

Rapport de stage

Implantation d'un algorithme de calcul d'enveloppes visuelles externes

Éric Colin de Verdière

Inria-Lorraine, 11 septembre - 11 décembre 1999

Introduction

Nous présentons ici le compte-rendu d'un stage qui s'est déroulé du 11 septembre au 11 décembre 1999, à l'Inria-Lorraine, à Nancy, sous la direction de Sylvain Petitjean et Sylvain Lazard. La problématique sur laquelle s'appuie le stage est celle de la compréhension d'une scène tridimensionnelle à partir d'images en deux dimensions, qui est une question importante en vision par ordinateur.

Considérons un objet dans l'espace, et un observateur astreint à se déplacer dans une certaine région en regardant les silhouettes de l'objet, c'est-à-dire l'image de l'objet projetée sur un écran.

Il se peut fort bien, dans certains cas, que l'observateur ne puisse pas reconstruire exactement l'objet à partir des informations dont il dispose. L'« enveloppe visuelle » de l'objet représente la meilleure approximation possible de cet objet quand l'observateur se déplace dans cette région.

L'objectif du stage est l'implantation d'un algorithme de calcul d'enveloppes visuelles pour une scène en deux dimensions quand la région d'observation entoure complètement l'objet et est située à une certaine distance de celui-ci. Cette approche se fonde sur un article de S. Petitjean [6].

Ce rapport est divisé en six parties. Après une présentation de l'équipe d'accueil, nous résumons la théorie des enveloppes visuelles nécessaire à la compréhension de l'algorithme ; les différentes étapes de celui-ci sont ensuite données. Puis viennent les détails concernant l'implantation : une petite présentation des logiciels utilisés, de nombreuses précisions d'implantation, et les résultats obtenus.

1 Présentation de l'équipe Isa du Loria

Ce stage s'est déroulé au sein de l'équipe Isa (Image, synthèse et analyse), au Loria (Laboratoire lorrain de recherche en informatique et ses applications, commun notamment à l'Inria et au CNRS), dans de très bonnes conditions – chaleureux merci à tous !

L'équipe Isa regroupe une trentaine de personnes et est constituée de plusieurs groupes de recherche qui travaillent sur les sujets suivants :

- la reconnaissance graphique : l'objectif principal est l'analyse de plans archi-

tecturaux afin de modéliser des bâtiments en trois dimensions, en évitant le plus possible l'intervention de l'utilisateur ;

- la réalité augmentée, dont le but est de pouvoir incruster sur des images d'une scène des objets de synthèse ; les applications sont nombreuses notamment en imagerie médicale ;
- la création d'images réalistes : comment simuler physiquement les phénomènes lumineux ? Des équations de radiosit  permettent de r pondre de fa on th orique   cette question, mais de nombreuses questions surviennent en pratique ;
- la reconstruction et la mod lisation de surfaces et de volumes, avec une application particuli re dans le domaine de la visualisation des couches g ologiques ;
- le graphisme haute performance, qui vise   optimiser l'affichage d'une sc ne par l'utilisation de machines   plusieurs processeurs et de mat riel acc l rant cet affichage ;
- la g om trie, notamment les probl mes de visibilit , mais aussi la mod lisation de sc nes   partir d' l ments de quadriques, au lieu de faces polygonales comme on le fait classiquement, et d'autres probl mes g om triques.

Ce stage s'inscrit donc dans ce dernier axe de recherche, puisque les enveloppes visuelles constituent un objet assez fondamental pour les probl mes de visibilit . Cependant, certaines des m thodes employ es concernent  galement des sujets voisins : les avanc es th oriques en visibilit  peuvent se montrer cruciales pour l'affichage rapide et r aliste de sc nes ; la question de la reconstruction d'objets se pose aussi dans le groupe s'int ressant   la mod lisation de couches du sous-sol   partir de certaines donn es.

Cela a aussi  t  l'occasion de se familiariser avec la biblioth que CGAL, qui tend    tre de plus en plus utilis e par la communaut  de g om trie algorithmique.

Cette  quipe travaille donc sur des sujets vari s, qui pourtant se compl tent assez bien : des approches utilis es par un groupe peuvent devenir une id e fructueuse pour des membres d'un autre groupe, comme l'ont montr  plusieurs discussions int ressantes, notamment lors du s minaire de l' quipe dans les Vosges.

2 Th orie des enveloppes visuelles

Nous donnons d'abord un aper u sur les enveloppes visuelles, en insistant sur les motivations qui ont conduit   l' tude de cet objet et en situant le sujet du stage par rapport aux diff rentes notions. Nous  tudions ensuite les enveloppes visuelles externes, qui sont un cas particulier d'enveloppes visuelles, avant d' tudier une propri t  qui nous servira pour l'implantation pratique de l'algorithme.

2.1 Introduction aux enveloppes visuelles

2.1.1 Les motivations de cette  tude

La compr hension d'une sc ne tridimensionnelle   partir d'images en deux dimensions est un probl me central en vision par ordinateur. De nombreuses approches

ont été proposées pour reconstruire le mieux possible un objet 3D à partir d'images de cet objet.

Ici, on s'intéresse à la notion de *silhouette* d'un objet S vu d'un point V , c'est-à-dire la vue qu'un observateur situé en V a de cet objet sur un écran (voir figure 1). Ainsi, l'observateur situé en V sait dans quelles directions sa vue est obstruée et dans quelles directions elle ne l'est pas, mais n'a pas de notion de distance. L'information de l'observateur est équivalente à la donnée du *cône de vue* de sommet V dont chaque demi-droite rencontre l'objet S .

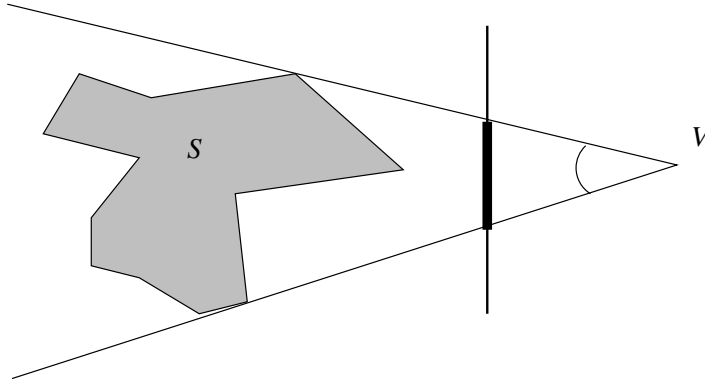


FIG. 1 – La silhouette d'un objet S vu du point V sur un écran et le cône de vue correspondant.

Ces notions de silhouette et de cône de vue sont importantes. Supposons qu'un observateur dispose de plusieurs vues d'un même objet, ce qui revient à se donner plusieurs cônes de vue d'origines V_1, V_2, \dots, V_n . La notion « la plus fine » qu'il a de l'objet S est alors l'intersection de ces cônes de vue; intuitivement, plus le nombre d'images est grand, meilleure est cette approximation de l'objet. Mais, même si le nombre de vues est infini, on ne peut pas toujours « reconstruire » exactement l'objet initial. Par exemple, considérons une sphère et une boule de même rayon. Un observateur se déplaçant autour de ces objets a les mêmes informations visuelles de ces objets et ne peut distinguer l'un de l'autre.

Plus généralement, on se donne une *région d'observation* R . L'*enveloppe visuelle* de l'objet S relativement à R – notée $VH(S, R)$ – est l'intersection des cônes de vue de S relativement à V quand le point V décrit R . Hors de $VH(S, R)$, l'observateur est certain qu'aucun point de l'objet n'est présent; $VH(S, R)$ est le plus petit ensemble vérifiant cela. Deux exemples d'enveloppes visuelles sont représentés sur la figure 2.

2.1.2 La définition

La théorie des enveloppes visuelles a été formellement étudiée par A. Laurentini [4]; les propositions des parties 2.1 et 2.2 ainsi que leurs démonstrations sont inspirées de cet article.

Définition 1 L'enveloppe visuelle $VH(S, R)$ d'un objet S borné dans \mathbb{R}^n , relativement à une région R , est l'ensemble des $P \in \mathbb{R}^n$ tels que : pour tout point $V \in R$, la demi-droite $[VP)$ rencontre S .

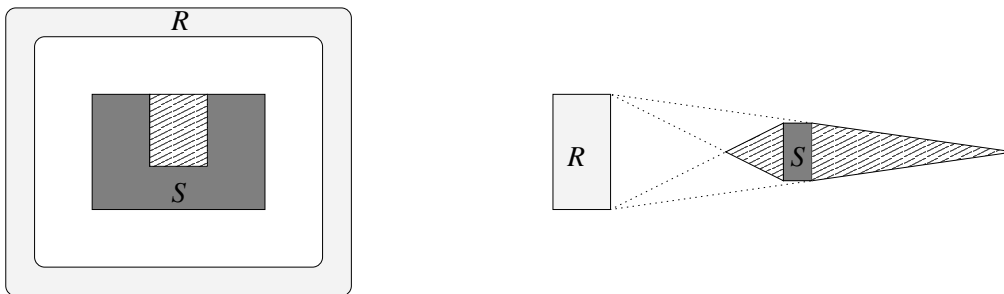


FIG. 2 – Deux exemples d’enveloppes visuelles. La partie en gris foncé correspond à l’objet S ; la région d’observation est R , en gris clair. L’enveloppe visuelle $VH(S, R)$ est constituée de l’objet lui-même et de la zone hachurée.

Proposition 2 Pour tout objet S , pour toutes régions R et R' , on a :

1. $S \subseteq VH(S, R)$: un objet est toujours contenu dans son enveloppe visuelle ;
2. si $R \subseteq R'$, $VH(S, R') \subseteq VH(S, R)$: plus une région d’observation est grande, meilleure est l’approximation de l’objet et plus petite est l’enveloppe visuelle.

2.1.3 L’objet du stage et ses liens avec les autres travaux

Dans ce stage, on s’intéresse aux enveloppes visuelles *externes*, c’est-à-dire au cas où la région d’observation est l’espace entier privé de l’enveloppe convexe $CH(S)$ de l’objet S – objet qui peut bien entendu ne pas être connexe. Nous en verrons des propriétés particulières à la section suivante. On définit également l’enveloppe visuelle *interne*, pour laquelle la région d’observation R est l’espace privé de l’objet S lui-même ; on peut étudier plus généralement l’enveloppe visuelle d’un objet S avec une région d’observation R quelconque.

L’objet du stage est l’implantation d’un algorithme de calcul d’enveloppes visuelles externes dans le plan (le calcul des enveloppes visuelles dans l’espace s’avère être nettement plus compliqué). Même si la motivation première de la définition de l’enveloppe visuelle est celle de la reconnaissance tridimensionnelle, l’étude dans le plan constitue une première approche de cet objet. D’autre part, elle peut se révéler utile à la conception d’un algorithme en trois dimensions, comme l’explique A. Laurentini [4].

L’algorithme implanté est fondé sur un article de S. Petitjean [6], qui a également proposé un algorithme de calcul des enveloppes visuelles internes et s’est intéressé à l’efficacité de la reconstruction d’objets 3D avec un nombre limité de points de vue (idée également abordée par A. Laurentini [5]).

On étudie maintenant le cas des enveloppes visuelles externes, qui est celui qui nous intéresse.

2.2 Enveloppes visuelles externes

Dans toute la suite, on pose donc $R = \mathbb{R}^n \setminus CH(S)$.

Notation. Nous noterons dans la suite $VH(S)$ l’enveloppe visuelle externe de S (c’est donc $VH(S, R)$). Cette notation se justifie par l’intérêt de ce choix précis de la région R , comme nous le verrons plus loin.

2.2.1 Une première propriété

Proposition 3 $VH(S) \subseteq CH(S)$.

Preuve. Soit $P \notin CH(S)$. On peut facilement trouver un point de vue V tel que la droite (VP) ne coupe pas S : tous les points se situant sur l'hyperplan H passant par P et normal à la droite (PQ) , où Q est le point de $CH(S)$ le plus proche de P , conviennent.

En effet, si tel n'est pas le cas (figure 3), soit $T \in S$ sur la droite (VP) . Le point R , projeté orthogonal de P sur (QT) , appartient à l'enveloppe convexe de S et est plus proche de P que ne l'est Q , ce qui est absurde. \square

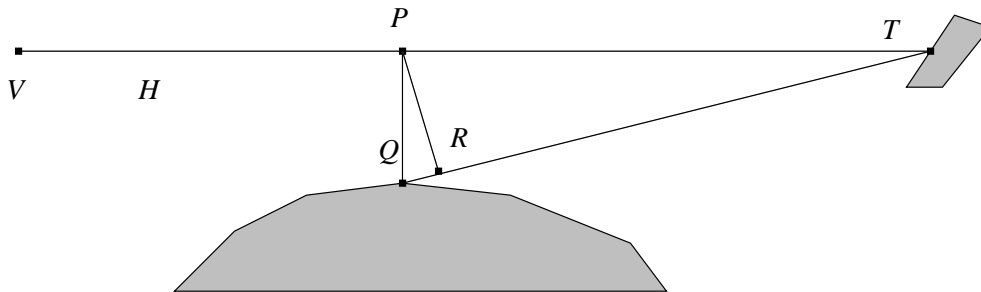


FIG. 3 – L'enveloppe visuelle externe d'un objet S est incluse dans son enveloppe convexe.

2.2.2 Une caractérisation simple

Nous donnons ici une caractérisation simple et fondamentale de l'enveloppe visuelle externe.

Proposition 4 Un point Q appartient à $VH(S)$ si et seulement si toute droite passant par Q contient au moins un point de S .

