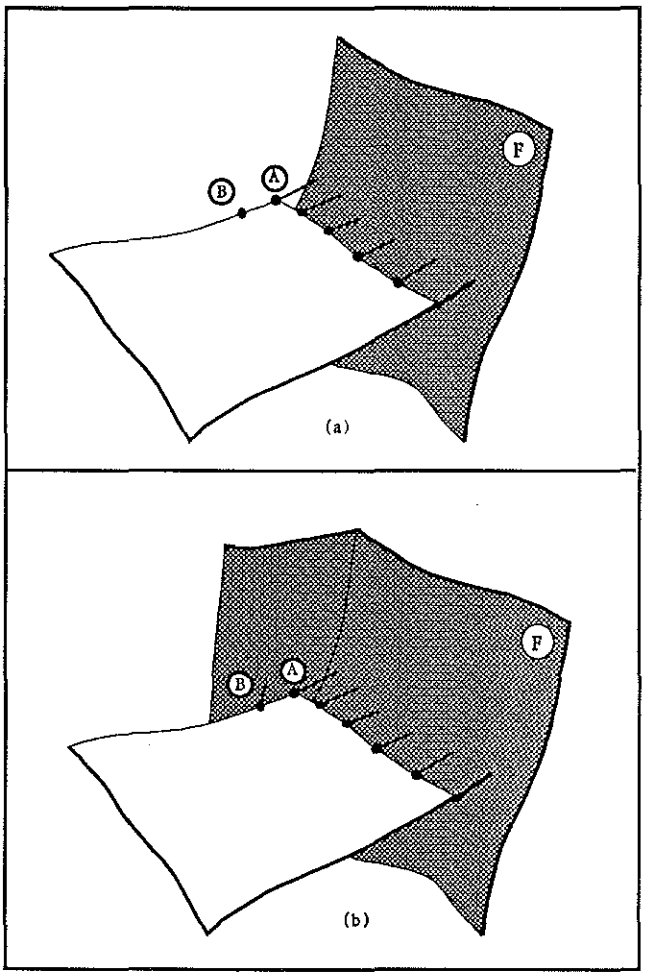

Figure 3.33 *illustration de l'ambiguité d'une ligne de contact.*

• **Comment définir les directions de tir ?**
Certes la définition des directions de tir est fonction de la forme et de la position relative des deux surfaces, mais il est possible, au lieu de les définir manuellement une à une, d'automatiser leurs initialisations comme suit:

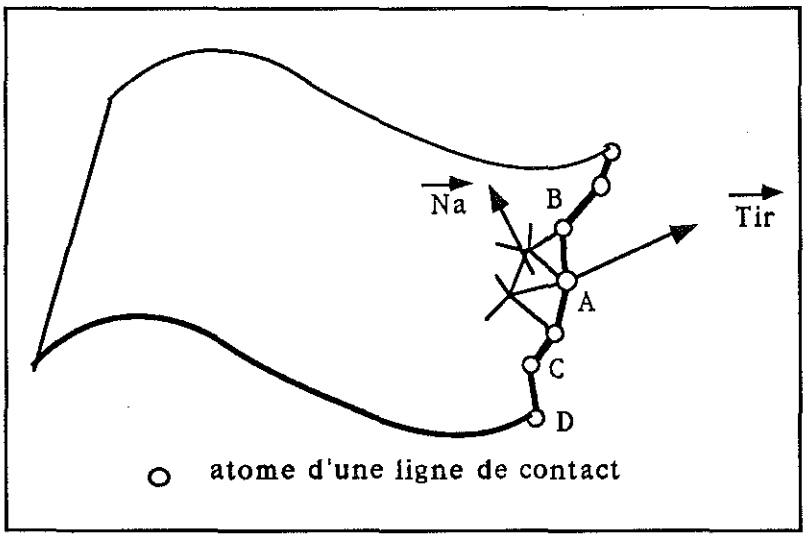

Figure 3.34 *calcul automatique des directions de tirs des contraintes OTS.*

Considérons à la figure 3.34, l'atome A entouré de deux satellites B et C. Si ces trois atomes appartiennent à la même ligne de contact, alors la direction de tir $\overrightarrow{Tir}$ est définie comme le produit vectoriel du vecteur $\overrightarrow{BC}$ avec le vecteur $\overrightarrow{Na}$, normale moyenne de la surface S, calculée sur l'ensemble des triangles dont A est un sommet. Si l'atome D est une des extrémités de la ligne de soudure alors le vecteur $\overrightarrow{Tir}$ associé est égal au produit vectoriel de $\overrightarrow{Nd}$ et de $\overrightarrow{CD}$. Le sens du vecteur $\overrightarrow{Tir}$ a peu d'importance, puisque lors du calcul des triangles interceptés, la fonction *TSURF_Shoot()*, précédemment évoquée page 56, effectue la recherche dans les deux directions opposées, mais néanmoins portées par ce vecteur.

Si désormais, l'ensemble des directions de tir sont initialisées, nous avons implémenté un système, qui, à l'image des directions de tir des fuzzy control lines, permet de les interpoler. Si certaines directions sont fixées, l'ensemble des directions laissées libres sont recalculées afin de minimiser au sens de DSI les écarts entre une direction de tir et ses directions de tir voisines.

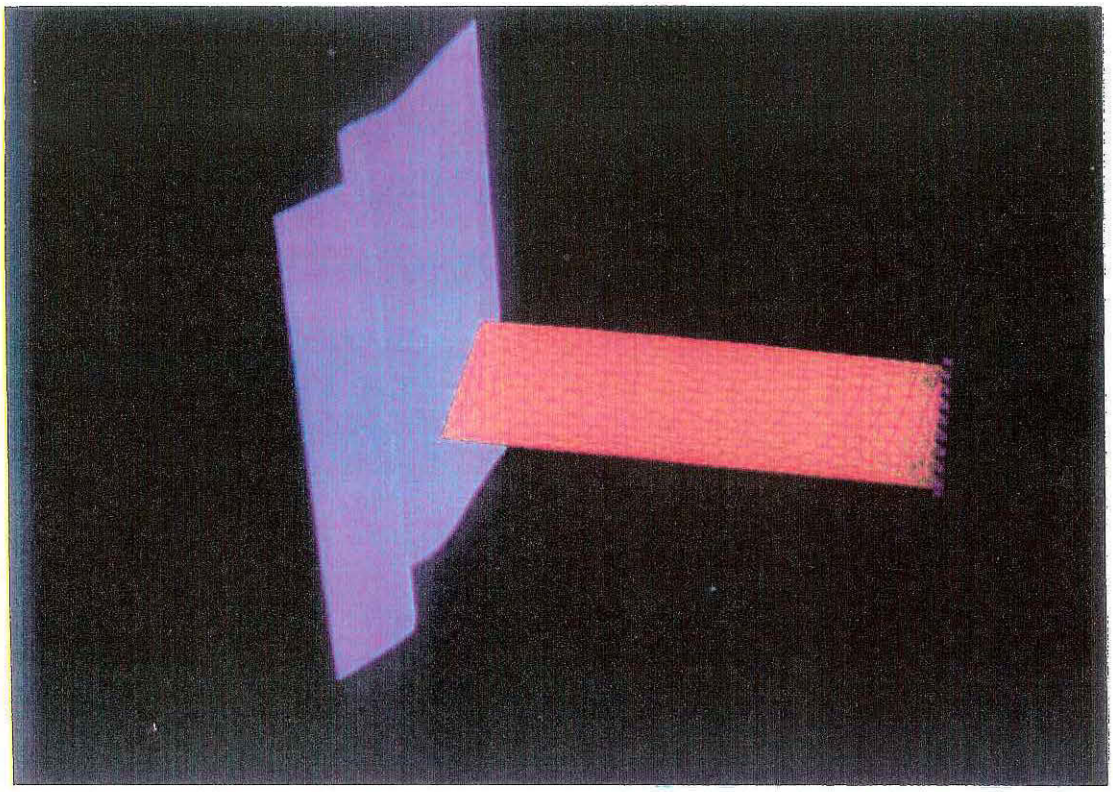

Figure 3.37 *Chaque atome du bord de la surface jaune est associé à une contrainte OTS, afin d'astreindre cette bordure à s'ajuster sur la surface bleu. Les directions de tirs sont représentées par les traits rouges. Les cubes verts sont des controlnodes qui fixent la bordure opposée de l'horizon*

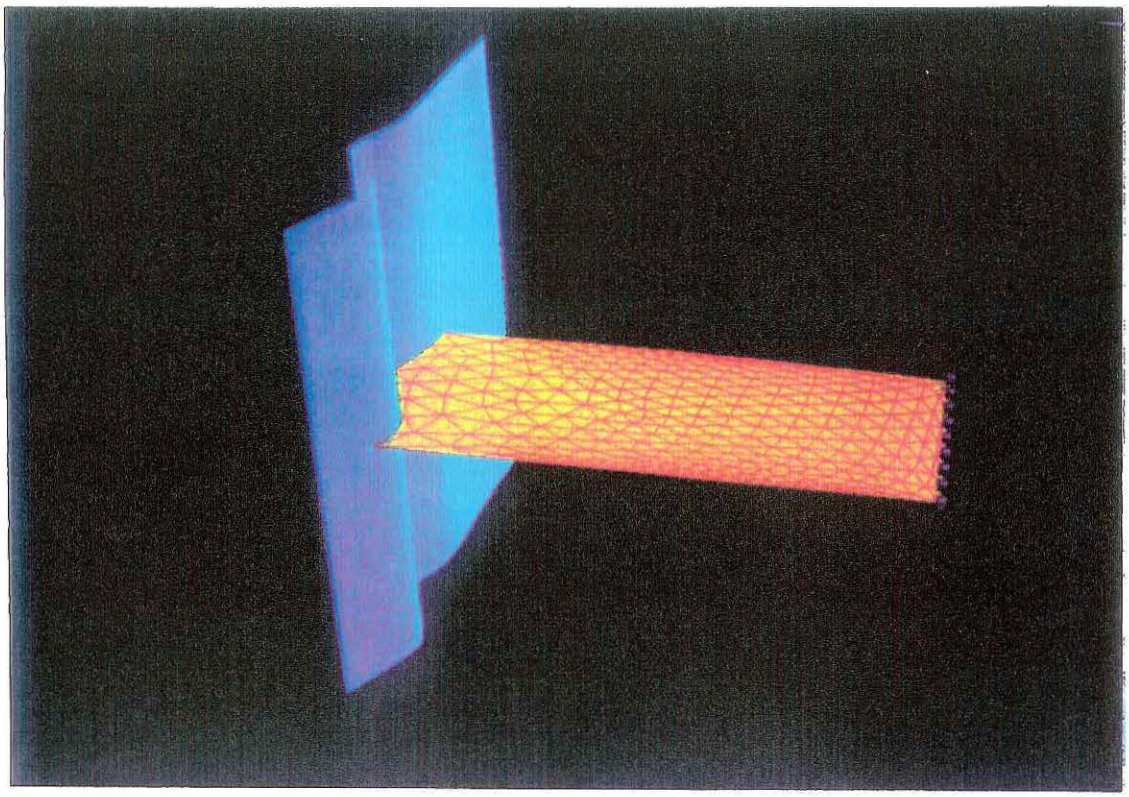

Figure 3.38 *Résultat de l'interpolation après dix itérations. Le contact entre les deux surfaces est respecté.*

