GlSurf

Maths et Dessins en OCaml

Christophe Raffalli

1: Laboratoire de Mathématiques,
Université de Savoie, 73376 Le Bourget-du-Lac cedex, France
christophe.raffalli@univ-savoie.fr

Résumé

GlSurf est un programme permettant de dessiner et manipuler des surfaces et des courbes implicites avec OpenGL. Nous décrivons l'algorithme de triangulation qui est à la base de ce programme et son implantation en Objective Caml.

1. Introduction

Cet article décrit une amélioration de l'algorithme « marching cube », son implantation en Objective Caml et l'utilisation d'OpenGL (via LablGL [2, 8]) pour visualiser des surfaces implicites. La figure 1 montre ce qu'on peut faire avec l'algorithme (il s'agit d'une surface cubique). L'écriture de GlSurf a été inspirée par Surf [7] qui permet le tracé de surfaces algébriques par lancer de rayon. L'apport de GlSurf est l'animation en temps réel, l'absence de limitation aux surfaces algébriques et le calcul effectif de la topologie de la surface.

Énoncé du problème mathématique : étant donné une fonction f de $\mathbb{R}^3 \mapsto \mathbb{R}$ et un cube dans $\mathbb{R}^3$, on désire obtenir une triangulation de la portion de surface d'équation $f(x, y, z) = 0$ à l'intérieur du cube. Une triangulation est un polytope à faces triangulaires ayant la même topologie que la surface et dont tous les sommets sont sur la surface.

Ce problème est extrêmement difficile dans le cas général et on ne peut garantir la correction de l'algorithme[1]. Toutefois, même pour des surfaces singulières (non lisses), notre algorithme permet une bonne visualisation (voir la figure 1 et la figure A en appendice) faisant de GlSurf un outil intéressant pour l'enseignement et la recherche.

Contexte algorithmique : la base de l'algorithme est le « marching cubes » [1, 4] qui divise l'espace en cubes et utilise les changements de signes de $f(x, y, z)$ aux deux extrémités d'une arête du cube pour trouver des points de la surface par dichotomie. Ensuite, il suffit de relier ces points (judicieusement) pour obtenir une triangulation de la surface à l'intérieur de ce cube. Si le cube initial a été suffisamment subdivisé, on obtient une bonne triangulation.

Le problème de cet algorithme est double :

1. Les portions de la surface qui passent près des sommets d'un cube génèrent de très petits triangles. Ce problème est résolu dans [5] en considérant que la surface coupe toutes les arêtes en leur milieu[2].

[1] Même dans le cas algébrique (si f est un polynôme), aucun algorithme exact *et* faisable n'existe pour l'instant.

[2] Si l'on désire conserver la propriété que tous les sommets des triangles sont sur la surface, on peut les déplacer dans la direction du gradient de f.

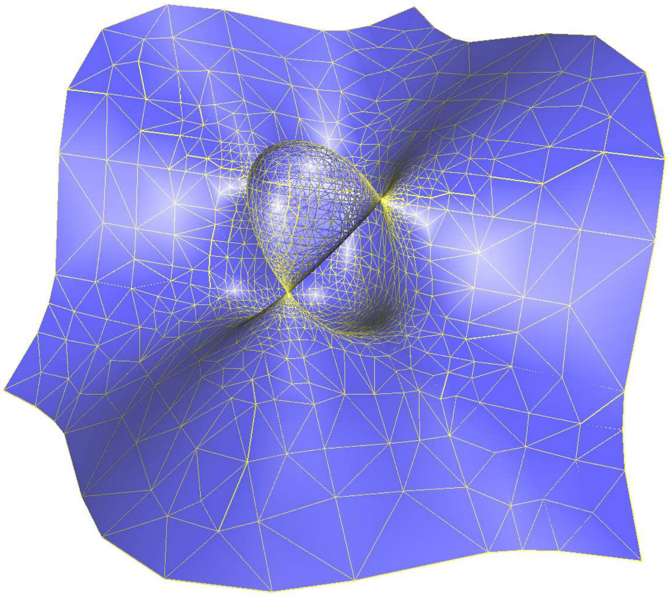

Fig. 1 – Une triangulation obtenue avec GlSurf

2. Les surfaces mathématiquement intéressantes possèdent souvent des structures de taille variable. Il faudrait donc des cubes de taille variable, ce que ne permet pas l'algorithme « marching cubes » (on obtient alors une surface avec de fines déchirures et donc on ne respecte pas la topologie de la surface). L'algorithme « marching triangles »[3] tente de résoudre ce problème, mais nécessite de trouver préalablement un point par composante connexe de la surface, ce qui peut être difficile.

Notre algorithme propose une nouvelle méthode pour extraire une triangulation à l'intérieur d'un cube dont les voisins peuvent être éventuellement plus petits que lui. De plus, pour éliminer les petits triangles, on déplace les sommets des cubes proches de la surface sur celle-ci (les cubes ne sont alors plus des cubes).

Le principal défaut de notre méthode est qu'elle consomme énormément de mémoire (100Mo et plus pour des surfaces un peu compliquées), par contre elle donne la plupart du temps la caractéristique d'Euler exacte pour ces surfaces.

Plan : la section 2 décrit plus précisément notre algorithme. Son implantation en Objective Caml (par rapport à des langages tels que C, C++) a permis facilement de représenter les données, de faire évoluer l'algorithme et de gagner du temps pour implanter la partie calcul formel du logiciel, la compilation des fonctions (voir section 4), la sauvegarde des triangulations pour éviter de les recalculer, etc. De plus, le logiciel obtenu est portable (testé avec succès sous Windows, Mac OS X, Linux et FreeBSD), la seule partie du code qui est dépendante de l'architecture étant la liaison dynamique de code C. Toutefois, nous verrons dans la section 5 que la représentation des données pourrait être améliorée.