Preuve. S'il existe une droite passant par Q et ne rencontrant pas S , bien sûr Q n'appartient pas à $VH(S)$. Réciproquement, si Q n'appartient pas à $VH(S)$:

- ou bien $Q \notin CH(S)$, et dans ce cas il est évident qu'il y a une droite passant par Q ne rencontrant pas S (voir la preuve de la proposition 3);
- ou bien $Q \in CH(S)$. Mais on sait qu'il existe un point $V \notin CH(S)$ tel que $[VQ) \cap S = \emptyset$. Les faits suivants:
 - $(VQ) \cap CH(S)$ est connexe;
 - $Q \in CH(S)$;
 - $V \notin CH(S)$,

nous garantissent que la demi-droite issue de V et de direction opposée à Q ne rencontre pas $CH(S)$, donc à plus forte raison S .

Le résultat est donc acquis dans les deux cas. \square

2.2.3 L'intérêt de la définition

Voyons maintenant en quoi la notion d'enveloppe visuelle externe est une notion assez générale : à première vue, ce choix de la région R semble assez arbitraire. Cependant, dans un certain nombre de cas pratiques, une région d'observation R' satisfait aux deux conditions suivantes :

1. R' est à une certaine distance de S ;
2. R' entoure complètement l'objet S .

Nous allons voir que, dans de tels cas, $VH(S, R') = VH(S, R)$, c'est-à-dire que si l'observateur se déplace dans une région R' satisfaisant aux deux conditions suivantes, il a la même connaissance de l'objet S que s'il se déplaçait dans R . Plus précisément :

Théorème 5 *Soit R' une région*

1. *ne rencontrant pas l'enveloppe convexe de S ;*
2. *entourant complètement S (la composante connexe de $\mathbb{R}^n \setminus R'$ contenant S est bornée).*

Alors $VH(S, R) = VH(S, R')$.

Preuve. Comme $R' \subseteq R$, on a déjà $VH(S, R) \subseteq VH(S, R')$ (proposition 2) ; il suffit de montrer l'autre inclusion. Soit $Q \notin VH(S, R)$; prouvons : $Q \notin VH(S, R')$. Nous distinguons deux cas :

- si Q se trouve dans R' ou dans une autre composante connexe de $\mathbb{R}^n \setminus R'$ que celle qui contient S , c'est évidemment terminé : dans les deux cas, on peut trouver une demi-droite contenant Q et ne rencontrant pas S ;
- sinon, Q est dans une composante connexe bornée de $\mathbb{R}^n \setminus R'$. Utilisons la proposition 4 : il existe une droite D passant par Q et ne rencontrant pas S . De plus, D rencontre R' en un point V' . La demi-droite $[V'Q)$, incluse dans D , ne rencontre pas S , ce qui permet de conclure.

D'où le résultat. □

2.3 Enveloppes visuelles externes dans le plan

Terminons par une propriété très simple, spécifique au plan. L'algorithme que nous avons implanté prend en entrée des polygones disjoints dans le plan et calcule l'enveloppe visuelle externe de la réunion de ces polygones. Très souvent, il est plus pratique de manipuler des polygones convexes que des polygones quelconques. La proposition suivante prouve qu'on peut le faire ; mais commençons par un lemme :

Lemme 6 *Soit $S \subseteq \mathbb{R}^2$, S étant connexe, et D une droite. $D \cap CH(S) \neq \emptyset$ si et seulement si $D \cap S \neq \emptyset$.*

Preuve. Un des sens est évident. Pour l'autre, procédons par l'absurde : supposons que D ne rencontre pas S . S connexe se trouve donc tout entier strictement d'un côté de D ; il en est de même de $CH(S)$. \square

Cette propriété est évidemment spécifique au plan (considérer un polygone non convexe dans l'espace).

Proposition 7 Soient S_1, \dots, S_n des parties connexes de \mathbb{R}^2 . Alors :

$$VH\left(\bigcup_i S_i\right) = VH\left(\bigcup_i CH(S_i)\right).$$

Preuve. Cette proposition est évidente d'après le lemme 1. \square

En particulier, si $S \subseteq \mathbb{R}^2$ est connexe, alors $VH(S) = CH(S)$.

3 Description de l'algorithme

Voici maintenant une description de l'algorithme que nous avons implanté. Il calcule l'enveloppe visuelle externe d'un ensemble de polygones du plan dont les enveloppes convexes sont deux à deux disjointes. Dans cette présentation, la scène est supposée en position générale (trois sommets des polygones ne sont jamais alignés et les polygones sont d'intérieur non vide). Le traitement des cas dégénérés sera étudié plus loin.

3.1 Notion de nombre visuel

3.1.1 L'idée générale

Les algorithmes de calcul d'enveloppes visuelles externes que nous connaissons utilisent de façon cruciale le résultat de la proposition 4, qui constitue une caractérisation simple de cet objet. A. Laurentini [4] et S. Petitjean [6] utilisent tous deux la notion de *nombre visuel*. Tout repose sur le fait qu'un point n'appartient pas à l'enveloppe visuelle (externe) s'il existe des droites *libres* passant par ce point, c'est-à-dire ne coupant aucun polygone. Considérons un point V ; nous allons voir que les droites libres passant par V peuvent être regroupées en familles. Notons $n(V)$ le nombre de ces familles. En fait, le plan peut être divisé en régions polygonales telles que deux points V et V' d'une même région vérifient $n(V) = n(V')$. L'enveloppe visuelle est la réunion des régions pour lesquelles $n = 0$, *i.e.*, aucune droite libre ne passe par un point de ces régions.

3.1.2 La définition

Soient P_1, \dots, P_n des polygones disjoints du plan et $S = \bigcup_i P_i$. Une droite D du plan est dite *libre* si $D \cap S = \emptyset$.

Il est clair que les droites libres passant par un point V peuvent être regroupées en familles (cf. figure 4). Formellement, l'ensemble des droites passant par un point V s'identifie naturellement à $\mathbb{R} \cup \infty$ (par la fonction qui à une droite associe sa pente), qui lui-même s'identifie au cercle unité S^1 . L'ensemble des droites libres

passant par V correspond donc à une certaine partie de S^1 , dont les composantes connexes s'identifient à ce que nous appelons les familles.

Le *nombre visuel* du point V est le nombre de ces familles.

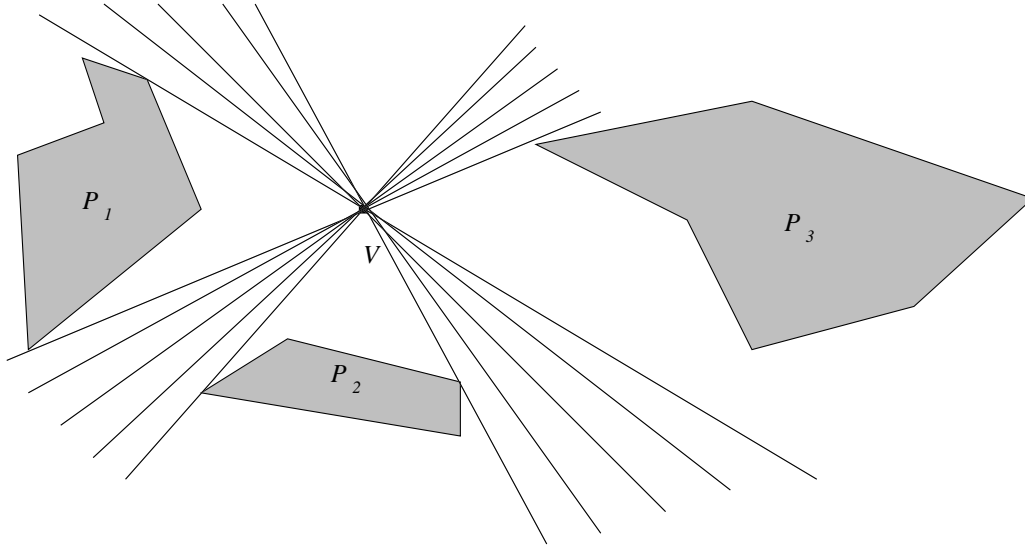


FIG. 4 – Par le point V passent deux familles de droites libres.

3.1.3 Une partition du plan

Nous l'avons vu, pour connaître l'enveloppe visuelle de S , il suffit de pouvoir connaître le nombre visuel en chaque point du plan. Pour cela, il faut nous demander dans quelles circonstances le nombre visuel peut varier. En fait, quand le point V se déplace, son nombre visuel ne varie que lorsqu'il franchit certains segments bien particuliers: ce sont les cas indiqués sur la figure 5. Les traits gras indiquent ces segments; de part et d'autre de ces segments, un $+$ et un $-$ indiquent de quel côté le nombre visuel est le plus grand. Les flèches prolongeant un segment indiquent que celui-ci est le support d'une droite libre (qui ne coupe aucun polygone).

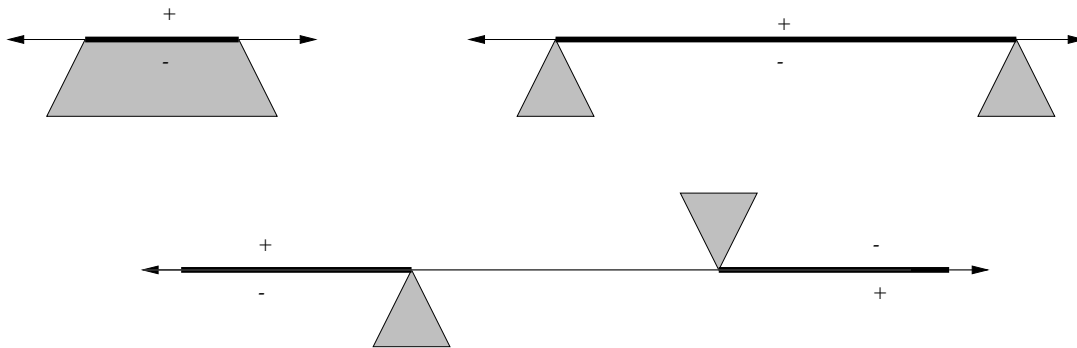


FIG. 5 – Les trois cas où le nombre visuel est différent de part et d'autre d'un segment.

La réunion de ces segments partitionne le plan en des zones de même nombre

visuel, c'est pourquoi nous appelons ces segments des *frontières*. Un exemple d'une telle partition est représenté sur la figure 6.

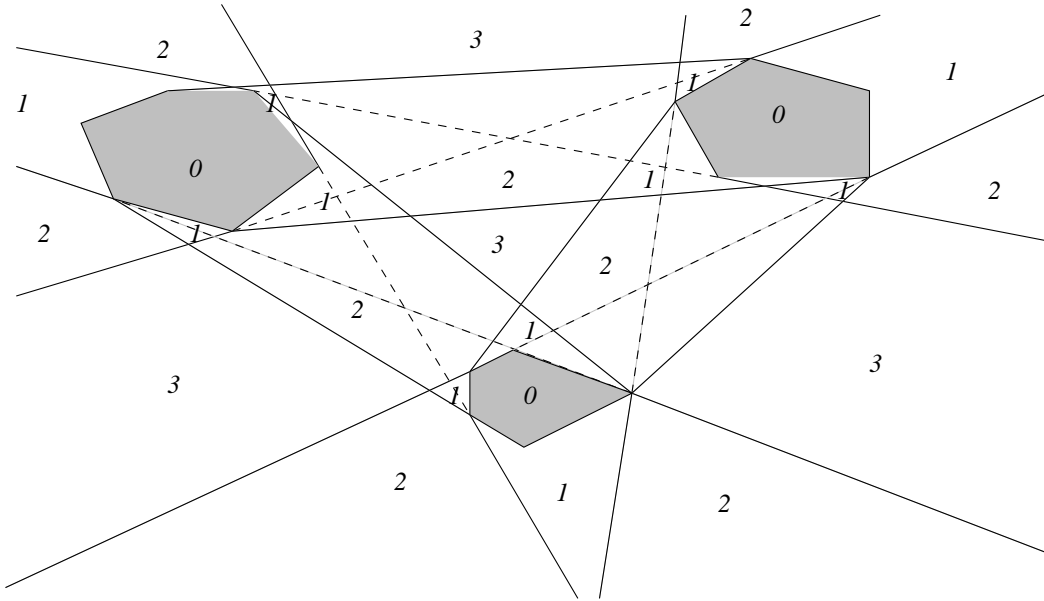


FIG. 6 – *Un exemple de partition du plan induite par les nombres visuels. Dans chaque région, le nombre visuel correspondant est indiqué. Les segments en traits discontinus ne sont présents que pour faciliter la compréhension du dessin et ne constituent pas des frontières.*

La généricité de la scène implique que le nombre visuel ne varie que d'une unité à chaque fois que l'on franchit une frontière (en d'autres termes, deux frontières ne se chevauchent jamais).

Notons bien que nous effectuons une partition du *plan* et que nous ne considérons pas d'office les polygones comme étant de nombre visuel 0 ; cela apparaîtra naturellement. Ainsi, tous les côtés des polygones n'ont pas à être insérés.

3.1.4 Quelques définitions

Nous utilisons maintenant la proposition 7 qui exprime que l'on peut remplacer les polygones par leurs enveloppes convexes (plus précisément, le lemme qui sert à démontrer cette proposition dit que les nombres visuels en tous points ne varient pas si l'on remplace les polygones par leurs enveloppes convexes). Cela permet d'indiquer simplement quelles sont les droites susceptibles de contenir des frontières. Pour voir cela, il nous faut d'abord donner quelques définitions :

Tout d'abord, nous appelons *droite tangente* à un polygone Q , toute droite *d'appui* de ce polygone, c'est-à-dire touchant ce polygone et telle que le polygone se trouve tout entier dans un demi-plan fermé défini par cette droite. Une droite *bitangente* est une droite tangente à deux polygones différents. La propriété intéressante est que, pour deux polygones convexes disjoints (et non dégénérés), il existe exactement quatre bitangentes : deux bitangentes *extérieures* (bitangentes qui laissent les deux polygones d'un même côté) et deux bitangentes *intérieures* (qui laissent un polygone de chaque côté), comme le montre la figure 7.