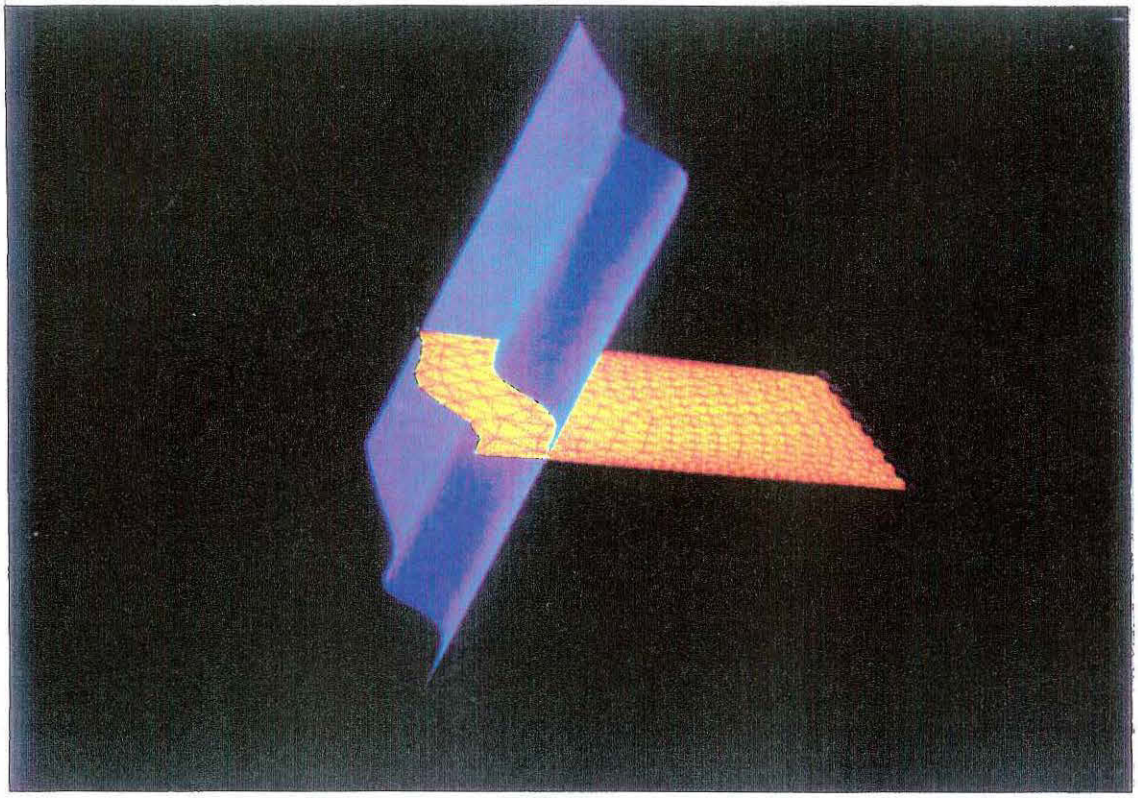

Figure 3.39 *La faille dessinée en bleu a été translatée, afin que les deux surfaces s'intersectent.*

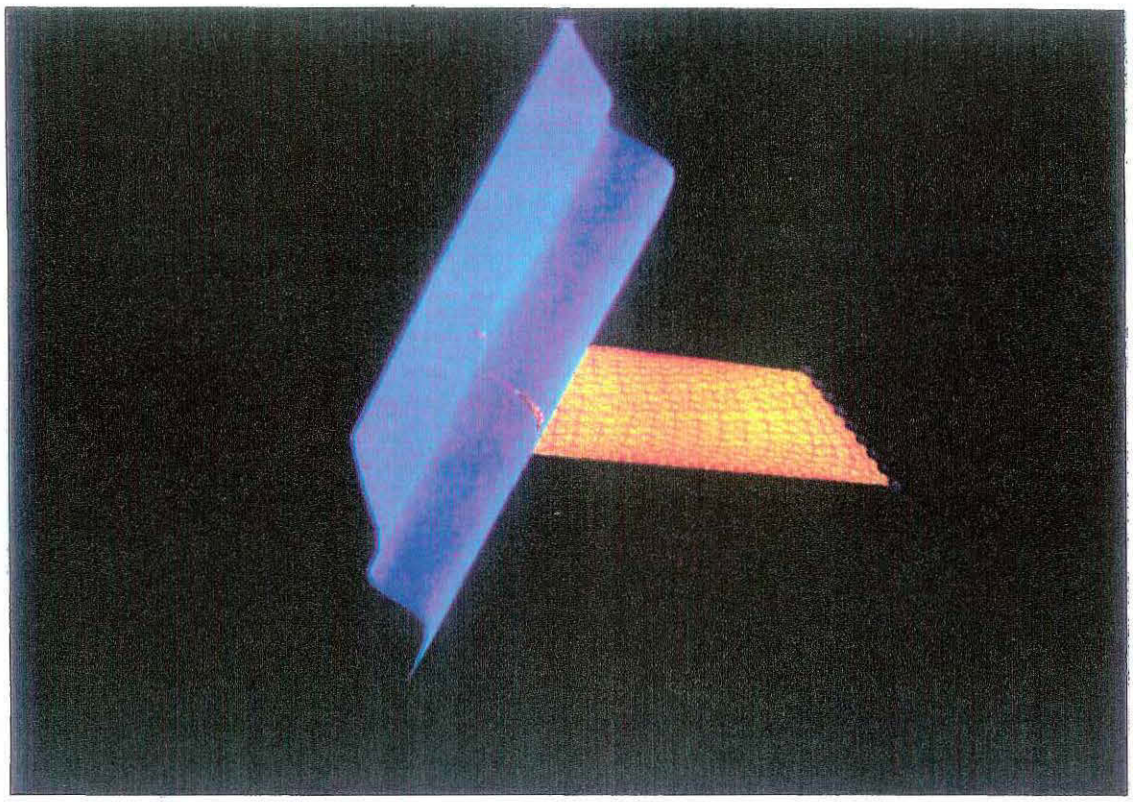

Figure 3.40 *Résultat après interpolation: Le contact entre les deux surfaces est restitué.*

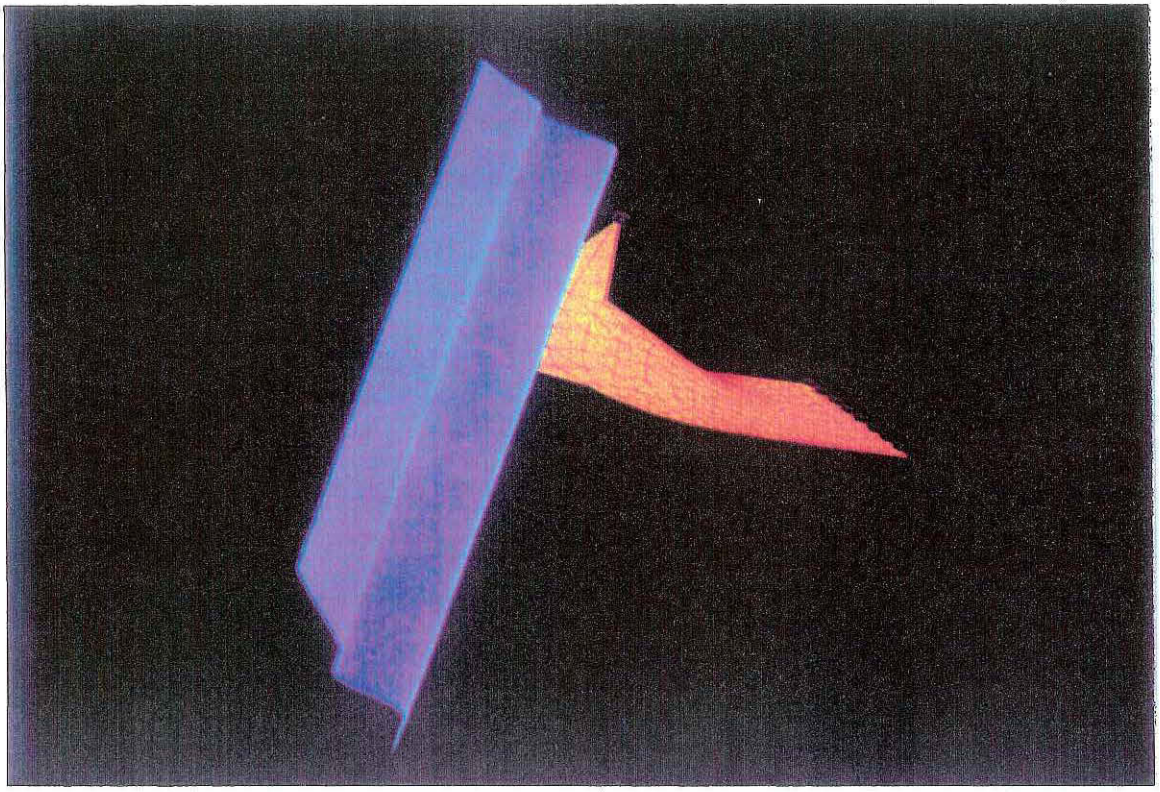

Figure 3.41 *Un control node est ajouté sur la surface jaune et provoque la déformation de l'horizon.*

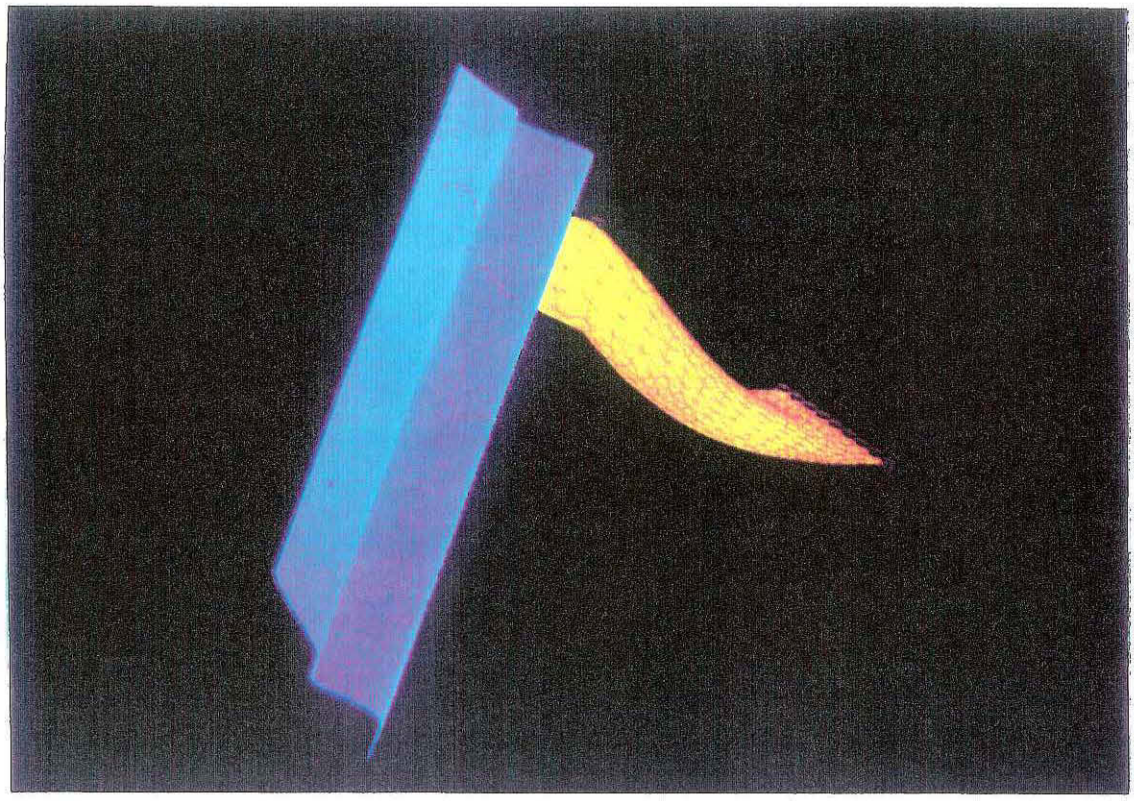

Figure 3.42 *Résultat après interpolation: le contact a glissé le long de la faille.*

Ce travail n'a pas été abordé car l'organisation et la gestion actuelle des objets de la base de données ne permet pas d'automatiser entièrement toutes les relations entre les objets. Cette organisation fait l'objet d'une étude en cours, qui est réalisée sur la base du modèle de Weiler (cf : [23]), et je pense qu'il est importants d'attendre ces résultats, avant de répondre de façon définitive au problème évoqué.

•modélisation de l'épaisseur d'une couche

L'utilisation de la contrainte *OTS* déborde le strict cadre de la représentation d'un contact entre un horizon et une faille, mais apporte une solution à d'autres problèmes , telle la modélisation de l' épaisseur d'une couche géologique. En effet, le contact entre un atome de la surface d'un horizon et un triangle de la faille, se résume à définir une distance nulle entre cet atome et le point d'impact. Par conséquent, il est tout aussi simple de respecter une distance qui serait soit une constante non nulle, soit le résultat d'une fonction. Dès lors, il est possible de définir la distance d'un atome d'une surface par rapport à un triangle d'une autre surface et de simuler ainsi l'épaisseur d'une couche, tel que l'illustre la figure 3.45. La contrainte *OTS* est d'ailleurs implémentée réellement sous cette forme.