L'utilisation d'OpenGL (via LablGL et LablGlut [2, 8]) pour visualiser et animer les surfaces n'avait pas d'alternative. Nous avons dû tout de même étendre LablGL pour gérer les tableaux afin d'économiser de la mémoire et développer un modèle de transparence ne nécessitant pas le tri des triangles pour les dessiner d'arrière en avant (voir section 3).

2. Description détaillée de l'algorithme

2.1. Représentation des données

Notre algorithme utilise une représentation impérative des cubes. Chaque cube est une structure contenant 6 faces et deux cubes partageant une face pointent sur la même face :

```
type cube =
    { face1 : face; face2 : face; face3 : face;
      face4 : face; face5 : face; face6 : face }
```

Chaque face est une structure contenant 4 arêtes. De plus, une face peut être divisée ou non et si elle est divisée, on trouve dans la structure les quatre sous-faces :

```
and face =
    { edge1 : edge; edge2 : edge; edge3 : edge; edge4 : edge;
      mutable face_status : face_status }

and face_status =
    FDivided of face * face * face * face
  | FNothing
```

Les arêtes sont une structure contenant 2 sommets, elles peuvent contenir un point de la surface, ou bien être divisée en deux arêtes, ou bien n'avoir aucune particularité :

```
and edge =
    { vertex1 : cvertex; vertex2 : cvertex;
      mutable status : edge_status }

and edge_status =
    Mesh_point of point
  | Divided of edge * edge
  | Nothing
```

Le type cvertex contient les coordonnées du point dans l'espace et des informations relatives à la phase deux de l'algorithme (voir plus loin).

```
and cvertex = {
  mutable cpos : point;
  mutable info : vertex_info;
}

and vertex_info =
  Initial
| Minsize of float
| Onsurface
| Onsurfacetmp of point

and point = float array
```

Voici les deux opérations de base sur un cube :

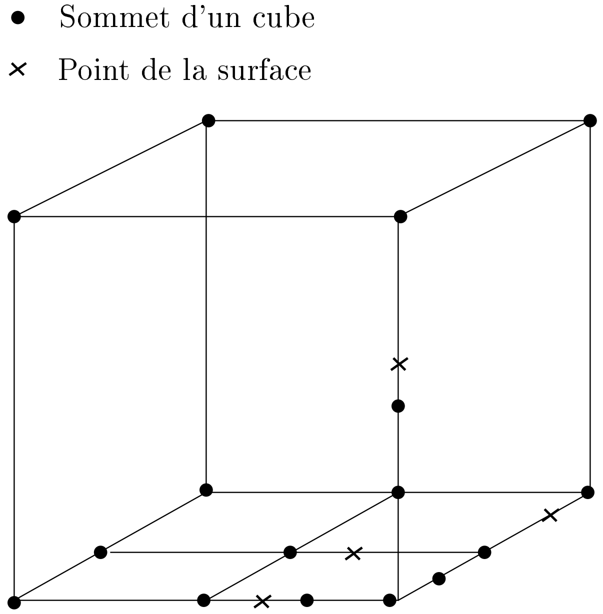

FIG. 2 – Un cube typique

La division du cube en 8 cubes, nécessitant éventuellement de diviser certaines faces et arêtes (si les cubes voisins n'ont pas encore été suffisamment divisés).

Pour ce faire, et pour ne pas mélanger les faces et les arêtes n'importe comment, on ne doit pas mettre dans un ordre quelconque les faces dans la structure `cube`, les arêtes dans la structure `face`, etc. En fait, n'importe quel choix convient, pourvu que ce choix ne soit pas relatif au cube que l'on regarde : si l'on considère une face comme la face supérieure d'un cube ou la face inférieure d'un autre cube, on doit choisir le même ordre pour les arêtes.

La recherche des racines : cela consiste à rechercher tous les points de la surface que l'on a trouvés sur une arête se trouvant sur les faces d'un cube. Les faces et les arêtes du cube pouvant avoir été subdivisées, il s'agit d'une petite fonction récursive assez simple.

La figure 2.1 représente un cube typique avec une face et des arêtes subdivisées. On a aussi indiqué les racines de f sur les arêtes.

2.2. Phase 1 : critère de division

On définit deux critères sur le cube : le premier critère (critère de division) permet de décider si l'on divise le cube en 8 pour l'explorer, le second (critère de conservation) décide si l'on conserve cette division. Le but de ces deux critères est de trouver toutes les composantes et les anses de la surface et de garantir que les points de la surface d'un même cube sont sur le même « morceau de surface ».

On doit aussi imposer une taille minimale au cube, sinon pour des surfaces singulières ou très proches de l'être, on risque de diviser les cubes à l'infini (il faut que cette taille minimale soit strictement supérieure à la taille du cube initial multiplié par la constante `epsilon_float`, sinon on va considérer des cubes dont les coordonnées des sommets ne sont plus distinguables par le type `float`) .

Critère de division : on utilise deux constantes `max_angle` et `out_max_angle`. Un cube satisfait le critère de division dans l'un des deux cas suivants :

1. Il existe deux points de la surface sur les faces du cube tels que le gradient de f en ces deux points fait un angle supérieur à `max_angle`. Le but de ce premier test est de diviser le cube si la surface est trop courbe.
2. Si $s = (x, y, z)$ est un sommet du cube, on dit qu'il est « proche de la surface » si $s - \frac{f(s)}{||\nabla f(s)||^2}.\nabla f(s)$ est à l'intérieur d'une sphère dont le centre est le centre du cube et le rayon une constante que l'on peut changer (au moins le rayon du cube c'est-à-dire $\sqrt{3}/2$ fois le côté du cube).