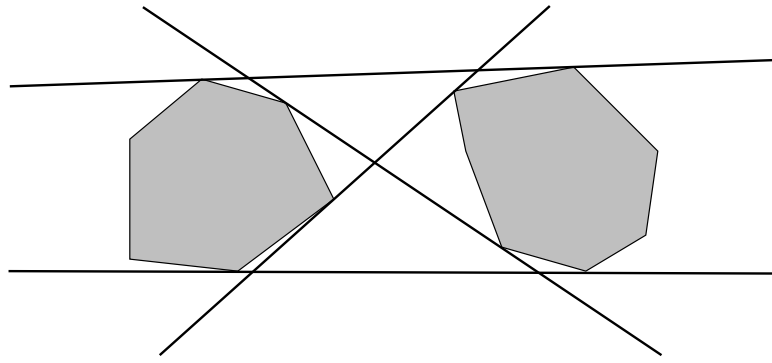


FIG. 7 – Les quatre bitangentes pour deux polygones convexes.

Revenons maintenant à la partition du plan induite par les nombres visuels, telle que nous l'avons décrite précédemment. Les définitions que nous venons de donner permettent de formaliser ce qui est indiqué par la figure 5 : les frontières de cette partition sont des portions de bitangentes libres et les côtés des polygones dont les supports sont libres.

L'efficacité de l'algorithme dépendra donc essentiellement de la façon dont on calculera les bitangentes libres de l'ensemble des polygones. Ceci est étudié dans la section suivante.

3.2 Calcul des frontières

3.2.1 Les différentes approches

Il y a bien entendu un algorithme simple (en $O(n^3)$, n étant le nombre de polygones – en supposant les polygones de complexité $O(1)$) qui permet de déterminer quelles sont les bitangentes libres (et d'en déduire immédiatement les frontières de la partition du plan décrite ci-dessus). C'est ce que fait A. Laurentini [4], en accélérant toutefois un peu le processus : on peut notamment éliminer un certain nombre de demi-droites qui sont en-dehors de l'enveloppe visuelle.

S. Petitjean [6] utilise un algorithme de M. Pocchiola et G. Vegter [7], qui calcule le graphe de visibilité des bitangentes, pour réduire la complexité de cette étape à $O((k+n) \log n)$, où k est la taille du graphe de visibilité et n le nombre de polygones, supposés de complexité bornée.

Nous nous fondons sur l'article de S. Petitjean pour implanter un algorithme de calcul d'enveloppes visuelles, mais n'utilisons pas l'algorithme de calcul du graphe de visibilité ; à la place, un *balayage rotationnel* autour de chaque polygone permet de calculer les bitangentes libres. Voyons ceci de plus près.

3.2.2 Le calcul des bitangentes libres

Les tangentes à un polygone. Concentrons notre attention sur un polygone P et cherchons à déterminer, parmi les bitangentes qui sont tangentes à P , celles qui sont libres. Nous ferons de même avec tous les polygones.

L'approche proposée est en quelque sorte un *balayage rotationnel*, par analogie avec la notion de balayage dans le plan tel que celui présenté dans [2] (dans ce même livre est présenté un algorithme de balayage rotationnel autour d'un point ; dans le

cas présent la situation est un peu différente puisque nous allons tourner autour du polygone P).

Nous allons considérer dans la suite des tangentes au polygone P . Convenons de les orienter de telle sorte qu'elles laissent le polygone P sur leur gauche (figure 8).

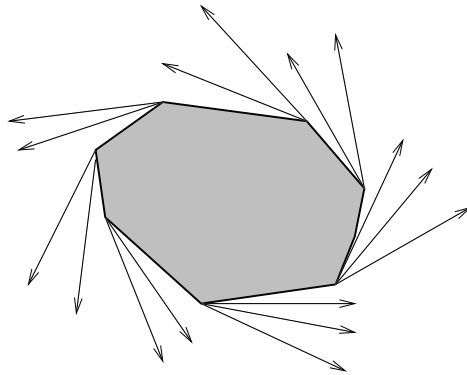


FIG. 8 – Une droite orientée tournant autour d'un polygone.

Le cas où il n'y a que deux polygones. Pour simplifier, supposons temporairement que la scène ne comporte qu'un autre polygone Q . Donnons-nous une tangente T à P quelconque. Comme P est convexe, il est désormais facile de savoir si T est libre : cela dépend de la direction de T (orientée pour laisser le polygone P à sa gauche) et des directions des quatre bitangentes à P et Q , orientées de la même façon (figure 9). Intuitivement, T est libre si et seulement si sa direction se situe hors des deux « intervalles » de S^1 limités par les directions de la bitangente 1 et de la bitangente 4 d'une part, et par celles de 3 et 2 d'autre part (en tournant dans le sens trigonométrique).

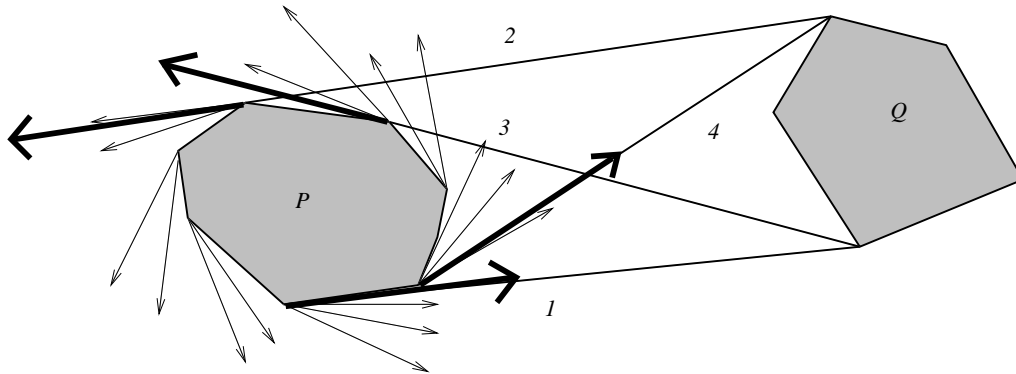


FIG. 9 – Déterminer si une droite tangente à P est libre. Les cas limites sont représentés en gras.

Le cas général. Voici maintenant le cas où il y a un nombre quelconque de polygones. On se concentre toujours sur un polygone particulier P . Pour Q décrivant l'ensemble des autres polygones, on calcule les intervalles comme indiqué dans le paragraphe précédent. Pour qu'une tangente à P soit libre vis-à-vis de tous les autres

polygones, il faut et il suffit que sa direction se situe hors de la *réunion* de tous les intervalles ainsi constitués. On peut appeler cet algorithme un « balayage » car on parcourt les bitangentes concernant P ordonnées selon leurs directions.

Donc, d'un point de vue algorithmique, il suffit de faire la réunion de certains intervalles dans S^1 . On se doute bien qu'il suffit plus ou moins pour cela de trier les extrémités des intervalles – nous verrons comment nous implantons cette réunion en détail plus loin. C'est pourquoi la complexité de ce balayage est $O(n \log n)$, où l'on a n polygones de complexité bornée. Procédant ainsi pour chaque polygone, nous obtenons une complexité égale à $O(n^2 \log n)$ pour le calcul des bitangentes libres.

Ceci étant effectué, il reste à calculer la partition (ou arrangement) du plan constituée de ces portions de bitangentes libres et des côtés des polygones qui sont libres et à parcourir la partition pour déterminer le nombre visuel de chaque cellule en suivant les indications de la figure 5. La complexité de cette étape est en $O((n + s) \log n)$, où s désigne le nombre de sommets de la carte planaire.

La complexité totale de l'algorithme dépend essentiellement de ces deux dernières phases : elle est en $O((n^2 + s) \log n)$. Nous en ferons une étude plus précise à la section 6.

3.3 Autre algorithme possible

Nous présentons ici, sans en donner les détails, un autre algorithme de calcul des enveloppes visuelles externes suggéré par M. Pocchiola, assez original en ce sens qu'il n'utilise pas la notion de nombre visuel comme le font les algorithmes de S. Petitjean [6] et A. Laurentini [4].

3.3.1 Le principe

L'idée est de calculer le complémentaire de l'enveloppe visuelle en tant que réunion des points qui sont sur des droites libres. Pour ce faire, il faut d'abord calculer l'ensemble des droites libres, en utilisant une paramétrisation bien choisie des droites dans le plan. Ensuite, il reste à effectuer la réunion des points situés sur ces droites libres.

Bien sûr, comme précédemment, on peut remplacer chaque polygone par son enveloppe convexe.

La paramétrisation choisie. Explicitons tout d'abord la paramétrisation choisie pour représenter des droites. Plus précisément, il s'agit d'une paramétrisation pour des droites orientées. On détermine une droite orientée dans le plan par deux paramètres : un angle $\theta \in [0, 2\pi[$ et une mesure algébrique r , comme représenté sur la figure 10. La valeur de θ est déterminée par la direction du vecteur porteur \vec{u} de la droite orientée, à laquelle on ajoute $\pi/2$; ce qui implique que r peut être positif ou négatif.

Les tangentes à un polygone convexe. Considérons maintenant un polygone convexe P et regardons les droites orientées tangentes à P qui laissent P à leur gauche. On peut commodément représenter ces droites dans un diagramme de droites orientées où θ est représenté en abscisse et r en ordonnée. On procède de même avec

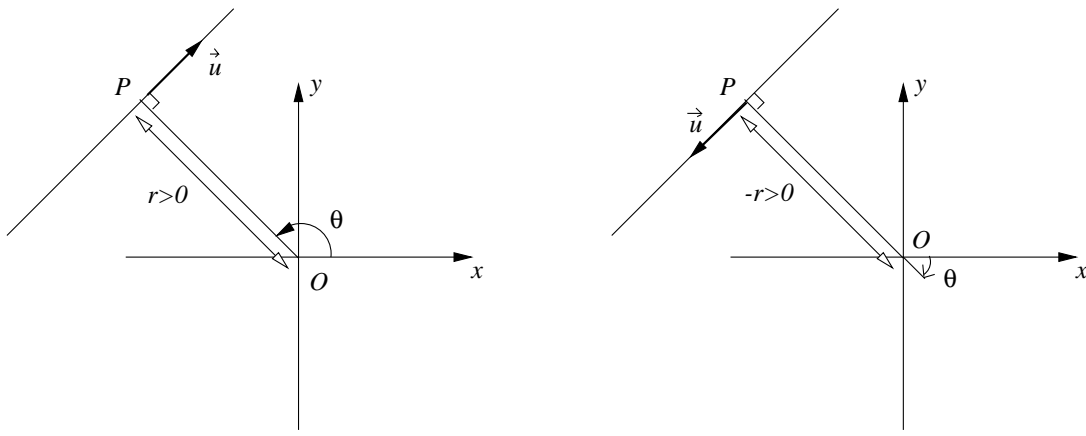


FIG. 10 – La paramétrisation d'une droite orientée dans le plan.

les tangentes qui laissent P à leur droite. On obtient un diagramme qui ressemble à la figure 11.

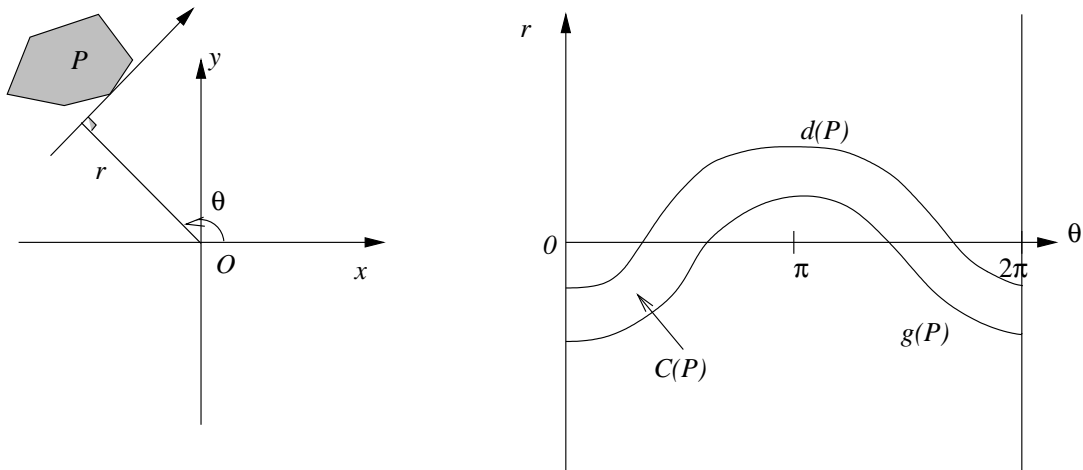


FIG. 11 – Le diagramme des tangentes à un polygone. La courbe $g(P)$ représente les tangentes orientées qui laissent le polygone à leur gauche, la courbe $d(P)$ celle qui laissent le polygone à leur droite.

L'intérêt de ce diagramme est que l'on se représente très bien les droites qui sont libres vis-à-vis du polygone P et celles qui le traversent : ces dernières sont naturellement représentées par les points du diagramme situés entre les deux courbes $d(P)$ et $g(P)$ (les courbes des tangentes orientées laissant respectivement P à leur droite et à leur gauche).

Nous pouvons d'ores et déjà noter la symétrie suivante : la transformation qui consiste à ajouter π et à changer r en son opposé fait correspondre les points de $g(P)$ aux points de $d(P)$. C'est normal, car c'est la transformation qui envoie une droite orientée sur la même droite d'orientation opposée.