•l'information est qualitative

A l'inverse des contraintes classiques qui sont quantitatives, la contrainte *OTS* repose sur une information qualitative, qui s'exprime simplement par le fait qu'une surface est astreinte à rester collée le long d'une autre. Son implémentation prouve que la méthode *DSI* est capable de tenir compte de données de natures très différentes, et bien souvent le savoir-faire d'un expert s'exprime plus en fonction de connaissances qualitatives liées à son expérience, qu'en fonction de mesures numériques.

3.4.6 conclusion

Détrompez-vous !!

L'objectif principal de cet exposé ne consistait pas à modéliser le glissement d'une surface sur une autre. Il nous est apparu bien plus important de prouver qu'avec un peu d'imagination la méthode *DSI* était capable de modéliser des comportements spécifiques, qui correspondent à des relations géologiques précises. Un horizon et une faille sont deux objets géologiques différents, mais ils sont tous deux représentés géométriquement par une surface triangulée. Seule la différence de leurs comportements traduit leur vraies nature physique: La faille découpe l'horizon et non l'inverse, tandis que l'horizon

glisse le long de la faille. Cette faculté de modéliser des comportements est essentielle, car elle permettra dans l'avenir de gérer des objets géologiques et non plus seulement leur simple représentation géométrique.

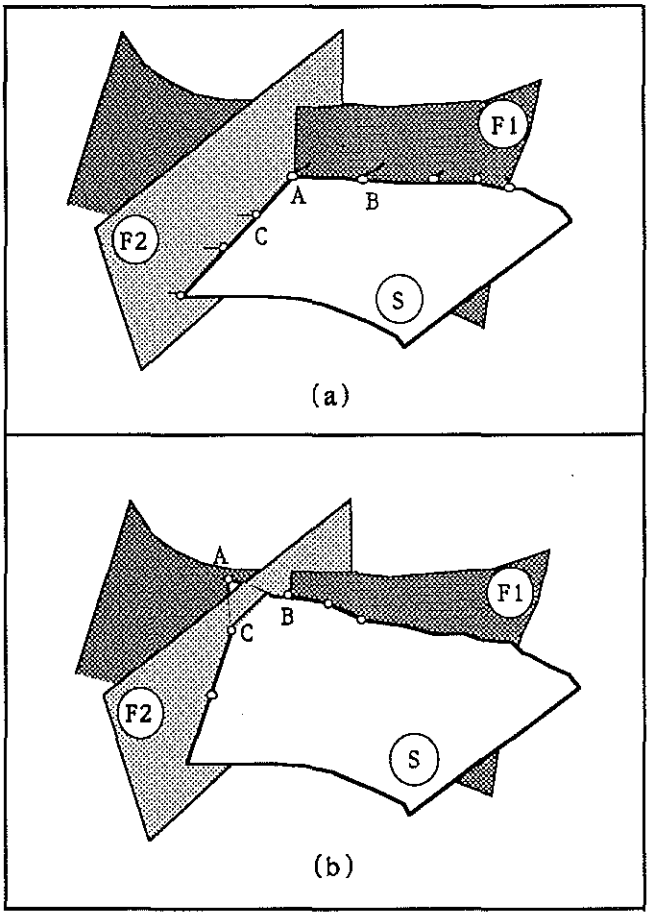

Figure 3.44 *schéma de contacts entre un horizon et deux failles qui s'intersectent.*

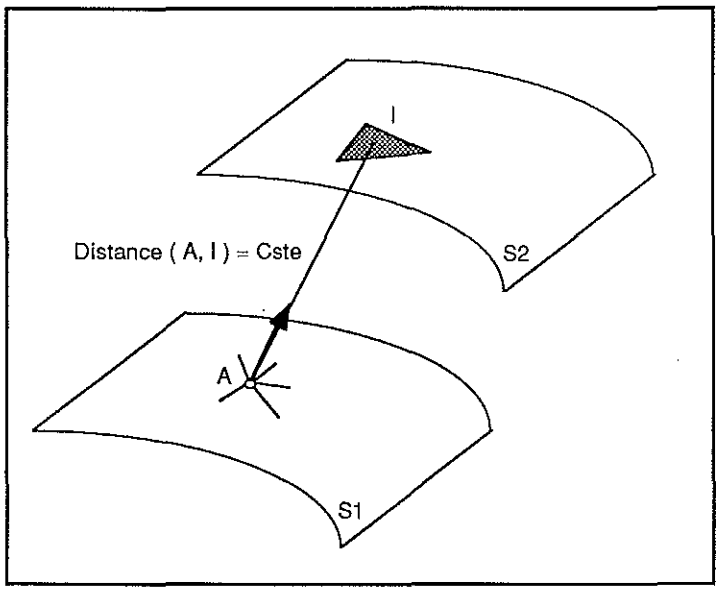

Figure 3.45 *la distance (A,I) est imposée entre les deux surfaces.*

3.5 Conclusion

De jour en jour, la méthode *DSI* s'enrichit de l'imagination et de la conception de nouvelles contraintes. Leurs compatibilité et leur complémentarité, nous permet d'envisager dans l'avenir la modélisation de situations de plus en plus complexes. La réalisation des contraintes "fuzzy controlpoint", "fuzzy control line" et "on-tsurf" fut guidée par les recommandations des sponsors du projet GOCAD et notre intention fut d'apporter des solutions concrètes aux difficultés rencontrées lors de l'etude de cas réels. Pour conclure, je tenais à vous présenter les résultats d'une étude réalisée au BRGM [6], qui illustre parfaitement les capacités de ces trois contraintes. Ce fantastique travail réalisé par Mr Ph.Renard, constitue la meilleure illustration, dont nous puissions rêver. Les horizons ont été reconstruits à l'aide de données filtrées de lignes sismiques, représentées en jaune sur la figure 3.47, et les contacts de ces horizons avec les failles ont été reconstitués grâce aux contraintes de type *OTS*.
Parole aux images !!

[6] Bureau de Recherches Géologiques et Minières

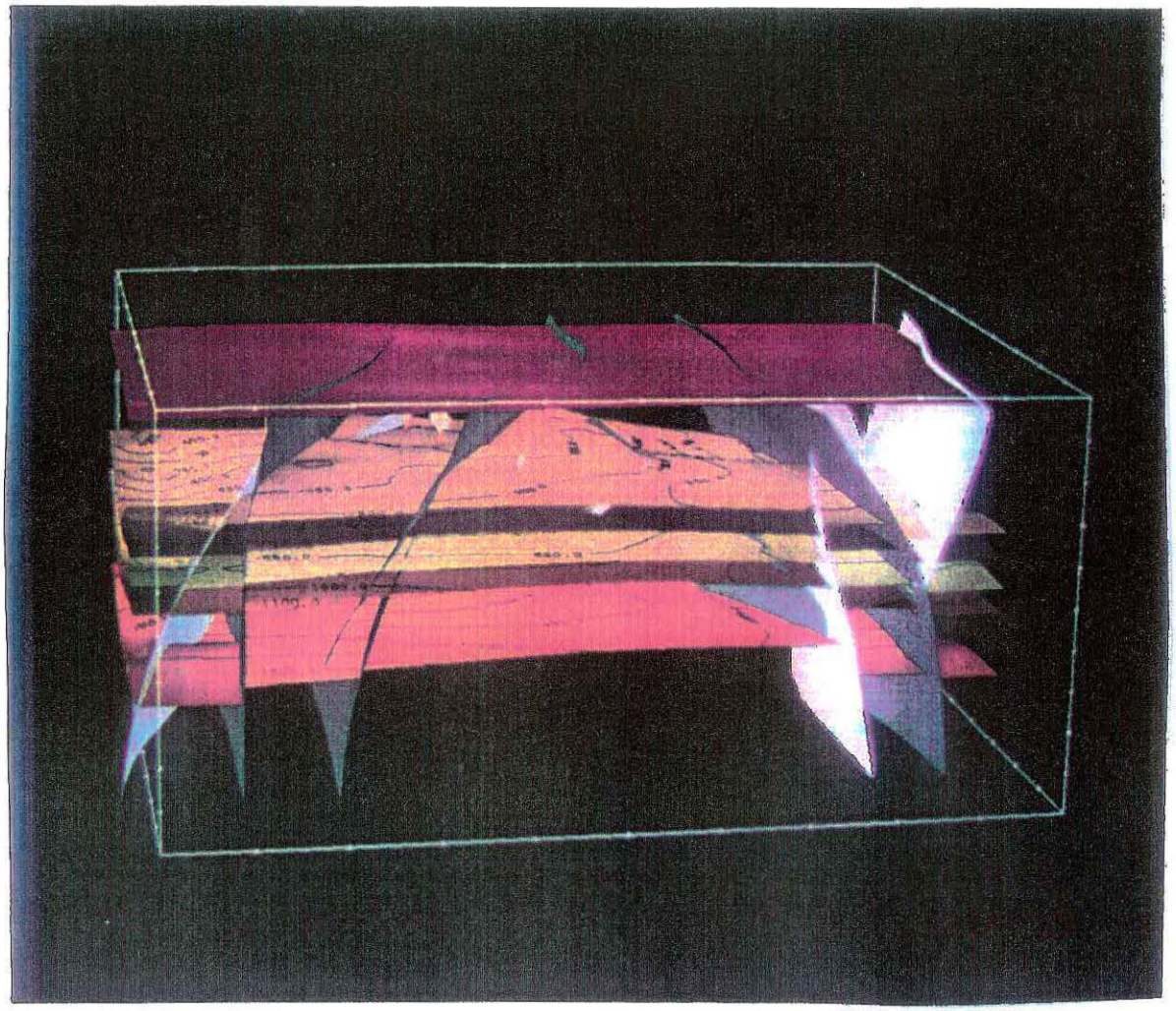

Figure 3.46 *modélisation d'un bassin géologique constitué de multiples horizons faillés*

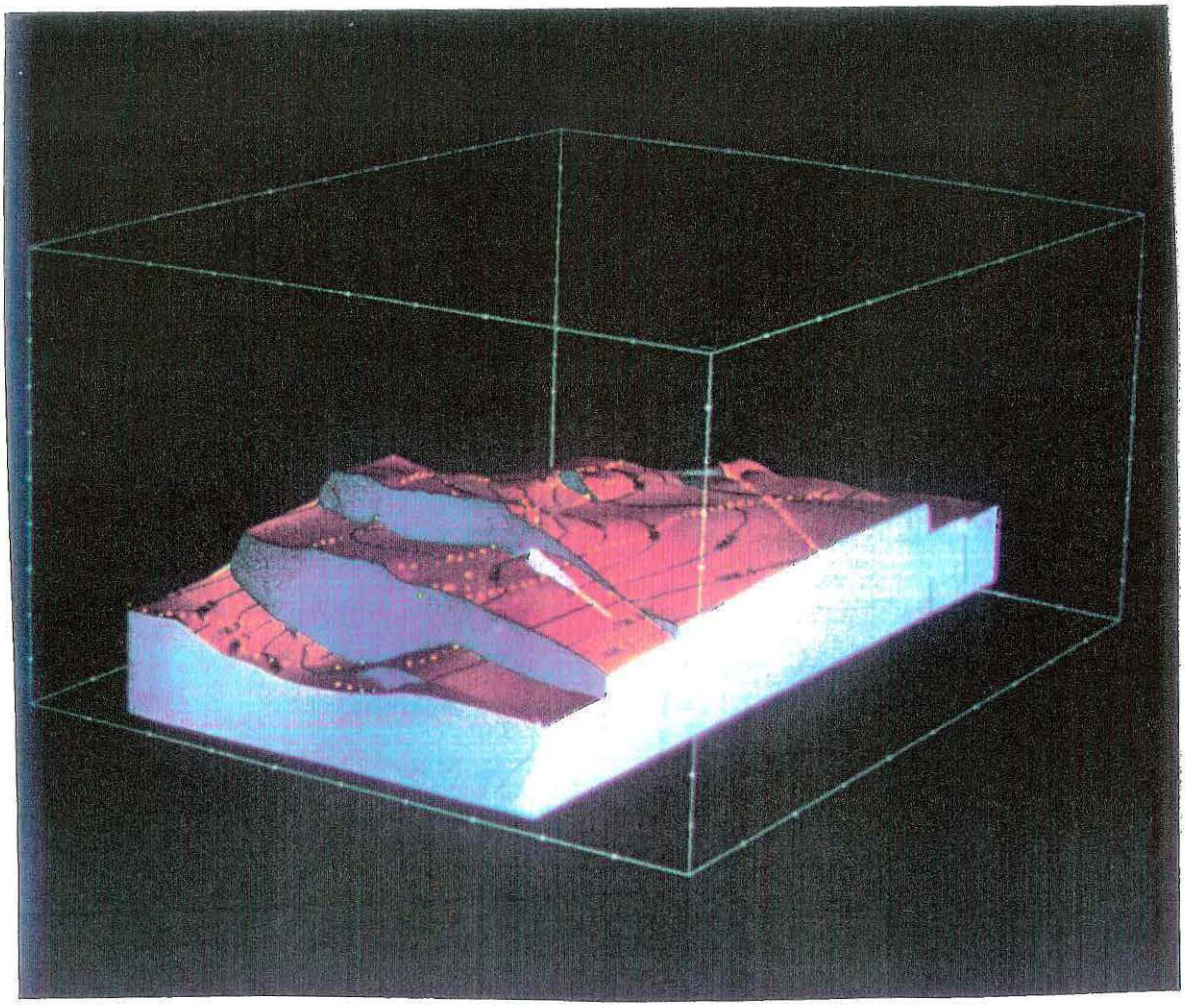

Figure 3.47 *bloc diagramme d'un des horizons du modèle, dont la structure tectonique est remarquablement mise en avant. Les lignes de points dessinées en jaunes correspondent aux contraintes de type Fuzzy Control Line, qui ont été utilisées lors de l'ajustement des horizons. Il est important de noter que la discontinuité topologique de la surface ne perturbe pas l'utilisation de ce type de contraintes.*