   Le second critère s'énonce alors ainsi : Il existe deux sommets du cube proches de la surface où f a le même signe ou un sommet du cube proche de la surface et un point de la surface tels que le gradient de f en ces deux points fait un angle supérieur à `out_max_angle`. Ce second test cherche à trouver des composantes (ou de petites anses) de la surface que l'on n'aurait pas encore détectées.

   On pourrait appliquer ce critère à d'autres points du cube que les sommets pour avoir plus de chance de détecter des composantes. Ceci n'est pas encore possible dans la version actuelle.

Critère de conservation : Il reprend seulement le premier point du critère précédent. En effet, si l'on a divisé un cube surtout pour rechercher de nouvelles composantes et que l'on n'en a pas trouvé, il est inutile de garder le cube divisé, cela consomme inutilement de la mémoire.

Pour implanter, il faut garder une « undo-list » des `face_status` et `edge_status` qui ont été modifiés afin de pouvoir restaurer leur ancienne valeur. De plus, pour économiser de la mémoire, en rendant inaccessible des pointeurs, dès que l'on conserve une division d'un cube, on sait que l'on conservera toutes les divisions de ses cubes « parents » et l'on peut alors simplifier la « undo-list ».

Un dernier problème se pose : il faut atteindre un point fixe. En divisant un cube, on peut découvrir de nouveaux points sur une face donnée, et ces points peuvent changer le critère de conservation pour l'autre cube partageant la même face. Il faudra donc le tester à nouveau. Dans la version actuelle, la recherche du point fixe se fait en testant tous les cubes plusieurs fois (au moins deux) jusqu'à ce qu'aucune division nouvelle ne soit conservée. On envisage, dans une prochaine version, de ne tester que les cubes voisins d'un cube dont on vient de conserver la division, car la recherche du point fixe par la méthode actuelle est parfois trop coûteuse.

Cette recherche de point fixe permet d'assurer que lorsque l'on détecte un morceau d'une composante (ou d'une anse) de la surface, on la triangulera en totalité. Il suffit donc que le critère de division soit assez puissant pour détecter une partie de chaque composante et de chaque anse de la surface.

2.3. Phase 2 : ajustement des sommets

Pour économiser des triangles, on tente de déplacer les sommets des cubes sur la surface en respectant les deux règles suivantes :

- Le déplacement d'un sommet du cube est inférieur à la moitié de la longueur de la plus petite arête d'un cube partant de ce sommet.
  Cette règle assure que les cubes ne sont pas trop déformés, ce qui est nécessaire (et non suffisant) pour garantir que les polygones que l'on considère à la phase 3 sont presque plats et convexes.
  Pour cela, le champ `info` de la structure `cvertex` représentant un sommet s d'un cube est positionné à la valeur `Min_size` x où x est la moitié de la longueur de la plus petite arête partant de s (Sauf pour les sommets qui sont déjà sur la surface pour lesquels ce champs est positionné à `Onsurface`). Ceci nécessite un premier examen de tous les cubes. Un second examen de tous les cubes permet de rechercher un point de la surface p assez proche de s et de positionner le champs `info` à `Onsurfacetmp` p si l'on en trouve un.

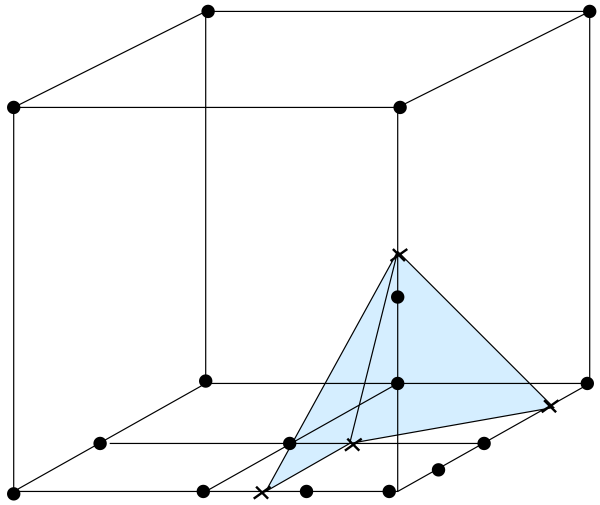

FIG. 3 – Extraction des triangles

– Le critère de conservation reste respecté, en utilisant la constante `adjust_angle` à la place de `max_angle`. En effet, il faut pouvoir déplacer un sommet, même si le cube satisfait de justesse le critère de conservation. On doit donc relâcher un peu la contrainte du critère de conservation.
  Cette règle permet d'éviter de rajouter dans un cube des points de la surface qui n'appartiennent pas au même « morceau de la surface » que les autres points déjà présents dans le cube.
  Remarque : on doit d'abord tenter de déplacer les sommets du cube les plus proches de la surface, pour éviter que le déplacement d'un sommet relativement loin empêche ensuite de déplacer des sommets plus proches.
  Pour ce faire, on parcourt à nouveau tous les cubes, et on vérifie si les sommets candidats au déplacement (le champ `info` vaut `Onsurface` p) satisfont le critère. Sinon, on repositionne le champs `info` à `Initial`.
  Finalement, par un dernier examen de tous les cubes, on déplace tous les sommets pour lesquels le champ `info` vaut `Onsurface` p et on repositionne le champs `info` à `Initial`.