Pour un polygone P , nous notons $C(P)$ l'ensemble des points du diagramme compris entre les courbes $d(P)$ et $g(P)$ correspondantes. Cela représente l'ensemble des droites qui ne sont pas libres vis-à-vis du polygone P .

Les droites libres du plan. Considérons maintenant une scène constituée de plusieurs polygones. Il est clair d'après ce qui précède que les droites non libres sont représentées dans notre diagramme par $C = \bigcup C(P)$ où P décrit l'ensemble des polygones. Deux courbes du diagramme des tangentes peuvent se couper : le point de concours représente une bitangente. Les droites libres du plan sont représentées dans ce diagramme par $L = ([0, 2\pi[\times \mathbb{R}) \setminus C$.

La réunion des points situés sur les droites libres. Il reste à effectuer la réunion des points des droites D quand D décrit l'ensemble L . Remarquons que L possède plusieurs composantes connexes. Ici, le terme « composante connexe » ne se réfère pas aux composantes connexes de L sur le diagramme, mais aux composantes connexes de L en identifiant tout point du diagramme de la forme (θ, r) avec le point de coordonnées $(\theta + \pi, -r)$, c'est-à-dire en identifiant une droite orientée avec la droite de direction opposée (on obtient une variété qui a la topologie d'un anneau de Möbius). Les composantes connexes ainsi définies correspondent aux « familles » de droites libres. Considérons une composante L_i et regardons l'ensemble Q_i des points qui sont sur une droite de L_i . Un exemple est représenté sur la figure 12.

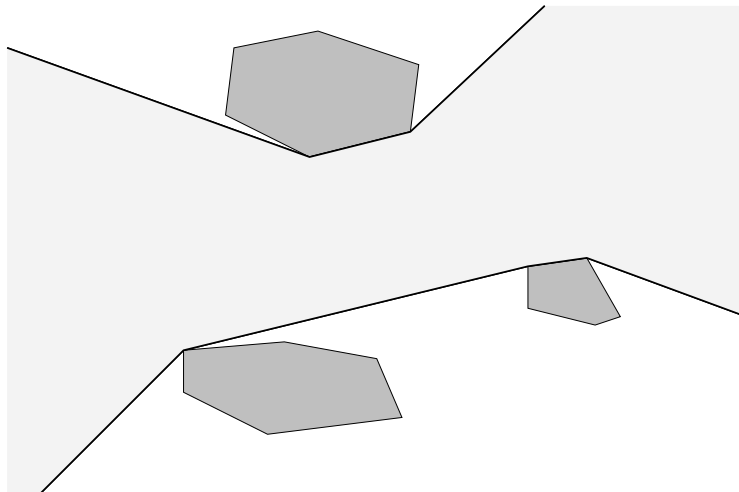


FIG. 12 – *En regardant plus particulièrement une composante connexe L_i de L , on s'aperçoit que les points qui appartiennent à une droite de L_i ont une forme caractéristique (ici représentée en gris clair).*

Nous remarquons que la région Q_i est limitée par des portions de bitangentes ou par des côtés libres de polygones. Ainsi, pour calculer l'ensemble Q_i à partir de L_i , il suffit de parcourir la frontière de L_i et de reporter les bitangentes (points de croisement entre deux courbes d ou g) situées sur cette frontière.

Ceci étant fait, on connaît l'ensemble des points situés sur au moins une droite libre, dont l'enveloppe visuelle externe est le complémentaire.

3.3.2 Les caractéristiques de cette méthode

Cette méthode présente de nets atouts sur le plan théorique :

- on ne calcule pas d'arrangement dans le plan. Le calcul d'une réunion de régions du plan est plus simple que le calcul de l'arrangement entier. Quand on

effectue la réunion des $C(P)$, toutes les bitangentes (intersections de courbes d ou g) ne sont pas calculées. Inévitablement, certaines bitangentes inutiles sont calculées, mais, en insérant au hasard les polygones, le nombre de bitangentes inutilement calculées ne devrait pas être trop grand ;

- la signification géométrique des régions Q_i apparaît bien ;
- cette méthode pourrait être adaptée pour qu'elle soit dynamique (on pourrait ajouter ou retirer des polygones sans tout devoir recalculer).

En revanche, cet algorithme paraît nettement plus complexe à implanter en pratique.

4 Aperçu de LEDA et CGAL

Voici une présentation des logiciels LEDA et CGAL rencontrés pendant ces trois mois de stage en vue de l'implantation de cet algorithme. Ce sont des bibliothèques d'algorithmes et de types de données, qui utilisent le langage C++ ; elles sont cependant assez différentes à la fois par la nature des outils proposés et par l'esprit général qui en régit l'organisation.

4.1 LEDA

LEDA (acronyme pour *Library of Efficient Data types and Algorithms*) est une bibliothèque originellement centrée sur la mise en place de types de données très généraux en informatique et en mathématiques et en l'utilisation de ces types de données par le biais d'algorithmes. L'objectif, selon la documentation même, est de « réduire la distance entre la recherche en algorithmique, son enseignement et l'implantation de ces algorithmes », en fournissant des outils directement opérationnels et utilisables rapidement.

De fait, LEDA n'est pas très compliqué à utiliser, peut-être au détriment d'une certaine souplesse que l'on a dans CGAL.

Nous présentons ici un petit tour d'horizon des diverses possibilités de LEDA :

4.1.1 Des nombres de précision arbitraire

Un atout important de LEDA est de pouvoir manipuler des nombres avec une précision arbitraire : on dispose ainsi du type `integer` (entiers arbitrairement longs) et des rationnels exacts, qui est le type de données effectivement utilisé par notre programme de calcul d'enveloppes visuelles. Tous les algorithmes de LEDA manipulant des nombres sont donc présents sous deux formes : une version utilisant les types de données internes et une version « exacte ».

4.1.2 Des algorithmes couramment employés en informatique

LEDA procure un grand nombre d'algorithmes qui facilitent la tâche du programmeur en gérant des structures de données abondamment étudiées dans la littérature mais dont l'implantation robuste nécessite un certain travail.

Ainsi, on y trouve plusieurs types de listes et tableaux, une gestion des ensembles, des arbres, des dictionnaires, des queues de priorité. . .

Sont également présentes des fonctions permettant de gérer avec simplicité tout ce qui concerne les entrées-sorties sur fichier, les chaînes et la gestion de la mémoire.

4.1.3 Les graphes

Les graphes constituent sans doute la structure la plus approfondie en LEDA, qui fournit des algorithmes d'une grande richesse permettant de les manipuler (graphes bipartis, algorithmes de flot maximal, arbres recouvrants, . . .).

De surcroît, les graphes ne sont pas seulement abordés en tant que structure combinatoire abstraite, mais l'aspect topologique (plongements dans le plan) y est également présent avec la notion de carte planaire et la définition des faces.

4.1.4 Des outils géométriques

On trouve les structures de données de base de la géométrie plane (par exemple les points, les segments, les polygones, les cercles) et des algorithmes les manipulant : enveloppes convexes, triangulations de Delaunay, dictionnaires en deux dimensions. . . En lien avec les graphes, il y a plusieurs implantations de calcul de l'arrangement constitué par un ensemble de segments (dans le plan) pouvant se couper.

Sont également présents des algorithmes de géométrie en trois dimensions.

4.1.5 Une interface graphique

Un des gros atouts de LEDA est de fournir aussi des fonctions permettant de programmer aisément une interface graphique en créant des boutons, des menus, des icônes. . . , ce qui serait relativement laborieux à faire soi-même en C++. La gestion des entrées-sorties (événements souris, affichage d'une fenêtre d'aide, etc.) est vraiment simple grâce à ces routines de base.

4.2 CGAL

CGAL (pour *Computational Geometry Algorithms Library*) est une bibliothèque d'algorithmes de géométrie. On le sait, il y a souvent un fossé entre la description théorique d'un algorithme géométrique et son implantation de façon robuste ; CGAL est en ce sens une « bibliothèque » qui regroupe ces algorithmes.

4.2.1 Quelques atouts

Voici les points forts de la philosophie de CGAL tels qu'ils sont indiqués dans la documentation :

- *robustesse* : la plupart du temps, un algorithme dépend du calcul de certains prédicats géométriques ; si ceux-ci ne sont pas calculés de façon exacte (par exemple à cause d'erreurs d'arrondi), les résultats obtenus peuvent être erronés. Pour cette raison, CGAL peut utiliser des types de données exacts (les rationnels de LEDA. . .), ou approchés (`float`, `double`, . . .) si l'utilisateur désire un calcul rapide au détriment de l'exactitude dans certains cas ;

- *souplesse*: CGAL se veut être le plus général possible, les applications pouvant être extrêmement variées. Dans cette optique, tous les types de données sont paramétrés par des *templates* C++. Par exemple, un algorithme d’enveloppe convexe pourra prendre en entrée des points dont les coordonnées sont cartésiennes ou homogènes, avec n’importe quel type de nombre, exact ou approché;
- *efficacité*: les algorithmes les plus efficaces sont implantés. Comme un algorithme donné ne peut pas être optimal dans chaque situation et selon tous les critères (espace mémoire, complexité dans les cas dégénérés...), plusieurs algorithmes sont parfois disponibles;
- *facilité d’utilisation*: l’utilisation parfois fastidieuse des *templates*, qui donne sa grande souplesse à CGAL, ne doit pas être un obstacle; certains types de données paramétrés sont très fréquemment utilisés (par exemple, le type *Point* en coordonnées cartésiennes et avec des nombres `double`) et on peut leur donner une abréviation.

4.2.2 Un tour d’horizon des modules de CGAL

Avant d’aborder plus en détail comment ces contraintes sont résolues, notamment la souplesse d’utilisation, on décrit ici sommairement les domaines concernant les algorithmes de CGAL. Sont présents :

- tous les types géométriques dans le plan ou dans l’espace (points, droites, polygones, ...) et les algorithmes de base qui les manipulent;
- les types de données concernant les cartes topologiques et les cartes planaires;
- des outils pour la géométrie dans le plan et/ou dans l’espace: enveloppes convexes, triangulations, fonctions d’optimisation géométrique (calcul du plus petit disque englobant, etc.), surfaces polyédriques;
- des fonctions de recherche multidimensionnelle assez générales;
- plusieurs outils pour travailler avec d’autres bibliothèques, des utilitaires (génération aléatoire, mesure du temps, etc.).

Il est possible d’utiliser CGAL avec l’interface graphique de LEDA: des fonctions CGAL permettent cela de façon assez transparente.

4.2.3 Des types de données paramétrables

Un des très grands mérites de CGAL est sa souplesse, obtenue par le fait que tous les types de données sont paramétrables. Comment cela est-il possible en pratique?

C++, un langage orienté objet. Tout repose sur le fait que C++ est un langage orienté objet qui permet de s’abstraire des types de données quand cela s’avère nécessaire. Par exemple, un algorithme de calcul d’enveloppes convexes ne doit pas avoir besoin de savoir si les coordonnées des points sont stockées en coordonnées homogènes ou cartésiennes, la fonction est essentiellement la même.

Les *templates* permettent de s'affranchir de cette contrainte. Un point dans le plan, représenté en coordonnées cartésiennes par deux éléments `double`, a le type suivant : `Point_2 < Cartesian < double > >`. Tous les algorithmes manipulant des points dans le plan, `Point_2`, peuvent utiliser ce type tout comme des points en coordonnées homogènes avec le type entier exact de LEDA.

L'utilisation de STL. STL (pour *Standard Template Library*) constitue un cadre de programmation en C++ ; l'idée est de pouvoir écrire des algorithmes « universels » qui fonctionnent de la même façon avec toutes les structures de données. Ainsi, un algorithme de tri pourra être le même que l'on stocke les données dans un tableau ou dans une liste.

La notion de base dans STL est celle d'itérateur. Une liste ou un tableau est constitué d'itérateurs. Si on note `iter` un itérateur, `iter++` donne l'itérateur suivant du tableau ou de la liste et `*iter` renvoie la valeur du tableau correspondant à cet itérateur. On voit bien avec cet exemple-ci l'analogie qu'il y a entre un itérateur et un pointeur ; la notation `*` est la même, simplement un itérateur permet de se déplacer dans la structure de données.

À noter que LEDA peut utiliser STL, mais son usage ne se révèle vraiment indispensable qu'avec CGAL.

Les Traits. Le caractère paramétrable de CGAL ne s'arrête pas aux structures de données. Souvent, un algorithme géométrique dépend du calcul de certains prédicats. L'implantation de ces calculs peut dépendre de beaucoup de choses : par exemple du type de données, d'un choix de l'utilisateur (un algorithme peut se révéler plus efficace qu'un autre dans certaines conditions).

Par exemple, le module CGAL *Arrangement* qui s'intéresse aux arrangements dans le plan ne dépend pas du type de courbes auquel on s'intéresse : une classe *Traits* fournie par l'utilisateur est chargée d'effectuer les opérations élémentaires sur les courbes (notamment calculer les intersections entre deux courbes, découper une courbe en courbes x -monotones, ...) et l'algorithme de calcul de l'arrangement fait appel à ces fonctions dans la classe de *Traits* quand cela est nécessaire. Dans le module *Arrangement* sont fournies quelques classes de *Traits*, notamment pour le cas des segments ; si l'utilisateur veut calculer un arrangement constitué par des courbes plus complexes, des coniques par exemple, il lui suffira d'écrire sa propre classe de *Traits*.

5 Implantation

Nous présentons ici, pas à pas, les détails d'implantation du programme ; en résumé, cette partie contient toutes les petites difficultés non évoquées dans la section 3 par souci de clarté et de concision.