Chapitre 4

Gestion de relations entre objets

4.1 Introduction

La représentation d'une faille marque une étape très importante dans le développement du système GOCAD, car elle soulève implicitement les problèmes liés aux relations entre objets. Auparavant, chaque objet de la base de données était géré individuellement, et les interactions existant entre différents objets n'étaient jamais prises en compte. Lors de la présentation de la contrainte *OTS* (cf page 78) nous avons pu constater que la faille est un objet géologique complexe, puisque par nature elle traverse plusieurs horizons et influence ainsi directement leurs comportements. Sa modélisation demeure donc irréalisable en l'absence d'un mécanisme capable de gérer des dépendances entres objets. Ce sujet fut maintes fois abordé, mais les solutions alors envisagées s'avéraient inadaptées, comme l'illustre par exemple le mécanisme d'importation ou d'exportation de vertex (cf page 12):
Lorsque deux objets partagent un ensemble de points, l'existence des vports d'importations et d'exportations au sein des vsets des objets permet de retrouver la trace de leurs inter-dépendances. Cependant ce principe est limité car il ne tient compte que d'un type unique de relation (l'existence de points communs). De toute évidence, la gestion des relations entre objets

doit être parfaitement indépendante de la nature des relations.

La littérature informatique divise les mécanismes de dépendances en deux grandes catégories:

- **les dépendances statiques**
  Une dépendance est statique lorsqu'elle est définie dès la compilation. Dans ce cas, elle est exprimée explicitement dans la structure des objets. Elle est connue préalablement et ne sera jamais rompue. Ce type de dépendance définit des relations entre classes d'objets.
  Exemple : Une "tsurf" est liée par nature à un "vset", puisque sa structure réserve un champ spécifique au vset qui lui est associé (cf page 8).

- **les dépendances dynamiques**
  Une dépendance est dynamique lorsqu'elle est gérée à l'exécution, c'est-à-dire en temps réel. Ces relations peuvent être créées, détruites, puis reconstruites à tout instant au cours d'une session de travail. Elles sont parfois temporaires, et à l'inverse des dépendances statiques, elles définissent des relations entre des instances de classes d'objets.
  Exemple: Les notions de tsurf et de pline sont parfaitement indépendantes et correspondent chacune à la définition d'un macro-objet de la base de données. Néanmoins, lorsqu'une contrainte de type Fuzzy Control Line est installée, une dépendance particulière est créée entre une surface spécifique "S" et une ligne "L".

  D'un point de vue informatique, la gestion des dépendances dynamiques est la plus complexe, car elle préserve l'interactivité du système.

Les contraintes $FCP\ (vset, tsurf)$, $FCL\ (pline, tsurf)$, $OTS\ (tsurf, tsurf)$ présentées dans cet ouvrage constituent une nouvelle famille de contraintes, car elles correspondent toutes à des relations qui mettent en jeu deux macro-objets distincts. Par conséquent, leurs implémentations au sein du système GOCAD nécessitent la conception préalable d'un mécanisme capable de gérer des dépendances dynamiques entre objets. La suite de ce chapitre lui est consacrée.

4.2 Principe adopté

Chaque dépendance dynamique, établie entre deux objets, représente un ensemble de données supplémentaires, qui leurs sont alloué. Le principe consiste à imaginer un espace mémoire destiné au stockage de ces informations, qui puisse être modulable en fonction de leur nature, leur nombre et leur taille. Cet espace, appelé **blackboard** est associé à tout objet de type **gobj**[1] (cf page 11). Les gobjs ou macro-objets, constituent une classe à part de la base de données; par convention, toute structure dont les quatres premiers champs sont identiques à la structure *GOBJ_t* décrite ci-dessous, sera considéré comme un gobj. La définition de cette classe d'objets permet de gérer par des mécanismes uniques l'ensemble des différents macro-objets du système. Rappelons, qu'une tsurf, une pline ou un vset sont chacun un gobj.

```
typedef struct GOBJ_t
        {
          int      type       ;
          char   * name       ;
          long     mailbox    ;
          SETK_c   blackboard ;
        } GOBJ_t ;

typedef struct GOBJ_t * GOBJ_c ;
```

Par souci de cohérence, les informations stockées dans le blackboard sont définies obligatoirement selon un format unique, appelé **token**. Le mécanisme de gestion des relations entre objets repose sur la complémentarité des notions de blackboard et de token; le premier représente un "container" et le second constitue son "contenu". Ces deux notions essentielles sont présentées ci-dessous.

4.2.1 notion de token

Les tokens représentent l'unique format de données susceptibles d'être contenues dans un blackboard. Toute structure dont le premier champ est un entier appelé *type* est considéré comme un **token**. Cependant cette famille se décompose en deux catégories :

1. **les tokens élémentaires**
   Nous appelons token élémentaire tout objet dont la structure se résume

[1] gobj signi e G ocad-O bject

à un unique entier, appelé *type* et représentée par la structure *TOKEN_t* suivante.

```
typedef struct TOKEN_t
               {
               int   type ;
               } TOKEN_t  ;

typedef struct TOKEN_t * TOKEN_c ;
```

2. **les tokens utilisateurs**
   Chaque utilisateur a la possibilité de définir ses propres tokens. Ils représentent des tokens élémentaires, dont la structure est étendue à de nouveaux attributs, qui caractérisent l'information à stocker dans le blackboard.
   Exemple : Supposons que l'utilisateur désire conserver temporairement un certain nombre de données telles que le nom d'un objet ou sa couleur. Dans ce cas, il doit définir une structure spécifique de token, notée par exemple *my_token*, dont il assume la cohérence.

```
#define MY_TOKEN  100

typedef struct MY_TOKEN_t
               {
               int   type                    ;  /* = MY_TOKEN */
               char * name_of_object         ;
               ...
               long   color_index_of_object ;
               } TOKEN_t  ;

typedef struct MY_TOKEN_t * MY_TOKEN_c ;
```

   Cette information une fois créée, c'est-à-dire lorsque la place en mémoire de la structure *MY_TOKEN_c* est allouée, et que ses champs sont initialisés, est insérée dans un blackboard, où à tout moment elle sera accessible.

Remarque :
Afin d'assurer la consistance du système, toute variable, *type*, d'un token doit caractériser de façon univoque la nature de l'information à stocker; en d'autres termes, deux informations de nature différente seront associées à deux tokens dont le champ *type* sera différent.

4.2.2 notion de blackboard

Le rôle du blackboard consiste à conserver en mémoire centrale des données dont on ne connait a priori ni la nature ni le nombre. Le choix de sa structure est important car elle définit l'organisation du stockage de l'information dans la mémoire. Deux principes ont guidé ce choix:

- la diversité des informations susceptibles d'être stockées nous impose d'adopter une structure modulaire. Les informations seront répertoriées selon leur nature dans des modules, qui seront eux-mêmes automatiquement insérés dans la structure du blackboard. Le nombre de ces modules, ainsi que le nombre d'informations de même type sera illimité, afin de tenir compte de la diversité des éléments.

- la structure adoptée doit préserver l'efficacité du mécanisme de gestion du blackboard, dont l'ajout, la recherche et le retrait d'un élément constituent les trois fonctions essentielles. C'est pourquoi la structure du blackboard ne doit pas être conçue aux détriments de la simplicité et de la rapidité de ces fonctions.

Avant de poursuivre, je désire ouvrir une parenthèse sur la notion de "container", car elle illustre parfaitement le cheminement et les étapes qu'il fallu franchir avant d'imaginer la structure actuelle du blackboard.
Parmi les outils qui organisent l'information en mémoire centrale, les tableaux, les listes, et les piles sont les plus utilisés mais ils ne correspondent malheureusement pas aux spécificités du blackboard. Plus généralement, la diversité de ces outils prouve qu'il n'existe pas de structure unique, qui soit adaptée à tous les type de données ou à toutes les utilisations. L'efficacité d'une liste est reconnue lors de l'ajout d'un élément, mais par contre lorsqu'un élément précis est recherché, aucun accès direct n'est possible, et l'ensemble de la liste doit être parcourue jusquà atteindre la position de l'objet désiré. A l'inverse le tableau offre, grâce à l'indice de ses éléments, un accès direct aux informations, mais lorsqu'un nouvel élément est rajouté la mémoire réservée est remise en cause. Parmi les outils utilisés pour organiser les informations d'une base de données, la notion de container telle qu'elle est définie dans GOCAD, est essentielle car elle détermine directement l'efficacité de la gestion de cette base. En effet, elle définit les accès, oriente le classement, évite la redondance, et assure la consistance même de l'information.
Exemple : Certains objets, tels les macro-objets ont une existence propre, d'autres tels que les vertex représentent des sous-objets dont l'accès et la

manipulation sont indirectes. En effet, je rappelle que tout vertex est obligatoirement contenu dans un vset, et cette convention assure la cohérence du mécanisme des importations et exportations de vertex (cf 26). De manière générale, tout container doit être adapté à l'information qu'il est appelé à stocker, ainsi qu'aux utilisations qui lui sont destinées.

La conception de nouveaux types de containers, spécialement adaptés aux besoins de la base de données du système GOCAD, furent menés par Mr Philippe Nobili, et ils aboutirent à la réalisation de deux nouvelles structures appelées **set** et **setk**. Ces nouveaux containers possèdent la remarquable faculté de rassembler en même temps les avantages d'une hash-table et d'une liste.

Notion de SET :

La structure de set fut exposée en détail au chapitre dédié aux exportations et importations de vertex (voir page 12), et son architecture est illustrée par la figure 2.5. Les informations stockées sont des variables ou des structures, accessibles par leurs adresses en mémoire. Cette information appelée, "*item*", est contenue dans une cellule, et l'agencement de ces cellules procure au set les caractéristiques combinées d'une hash-table et d'une liste. L'adresse en mémoire centrale de chaque item est utilisée afin de définir la place de sa cellule dans le tableau, selon le codage suivant :

soient	indice(item)	= indice de item dans le tableau
	size	= taille du tableau du set
	addr(item)	= adresse de item

indice(item) = (long)(adrr(item)) % size

Lorsque plusieurs items sont associés au même indice, leurs cellules sont chaînées sous forme d'une liste, grâce au pointeur *next_c*. Ainsi, la place de chaque item est unique et il ne peut en aucun cas être stocké deux fois dans le même set. Cette organisation, appelée hash-table, est couplée à une structure de simple liste, puisqu'en même temps l'ensemble des cellules sont chaînées, grâce à un second pointeur, *next_r*. Enfin, seules les quatre fonctions utilisateur suivantes sont destinées à la gestion de cet outil.

```
BOOLEAN_t  SET_Add_Item(set_of_item,item)
     SET_c  set_of_item ;
     PNTR_t item        ;

BOOLEAN_t  SET_Rem_Item(set_of_item,item)
```

```
        SET_c  set_of_item  ;
        PNTR_t item         ;

BOOLEAN_t  SET_Retrieve_Item(set_of_item,item)
        SET_c  set_of_item  ;
        PNTR_t item         ;

ITERATOR_t  SET_Init_Next_Item(set_of_item)
        SET_c  set_of_item  ;
```

1. *SET_Add_Item(set_of_item,item)*
   Cette fonction ajoute l'élément *item*, dans le set *set_of_item*. L'indice, calculé à partir de l'adresse du pointeur *item*, définit une entrée unique dans le tableau où la cellule contenant *item* est insérée.