2.4. Phase 3 : extraction des triangles

L'extraction des triangles, qui procède cube par cube, repose sur les hypothèses suivantes : tous les points de la surface d'un même cube appartiennent au même morceau de la surface (que l'on va donc trianguler), ce morceau est relativement plat et le cube est peu déformé. Pour des cubes qui ont atteint la taille minimum autorisée et que l'on n'a pas pu subdiviser, ces conditions ne sont pas respectées et les triangles extraits ne représentent pas, en général, fidèlement la surface.

Pour extraire les triangles d'un cube, on procède en deux étapes : on construit un polygone passant par les points de la surface que l'on a trouvés sur ce cube et séparant les sommets positifs des sommets négatifs. Attention, cela n'est pas simple, car il y a plusieurs solution et il faut choisir le polygone le plus plat. Une fois construit le polygone, il faut le trianguler et là encore ce n'est pas évident, car il n'est ni tout à fait plat ni tout à fait convexe en général, il faut donc rechercher les meilleurs triangles.

La figure 2.4 reprend le cube de la figure 2.1 et montre les triangles extraits de ce cube.

Nous ne détaillerons pas davantage cette partie de l'algorithme qui est assez technique et pour tout dire pas encore assez travaillée pour une présentation élégante.

3. Utilisation d'OpenGL

L'utilisation d'OpenGL ne pose pas de problème particulier grâce à l'utilisation de LablGL et LablGlut [2, 8]. Une exception toutefois, la notion de « tableau OpenGL » n'était pas intégrée.

Cette notion permet de stocker dans plusieurs tableaux les coordonnées des sommets, de leur normale, leur couleur, etc. Le gain est double :

1. La rapidité : ces tableaux sont réputés pour être aussi rapides que la notion de liste pré-compilée (`Gllist`).
2. Le gain en mémoire : les tableaux sont utilisables par OpenGl et le programme principal, évitant ainsi de stocker en double la liste des triangles.

Le problème principal est de réaliser une interface « type safe » pour ces fonctions OpenGL qui font des tests sur les types durant l'exécution (runtime). Toutefois, le module `Raw` de LablGL permet de résoudre ce problème. Ce module fournit un type de tableau `'a Raw.t` où `'a` est un variant polymorphe indiquant ce que le tableau contient (`[`int]` pour un entier 32 bits, `[`uint]` pour un entier non signé 32 bits, etc).

On trouvera en appendice B l'interface du module `GlArray` que nous avons créé pour GlSurf en utilisant le module `Raw` et qui sera intégré dans la prochaine distribution de LablGL[3].

Voici par exemple le code permettant de transformer une liste de triangles en une structure `surface` directement utilisable par OpenGL, comme le montre la fonction `draw_surface`. Ce code, comme tout code OpenGL, est essentiellement impératif puisqu'il contrôle «l'automate» d'OpenGL. En fait, les codes openGL se ressemblent beaucoup, quelle que soit le langage dans lequel ils sont écrits. On remarquera aussi l'utilisation d'une précision réduite à 3 octets pour stocker les normales et le paramètre `` `three `` de la fonction `GlArray.vertexPointer` indiquant qu'un sommet a trois coordonnées (il pourrait y en avoir deux ou quatre) :

```
type surface = {
  svertex : [`double] Raw.t; (* tableaux des coordonnées des sommets  *)
  snormal : [`byte] Raw.t;   (* tableaux des coordonnées des normales *)
  selements : [`uint] Raw.t; (* index des sommets des triangles       *)
}

let drawSurface s =
  GlArray.enableClientState `vertex;
  GlArray.vertexPointer `three s.svertex;
  GlArray.enableClientState `normal;
  GlArray.normalPointer s.snormal;
  GlArray.drawElements `triangles (Raw.length s.selements) s.selements;
  GlArray.disableClientState `vertex;
  GlArray.disableClientState `normal

let triangles_to_surface grad tl =
  (* une table de hash pour partager les sommets *)
  let tbl = Hashtbl.create 10001 in
  let count = ref 0 in
  let find_point p = try Hashtbl.find tbl p
    with Not_found -> let r = !count in incr count; Hashtbl.add tbl p r; r
  in
```

[3]Nous avons aussi contribué à l'intégration de LablGlut dans LablGL qui sera aussi disponible dans la prochaine version de LablGL

```
(* allocation du tableau des triangles: 3 entiers non signés par triangles *)
let selements = Raw.create ‘uint (List.length tl * 3) in
let pos = ref 0 in
let treat_one_triangle (v1, v2, v3) =
  Raw.set selements ~pos:!pos (find_point v1); incr pos;
  Raw.set selements ~pos:!pos (find_point v2); incr pos;
  Raw.set selements ~pos:!pos (find_point v3); incr pos;
in
List.iter treat_one_triangle tl;
let nb_vertex = !count in
(* allocation du tableau des sommets: 3 doubles par sommets *)
let svertex = Raw.create ‘double (3*nb_vertex) in
(* allocation du tableau des normales: 3 octets par sommets *)
let snormal = Raw.create ‘byte (3*nb_vertex) in
let treat_one_vertex ([|v1;v2;v3|] as p) index =
  Raw.set_float svertex ~pos:(3*index) v1;
  Raw.set_float svertex ~pos:(3*index+1) v2;
  Raw.set_float svertex ~pos:(3*index+2) v3;
  let g = grad p in (* calcul du gradient *)
  let [|x1;x2;x3|] = 127. ** g // norm g in
  let float_to_ubyte x =
    if x <= -127.0 then -127 else if x >= 127.0 then 127 else truncate x
  in
  Raw.set snormal ~pos:(3*index) (float_to_ubyte x1);
  Raw.set snormal ~pos:(3*index+1) (float_to_ubyte x2);
  Raw.set snormal ~pos:(3*index+2) (float_to_ubyte x3)
in
Hashtbl.iter treat_one_vertex tbl;
let s = {selements = selements; snormal = snormal; svertex = svertex} in
s
```

On remarque l'utilisation de variants polymorphes (`‘double`, `‘byte`, etc) permettant du côté C de passer le type des éléments du tableau à OpenGL et côté Objective Caml d'avoir une interface « type-safe ». Il semble que les variants polymorphes soient la bonne solution pour ce genre de problèmes.