Avant tout calcul, on s'empresse d'oublier les polygones en les remplaçant avantageusement par leurs enveloppes convexes, en vertu de la proposition 7 et de ce qui a été dit avant.

L'entrée de l'utilisateur est également vérifiée : en l'occurrence, il s'agit de s'assurer que les enveloppes convexes des polygones sont deux à deux disjointes. Ensuite, on peut vraiment entrer dans le vif du sujet...

Par souci de clarté, on suppose d'abord tous les polygones en position générale (trois sommets distincts ne sont jamais alignés et les polygones sont d'intérieur non vide), les modifications à apporter pour les cas dégénérés seront détaillées plus loin.

5.1 Calcul des bitangentes

L'algorithme utilisé pour le calcul des bitangentes de deux polygones convexes disjoints est linéaire en la complexité des deux polygones. Son fondement réside en des rotations successives autour de chacun d'eux.

Il s'inspire de l'ouvrage d'H. Edelsbrunner [3], pp. 146-147, où l'auteur calcule les bitangentes extérieures seulement. Nous résumons d'abord ce texte, avant de donner l'implantation choisie, un peu différente.

5.1.1 L'idée d'Edelsbrunner

Plaçons-nous dans un repère orthonormé; supposons que les deux polygones (convexes) P et Q soient séparés par la droite verticale $x = 0$ (les points de P étant d'abscisse strictement négative, les points de Q d'abscisse strictement positive).

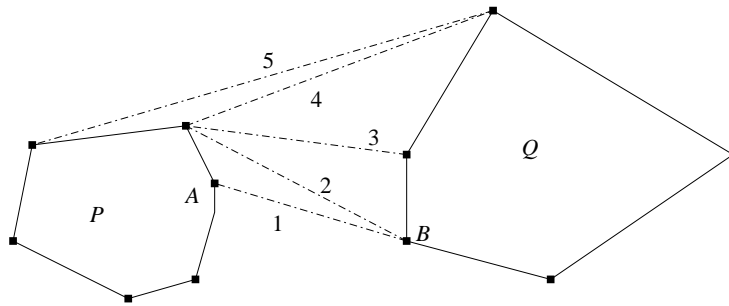


FIG. 13 – L'algorithme d'Edelsbrunner.

Pour calculer une bitangente extérieure, considérons deux points courants A et B situés sur P et Q respectivement. Initialement, choisissons-les de telle sorte qu'ils se voient » (*i.e.*, $]AB[$ ne rencontre aucun des polygones). Convenons d'ordonner les sommets de P et Q dans le sens trigonométrique. La figure 13 illustre le déroulement de l'algorithme, qui est le suivant :

Tant que $\text{succ}(A)$ ou $\text{pred}(B)$ se trouve au-dessus de (AB) :

- si $\text{succ}(A)$ se trouve au-dessus de (AB) , faire $A = \text{succ}(A)$;
- sinon, faire $B = \text{pred}(B)$.

Intuitivement, on « tourne » tant qu'il le faut autour de Q , en ajustant à chaque fois le point courant de P de telle sorte que la droite (AB) soit toujours tangente à P . À chaque pas, les points A et B se « voient ».

La seule difficulté est de prouver que l'algorithme termine (ce qui n'est pas présenté dans [3]...). Prouvons d'abord qu'on ne peut pas faire un tour complet autour du polygone P . Pour cela, imaginons que les points A et B varient de façon continue au cours de l'algorithme (*i.e.*, qu'ils longent les arêtes au lieu de sauter d'un sommet

à l'autre). La droite (AB) est donc toujours tangente au polygone P et $[AB)$ est toujours dirigée vers la droite, ce qui rend impossible le fait de faire un tour complet autour de P .

Maintenant, il est également impossible de faire un tour complet autour du polygone Q . En effet, si tel était le cas, on retrouve une position des points A et B précédemment rencontrée. C'est impossible. En effet, regardons la droite $x = 0$ séparant les deux polygones : on constate qu'à chaque déplacement de A ou B , le point d'intersection de la droite (AB) avec ce segment a une ordonnée qui croît strictement, ce qui rend toute « boucle » impossible.

On en déduit bien que l'algorithme est de complexité linéaire en la taille des polygones.

Celui-ci s'adapte sans difficulté pour obtenir des bitangentes intérieures ; pour la preuve, il suffit de considérer une droite verticale d'abscisse plus petite que toutes celles des sommets des polygones, au lieu de regarder $x = 0$.

5.1.2 L'implantation choisie

On utilise toutefois une implantation un peu différente de l'approche que nous venons de présenter. L'inconvénient de celle-ci est en effet que l'on doit disposer initialement de deux points « qui se voient ». Cette petite complication est contournée dans l'algorithme ci-dessous.

Reprenons l'approche décrite précédemment. Nous constatons en fait qu'il est inutile d'ajuster le point courant de P à chaque fois que l'on passe d'un sommet de Q à son voisin : on peut d'un coup déplacer le point Q tant que cela est nécessaire, puis ajuster la position du point P , et ainsi de suite. Cette idée simple va prendre un sens plus précis dans les lignes qui suivent.

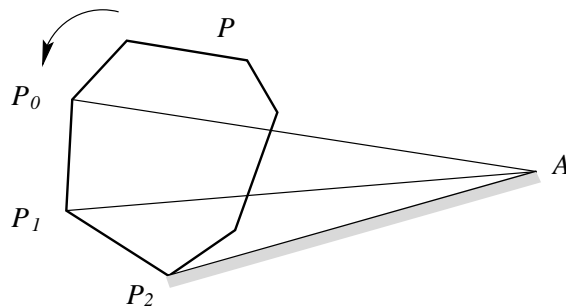


FIG. 14 – Une rotation : c'est la procédure qui, étant donné un polygone convexe P et un point A situé à l'extérieur de P , trouve une tangente au polygone P passant par A .

On définit d'abord une fonction de *rotation* autour d'un polygone P qui permet, étant donné un point A situé à l'extérieur de celui-ci, de tourner autour de P jusqu'à obtenir une droite passant par A et tangente au polygone (cf. figure 14).

En entrée, on spécifie :

- le polygone P ;
- le sommet de départ P_0 ;

- le point A ;
- le sens de rotation (qui ne change pas le résultat de l’algorithme mais permet d’obtenir la complexité linéaire indiquée précédemment, en tournant dans le bon sens) ;
- la position voulue du polygone par rapport à la tangente (le polygone doit être du côté opposé à la partie grisée, sur la figure).

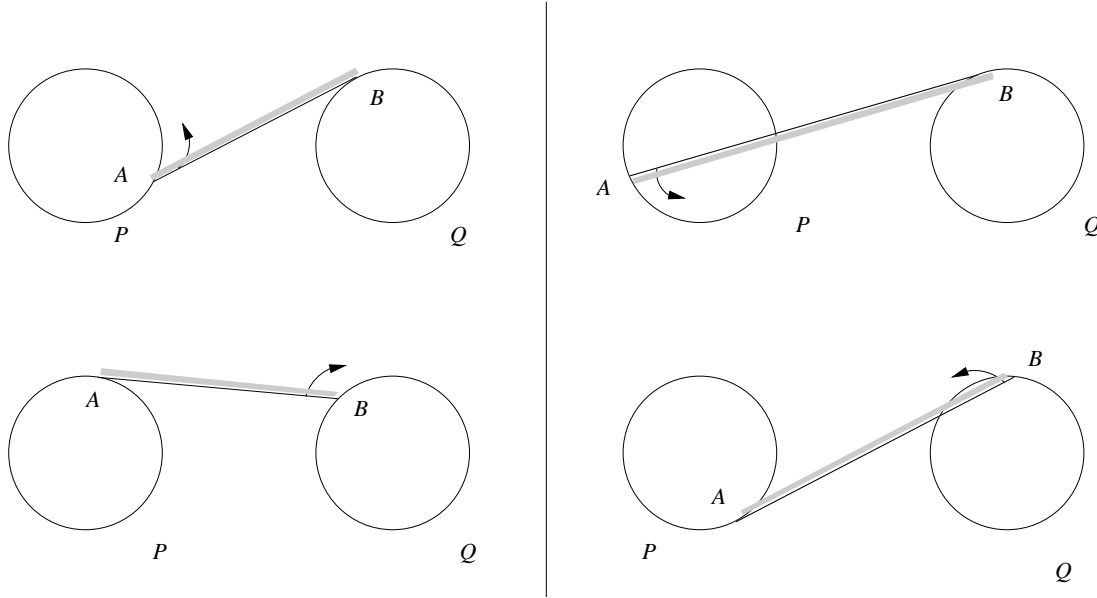


FIG. 15 – *La succession des rotations pour obtenir les bitangentes. À gauche : pour une bitangente extérieure ; à droite : pour une bitangente intérieure.*

L’algorithme est alors le suivant. L’idée est alors de prendre deux points courants A et B sur les polygones P et Q et d’effectuer des rotations : d’abord en bougeant A , relativement au point B , puis en bougeant B , relativement au point A , et ainsi de suite, comme le montre la figure 15. On choisit bien sûr les sens de rotations et les positions des polygones par rapport aux tangentes en fonction du type de bitangente voulue (intérieure ou extérieure).

Plus précisément, partons de deux sommets quelconques A et B sur P et Q respectivement. Après les deux premières rotations, l’invariant suivant est obtenu et conservé (voir la figure 15) :

- dans le cas des bitangentes extérieures : $]AB[$ ne rencontre aucun des deux polygones P ou Q ;
- dans le cas des bitangentes intérieures :
 - après avoir bougé A , la demi-droite incluse dans (AB) , issue de B et ne passant pas par A , ne rencontre pas Q ;
 - de même en échangeant les rôles de A et B .

Grâce à cette approche, dans le cas des bitangentes extérieures, les deux premières rotations permettent automatiquement de faire en sorte que les deux points A et B « se voient », ce qui évite de créer une procédure séparée qui effectue cela. Dans le cas des bitangentes intérieures, la situation est un peu différente puisqu'on ne se place pas dans une situation où les deux points se voient mutuellement ; mais l'invariant suffit à assurer la bonne marche de l'algorithme (la preuve est exactement la même que celle de la section 5.1.1).

La complexité, linéaire en le nombre de sommets des deux polygones, s'obtient de la même façon : après les deux premières rotations, les sens de parcours des polygones sont choisis comme dans la section 5.1.1, ce qui fait qu'après cette phase d'initialisation, chaque sommet est considéré au plus une fois.

On effectue ce calcul pour chaque paire de polygones, d'où une complexité en $O(m^2)$, où m est le nombre total de sommets.

5.2 Élimination des bitangentes non libres

Le problème d'implantation, nous l'avons vu, consiste essentiellement à savoir effectuer une réunion d'intervalles sur l'ensemble des directions possibles des vecteurs non nuls de \mathbb{R}^2 (identifié à S^1). En prévision du traitement des cas dégénérés, il s'avère judicieux de considérer ces intervalles comme des *ouverts*, même si cela n'aura une importance que lorsque nous aborderons cette question.

On propose pour effectuer cette réunion d'intervalles une approche du type « diviser pour régner », fondée sur le principe du tri-fusion.

5.2.1 Le cas de \mathbb{R}

Par souci de clarté, voyons d'abord comment on procéderait si l'on avait à effectuer une réunion d'intervalles ouverts bornés de \mathbb{R} . Nous proposons une solution récursive à ce problème en créant une fonction `reunion`. Voyons d'abord les structures de données choisies pour représenter des intervalles et des réunions d'intervalles. En fait, une réunion d'intervalles ouverts bornés de \mathbb{R} se représente commodément sous forme d'une liste contenant alternativement des débuts et des fins d'intervalles. Nous considérons donc pour l'instant les deux types de données suivants :

- un type `Elt_Interv` qui est soit un début, soit une fin d'intervalle ;
- un type qui correspond à une réunion d'intervalles ouverts disjoints : il s'agit d'une liste d'éléments de type `Elt_Interv` (contenant en fait alternativement des débuts et des fins d'intervalles). La sortie de l'algorithme est de ce type.

Notre fonction récursive `reunion` prend en entrée plusieurs *réunions d'intervalles* (c'est-à-dire une liste de listes d'`Elt_Bit`) et se charge de les fusionner, en renvoyant donc une *réunion d'intervalles*, soit une liste d'`Elt_Bit`.

Ainsi, si l'utilisateur doit réunir n intervalles, il fournit à cette fonction une liste de n éléments, chacun de ces éléments étant un intervalle (donc une liste avec exactement un élément `Elt_Bit` de début et un élément `Elt_Bit` de fin).

L'algorithme est alors le suivant :

1. si la liste en entrée contient zéro ou une sous-liste (réunion d'intervalles), on conclut naturellement ;

2. sinon, diviser cette liste en deux et réunir chacune de ces listes par appel récursif; on obtient deux réunions d'intervalles stockées sous forme de listes d'extrémités d'intervalles L_1 et L_2 (en écrivant à chaque fois s'il s'agit d'un début ou d'une fin d'intervalle). Il s'agit maintenant de fusionner L_1 et L_2 ;
3. tant que L_1 et L_2 ne sont pas vides :
 - regarder a_1 , première extrémité d'intervalle de L_1 , et a_2 , celle de L_2 ; supposons par exemple $a_1 < a_2$.
 - si a_2 est un début d'intervalle, écrire a_1 dans la liste **resultat** ;
 - effacer a_1 de la liste L_1 ;
4. recopier la liste (L_1 ou L_2) qui n'est pas vide à la fin de **resultat**.

Pour démontrer la correction de l'algorithme, le point crucial est de montrer que l'opération de fusion fonctionne correctement (ensuite, une simple récurrence sur le nombre de sous-listes permet de conclure).