2. *SET_Rem_Item(set_of_item,item)*
   Cette fonction retire l'élément *item* du set *set_of_item*. L'indice, calculé à partir de l'adresse de *item*, procure un accès direct à la case de la hash-table où est stockée la cellule contenant *item*. La liste des éléments attachés à la même case est parcourue par le pointeur *next_c* et la cellule contenant *item* est otée de cette liste. En aucun cas, l'élément *item* n'est détruit physiquement de mémoire centrale.

3. *SET_Retrieve_Item(set_of_item,item)*
   Cette fonction vérifie si l'élément *item* est contenu dans le set *set_of_item*. La recherche de l'élément *item* est effectuée dans la liste chaînée par *next_c*, située à la case indicée par l'adresse de *item*. La structure de hash-table permet ainsi d'optimiser la recherche d'un élément. Cette organisation se révèle d'autant plus efficace que le volume de données à stocker est important.

4. *SET_Init_Next_Item(set_of_item)*
   Cette fonction permet de parcourir l'ensemble des éléments d'un set. Dans ce cas, l'utilisation du pointeur *next_r* permet d'optimiser cette opération car il évite automatiquement de parcourir les cases vides de la hash-table.

Notion de SETK :

Le setk possède une structure très proche de celle d'un set. Chaque information, appelée *item*, est stockée dans une cellule, en même temps qu'une clé, *key*, qui lui est associée. Toute clé est un entier de type long défini par l'utilisateur. Cette clé, et non plus l'adresse de *item*, est utilisée

afin de calculer la place de l'information dans la hash-table, selon la formule de "hashing" suivante.

soient
- indice(item) = indice de item dans le tableau
- size = taille du tableau du setk
- key(item) = clé associée àl'item

indice(item) = key(item) % size

Ce principe permet à l'utilisateur de gérer lui-même la position de ses informations dans une structure de type setk. Les quatre fonctions suivantes, similaires à celles du set, gèrent ce mécanisme :

```
BOOLEAN_t    SETK_Add_Item(setk_of_item,item,key)
     SETK_c  setk_of_item  ;
     PNTR_t  item          ;
     long    key           ;

BOOLEAN_t    SETK_Rem_Item(setk_of_item,item)
     SETK_c  setk_of_item  ;
     PNTR_t  item          ;

BOOLEAN_t    SETK_Retrieve_Item(setk_of_item,item)
     SETK_c  setk_of_item  ;
     PNTR_t  item ;

ITERATOR_t   SETK_Init_Next_Item(setk_of_item)
     SETK_c  setk_of_item  ;
```

L'existence de ces deux types de containers fut décisive lors de la conception de la structure du blakboard. En effet, le setk permet de classer facilement les informations suivant leur type, puisqu'elles sont exprimées sous la forme d'un token, dont l'attribut *type* sera utilisée comme clé du setk. Puis toutes les informations de même nature sont regroupées au sein d'un unique set, de telle sorte que la structure du blackboard constitue "un setk de set". Comme l'illustre la figure 4.2, ce type d'organisation reflète la possibilité de classer sous forme de modules les informations selon leur nature et leur nombre, mais aussi d'étendre cette espace de la mémoire centrale suivant la diversité des données.

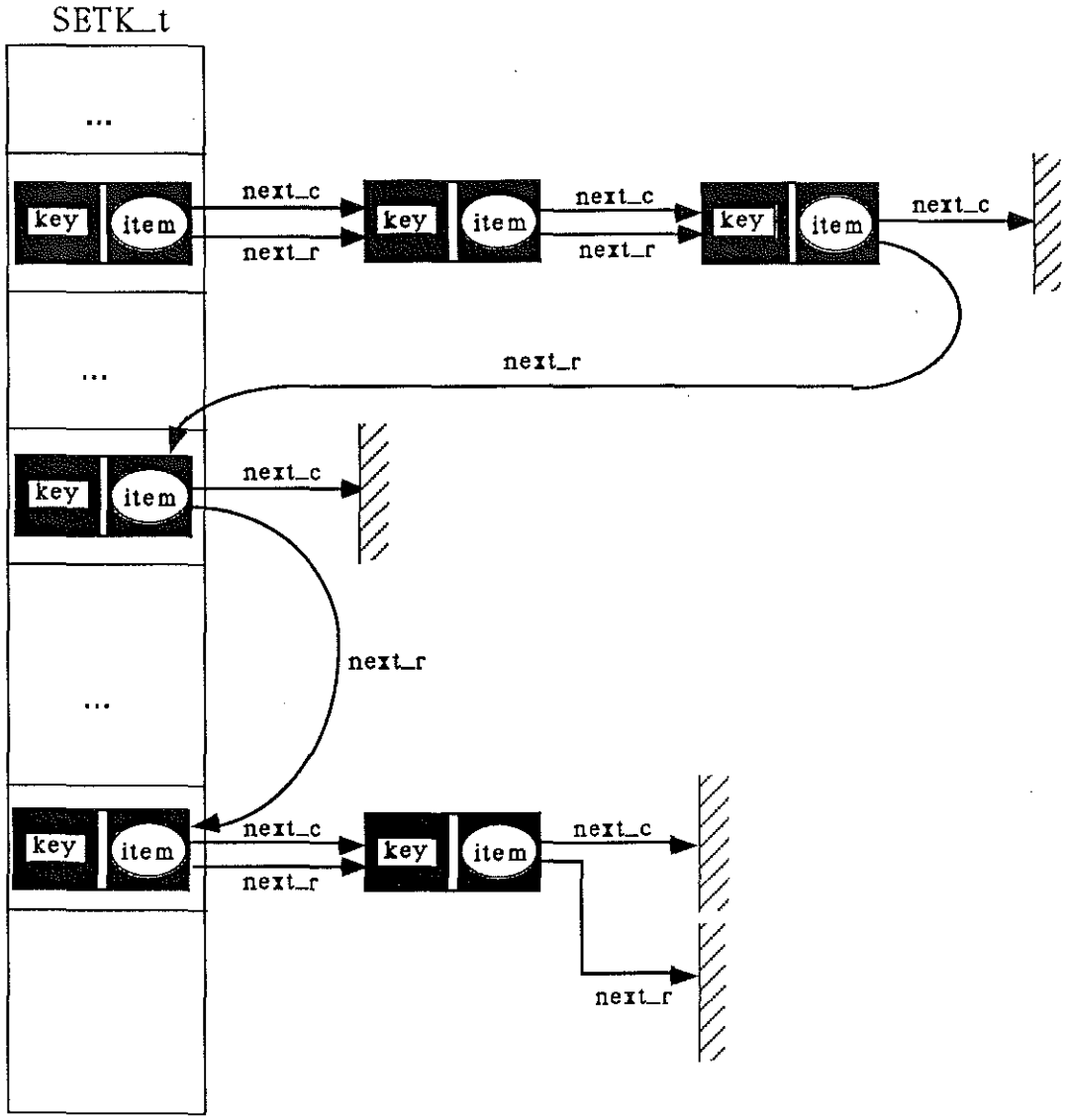

Figure 4.1 *schéma d'un setk.*

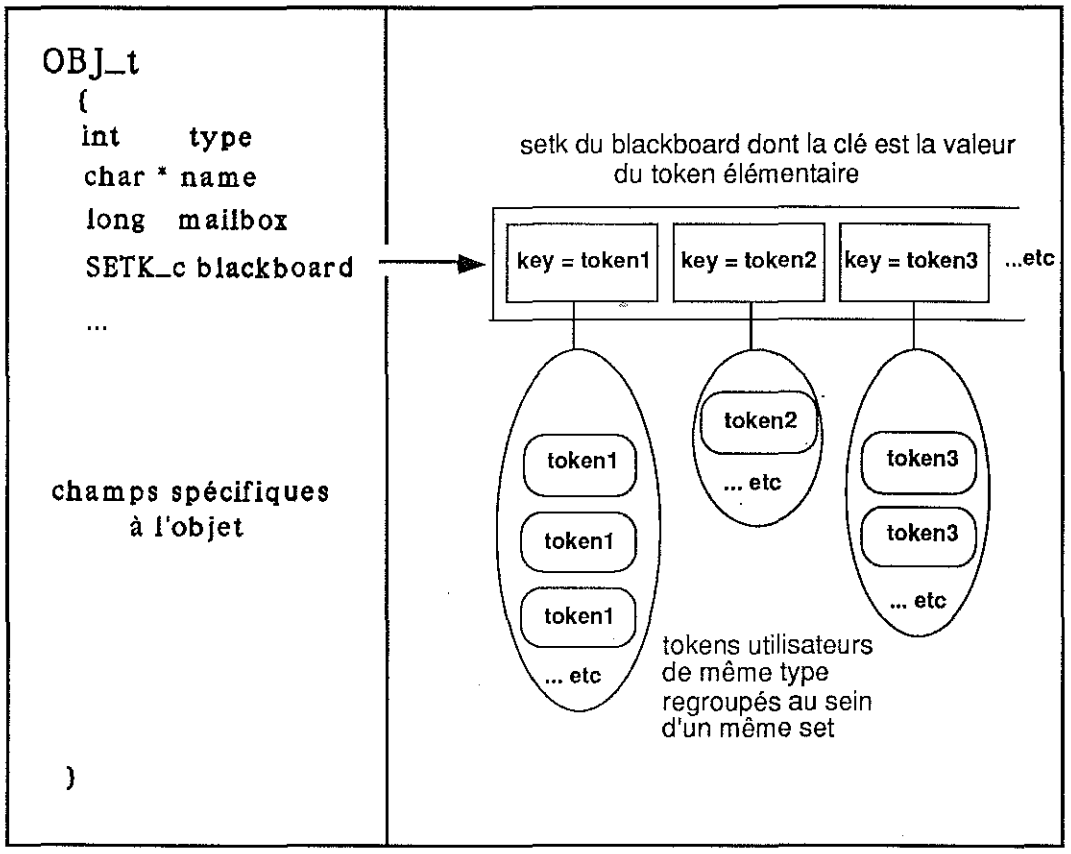

Figure 4.2 *schéma d'un blackboard.*

4.2.3 fonction implantées

Seules trois fonctions sont disponibles à l'utilisateur. Leur nombre, leur nom, et leur paramètre expriment la simplicité et la robutesse du mécanisme.

BOOLEAN_t GOBJ_Add_Token(gobj,token)

GOBJ_c	*gobj*
TOKEN_c	*token*

Cette fonction ajoute le token *token* dans le blackboard de l'objet *gobj*. Soit *type* la valeur du champ type de *token*. Si le blackboard de *gobj* est nul, il est initialisé, c'est-à-dire que l'espace réservé à sa hash-table est allouée en mémoire centrale. Ensuite, le set situé à l'indice *type* est recherché dans le setk du blackboard. S'il n'existe pas, une structure de set est initialisée et stockée dans le blackboard de *gobj* à l'indice *type*. Enfin, l'élément *token* est ajouté dans tous les cas de figures à ce set. En cas d'erreur, *FALSE* est retourné.

BOOLEAN_t GOBJ_Rem_Token(gobj,token)

GOBJ_c	*gobj*
TOKEN_c	*token*

Cette fonction suprime le token *token* du blackboard de l'objet *gobj*. Soient *type* la valeur du champ type de *token*, et *set*, le set stocké dans le blackboard de *gobj* à l'indice *type*. L'élément *token* est supprimé de ce set. Si le nombre d'éléments stockés dans ce set devient nul, il est lui-même supprimer du blackboard de *gobj*. De même, si le nombre de sets stockés dans le blackboard devient nul, l'espace alloué au setk est libéré et le champ blackboard de *gobj* est remis à nul. En cas d'erreur, *FALSE* est retourné. Je rappelle que les éléments contenus dans un set ou un setk, ne peuvent en aucun cas être dupliqués.