Pour terminer cette section sur OpenGL, une petite remarque : notre logiciel permet aussi de visualiser plusieurs surfaces et des courbes simultanément. Dans ce cas, il est pratique de dessiner certaines surfaces transparentes. Tous les modèles de transparence que nous avons trouvés dans la littérature nécessite de trier les triangles pour les dessiner de l'arrière vers l'avant, ce qui peut être long. Nous avons trouvé un modèle de transparence qui évite ce problème (voir appendice C).

4. Compilation et édition de lien dynamique

Ce genre de logiciel nécessite toutes les optimisations possibles. Nous avons donc choisi de "compiler" (si un compilateur est disponible) les équations des surfaces à tracer. Pour ce faire, nous utilisons le module Expression de notre bibliothèque de calcul formel, ce qui nous permet de calculer et simplifier les dérivées formelles[4] de f (ceci est nécessaire pour calculer le gradient).

[4] Pour certaines formes de f, la dérivation numérique est plus rapide, si la taille de la dérivée est au moins trois fois plus importante que celle de f. Nous envisageons de proposer en option la dérivation numérique à la place de la dérivation formelle pour une version future.

Ensuite, nous produisons un fichier C calculant ces fonctions, que nous compilons et lions avec notre programme. Il faut noter que bien que la liaison dynamique de code Objective Caml ne soit pas disponible pour les programmes compilés avec `ocamlopt` (le compilateur de code natif), la liaison dynamique de programme C fonctionne très bien.

Sur l'exemple donné en appendice A, le temps de calcul de la triangulation de la surface sans compilation est de 1 minute et 21 secondes alors qu'il n'est que de 21 secondes avec la compilation (soit un facteur 4 !).

5. Économiser de la mémoire

Le problème principal de notre algorithme est sa consommation mémoire. L'exemple de l'appendice A nécessite dans la version 2.1 en cours de développement 37Mo de mémoire (7,2 million de mots pour le tas Objective Caml, et une partie du reste pour les tableaux OpenGL alloués par `malloc`). Ceci pour une surface comprenant 46021 triangles, l'espace ayant été divisé en 51996 cubes.

Ceci fait une moyenne de 138 mots par cube pour le tas, ce qui est conforme à ce que l'on observe sur d'autres exemples. Cela parait beaucoup ! Il fallait donc vérifier l'absence de fuite de mémoire. L'étude détaillée qui suit montre que si l'on tient compte de la mémoire occupée par les cubes, les faces, les arêtes et les sommets et aussi du fait que lorsque les cubes n'ont pas tous la même taille il faut stocker une grande face (resp. arête) subdivisée en quatre (resp. deux) petites qui peuvent elles-même être subdivisées, cette quantité mémoire semble légitime.

Il faut noter que la même surface dans les premières versions du logiciel nécessitait plus de 100Mo de mémoire. Les deux modifications majeures qui ont permis de réduire la consommation mémoire furent :

1. Initialement, nous utilisions une table de hachage pour trouver l'arête reliant deux sommets, nous n'avions pas de type `face` et un cube était une structure contenant directement ses huit sommets.

   Cette représentation des données peut paraître plus économique, mais il n'en est rien : la table de hachage prend sans doute autant de place que les faces de la nouvelle version, mais surtout nous n'avons trouvé aucun moyen raisonnable de supprimer les arêtes devenues inutiles de la table.

   Une arête devient inutile si ses deux sommets ne font plus partie d'un même cube accessible dans la mémoire. Cette condition est trop compliquée pour toutes les notions de table hachage faible («weak hash table») que nous connaissons.
2. Nous avons aussi modifié l'algorithme : la première version de l'algorithme n'avait pas de critère de conservation et un cube, une fois divisé le restait jusqu'à la fin. Là encore la gestion des « undo list » permettant de défaire la division du cube pouvait sembler coûteuse en mémoire, alors qu'il s'agit en fait d'un gain substantiel (10 à 30% suivant les surfaces).

Sans le typage fort d'Objective Caml, ces modifications successives et profondes de l'algorithme auraient été pénibles, alors qu'ici il suffisait de se laisser guider par les erreurs de types !

Une étude détaillée sur un cas simple. Considérons un groupe de $2 \times 2 \times 2 = 8$ cubes, dont un des cubes est lui-même divisé en 8 (voir figure 5). Dans cette situation, on compte :
- $(8 - 1) + 8 = 15$ cubes (le cube divisé ne compte plus).
- $(4 \times 3 \times 3 - 3) + 4 \times 3 \times 3 = 69$ faces dont 3 sont divisées. Le -3 correspond aux trois faces divisées sur le bord. Les trois faces divisées qui font encore partie d'un grand cube sont conservées.
- $(6 \times 3 \times 3 - 3) + 6 \times 3 \times 3 = 105$ arêtes dont 9 sont divisées (même remarque que pour les faces).
- $9 \times 3 + 5 \times 3 = 42$ sommets.