Pour cela, montrons qu'à chaque événement, l'algorithme écrit correctement ce qu'il faut dans la liste résultat, par inspection de toutes les possibilités (toujours en supposant $a_1 < a_2$) :

- Si a_1 et a_2 sont des débuts d'intervalles : alors a_1 constitue un début d'intervalle dans cette réunion ;
- Si a_1 est une fin et a_2 est un début, a_1 constitue une fin d'intervalle dans cette réunion ;
- Si a_2 est une fin d'intervalle, alors a_1 appartient à la réunion des intervalles représentée par la liste L_2 en début d'appel et ne doit pas figurer dans le résultat.

La complexité de cet algorithme est évidemment la même que celle du tri-fusion, soit $n \log n$, où n est le nombre d'intervalles.

5.2.2 Le cas de S^1

Cet algorithme simple se généralise sans trop de difficultés à des intervalles ouverts de S^1 . Pour cela, fixons-nous une origine (par exemple en identifiant S^1 à $[0, 2\pi[$) et convenons de stocker toutes les listes d'extrémités d'intervalles relativement à cet ordre. Par exemple, si l'on devait stocker l'intervalle de S^1 correspondant aux angles $] -\frac{\pi}{4}; \frac{\pi}{4}[$ en choisissant 0 pour origine, on stockerait d'abord $\frac{\pi}{4}$ en tant que fin d'intervalle, puis $2\pi - \frac{\pi}{4}$ en tant que début d'intervalle.

Une fois cette convention choisie, la seule modification à effectuer est l'étape numéro 4 de l'algorithme : supposons par exemple que la liste vide soit L_1 . Alors, selon que le dernier élément de L_1 était une fin ou un début d'intervalle, il faut ou non recopier la liste L_2 dans **resultat**.

Bien entendu, en pratique, on ne calcule surtout pas les pentes des bitangentes ; on écrit simplement une fonction qui, étant donnés deux segments (non réduits à des points), renvoie si l'un est d'angle polaire strictement plus petit que le second.

5.3 Insertion des côtés libres des polygones

On rappelle qu'on doit insérer dans l'arrangement, outre certaines parties des droites bitangentes libres, les côtés des polygones dont les supports sont libres.

Avec ce qui précède, il y a un moyen simple et relativement élégant pour savoir si les côtés d'un polygone sont libres ou non. Considérons le polygone P ; on veut savoir quels sont les côtés de ce polygone qui sont libres.

Notons qu'avec le « balayage rotationnel » effectué autour du polygone P , on sait dans quelles directions les tangentes à ce polygone sont libres. Mais les côtés du polygone P sont précisément des tangentes : on sait donc, par simple inspection de la réunion obtenue, quels sont les côtés libres.

Mais il y a une façon encore plus efficace de procéder. Modifions une nouvelle fois un tout petit peu la procédure `reunion` : il y a pour l'instant deux types d'extrémités d'intervalles (début et fin) ; créons-en un troisième, qui s'appellera « rien ». Au lieu de donner à la procédure `reunion` des intervalles, on lui donnera en outre des éléments « rien » ne faisant référence à aucun intervalle, un pour chaque côté de polygone, qui témoignera simplement du fait qu'un événement se produit dans cette direction.

Durant la fusion, un élément « rien » de la liste L_1 doit disparaître si et seulement si, à cet endroit, on est dans un des intervalles de la liste L_2 , ce qui revient à dire que le côté du polygone représenté par cet élément n'est pas libre à cause d'un des intervalles stockés dans L_2 .

5.4 Calcul des nombres visuels

Ce calcul est apparemment simple du point de vue de l'implantation, mais c'est celle qui a pris le plus de temps. La raison en est qu'on a besoin (à moins de la programmer soi-même. . .) d'une bibliothèque qui gère les arrangements dans le plan. Une partie non négligeable du stage (4 semaines) a été consacrée à la recherche de ces fonctionnalités dans CGAL et LEDA. Petit historique des recherches engagées. . .

5.4.1 Les cartes planaires dans CGAL

La version publique courante de CGAL (2.0) permet de créer des « cartes planaires ». Une carte planaire est un ensemble de sommets, arêtes et faces plongées dans le plan, liées par des relations d'adjacence (qui permettent donc, par exemple, de parcourir une face par la liste de ses arêtes). Les sommets sont des points du plan, les arêtes sont des courbes x -monotones qui relient deux sommets et les faces sont les composantes connexes maximales du complémentaire des arêtes.

En lisant attentivement la documentation, on s'aperçoit toutefois que l'utilisateur ne peut insérer dans la carte planaire une nouvelle courbe que si elle ne coupe aucune arête déjà existante (en dehors de ses extrémités). Ceci rend le module `Planar_Map` de CGAL *a priori* peu intéressant pour ce qu'on veut faire.

5.4.2 Les arrangements dans CGAL

Toutefois, il existe dans la version interne de CGAL un module `Arrangement`, utilisant de manière interne le module `Planar_Map`, et gérant ce problème des intersections.

Les spécifications du module *Arrangement*. Ce module est plus général que celui des cartes planaires :

- l'utilisateur insère des courbes quelconques du plan, pouvant se couper ; charge au module de repérer ces intersections et de mettre à jour les informations sur les faces, arêtes et sommets ;
- les courbes ne sont plus nécessairement x -monotones ;
- le module gère le problème des cas dégénérés (points triples...), y compris dans le cas où deux courbes se superposent (une structure de données à part permet à l'utilisateur de le savoir).

Le fonctionnement. Il y a en fait création d'une arborescence à trois niveaux. Le niveau supérieur est constitué des courbes insérées par l'utilisateur. Ces courbes sont subdivisées dans le niveau du milieu en courbes x -monotones. Celles-ci sont à nouveau découpées dans le niveau du bas en arêtes (portions de courbes x -monotones qui ne sont coupées par aucune autre courbe). L'algorithme est incrémental, c'est-à-dire que l'utilisateur peut insérer petit à petit des courbes et regarder à chaque étape l'arrangement.

Les points sont stockés, au choix de l'utilisateur, en coordonnées cartésiennes ou homogènes, avec n'importe quel type, exact ou approché (**double** en coordonnées cartésiennes, **long** ou type **integer** – entiers arbitrairement longs de LEDA – en coordonnées homogènes, ...).

Les opérations élémentaires concernant les courbes (découpe en courbes x -monotones, intersection(s) entre deux courbes, ...) ne sont pas incluses directement dans le module *Planar_Map* mais sont gérées par une classe C++ appelée **Traits** (comme il y en a dans tous les modules CGAL). Plusieurs types de **Traits** sont fournis dans la version interne de CGAL : des **Traits** pour les arrangements de segments en type de données exact (avec trois variantes) et des **Traits** pour les arrangements de cercles. Rien n'empêche l'utilisateur de créer ses propres **Traits** : c'est d'ailleurs ce qui constitue un des points forts de CGAL en lui donnant toute sa flexibilité.

Les problèmes rencontrés. Pour tester ce module et suite aux problèmes rencontrés, nous avons créé une procédure de test toute simple : dans un premier temps, il y a insertion d'une figure en forme d'échelle dans le plan, légèrement tordue pour éviter les segments horizontaux ou verticaux et les cas dégénérés, avec création de 50 barreaux. Ensuite, un segment vient s'introduire au milieu de l'échelle en coupant un nombre à choisir de barreaux.

Voici une synthèse des problèmes, qui sont bien entendu apparus au fur et à mesure. Deux versions de ce module ont été testées (1.03, mi-octobre 1999 et 1.04, 9 novembre 1999).

Un module en développement. Tout d'abord, il s'agit d'une version interne ; le module n'est pas vraiment terminé, on constate en particulier :

- des erreurs internes du compilateur (*egcs 1.1.1*) avec certaines classes de **Traits** ;

- des temps de calcul très aléatoires ; ainsi, dans la version 1.03, en type exact pour des coordonnées homogènes, le test décrit ci-dessus donne les temps suivants (la colonne de gauche contient le nombre de barreaux coupés par le segment qu'on insère au milieu de l'échelle, la colonne de droite le temps mis pour insérer ce segment) :

nombre d'intersections	temps (s)
≤ 27	≤ 30
28	130
29	13
30	28
31	16

La nécessité de calculs exacts. Il faut tout faire avec un type de données exact : dans la version 1.03, si on utilise les `float` par exemple, même dans l'exemple simple de l'« échelle », où les intersections sont assez éloignées les unes des autres et où il n'y a pas de cas dégénérés, le programme échoue en donnant des « erreurs de précondition » de CGAL (une fonction reçoit en entrée des données incohérentes entre elles, du type : un point se trouve au-dessus de la courbe alors qu'il devrait être au-dessous). Quand on inhibe les préconditions (une option du compilateur est disponible), le programme boucle indéfiniment. Dans la version 1.04, des erreurs se produisent même si un type exact est utilisé ! Ce problème semble avoir été corrigé depuis.

La lenteur. Enfin, comme cela a été vu en partie précédemment, il faut noter une grande lenteur : dans certains cas, il fallait plus d'une heure pour couper 5 barreaux de l'échelle ! Ce phénomène s'aggrave si les coordonnées des points sont « compliquées » (ce qui est le cas dans le programme d'enveloppes visuelles, les points étant entrés à la souris). Après prise de contact avec les concepteurs du module par l'intermédiaire de Mariette Yvinec et Sylvain Pion à l'Inria Sophia-Antipolis, plusieurs problèmes ont été repérés :

- les calculs d'intersection entre courbes se faisaient initialement dans le niveau « du bas » dans l'arborescence (c'est-à-dire au niveau des *arêtes* préexistant lors de l'insertion de la nouvelle courbe). Il en résultait une explosion de la complexité des coefficients en type exact (les numérateurs et dénominateurs des extrémités des arêtes étant généralement plus grands que ceux des courbes initiales). Ce problème a été résolu par un pointeur sur la donnée d'origine ;
- le type `rational` de LEDA ne simplifie pas automatiquement les fractions ; l'algorithme implanté dans CGAL non plus. Ceci accentuait donc encore la lenteur des calculs ;
- la stratégie de localisation d'un point dans une face utilisée dans le module *Planar_Map* (lui-même utilisé par *Arrangement* de façon interne) se trouvait être peu efficace pour ce problème ;
- le module *Arrangement* n'utilise pas les filtres du noyau CGAL (un filtre est un moyen de répondre rapidement à un calcul de prédicat : on ne fait le calcul exact que quand le calcul approché s'avère ne pas être suffisamment précis).

Épilogue. Ces problèmes semblent avoir été résolus dans la version 1.06 de *Planar_Map*...

Cette étude assez poussée, en tant qu'utilisateur, du module *Arrangement* de CGAL, a conduit à une visite de deux jours à l'Inria Sophia-Antipolis (les 18 et 19 novembre 1999) pour y présenter ces difficultés et assister à une présentation sur les spécifications formelles des cartes planaires par J.-F. Dufourd, professeur à Strasbourg. S'en est ensuivie une discussion sur les cartes planaires en général et dans CGAL en particulier avec tous les participants. Enfin, la soutenance de Sylvain Pion concernant le calcul des prédicats géométriques grâce à l'arithmétique filtrée (méthode programmée par Sylvain Pion en particulier dans CGAL) a fourni un assez bon éclairage de la façon dont les prédicats étaient calculés en CGAL.

5.4.3 Les arrangements en LEDA

Après avoir renoncé à CGAL, nous nous sommes tourné vers LEDA (il a donc bien sûr fallu « traduire » le code déjà écrit de CGAL vers LEDA...). De façon générale, LEDA est plus simple d'utilisation, mais nettement moins paramétrable que CGAL. L'algorithme `SWEEP_SEGMENTS` de LEDA prend en entrée une liste de segments et calcule l'arrangement correspondant dans le plan. C'est l'algorithme de Bentley et Ottmann [1], dont la complexité est $O((n + s) \log n)$, où n est le nombre de segments et s le nombre de sommets de la carte planaire.

L'implantation de cet algorithme fonctionne correctement. Toutefois, il est nécessaire dans notre problème de stocker de l'information dans les segments pour savoir de quel côté de l'arête le nombre visuel était le plus grand.

Pour résoudre cette difficulté, la première tentative a été de créer une classe dérivée de la classe `segment` en ajoutant l'information requise, puis de donner à l'algorithme une liste d'objets de cette nouvelle classe: cela n'a pas fonctionné, le compilateur n'acceptant pas une liste d'objets de cette classe dérivée de `segment` au lieu d'une liste d'objets de la classe `segment` elle-même.

Ensuite, nous avons tenté de modifier le code de la fonction `SWEEP_SEGMENTS` de LEDA afin de stocker l'information; cela s'est avéré possible, mais au cours de la modification deux choses sont apparues:

- les segments que l'on donne à l'algorithme sont orientés et une option permet que les arêtes de l'arrangement soient orientées dans le même sens que les segments de départ (ce qui résolvait parfaitement notre problème);
- sur chaque arête de l'arrangement sont stockées les coordonnées des extrémités des segments, ce qui n'est pas utile dans le cas des enveloppes externes mais pourrait servir par exemple dans le cas des enveloppes internes (notamment pour repérer si une arête est une partie d'un côté d'un polygone, puisqu'on ne doit pas traverser une telle arête). À noter que ce stockage n'apparaît pas dans la documentation de LEDA.

Donc l'implantation a été possible sans modifier le code de LEDA: l'orientation d'un segment détermine de quel côté le nombre visuel est le plus grand. Cependant, ce problème simple et déroutant aurait pu être beaucoup plus ennuyeux (si nous avions voulu stocker un entier sur chaque segment et pas seulement un booléen correspondant à une orientation) et aurait trouvé sans nul doute une solution élégante en CGAL si le module *Arrangement* était terminé.