ITERATOR_t GOBJ_Init_Next_Token(gobj,token_type)

GOBJ_c	*gobj*
int	*token_type*

Cette fonction permet de parcourir l'ensemble des tokens de type *token_type* contenus dans le blackboard de l'objet *gobj*. Si *token_type* est nul, l'ensemble des tokens du blackboard de *gobj* sont restitués.

Remarque:
Contrairement à leurs sous-classes, les gobjs ou les tokens élémentaires sont

des notions abstraites, car ils ne représentent pas directement un objet concret et distinct. Une tsurf, ou une pline, constituent une sous-classe des gobjs, car leurs structures représentent une extension de la structure de gobj; de même un token utilisateur est une sous-classe du token élémentaire. La conversion automatique des types des paramètres d'une fonction, constitue une des propriétés les plus intéressantes du langage C, car elle permet de simuler un héritage, et procure ainsi la transparence du mécanisme de gestion du blackboard. Lorsque la définition des paramètres d'une fonction, fait référence à une classe d'objets, le compilateur C accepte que l'argument transmis soit un membre de l'une des ses sous-classes. Ce point est important car il signifie que des fonctions agissant sur une classe d'objets peuvent être utilisées par l'ensemble de leurs sous-classes.
Exemple : L'appel à la fonction GOBJ_Add_Token(tsurf, my_token) entraine automatiquement la conversion de la tsurf en gobj et la conversion de my_token en token.

4.2.4 commentaires

Avant d'aborder dans le détail la gestion des dépendances entre objets, je vous invite à prendre du recul et à réfléchir sur la vraie nature des structures adoptées: Considérons la structure d'une surface.

```
typedef struct TSURF_t
        {

        /* partie commune aux gobjs */

        int         type       ;
        char      * name       ;
        long        mailbox    ;
        SETK_c      blackboard ;

        /* partie specifique a la classe des tsurfs */

        int         nb_trgl    ;
        int         nb_atom    ;
        TRGL_c      p_trgl     ;
        ATOM_c      p_atom     ;
           ...
        VSET_c      p_vset     ;
        }
```

La structure de chaque macro-objet de la base de données GOCAD, est similaire à celle d'une surface et peut être décomposée en trois grandes par-

ties.

- **une partie commune à tous les objets et statique**
  Cette partie est constituée des quatre premiers champs de tout objet. Elle est commune à tous les gobjs et définie par convention explicitement en tête de toute structure d'un macro-objet. Cette partie constitue la zone commune et statique de tout objet.

- **une partie spécifique à une classe d'objets et statique**
  Cette partie contient les attributs spécifiques d'une classe d'objets, et définit la nature de l'objet. Ceux sont tous les champs qui sont définis à la suite de la structure de gobj.
  Exemple : L'attribut "nombre de triangles" est une notion directement et définitivement rattachée à la notion de surface. Dans le cas d'une pline, l'attribut "nombre de segments" lui serait substitué.

- **une partie spécifique à chaque objet et dynamique**
  Il s'agit du blackboard de chaque objet. En effet, le blackboard constitue à lui seul une partie à part de chaque objet, car il offre par nature une structure modulaire dont la taille peut évoluer au fur et à mesure des informations qui y sont stockées. Il constitue ainsi une zone dynamique, où certains attributs qui ne sont pas communs à la classe de l'objet peuvent être stockés. Le blackboard permet en définitive d'accroitre et d'ajuster la taille d'une structure afin d'accéder à un certains nombre d'informations qui ne furent pas prévus lors de la conception de la structure de la classe de cet objet. Ces informations sont spécifiques à une instance de la classe.

En résumé,

le mécanisme du blackboard permet détendre artificiellement et durant l'éxécution la structure d'un objet.

Ce principe soulève deux remarques:

- Le language informatique utilisé, c'est-à-dire le language C, influenca fortement la conception du blackboard, car il permet une grande souplesse dans la création et l'utilisation des structures, notamment grâce à l'opérateur "cast".

- Je désire mettre en garde le lecteur sur une utilisation excessive du blackboard. Cet outil offre la possibilité d'étendre une structure de

telle manière qu'il est tout à fait possible d'imaginer que la structure d'un objet se résume à son blackboard. C'est pourquoi, j'insiste sur le fait que l'utilisation du blackboard doit être réservée aux situations où elle s'avère strictement nécessaire.

4.3 Gestion des contraintes

Le mécanisme du blackboard fut conçu à l'origine pour implémenter de façon cohérente les contraintes *FCP*, *FCL* et *OTS* (cf pages 48 et 78), car leur conception et leur utilisation soulèvent deux types de problèmes:

1. **Comment accéder rapidement aux objets associés ?**
   Ces contraintes ont la particularités de mettre en jeu deux objets différents. Or, certaines actions appliquées sur un objet provoquent des répercutions sur l'autre, dont il est nécessaire de tenir compte. Le mécanisme doit donc proposer un moyen efficace de parcourir pour un objet donné, l'ensemble des objets qui lui sont liés.

2. **Comment tenir compte des modifications topologiques?**
   Lors de la présentation de la méthode *DSI* (cf page 40), nous avons annoncé que les contraintes étaient exprimées localement au niveau des atomes, sous forme d'une liste qui est définie explicitement dans la structure de chaque atome (cf page 5). Or, le système GOCAD met à la disposition de l'utilisateur, un certains nombre d'opérateurs qui modifie la topologie des objets et notamment le maillage des surfaces (cf [13] et [5]). Lorsqu'ils sont appliqués sur une surface, certains atomes sont créés, tandis que d'autres disparaîssent. Que se passe-t-il alors pour les contraintes qui étaient définis sur des atomes, destinés à être détruits?

Ces deux problèmes ont été résolus, grâce à une utilisation judicieuse du blackboard et des informations, concernant ces contraintes qui y sont stockées.

4.3.1 token émetteur et récepteur

Considérons l'exemple de la contrainte Fuzzy Control Point (cf page 48). Elle représente un lien entre une surface et un vset, néanmoins cette relation n'est pas symétrique, puisque la surface correspond à l'objet qui reçoit les contraintes, tandis que le vset est à leur origine; en d'autres termes la relation

est orientée d'un objet "maître" vers un objet "esclave". Cette remarque s'appliquant à l'ensemble des contraintes évoquées, nous avons cherché à définir un schéma d'organisation standard, tel un canevas, qui puisse être utilisé lors de la conception de nouvelles contraintes. Son principe repose sur la distinction entre deux catégories de tokens : **les tokens émetteurs** et **les tokens récepteurs**. Le token émetteur caractérise l'information contenue dans l'objet "maître", tandis que le token récepteur définit celle de l'objet "esclave".

Exemple: La contrainte de type *FCP* est associée à deux types de tokens; l'émetteur, appelé *TKE_FCP*, est enregistré dans le blackboard du vset et le récepteur, appelé *TKR_FCP*, dans celui de la surface. Ils sont définis successivement par les structures suivantes :

```
#define TKE_FCP 510

typedef struct TKE_FCP_t
        {
         int       type  ;
         TSURF_c   tsurf ;
         TKR_FCL_c mate  ;
        } TKE_FCL_t ;
```

```
#define TKR_FCP 520

typedef struct TKR_FCP_t
        {
         int       type       ;
         VSET_c    vset       ;
         TKE_FCL_c mate       ;
         SET_c     set_of_fcp ;
         } TKR_FCL_t ;
```

- *type* est un entier qui caractérise à la fois la nature de la relation et le fait que l'objet soit émetteur ou récepteur. Il est égal aux valeurs définies respectivement par les opérateurs "define" et cette valeur est utilisée comme clé du setk des blackboards.

- *tsurf* ou *vset* représente, selon le cas, l'objet associé. Dans le token émetteur, rattaché au vset, la valeur du champs *tsurf* procure un accès direct à la surface associée et vice versa.

- *mate* est un pointeur sur la structure du token qui lui est réciproque. Il procure un accès du token émetteur vers le token récepteur et vice versa.

- *set_of_fcp* est un pointeur sur un set, où sont répertoriées toutes les structures de type *FCP_INFO_t* (cf page 55), qui représente le coeur de l'information d'une contrainte *FCP*. On peut noter que ces données sont très facilement accessibles par le vset grâce au pointeur, *mate*; ceci évite la redondance de l'information.

Le canevas présenté consite à répartir l'information qui caractérise une contrainte, au sein de deux catégories de tokens, dont les attributs essentiels sont le type, l'objet associé et le token associé. Ce principe reste une proposition, et tout utilisateur peut organiser, de toute autre manière, le contenu du blackboard selon ses désirs.

Cette organisation apporte une solution au premier problème évoqué puisque le parcours des tokens stockés dans le blackboard d'un objet, définis automatiquement la liste des objets qui lui sont associés et la nature de leur relation.

La réponse au second problème est plus délicate et mérite d'ouvrir une parenthèse:

L'implémentation des contraintes ou le paradoxe de *DSI*!!

En effet, il apparait tout à fait incohérent de définir des contraintes au niveau des atomes, car ce principe induit automatiquement un lien étroit entre elles et le maillage d'une surface, c'est-à-dire entre la géométrie et la topologie. De façon logique, une surface doit être contrainte globalement sans se soucier de son maillage, sinon la moindre modification topologique entraine une incohérence totale du système. Par conséquent, il apparaît primordial de déconnecter les contraintes des maillages.

Ce problème est complexe car il est au coeur d'une mésentente sur le terme de contrainte. Plus précisemment, il est important de faire une distinction entre **l'origine** de la contrainte, et sa **localisation**.

- **l'origine de la contrainte**
  Elle réunit l'ensemble des informations qui caractérisent totalement cette contrainte, en dehors de toute référence au maillage. Cette information sera stockée dans le blackboard de la surface afin de la séparer totalement des atomes.
  Exemple: dans le cas de la contrainte de type *FCP*, il s'agit de l'ensemble des vertex du vset, des directions de tirs associées, et de la surface définie comme cible. Ces informations sont entièrements contenues dans le set, appelée *set_of_fcp*, qui contient les structures de type *FCP_INFO_t* (cf page 55).

- **la localisation de la contrainte**
  Elle représente l'ensemble des informations contenues dans la liste des contraintes de chaque atome de la surface. Elle sont utilisées lors du

processus d'interpolation par DSI, afin de définir la position dans l'espace de chaque atome.

Désormais le principe adopté consiste à diviser l'opération en trois étapes:

1. **Avant** : toute opération modifiant la topologie d'une surface, l'ensemble des contraintes locales sont détruites afin de déconnecter la contrainte du maillage, et ce pour tous les types de contraintes. Par contre les informations concernant leurs origines sont conservées dans le blackboard de la surface.
   Cette opération est réalisée par la fonction *GOBJ_Unset_Cnstr()*.

2. **Pendant** : l'opérateur modifie la topologie de la surface.

3. **Après** : Le système doit être capable de réinstaller des contraintes locales sur le nouveau maillage, grâce aux informations stockées dans le blackboard. Cette opération qui consiste à une nouvelle initialisation des contraintes, est réalisée par la fonction *GOBJ_Reset_Csntr()*.

Ce mécanisme soulève plusieurs remarques:

- Ce principe permet de tenir compte de toutes les opérations réalisées sur les maillages en protégeant la consistance du système. Certes le coût de ces fonctions est parfois élevé, mais il faut considérer que toute modification topologique est une modification importante de votre modèle.

- Désormais, le double stockage des informations concernant les contraintes est parfaitement justifiable. Le principe décrit ci-dessus met clairement en évidence l'utilité de conserver l'origine de la contrainte dans le blackboard. Inversement, les contraintes sont stockées localement au niveau des atomes, afin d'optimiser les performances de l'interpolateur DSI, puisque dans ce cas les atomes y ont un accès direct.

- On peut constater que la dissociation des contraintes, fut déjà exprimée à travers les mécanisme de relaxation automatique, décrits lors de la présentation des Fuzzy Control Points (cf page 54). En effet, la possibilité de décaler les contraintes d'un triangle sur son voisin, au cours du processus d'interpolation, démontre l'utilité de différencier l'origine de la contrainte de sa localisation.

Cette description du mécanisme ne répond que partiellement au problème évoqué, puisque nous n'avons pas abordé le sort des atomes et des triangles destinés à être détruits. Certes, cette destruction peut être prise en compte directement par l'opérateur qui modifie le maillage d'une surface, cependant le blackboard permet de façon élégante et efficace d'automatiser cette opération. Ce mécanisme reflète une seconde utilisation du blackboard, et introduit la notion de tokens temporaires.