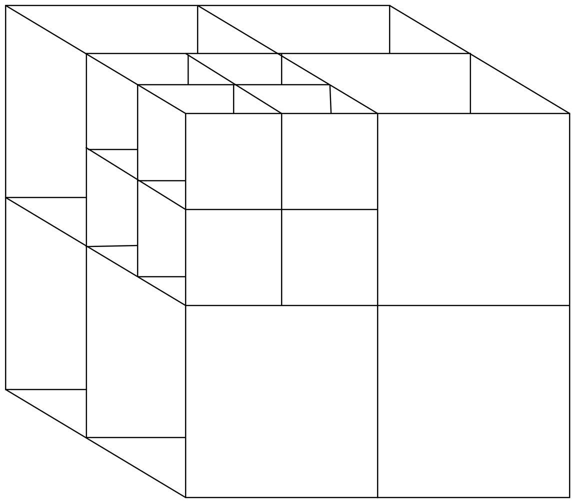

Fig. 4 – Cas d'étude

On peut alors faire le bilan du nombre de mots utilisés. Pour faire ce calcul, il ne faut pas oublier que chaque bloc contient un mot invisible (son « entête ») nécessaire au « runtime » d'Objective Caml. Il faut aussi remarquer qu'en plus de la taille de la structure cube elle même, on doit stocker une liste de tous les cubes.

type	taille unitaire (en mots)	taille totale (en mots)
`cube list`	3	45
`cube`	7	105
`face`	6	416
`face_status` (face divisée uniquement)	5	15
`edge`	4	420
`edge_status` (arête divisée uniquement)	3	27
`sommets`	10	420
total		1448

Cette représentation des données n'est pas optimale, en effet, 258 mots sont utilisés uniquement pour les entêtes des blocs Objective Caml. Si nous avions programmé notre algorithme en C avec un GC maison spécialisé pour notre structure de donnée, et stockant son bit de marquage dans le bit de point faible d'un pointeur, nous gagnerions 18% de mémoire (il faudrait quand même garder des entêtes pour les types `edge_status` et `vertex_info` qui possèdent plus d'un constructeur non constant. Cela ne représente que 9 mots sur notre cas d'étude).

Ce gain n'est pas négligeable, mais il est raisonnable vu ce que l'on gagne en programmant avec un langage de haut niveau avec GC transparent. De plus les entêtes permettent aussi d'avoir une fonction polymorphe d'écriture dans un fichier que nous utilisons ... pour les triangulations mais pas pour les cubes.

Au final, je pense qu'il vaut mieux perdre un peu de mémoire et rester en Objective Caml. Toutefois, Même dans le cadre du runtime d'Objective Caml, la représentation n'est pas toujours optimale. Une optimisation est possible pour stocker les coordonnées des sommets : si les sommets ne sont pas encore déplacés, les coordonnées sont des divisions entières de la taille du cube initial et on pourrait les stocker, sur trois fois 10 bits, à l'intérieur d'un entier, si le cube est divisé en deux, moins de dix fois.

Le type `point` serait alors

```
type point =
  Int of int
| Floats of float array
```

Dans ce cas, si l'on peut utiliser la représentation entière, un objet de type `cvertex` prend 5 mots (on gagne 5 mots) sinon il occupe 12 mots (on perd 2 mots). Ce n'est pas très intéressant en général. Sur notre exemple très favorable, on gagne 210 mots (15%).

En Objective Caml, un entier (représenté par un mot impair) ne peut se confondre avec un pointeur sur un tableau (représenté par un mot pair). On pourrait donc éviter une indirection et utiliser 3 mots (gain de 7) dans le premier cas et 10 dans le second (pas de perte). Au total, on gagne alors 294 mots (20%) sans rien perdre lorsque l'on doit utiliser des flottants. De plus, on pourrait rajouter un troisième cas avec un tableau de trois entiers (lorsque 10 bits ne suffisent plus), car on peut aussi distinguer un tableau de flottants d'un tableau d'entiers en Objective Caml (car les tableaux de flottants sont un cas particulier pour le « runtime » d'Objective Caml).

Nous envisageons éventuellement de recourir à cette représentation en utilisant le module Obj, quand nous aurons exploré toutes les autres améliorations possibles de l'algorithme, ce qui n'est pas pour tout de suite.

Une autre solution, plus satisfaisante, serait qu'Objective Caml optimise la représentation des type de données quitte à produire un fichier `xxx.h` contenant des macros permettant d'utiliser ce type en C si nécessaire (puisque l'on ne peut plus deviner la représentation d'un type de donnée). On pourrait alors programmer proprement tout en choisissant la représentation la plus compacte possible de nos données compatible avec le runtime du langage.

6. Perspectives

Nous pensons continuer le développement de GlSurf et plusieurs directions sont possibles :

1. Amélioration de l'algorithme actuel, par exemple pour un meilleur traitement des surfaces singulières.
2. Ajout d'un algorithme spécifique pour les surfaces algébriques (avec plus de garanties de correction du résultat).
3. Ajout d'un module lancé de rayon pour un meilleur rendu (en perdant la possibilité d'animation temps réel).
4. Possibilité de faire des animations (déplacement des surfaces, déformations, etc).
5. Traitement des courbes et des surfaces paramétrées.

Bien sûr, toute personne qui souhaite contribuer au développement de GlSurf, dans les directions ci-dessus ou d'autres, sera la bienvenue !