5.4.4 L'implantation finalement retenue

Une fois ceci fait, il est bien sûr facile d'afficher l'enveloppe visuelle externe (régions de nombre visuel 0). À noter tout de même la légère difficulté d'implantation suivante : dans l'algorithme, il faut parfois insérer des demi-droites dans l'arrangement, pas seulement des segments. En CGAL, une solution élégante aurait pu être trouvée par le biais des `Traits`, pour permettre d'insérer segments ou demi-droites ; en LEDA, c'est impossible et on a donc eu recours à un artifice. Tout repose sur le fait que l'enveloppe visuelle externe est incluse dans l'enveloppe convexe de l'ensemble des polygones (proposition 3) ; on a donc créé un rectangle autour de cette enveloppe convexe et tronqué toutes les demi-droites dans cette boîte.

D'où certains petits détails simples mais peu intéressants et fastidieux de programmation, notamment dans le parcours des faces : il faut s'arranger pour ne pas prendre la face non bornée comme face de départ, ne jamais traverser une arête de la boîte, etc.

Bien entendu, en parcourant les différentes régions, on obtient les nombres visuels à une constante additive près seulement (le nombre visuel de la région de départ est inconnu). Toutefois il est clair que le plus petit nombre visuel présent est 0, ce qui permet de déterminer cette constante.

5.5 Cas dégénérés

On présente ici les détails d'implantation utilisés pour prendre en compte les cas dégénérés. Bien que ces détails aient été traités au fur et à mesure de l'implantation, il semble plus clair de les donner ici après une présentation générale de l'algorithme.

5.5.1 Le calcul des bitangentes

Peu de cas dégénérés peuvent se produire dans le calcul des bitangentes. En particulier, l'algorithme de calcul d'enveloppes convexes implanté dans LEDA nous garantit que, parmi les sommets d'un polygone convexe obtenu à l'aide de cet algorithme, trois d'entre eux ne peuvent être alignés.

Les seuls cas dégénérés possibles sont représentés sur la figure 16.

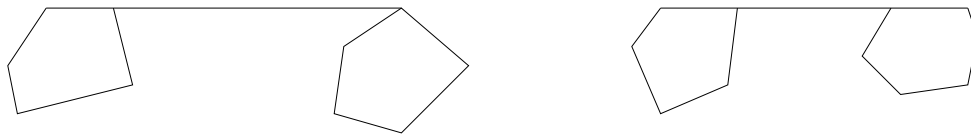


FIG. 16 – Les deux cas dégénérés dans le calcul des bitangentes.

Plus précisément, il faut aussi traiter un peu séparément le cas où le polygone est réduit à un segment (il est impossible d'entrer un polygone-point à la souris en LEDA...), mais ça ne pose aucune difficulté.

En tout état de cause, il ne faut pas renvoyer deux sommets porteurs de la bitangente, mais dans les cas dégénérés il faut, pour chacun des deux polygones, renvoyer le ou les points qui sont sur la droite bitangente.

5.5.2 L'élimination des bitangentes non libres

Il nous faut ici gérer les problèmes qui apparaissent quand trois sommets sont alignés. C'est impossible si les sommets appartiennent à un seul polygone, comme nous l'avons vu, mais cela est naturellement possible si plusieurs polygones entrent en jeu.

Précisons d'abord ce qu'on entend par « bitangentes libres » : à ce stade, on considère les polygones comme des *ouverts* ; si une droite est tangente à trois polygones, elle est considérée comme libre vis-à-vis de ces polygones et apparaîtra trois fois. Ceci est cohérent avec le fait qu'on effectue une réunion d'intervalles ouverts.

Lors de la fusion des intervalles, il faut prendre garde aux cas particuliers quand deux mêmes valeurs apparaissent dans L_1 et L_2 ; il faut en fait créer un nouveau type d'extrémité d'intervalle correspondant à la fin d'un intervalle *et* au début d'un autre. Les modifications à apporter sont mineures. Si jamais une même direction correspond aux deux bitangentes, il faut noter dans le type d'extrémité d'intervalle les *deux* bitangentes correspondantes. En d'autres termes, la structure de données qui contient le type d'extrémité d'intervalle contient aussi une liste d'éléments auxiliaires correspondant chacun aux informations relatives à une bitangente.

5.5.3 Le calcul des nombres visuels

Le problème est, étant donnée la liste des bitangentes (ou côtés de polygones) sur une droite donnée, de déterminer quelles sont les portions de cette droite à insérer dans l'arrangement.

Plusieurs tentatives. Plusieurs approches ont été tentées : tout d'abord, nous pensions qu'il suffisait de considérer une bitangente active seulement si l'intérieur du segment joignant les deux sommets des polygones ne touchait aucun polygone. Cette approche mène à des résultats faux, comme le montre la figure 17.

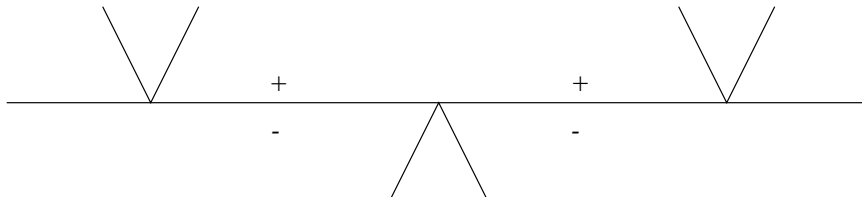


FIG. 17 – Il serait faux de considérer qu'une bitangente est libre si l'intérieur du segment joignant les deux sommets des polygones ne touche aucun polygone. Ici sont affichées les variations des nombres visuels pour la droite horizontale si l'on suivait cette idée. En fait, il n'est pas difficile de s'apercevoir qu'aucune variation de nombre visuel n'intervient de part et d'autre de la droite.

Finalement, la méthode utilisée a consisté à faire une étude à part des cas dégénérés. Considérons une droite (horizontale par exemple) tangente à plusieurs polygones. Certains polygones sont au-dessus de la droite, d'autres sont au-dessous. Ordonnons les sommets des polygones qui touchent cette droite ; on écrit H si le sommet appartient à un polygone situé au-dessus de la droite et B sinon ; puis on regarde la suite des H et des B obtenue.

La remarque cruciale est la suivante : si l'on peut extraire de cette suite BHB ou HBH , il n'y a aucun segment à insérer dans l'arrangement ; en effet, aucune droite « proche » de la droite horizontale considérée n'est libre.

On est donc ramené aux cas suivants : H^+ , B^+ , H^+B^+ , B^+H^+ , le $+$ signifiant « au moins un » pour le caractère précédent. Leur traitement est résumé sur la figure 18 (ces cas ressemblent bien sûr aux cas des bitangentes extérieures et intérieures).

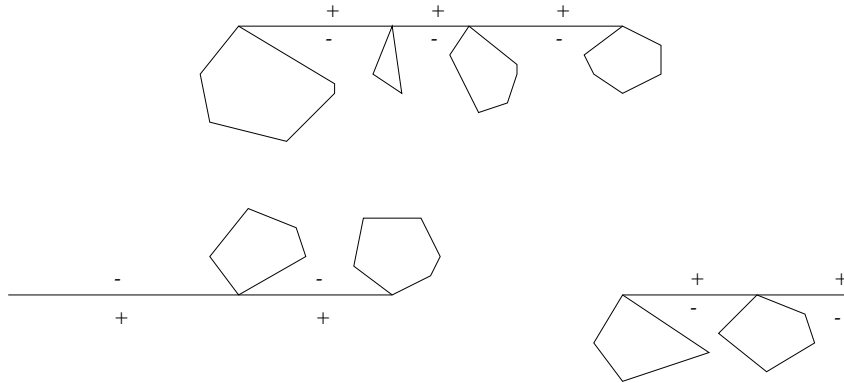


FIG. 18 – Les deux cas dégénérés à considérer.

En pratique, la séquence des H et des B n'est jamais calculée : cela nécessiterait un tri des abscisses des points. Il suffit en fait de calculer le plus petit segment de la droite contenant des H et le plus petit segment de la droite contenant des B :

- si ces deux segments se chevauchent, il n'y a rien à insérer dans l'arrangement ;
- si l'un des deux segments est vide, on est dans le premier cas de la figure ;
- sinon, on est dans le second cas.

À noter que ceci se généralise sans difficulté au cas où l'arête d'un (ou de plusieurs) polygones est incluse dans la bitangente elle-même.

Le cas des polygones-segments. Il demeure toutefois un cas dégénéré : celui où il y a des polygones réduits à des segments et inclus dans la droite ; ils constituent en quelque sorte à la fois des H et des B avec les notations ci-dessus. Les considérer comme des polygones ordinaires mène à des résultats faux. Un exemple est représenté sur la figure 19 : on s'aperçoit qu'à certains endroits le nombre visuel peut varier de plus d'une unité, ce que l'on n'obtient pas avec l'approche précédente.

Ce cas n'a pas été traité dans le programme faute de temps ; voici cependant l'approche théorique. Considérons une droite (toujours horizontale) source de cas dégénérés (*i.e.*, il y a plus d'une bitangente intervenant sur cette droite).

D'abord, il faut supprimer par la pensée les polygones-segments inclus dans la droite et calculer les variations des nombres visuels sur cette droite avec l'approche précédente.

Ensuite, pour chaque polygone-segment inclus dans la droite, superposons aux variations de nombre visuel précédentes d'autres variations induites par ces segments (par exemple, les 5 segments de la figure 19 font que le nombre visuel, en passant de A à B , varie de 5 au lieu de rester constant). Pour cela, regardons comme avant

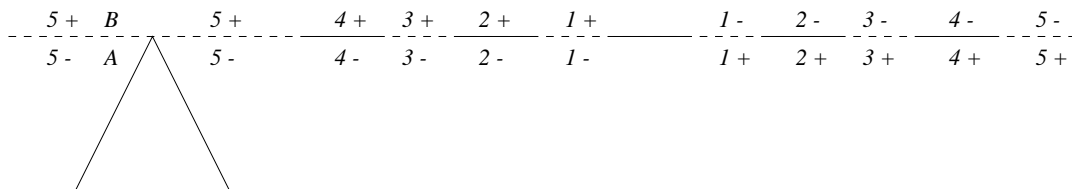


FIG. 19 – Considérer les polygones-segments comme des polygones ordinaires mène à des résultats erronés, puisque le nombre visuel ne pourrait varier que d’une unité à chaque fois. Ici, on représente un cas où le nombre visuel peut connaître de plus grandes discontinuités avec cinq polygones-segments sur la même droite. La notation $n+$, $n-$ signifie que le nombre visuel varie de n en traversant l’arête considérée.

la suite des H et des B , sans inclure dans cette suite les polygones-segments, en marquant par une étoile la position du segment. Voici les principaux cas (le lecteur en déduira sans difficulté les autres) :

- si on peut extraire de la suite la sous-suite BHB ou HBH , il ne faut rien insérer ;
- pour $B^+ \star B^+$, on n’insère rien ;
- le cas $B^+ \star$ est représenté sur la figure 20 ;

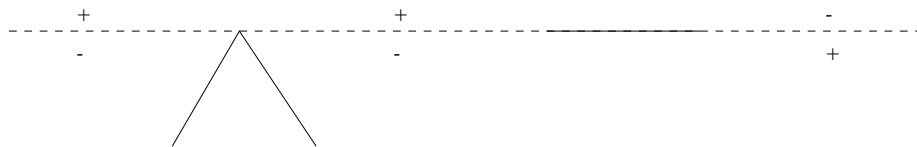


FIG. 20 – Un polygone-segment qui crée de nouvelles variations de nombre visuel : le cas « $B^+ \star$ ».

- le cas $B^+ \star H^+$ est représenté sur la figure 21.

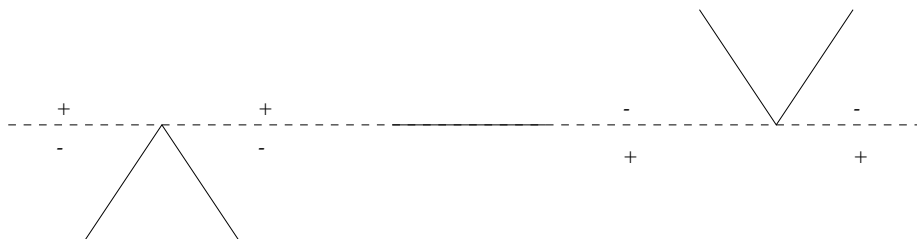


FIG. 21 – Un polygone-segment qui crée de nouvelles variations de nombre visuel : le cas « $B^+ \star H^+$ ».

Ceci conclut le traitement des cas dégénérés, qui, on le voit, est assez complexe comme dans de nombreux algorithmes géométriques.

6 Résultats

Enfin, voici les résultats de l'implantation de cet algorithme. On trouvera d'abord une brève description de l'interface graphique, avec quelques images du programme, puis une étude théorique de la complexité ainsi que des résultats chiffrés sur les performances obtenues en pratique.

6.1 Interface du programme

L'utilisateur dispose de plusieurs boutons; outre le lancement du calcul, une aide et la possibilité de quitter le programme, il peut effacer les polygones, afficher une grille pour tester les cas dégénérés, afficher les arêtes de l'arrangement. Il peut également afficher toutes les régions d'un nombre visuel donné (de 0 à 9).

L'interface a été programmée en LEDA; la création de fenêtres et de boutons est relativement aisée.

Voici quelques exemples de captures d'écran de ce programme (figures 22, 23 et 24).