4.3.2 token temporaire

Le principe repose sur le constat suivant:
Considérons une surface triangulée, et supposons que les atomes qui la constituent, avant et après toute modification topologique, soient conservés séparémment. Alors, par simple comparaison, il est très simple d'extraire, à la fois les nouveaux atomes apparus, et les atomes destinés à disparaitre. On peut noter que cette remarque concerne aussi les triangles.

La mise en œuvre de ce principe nécessite l'introduction d'un nouveau type de token, appelé, **token temporaire**, dont la structure est présentée ci-dessous:

```
#define TK_TMP 600

typedef struct TK_TMP_t
               {
                int    type             ; /*  = TK_TMP  */
                SET_c  set_of_new_atom  ;
                SET_c  set_of_new_trgl  ;
                SET_c  set_of_free_atom ;
                SET_c  set_of_free_trgl ;
               } TK_TMP_t ;

typedef struct TK_TMP_t * TK_TMP_c
```

- *type* est l'entier qui caractérise la notion de token temporaire.
- *set_of_new_atom* représente le set des nouveaux atomes apparus.
- *set_of_new_trgl* représente le set des nouveaux triangles apparus.
- *set_of_free_atom* représente le set des atomes destinés à disparaître.
- *set_of_free_trgl* représente le set des triangles destinés à disparaître.

L'enchainement des opérations est définit de la manière suivante:

```
1 -> GOBJ_Create_TK_TMP(tsurf)
2 -> GOBJ_Unset_Cnstr(tsurf)

3 -> GOBJ_Change_Topology(tsurf)

3 -> GOBJ_Update_TK_TMP(tsurf)
4 -> GOBJ_Reset_TK_TMP(tsurf)
5 -> GOBJ_Destroy_TK_TMP(tsurf)
```

Considérons une surface, *tsurf*, et un opérateur, appelé *GOBJ_Change_Topology()* qui modifie son maillage. Le mécanisme décrit la gestion des atomes mais il s'applique évidemment de façon identique sur les triangles.

- *GOBJ_Create_TK_TMP()*
  Avant de retirer les contraintes locales de la surface, cette fonction initialise une structure de type *TK_TMP_t*. Puis tous les atomes préexistants dans la surface sont respectivement répertoriés dans les set *set_of_free_atom*.

- *GOBJ_Update_TK_TMP()*
  Cette fonction est appelé, aprés la modification topologique, et avant de restituer les contraintes locales sur le nouveau maillage. A ce stade du procédé, tous les anciens atomes sont contenus dans le blackboard de la surface par l'intermédiaire du set *set_of_free_atom*, alors que tous les atomes qui constituent désormais la surface sont accessibles par la liste de ses atomes.
  La fonction *GOBJ_Update_TK_TMP()* a pour but de répertorier les nouveaux atomes créés et ceux qui sont appelés à disparaître. Lors du parcours des atomes constituant le nouveau maillage, deux cas se présentent:

  1. si cet atome existe dans le set *set_of_free_atom*, alors il est oté du set, puisqu'il ne doit pas disparaître.
  2. si cet atome n'existe pas dans le set *set_of_free_atom*, alors il est ajouté dans le set *set_of_new_atom*, puisque celà signifie qu'il vient d'être créé.

  Ce mécanisme est appliqué sur l'ensemble des atomes du nouveau maillage, et ainsi, au terme de cette opération, les set *set_of_free_atom*

et *set_of_new_atom* contiennent respectivement les atomes qui doivent être détruits et les nouveaux atomes.

- *GOBJ_Destroy_TK_TMP()*
  Cette fonction, appelée à la suite de la réinstallation des contraintes, se charge de détruire elle même les atomes et triangles qui sont contenus dans les sets *set_of_free_atom* et *set_of_free_trgl*. Elle peut ainsi libérer sans problème tous les attributs rattachés à cet atome. Puis, elle supprime le token *TK_TMP* de la structure du blackboard de la surface, avant de le détruire.

Ce principe offre un grand intérêt, puisque toute destruction physique d'un atome ou d'un triangle est automatisé et ainsi l'encapsulation du logiciel se trouve renforcée. Le source code est par conséquent simplifié, et si un nouvel opérateur, modifiant les maillages, est implémenté, le programmeur n'aura plus à se soucier du problème de libération de la mémoire, allouée aux triangles et aux atomes. On peut constater, que ce mécanisme est l'illustration d'un nouvel emploi du blackboard, puisque les tokens temporaires ne sont rattachés qu' à un seul objet, et que comme leur nom l'indique, leur existence est limité à l'exécution d'un mécanisme particulier.

Le mécanisme de gestion des contraintes s'articule donc autour de l'utilisation de plusieurs mécanismes, qui utilisent plusieurs catégories de tokens. Ce principe nous a parmis d'implémenter de façon cohérente l'ensemble des contraintes qui mettent en action plusieurs objets, car il disconnecte totalement les contraintes des maillages.

4.4 Conclusion

La conception d'un mécanisme capable de gérer des dépendances entre objets devenait de plus en plus impératif, au fur et à mesure que l'implémentation de nouvelles contraintes étaient envisagées. Elle représente, à mon sens, une des principales difficultées que j'ai rencontrée lors de mes travaux. La complémentarité du blackboard et des tokens ouvre une nouvelle voie au développement du projet GOCAD, car désormais l'ensemble des problèmes liés aux relations entre objets peuvent être abordées de façon concrète. A l'heure de cette rédaction, les applications basées sur le mécanisme du blackboard se multiplient au sein du logiciel, et je pense que sa simplicité et sa robustesse n'y sont pas étranger. Ces applications vont permettre au projet d'évoluer sans aucun doute vers un système de plus en plus "intelligent".

Chapitre

5

Mécanisme général de sauvegarde binaire

5.1 Exposé du problème

A tout moment, l'état d'un projet du système GOCAD, peut être sauvegardé. Cette sauvegarde consiste à stocker chaque objet du projet dans un fichier binaire, afin de conserver la trace du contenu de la base de données du système. Les adresses et les variables utiles y sont codées. La difficulté consiste à retrouver, après chargement l'état exact du projet. Notamment, il est important de restituer toutes les relations existantes entre les différents objets. Le but de chapitre est de vous présenter un mécanisme général de sauvegarde, spécialement adapté aux bases de données qui gèrent des dépendances entre objets.

D'un point de vue informatique, la mémoire centrale est un espace dédié au stockage de l'information, qui est organisée, comme l'illustre la figure 5.1, à la manière d'un tableau. Chaque information est contenue dans une cellule(bloc mémoire), qui est repérée par un indice appelé son **adresse**. Cette adresse est un entier et la taille des cellules s'adaptent à la taille de l'information stockée. Considérons une information contenue dans une cellule. Lorsque deux objets partagent cette information, ils accèdent tous les deux à la cellule considérée, par l'intermédiaire d'un **pointeur**, qui a

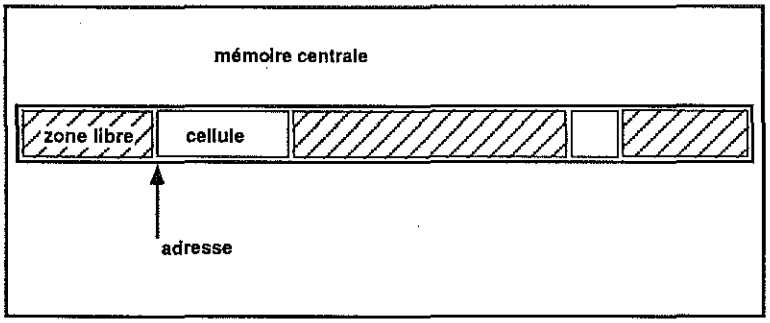

Figure 5.1 *shéma de la mémoire centrale.*

pour but de référencer l'adresse du début de la cellule. On parle dans ce cas d'une information "pointée" par deux objets différents.

Sauvegarder l'état d'un projet consiste à stocker sur fichiers l'image de la mémoire centrale. Inversement, son chargement, doit restituer dans sa totalité le contenu de cette mémoire centrale. Le mécanisme adopté doit, par conséquent, préserver l'unicité et la cohérence des adresses restituées.

Le codage des adresses : La façon la plus simple de coder un pointeur, consiste à stocker, sur fichier binaire, la valeur de son adresse. Cependant, ce principe n'assure pas la consistance d'un projet.

L'exemple de la figure 5.2 illustre ce problème. Lors de la première session un objet A d'adresse 948, fut créé, puis le projet sauvé. Lors de la deuxième session, sans installer A en mémoire centrale, un deuxième objet B est construit. Or, l'allocateur dynamique de mémoire a la possibilité d'attribuer l'adresse 948 à B. Le projet est sauvé une seconde fois. Désormais, le système ne peut recharger à la fois A et B sans faire apparaitre de conflit.

Si tous les objets étaient automatiquement et en totalité chargés en mémoire centrale, le projet serait cohérent. Cependant, cette contrainte n'est pas acceptable, car elle induit un encombrement mémoire trop important, et inutile dans la majorité des cas.

5.2 Principe adopté

Le principe consiste à associer à chaque projet un **compteur universel**. Ce compteur allouera à chaque pointeur sauvegardé un identificateur (**id**), c'est-à-dire un entier qui assurera l'unicité de cette valeur tout au long du

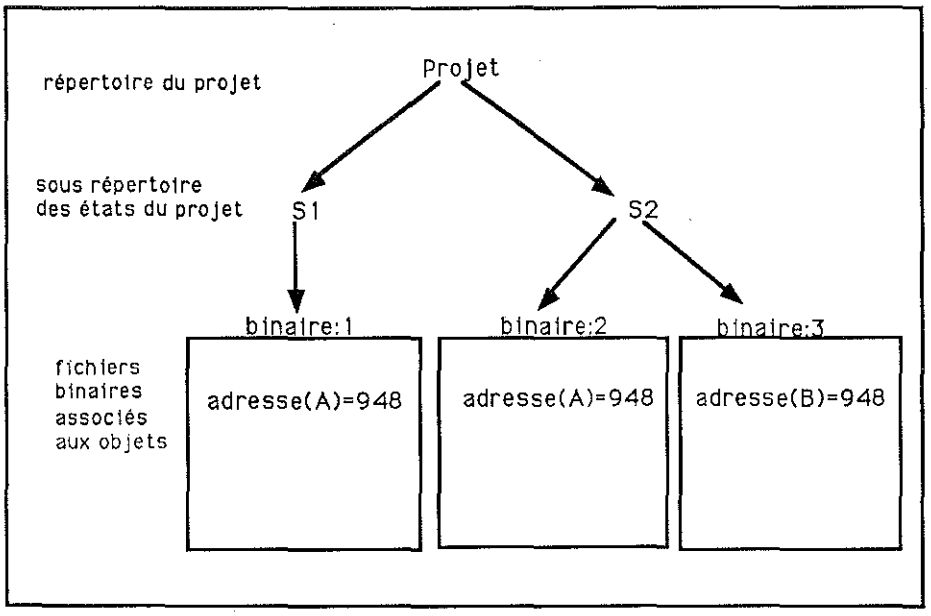

Figure 5.2 *architecture d'un projet.*

projet. Les relations entre les ids et les adresses sont enregistrées dans des structures de type id_htable, décrites ci-dessous.

Description des mécanismes de sauvegarde et de chargement:
Lors de la sauvegarde, deux cas se présentent :

1. **L'adresse n'est associée à aucun id:**
   Le compteur universel est incrémenté d'une unité, et un id est initialisé à cette valeur. Il sera désormais associé à cette adresse pour la durée totale du projet. Ce nouvel id est aussi stocké sur le fichier de sauvegarde.

2. **L'adresse est déjà associée à un id:**
   Cet id provient donc d'une sauvegarde précédente, dans laquelle il fut attribué à cette adresse.
   L'id est stocké sur le fichier de sauvegarde, afin de préserver l'unicité de la relation $(id, adresse)$.

De même lors du chargement deux cas sont à envisager :

1. **L'id lu sur le fichier de sauvegarde n'est associé à aucune adresse:**
   La place mémoire nécessaire est donc allouée, et la valeur de l'adresse retournée est associée à l'id.

2. **L'id est déjà associé à une adresse:**
   Cette adresse est conservée. Ainsi, le même id lu dans deux fichiers binaires différents appartenant au même projet est associé à une adresse unique en mémoire centrale, afin de respecter les dépendances entre objets, et par là-même la cohérence du projet.