A. Un exemple de script GlSurf

```
let size = 20;
let min_size = 1e-6;
let max_angle = pi/6;
let out_max_angle = pi/2;
let tetra_angle = pi - (109.47122063449069174*pi/180);
let cz = cos(tetra_angle);
```

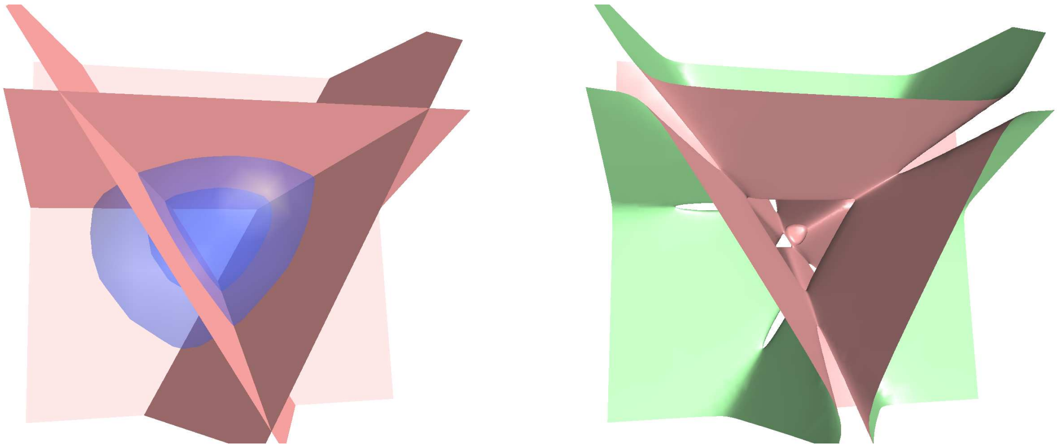

FIG. 5 – Les deux images produites par le script de l'appendice A

```
let cy = sin(tetra_angle);

(* quatre plans *)
let pt1(x,y,z) = z - 1;
let pt2(x,y,z) = -1*cy*y + cz*z + 1;
let pt3(x,y,z) = sin(2*pi/3)*cy*x -cos(2*pi/3)*cy*y + cz*z + 1;
let pt4(x,y,z) = sin(4*pi/3)*cy*x -cos(4*pi/3)*cy*y + cz*z + 1;

(* triangulation des quatres plans *)
surface pt1 = pt1(x,y,z);
surface pt2 = pt2(x,y,z);
surface pt3 = pt3(x,y,z);
surface pt4 = pt4(x,y,z);

(* leur produit *)
let q0(x,y,z) =  pt1(x,y,z)*pt2(x,y,z)*pt3(x,y,z)*pt4(x,y,z);

(* deux sphères*)
let r1 = 7;
let s1(x,y,z) = x^2 + y^2 + z^2 - r1^2;
let r2 = 4;
let s2(x,y,z) = x^2 + y^2 + z^2 - r2^2;

(* triangulation des deux sphères (transparente) *)
begin
  let transparent = true;
  let lmodel_twoside = false;
  vector front_diffuse = (0.2,0.3,0.8,0.5);
  surface s1 = s1(x,y,z);
  surface s2 = s2(x,y,z);
end

(* la quartique finale *)
```

```
let ra = 0;
let a = 0.001;
let q(x,y,z) = q0(x,y,z) + a*s1(x,y,z)*s2(x,y,z) + ra;

(* calcule de la triangulation de la quartique*)
surface q =  q(x,y,z);

(* dessin des quatres plans et des deux sphères *)
draw pt1,pt2,pt3,pt4,s1,s2;
(* dessin de la quartique seule *)
draw q;
```

B. Le module GlArray

```
(* Les trois types de tableaux *)
type kind =
  [ `color | `edge_flag | `index | `normal | `texture_coord | `vertex ]

(* Fonctions indiquant à OpenGL d'utiliser ou nom un tableau *)
external enableClientState : kind -> unit = "ml_glEnableClientState"
external disableClientState : kind -> unit = "ml_glDisableClientState"

(* GlArray.vertexPointer size tbl indique à OpenGL que tbl est un tableau
donnant les coordonnées des sommets (size coordonnées par sommets *)
external vertexPointer :
  [< `two | `three | `four] ->
  [< `double | `float | `int | `short ] Raw.t -> unit
  = "ml_glVertexPointer"

(* Comme vertexPointer mais pour les normales (toujours 3 coordonnées) *)
external normalPointer :
  [< `byte | `double | `float | `int | `short ] Raw.t -> unit
  = "ml_glNormalPointer"

(* Comme vertexPointer mais pour les "edge_flag" *)
external edgeFlagPointer : [ `bitmap ] Raw.t -> unit = "ml_glEdgeFlagPointer"

(* Comme vertexPointer mais pour les textures *)
external texCoordPointer :
  [< `one | `two | `three | `four] ->
  [< `double | `float | `int | `short ] Raw.t -> unit
  = "ml_glTexCoordPointer"

(* Comme vertexPointer mais pour les couleurs *)
external colorPointer :
  [< `three | `four] ->
  [< `byte | `double | `float | `int | `short | `ubyte | `uint | `ushort ] Raw.t
  -> unit = "ml_glColorPointer"

(* Comme vertexPointer mais pour les index de couleurs *)
```

```
external indexPointer :
  [< ‘double | ‘float | ‘int | ‘short | ‘ubyte ] Raw.t -> unit
  = "ml_glIndexPointer"

(* GlArray.arrayElement i
   indique à OpenGl de dessiner l’élément i des tableaux *)
external arrayElement : int -> unit = "ml_glArrayElement"

(* GlArray.drawArrays shape i j
   indique à OpenGl de dessiner les éléments i à j des tableaux *)
external drawArrays : GlDraw.shape -> first:int -> count:int -> unit
  = "ml_glDrawArrays"

(* GlArray.drawElements shape i tbl
   indique à OpenGl de dessiner les éléments tbl[0] à tbl[i-1] des tableaux *)
external drawElements :
  GlDraw.shape -> count:int -> [< ‘ubyte | ‘uint | ‘ushort ] Raw.t -> unit
  = "ml_glDrawElements"
```

C. Un modèle de transparence

Les modèles de transparence que j’ai pu trouver dans la littérature OpenGL nécessitent tous de dessiner les triangles de l’arrière vers l’avant. Le coût du tri des triangles étant trop important, j’ai chercher un modèle de transparence symétrique (le résultat est indépendant de l’ordre des triangles) et produisant un rendu raisonnable.