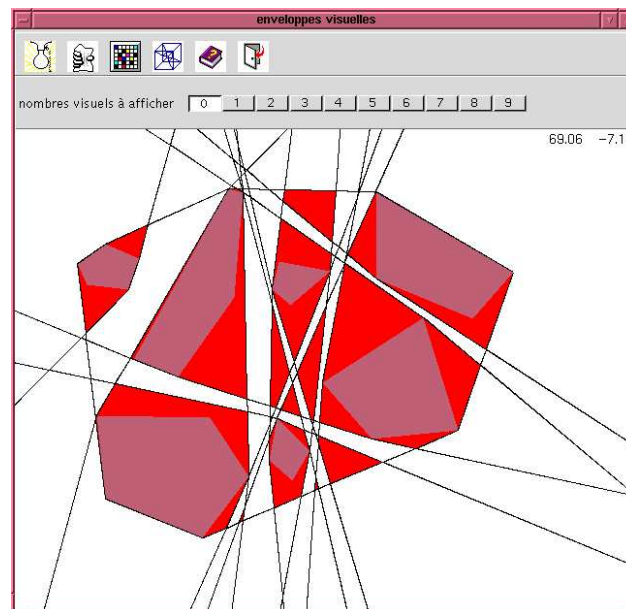


FIG. 22 – *Un exemple typique d'utilisation.*

6.2 Complexité de l'algorithme

Cet algorithme est plus coûteux en temps que l'algorithme de l'article de S. Petitjean [6], mais se révèle plus simple à implanter. Récapitulons les étapes de l'algorithme pour étudier sa complexité. Nous notons m le nombre total de sommets des polygones et n leur nombre.

1. Le calcul des bitangentes (sans se préoccuper du fait qu'elles soient libres ou non) a une complexité $O(m^2)$;

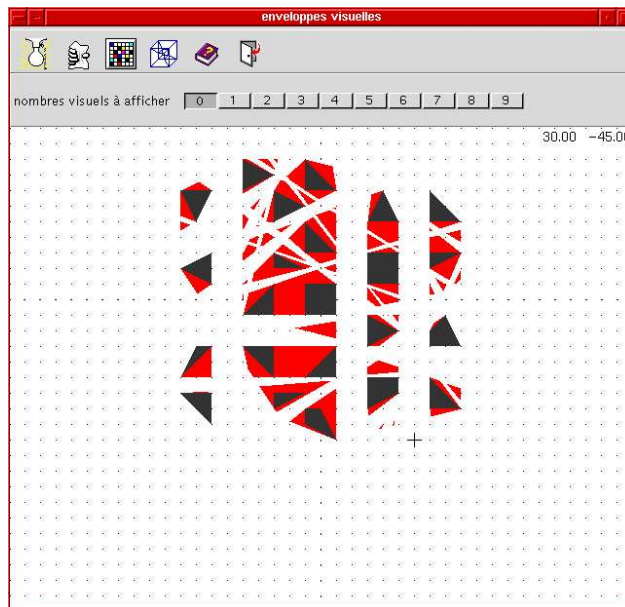


FIG. 23 – *Un cas vraiment dégénéré...*

2. la réunion des intervalles servant à ne conserver que les bitangentes et les côtés libres est de complexité $O((n^2 + m) \log m)$;
3. le calcul de la carte planaire se fait en $O((p + s) \log p)$, où p est le nombre de segments insérés et s le nombre de sommets de la carte planaire. Ici, $p = O(n^2 + m)$, d'où une complexité de $O((n^2 + m + s) \log m)$.

D'où la complexité finale : $O((n^2 + s) \log m + m^2)$. En pratique, très souvent, on considère que les polygones ont un nombre de sommets borné (donc $m = O(n)$) et on obtient donc le résultat suivant : $O(n^2 + s) \log n$, comme annoncé.

Dans le pire des cas, $s = O(n^4)$ et l'algorithme est en $O(n^4 \log n)$. Mais on conçoit bien que, dans la plupart des cas pratiques, s est nettement plus faible ; la complexité de l'algorithme dépend vraiment du résultat calculé.

Cet algorithme est donc de complexité intermédiaire entre celui d'A. Laurentini [4] ($O(n^3 + s \log s)$) et de S. Petitjean [6] ($O(n \log n + r + s \log s)$, où r désigne la taille du graphe de visibilité).

6.3 Performances chiffrées

Le code C++ (commenté) fait un peu plus de 1300 lignes (37 ko). La taille du fichier exécutable dépend énormément de la version de LEDA utilisée et du compilateur (300 ko à 3 Mo) mais ne signifie pas grand-chose puisque le programme fait de toute façon appel aux bibliothèques de LEDA qui ne sont pas incluses dans ce fichier.

La place mémoire utilisée par le programme est assez importante : 10 Mo en mémoire vive, auxquels s'ajoute apparemment de l'espace en mémoire virtuelle. Le temps de calcul n'est pas instantané : sur des stations de calcul importantes, quelques secondes sont nécessaires dans des cas compliqués (quelques dizaines de polygones).

Mis à part le cas des segments dans le calcul de l'enveloppe visuelle, aucune erreur « suspecte » n'a été décelée dans le programme lui-même : le calcul est fait

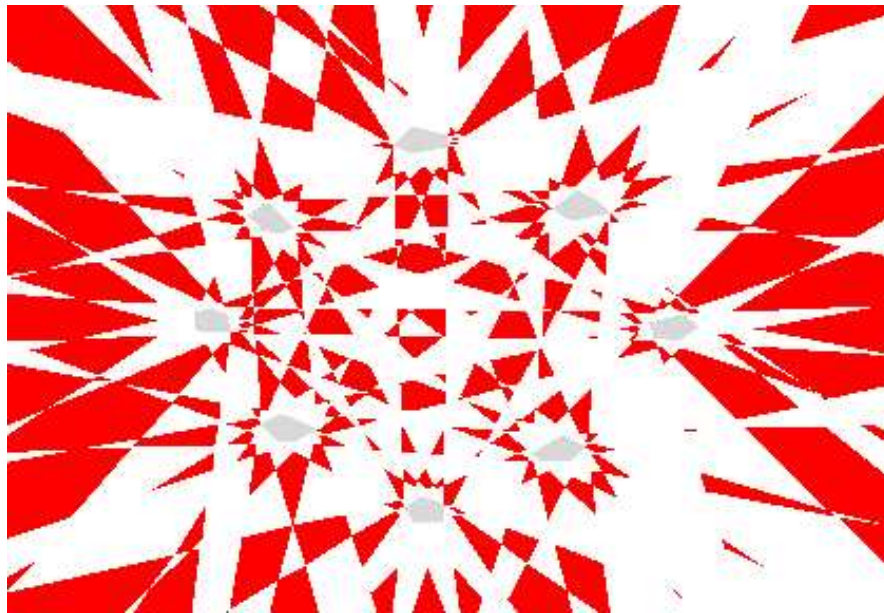


FIG. 24 – Ici, les régions grisées sont celles de nombre visuel 5.

de façon exacte avec les rationnels de LEDA et aucun résultat incorrect n'a été trouvé. Seuls quelques problèmes subsistent mais sont liés à des erreurs de LEDA (par exemple, une tentative d'agrandissement de la fenêtre peut ne pas fonctionner de façon satisfaisante : ce phénomène est entièrement géré par LEDA).

Toutefois, un préalable indispensable est de compiler correctement le programme : des erreurs dues à des différences de versions de compilateur ayant servi à compiler LEDA et le programme peuvent empêcher son lancement.

Conclusion

Nous avons pu voir, durant ces trois mois de stage, le processus nécessaire pour implanter un algorithme à partir d'un article. C'est intéressant en soi, car cela permet de mesurer le « fossé » qu'il y a entre la conception théorique d'un algorithme et sa mise en application. Les difficultés suivantes, qui ont été rencontrées et surmontées, illustrent bien ce fait :

- l'étude en détail de deux bibliothèques en vue de leur utilisation pour la programmation de notre algorithme, s'est révélée beaucoup plus fastidieuse que prévu (4 semaines) ;
- la gestion des cas dégénérés (qui, d'ailleurs, pourrait être complétée en ce qui concerne l'insertion des polygones-segments), a mobilisé également une part importante du temps disponible et prend une certaine place dans le code du programme.

Si ce stage était avant tout un stage de programmation, des questions théoriques parfois non évidentes n'ont pas manqué de se poser : on voit bien, en lisant la section 4, tous les détails à affiner en vue d'une implantation à partir des étapes données dans la section 2 !

L'intérêt d'implanter un tel algorithme est évidemment avant tout de permettre de prouver qu'il est utilisable en pratique. Ce programme pourrait également être utilisé à des fins didactiques : il est très simple, à partir de quelques images du programme, d'expliquer la signification géométrique d'une enveloppe visuelle et d'en donner des exemples typiques. Comme nous l'avons dit, la conception d'un tel algorithme dans le plan peut ouvrir la voie vers la rédaction d'un algorithme de calcul d'enveloppes visuelles en trois dimensions.

Enfin, plusieurs prolongements possibles n'ont pas été abordés faute de temps, mais ne manquent pas d'intérêt :

- un algorithme analogue (voir [6]) permet de calculer les enveloppes visuelles internes : il y a quelques complications par rapport aux enveloppes visuelles externes, toutefois la programmation d'un tel algorithme dans le cas de polygones *convexes* devrait être assez aisée à partir du programme déjà fait ;
- plus généralement, on pourrait étudier le calcul des enveloppes visuelles quand la région d'observation peut avoir des formes plus variées que pour les enveloppes externes ou internes.

À plus long terme et à un niveau plus théorique, on peut s'intéresser aux questions suivantes :

- les problèmes de visibilité en général (construction du graphe de visibilité par exemple) : l'affichage efficace d'une scène complexe pourrait passer par le calcul de cet objet – malgré certaines difficultés liées à son coût. Ici intervient de façon cruciale le calcul des bitangentes (et tritangentes dans le cas 3D) ;
- quels sont les objets qui ont même enveloppe visuelle ?
- dans quels cas peut-on reconstituer parfaitement un objet avec un nombre fini d'images ? Si c'est possible, quel est le nombre minimal d'images ? Si non, avec quelle précision peut-on approcher l'objet avec un nombre fixé de points de vue ? (certaines de ces questions sont abordées dans [5]) ;
- étant donnée une enveloppe visuelle, que peut-on dire de l'objet, *i.e.*, quels points de l'enveloppe visuelle se trouvent nécessairement à l'intérieur de l'objet ?

Références

- [1] J.L. Bentley, T. Ottmann, *Algorithms for Reporting and Counting Geometric Intersections*, IEEE Trans. on Computers C 28, pp. 643-647, 1979.
- [2] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 1997.
- [3] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, 1987.
- [4] A. Laurentini, *The Visual Hull Concept for Silhouette-Based Image Understanding*, IEEE Trans. on Pattern Analysis and Machine Intelligence, vol. 16, no. 2, Février 1994.
- [5] A. Laurentini, *How Many 2D silhouettes It Takes to Reconstruct a 3D Object?*, Computer Vision and Image Understanding, vol. 67, pp. 81-87, 1997.
- [6] S. Petitjean, *A Computational Geometric Approach to Visual Hulls*, International Journal on Computational Geometry and Applications, 8(4) :407-436, 1998.
- [7] M. Pocchiola, G. Vegter, *A Simple Sweep Algorithm for Visibility Graphs of Curved Obstacles*, Manuscrit, 1994.

Table des matières

1	Présentation de l'équipe Isa du Loria	1
2	Théorie des enveloppes visuelles	2
2.1	Introduction aux enveloppes visuelles	2
2.1.1	Les motivations de cette étude	2
2.1.2	La définition	3
2.1.3	L'objet du stage et ses liens avec les autres travaux	4
2.2	Enveloppes visuelles externes	4
2.2.1	Une première propriété	5
2.2.2	Une caractérisation simple	5
2.2.3	L'intérêt de la définition	6
2.3	Enveloppes visuelles externes dans le plan	6
3	Description de l'algorithme	7
3.1	Notion de nombre visuel	7
3.1.1	L'idée générale	7
3.1.2	La définition	7
3.1.3	Une partition du plan	8
3.1.4	Quelques définitions	9
3.2	Calcul des frontières	10
3.2.1	Les différentes approches	10
3.2.2	Le calcul des bitangentes libres	10
3.3	Autre algorithme possible	12
3.3.1	Le principe	12
3.3.2	Les caractéristiques de cette méthode	14

4	Aperçu de LEDA et CGAL	15
4.1	LEDA	15
4.1.1	Des nombres de précision arbitraire	15
4.1.2	Des algorithmes couramment employés en informatique	15
4.1.3	Les graphes	16
4.1.4	Des outils géométriques	16
4.1.5	Une interface graphique	16
4.2	CGAL	16
4.2.1	Quelques atouts	16
4.2.2	Un tour d’horizon des modules de CGAL	17
4.2.3	Des types de données paramétrables	17
5	Implantation	18
5.1	Calcul des bitangentes	19
5.1.1	L’idée d’Edelsbrunner	19
5.1.2	L’implantation choisie	20
5.2	Élimination des bitangentes non libres	22
5.2.1	Le cas de \mathbb{R}	22
5.2.2	Le cas de S^1	23
5.3	Insertion des côtés libres des polygones	24
5.4	Calcul des nombres visuels	24
5.4.1	Les cartes planaires dans CGAL	24
5.4.2	Les arrangements dans CGAL	24
5.4.3	Les arrangements en LEDA	27
5.4.4	L’implantation finalement retenue	28
5.5	Cas dégénérés	28
5.5.1	Le calcul des bitangentes	28
5.5.2	L’élimination des bitangentes non libres	29
5.5.3	Le calcul des nombres visuels	29
6	Résultats	32
6.1	Interface du programme	32
6.2	Complexité de l’algorithme	32
6.3	Performances chiffrées	33