Ces mécanismes garantissent, pour la durée totale du projet, l'unicité de la relation $(id, adresse)$. Seule la destruction de l'adresse peut rompre cette association. En effet, si au cours d'une session de travail, l'adresse *addr* liée à l'identificateur *id* est libérée de mémoire centrale, alors le couple $(addr, id)$ n'a plus de raisons d'exister. Désormais, l'allocateur dynamique peut redistribuer *addr* lors de la création d'un nouvel objet. Dans ce cas une nouvelle relation $(addr, id\_new)$ est crée, mais l'adresse est cette fois associée à un nouvel identificateur.

Remarque : Une adresse en mémoire assure l'unicité d'une variable durant une session de travail, c'est-à-dire lors de l'execution, tandis que l'id préserve son unicité durant toute l'existence du projet.

5.3 Notion de id_htable

Les relations entre id et adresse sont stockées dans des structures de type *id_htable_t*. Cette structure correspond à une hash-table, dans laquelle l'accès aux éléments du tableau est indexé par une clé.

```
typedef struct ID_HTABLE_t
{
  long                nb_item ;
  long                size    ;
  struct ID_NODE_t  ** pp_node ;
} ID_HTABLE_t ;

typedef ID_HTABLE_t * ID_HTABLE_c ;
```

- *nb_item* représente le nombre courant de valeurs stockées.
- *size* représente la taille de la hash-table, c'est-à-dire son nombre d'entrées.

- *pp_node* est un double pointeur de type *ID_NODE_t* , décrit ci-dessous.

La structure d' id_node est un container de pointeurs nommés *item*. Chaque container est indexé par une clé, *key*, qui définit sa position dans la hash-table. La valeur de l'index est calculée par la formule de hashing suivante:

$$\boxed{index(key) = key\%size\_of\_id\_htable}$$

Deux clés différentes peuvent avoir le même index.

```
typedef struct ID_NODE_t
{
  struct ID_NODE_t  * next  ;
  PNTR_t              item  ;
  unsigned long        key  ;
 } ID_NODE_t ;

typedef ID_NODE_t * ID_NODE_c ;
```

- *next* est un pointeur qui procure un accès à l'élément suivant de la liste des id_nodes regroupés sur la même case de la hash-table.
- *item* est un pointeur qui correspond au contenu stocké.
- *key* est la clé d'accès à ce niveau de la hash-table.

La gestion de la structure ID_HTABLE_t est assurée par les cinq fonctions suivantes:

- *ID_HTABLE_c ID_HTABLE_Create(size)*
  alloue en mémoire centrale une nouvelle structure de type id_htable, dont la taille est égale à l'entier *size*.
- *BOOLEAN_t ID_HTABLE_Add_Item(id_htable, item, key)*
  ajoute l'élément *item* dans la structure *id_htable*, relativement à la clé *key*.
- *BOOLEAN_t ID_HTABLE_Rem_Item(id_htable, key)*
  ôte l'élément indexé par *key* de la structure *id_htable*.
- *PNTR_t ID_HTABLE_Retrieve_Item(id_htable, key)*
  retourne l' élément indexé par *key* de la structure *id_htable*. *Null* est retourné si la case indexée est vide.

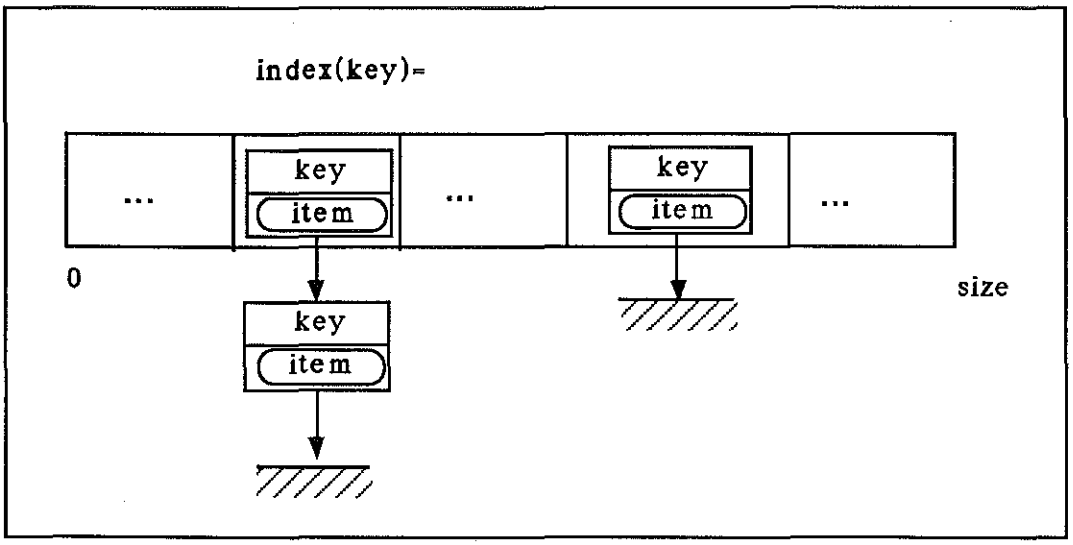

Figure 5.3 *schéma d'une structure ID_HTABLE_t.*

- *BOOLEAN_t ID_HTABLE_Destroy(id_htable)*
  libère de mémoire centrale, la place allouée pour la structure *id_htable*.

long size ; ID_HTABLE_c id_htable ; PNTR_t item ; long key ;

5.4 Le compteur universel

Le compteur universel correspond à une variable, de type long, nommée **universal_counter**. Chaque fois qu'une nouvelle adresse est sauvée, le système lui associe un id qui correspond à la valeur courante du compteur universel, incrémenté d'une unité. Cet id est attribué pour la durée totale du projet. La gestion du compteur universel est indissociable de celle du projet, dont le principe est explicité ci-dessous :

- *Création du projet*
  quand un projet est créé, la variable *universal_counter* est initialisée à zéro. Puis deux id_htables sont allouées. Elles enregistront les liens entre les adresses et les ids, pour la durée totale du projet.

  1. ID_HTABLE est une id_htable utilisée lors du chargement. Les *adresses* sont stockées en tant qu'*item*, et indexées par leurs *ids*

respectifs.

2. ADDR_HTABLE est une id_htable utilisée lors de la sauvegarde. les *ids* sont stockés en tant qu' *item*, et indexés par leurs *adresses* associées.

- *Sauvegarde du projet*
  A chaque sauvegarde, un fichier nommé **.universal** est créé. La valeur courante de *universal_counter* y est inscrite.

- *Chargement d'un projet*
  Lors du chargement d'un projet, le fichier **.universal** est ouvert, afin d'initialiser la valeur du compteur, *universal_counter*.

- *Fermeture du projet*
  Lors de la fermeture du projet les id_htables sont détruites de mémoire centrale.

5.5 Fonctions utilisateur implantées

Le fonctionnement du mécanisme de sauvegarde et de chargement, est géré par deux fonctions essentielles.

1. *long ADDR2ID(PNTR_t addr)*
   Cette fonction est utilisée lors de la sauvegarde pour associer une adresse à un id.

   Elle consulte la table ADDR_HTABLE. l'id stocké dans la case indexée par *addr* est retourné.
   Si cette case est vide, l'id retourné est initialisé à la valeur de *universal_counter*, incrémenté d'une unité.
   La nouvelle relation $(id, addr)$ est ensuite mise à jour:
   ce nouvel *id* est ajouté à la table ADDR_HTABLE, indexé par l'adresse *addr*, et de même l'adresse *addr* est stockée dans la table *id_htable*, indexée par *id*.

2. *PNTR_t Id2Addr(long size , long id)*
   Cette fonction associe un id à une adresse de la mémoire centrale.

   Elle est utilisée sous la forme ci-dessous:
   Soit x le type de la structure ou variable à allouer.

```
#define ID2ADDR( x , id )  ( x* ) Id2Addr( sizeof(x), id )
```

S'il existe une adresse indexée par *id* dans la table ID_HTABLE, alors cette adresse est retournée, sinon une allocation mémoire de taille égale à *size*, est effectuée. L'adresse ainsi obtenue est unique et sera associée de façon permanente à l'id.
Cette nouvelle relation $(id, addr)$ est de même mise à jour.

5.6 Optimisation du mécanisme

Imaginons deux objets A et B. Ils partagent une même structure *item*.
Lors du chargement, l'initialisation de cette structure peut être envisagée, selon deux méthodes :

1. L'ensemble de la structure *item* est sauvée, dans les deux fichiers binaires associés respectivement à A et B. L'utilisation des fonctions ID2ADDR() et ADDR2ID(), assure la cohérence du projet.

2. La structure *item* est le résultat d'une fonction *F()*,qui se charge de l'initialiser. Dans ce cas il est inutile de sauver son contenu.

Cette seconde approche nécessite l'utilisation de la nouvelle fonction, ADDR_Attached_to_Id().
Elle assure la cohérence des pointeurs alloués aux objets A et B.

- *BOOLEAN_t ADDR_Attached_to_Id(PNTR_t addr, long id)*

Son principe est détaillé dans l'exemple suivant:

Supposons que la structure *item* soit initialisée lors du chargement de A.
Deux cas se présentent :

1. **A est chargé avant B:**

```
A_Load()
 {
  ...
  fread( file_a , id)
  addr = A ->item =  F();
  ADDR_Attached_to_id( addr , id );
  ...
 }
```

```
B_Load()
 {
  ...
  fread( file_b , id )
  addr = B ->item = ID2ADDR( item\_t , id ) ;
  ...
 }
```

Lors de la sauvegarde les deux objets ont stockés le même id sur leur fichier respectif. La fonction *F()* initialise la structure *item* et retourne l'adresse *addr*. Puisque *id* n'est associée à aucune adresse, ADDR_Attached_to_Id() lui attribue *addr*. Ainsi lors du chargement de *B*, ID2ADDR() retrouve l'adresse *addr*, indexé par *id*.

2. *B* est chargé avant *A*:

```
B_Load()
 {
  ...
  fread( file_b , id )
  addr1 = B ->item = ID2ADDR( item\_t , id ) ;
  ...
 }

A_Load()
 {
  ...
  fread( file_a , id)
  addr2 = A ->item =  F();
  ADDR_Attached_to_id( addr2 , id );
  ...
 }
```

Lors du chargement de *B*, un première adresse *addr1* est associée à *id*. Lors du chargement de *A*, la fonction *F()* retourne une deuxième adresse *addr2*, où sont initialisés tous les champs de la structure *item*. La première action de la fonction ADDR_Attached_to_Id() consiste à vérifier si une adresse est déjà associée à *id*. Cette adresse est repréntée dans le cas présent par *addr1*. Afin que *B* et *A* partage effectivement le même *item*, cett adresse est reinitialisé à la valeur de *addr2*, par l'opération suivante:

```
&addr1 = addr2 ;
```

Ainsi, quelque soit l'ordre de chargement des objets, toute structure, même partagée, peut être initialisée par une fonction. Cette possibilité est d'autant plus intéressante que la structure est conséquente. Elle permet à l' utilisateur de réaliser un compromis entre ce qu'il désire stocker sur disque, et ce qui peut être recalculé, lors du chargement.

5.7 Conclusion

Grâce aux relations univoques $(identificateur, adresse)$, le système est capable de gérer de façon cohérente un projet, c'est-à-dire d'initialiser toutes les dépendances entre objets, en mémoire centrale. Il convient donc parfaitement aux besoins fixés par le système GOCAD.
Ce principe offre trois avantages :

- Sa gestion nécessite peu de fonctions, puisque seules trois d'entre elles sont accessibles à l'utilisateur.
- Le coût tant en vitesse d'exécution qu'en espace mémoire reste peu élevé. D'une part, la structure de l'id_htable permet des accès quasi direct aux données. D'autre part, l'espace mémoire nécessaire correspond à deux id_htables dont les tailles sont modulables. Les informations stockées sont soit des entiers de type long, soit des adresses; elles sont par conséquent peu coûteuses.
- le mécanisme est indépendant de l'allocateur dynamique de mémoire. En effet, c'est uniquement lors d'une sauvegarde ou d'un chargemnet, que les adresses sont associées à des ids. En dehors de ces opérations, le mécanisme est passif et ne ralentit en aucun cas l'allocation de mémoire.

Enfin, ce principe est général et indépendant des structures utilisées par le système GOCAD. Il peut être installé sur tout autre système, dès lors qu'il s'agit de sauvegarder une base de données sur disque magnétique.