En regardant la spécification d’OpenGL [6], on voit que la couleur d’un point après dessin d’un triangle transparentest donnée par la formule suivante, les facteurs S et D étant calculés par la «fonction de mélange» (`blend_func`) :

$$C = C_S S + C_D D \text{ où } C_S \text{ est la couleur du triangle et } C_D \text{ la couleur initiale.}$$

Si l’on dessine deux triangles successivement, en fonction de l’ordre, on obtient :

$$C = C'_S S + (C_S S + C_D D)D \text{ ou } C_S S + (C'_S S + C_D D)D$$

Pour que le résultat ne dépendent pas de l’ordre des triangles, il y a deux solutions : $D = 1$ ou $C'_S = C_S$. La solution $D = 1$ seule n’est pas satisfaisante, car la couleur ne faisant qu’augmenter, on dépasse rapidement 1 et on obtient alors du blanc (phénomène de saturation).

Pour obtenir un résultat convenable, on peut dessiner en deux étapes :

1. Une première fois avec S la transparence du triangle et $D = 1 - S$, mais on ne garde dans C_S et C'_S que la couleur d’ambiance du triangle qui ne dépend pas de la position du triangle dans l’espace.
2. Une second fois avec S la transparence du triangle et $D = 1$, mais on ne garde dans C_S et C'_S que les autres composantes de la couleur du triangle (couleur de diffusion et tâches spéculaires).

Remarque, cela suppose que tous les triangles transparents ont la même couleur ou tout au moins que l’on trie les triangles par couleur pour les dessiner (mais ça, on peut le faire une fois pour toute).

Techniquement, après avoir dessiner tous les objets non transparents, on interdit l’écriture dans le `depth_buffer` (qui est utilisé pour détecter les parties cachées), on autorise le `blend`, et on dessine les triangles avec la `blend_func src_alpha + one_minus_src_alpha` alors que l’on ne garde que la couleur ambiante des triangles et une couleur de diffusion noire avec un paramètre de transparence adéquate. Ensuite, on redessine les mêmes triangles avec la `blend_func src_alpha + one` et une couleur ambiante nulle.

Cela donne le code suivant :

```
Gl.enable `depth_test;
Gl.enable `blend;
GlFunc.depth_mask false;
GlDraw.polygon_mode `both `fill;
let (_,_,_,fa) = s.front_diffuse in
let (_,_,_,ba) = s.back_diffuse in
GlLight.material ~face:`front (`ambient s.front_ambient);
GlLight.material ~face:`back (`ambient s.back_ambient);
GlLight.material ~face:`front (`diffuse (0.0,0.0,0.0,fa));
GlLight.material ~face:`back (`diffuse (0.0,0.0,0.0,ba));
GlLight.material ~face:`both (`shininess 0.0);
GlLight.material ~face:`both (`specular (0.0,0.0,0.0,0.0));
GlFunc.blend_func `src_alpha `one_minus_src_alpha;
Glarray.drawSurface tl;
GlLight.material ~face:`both (`ambient (0.0,0.0,0.0,0.0));
GlLight.material ~face:`front (`diffuse s.front_diffuse);
GlLight.material ~face:`back (`diffuse s.back_diffuse);
GlLight.material ~face:`front (`shininess s.front_shininess);
GlLight.material ~face:`back (`shininess s.back_shininess);
GlLight.material ~face:`front (`specular s.front_specular);
GlLight.material ~face:`back (`specular s.back_specular);
GlFunc.blend_func `src_alpha `one;
Glarray.drawSurface tl;
```

Références

[1] Jules Bloomenthal. An implicit surface polygonizer. In Paul Heckbert, editor, *Graphics Gems IV*, pages 324–349. Academic Press, Boston, 1994.

[2] Jacques Garrigue. LablGL. A library for Objective Caml.

[3] A. Hilton and J. Illingworth. Marching triangles : Delaunay implicit surface triangulation.

[4] W.E. Lorensen and H.E. Cline. Marching Cubes : a high resolution 3D surface reconstruction algorithm. *Computer Graphics*, 21(4) :163–169, 1987. Proc. of SIGGRAPH.

[5] C. Montani, R. Scateni, and R. Scopigno. Discretized marching cubes. In R. D. Bergeron and A. E. Kaufman, editors, *Visualization '94 Proceedings*, pages 281–287, Washington D. C., USA, 1994. IEEE Computer Society, IEEE Computer Society Press.

[6] Mark Segal and Kurt Akely. *The OpenGL Graphics System : A Specication (Version 1.2)*, 1998.

[7] Endrass Stephan, Huelf Hans, Oertel Ruediger, Schmitt Ralf, Schneider Kai, and Beigel Johannes. Surf. A software to draw algebraic implicit surface by ray tracing.

[8] Isaac Trotts. LablGlut. A library for Objective Caml.