BAZEILLE Stéphane
MOUGEL Baptiste

IUP3

ALGORITHMES

POUR LA

VISUALISATION

SCIENTIFIQUE

EN

Année 2003/2004

TABLE DES MATIERES

Home.

Les algorithmes pour la visualisation scientifique par BAZEILLE Stéphane et MOUGEL Baptiste

Etudiants en Maîtrise Génie Informatique option Imagerie Numérique a l'IUP GI La Rochelle.

Rapport final résumant l'ensemble du module du Module Visualisation Scientifique encadré par M. M.EBOUEYA.L'ensemble des programmes présentés sont écrits en langage C++ avec et utilisent la librairie graphique Open GL.

Introduction.

Qu'est ce que la visualisation scientifique ?

La visualisation scientifique est un processus d'exploration de transformation et de visionnage de données sous forme d'images(ou sous d'autres formes, matrice de données par exemple …) pour comprendre la nature profonde de ces données.

Dans ce document nous allons étudier trois algorithmes, le marching square (visualisation en 2D), et deux de ses extensions en visualisation 3d, le marching cubes et le marching Tetrahedra. Ces trois algorithmes sont donc assez similaire et répondent aux problématiques de visualisation scientifiques notamment la représentation d'isosurface.

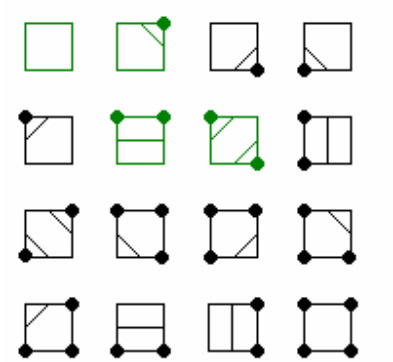
Pour chacun de cet algorithme nous analyserons leurs caractéristiques et leurs mises en œuvre, puis expliquerons quelques exemples d'utilisation, pour conclure nous tenterons de les comparer et de définir leurs points forts et leurs points faibles.

Marching Square.

Algorithme.

L'algorithme du Marching Square traite chaque cellule d'un maillage indépendamment des autres. Dans chacune , il détermine pour une valeur de niveau (ou seuil) donnée , si la (ou les) lignes passe(nt) dans la cellule , en comparant les valeurs aux sommets de la cellule et de celle du niveau.

Dans cet algorithme , un certain nombre de cas peuvent être déduits d'autres par permutations des sommets de la cellule traitée. On peut ainsi se ramener a quatre cas de base.
Ci dessous les 16 cas du Marching Square.



Les 16 cas du marching square
En vert les 4 cas de base.

L'algorithme se décompose en 5 étapes :

- Sélectionner la cellule.
- Pour chaque sommet déterminer son état (+) ou (-) par rapport a la valeur de référence.
- Créer un index en stockant l'état binaire de chaque sommet dans un bit.
- Utiliser cet index pour consulter l'état topologique de la cellule dans la table des cas.
- Calculer la place du contour su chacune des arêtes de la table des cas.

Programmation.

Dans notre programme nous disposons des structures suivantes :

- Structure POINT2D contenant les 2 coordonnées x et y d'un point.
- Structure GRILLE contenant 4 Point2D et leurs 4 poids associés
- Structure GRILLEVAL contenant un Point2D et son poids associé.

Cette structure est utilisé lors de la lecture du fichier de donnée ci dessous.

Fichier Data utilisé pour le Marching Square.

```
7 7
0-0-3.01916--0-1-2.39619--0-2-2.05086--0-3-2.19015--0-4-2.69360--0-5-3.35122--0-6-3.84358--
1-0-3.02469--1-1-2.53018--1-2-2.22640--1-3-2.32983--1-4-2.74447--1-5-3.27486--1-6-3.67591--
2-0-3.01960--2-1-2.85490--2-2-2.76266--2-3-2.79300--2-4-2.92218--2-5-3.07573--2-6-3.20750--
3-0-3.01939--3-1-3.29857--3-2-3.42150--3-3-3.33166--3-4-3.12826--3-5-2.81074--3-6-2.60939--
4-0-2.98634--4-1-3.57370--4-2-3.89092--4-3-3.75675--4-4-3.27829--4-5-2.66569--4-6-2.20287--
5-0-2.96770--5-1-3.56760--5-2-3.90497--5-3-3.77949--5-4-3.29418--5-5-2.68257--5-6-2.19999--
6-0-2.97887--6-1-3.29238--6-2-3.50024--6-3-3.43863--6-4-3.17162--6-5-2.82848--6-6-2.57565--
```

Explication des fonction utilisées :

- POINT2D Interpolation (Point2D 1, Point2D 2, Poids 1, Poids 2 , Isovaleur)

Cette fonction calcule les coordonnés d'un point entre 2 sommets par rapport a leurs poids et a l'isovaleur .

- VOID TraceSquare (Isovaleur, Grille g)

Cette fonction parcours les 4 sommets de la grille et calcul si elle existe les interpolations associés.
Elle dessine ensuite si elle(s) existe(nt) la ou les 2 lignes traversant la grille.
C'est donc cette fonction qui dessine 1 des 16 cas vu précédemment.

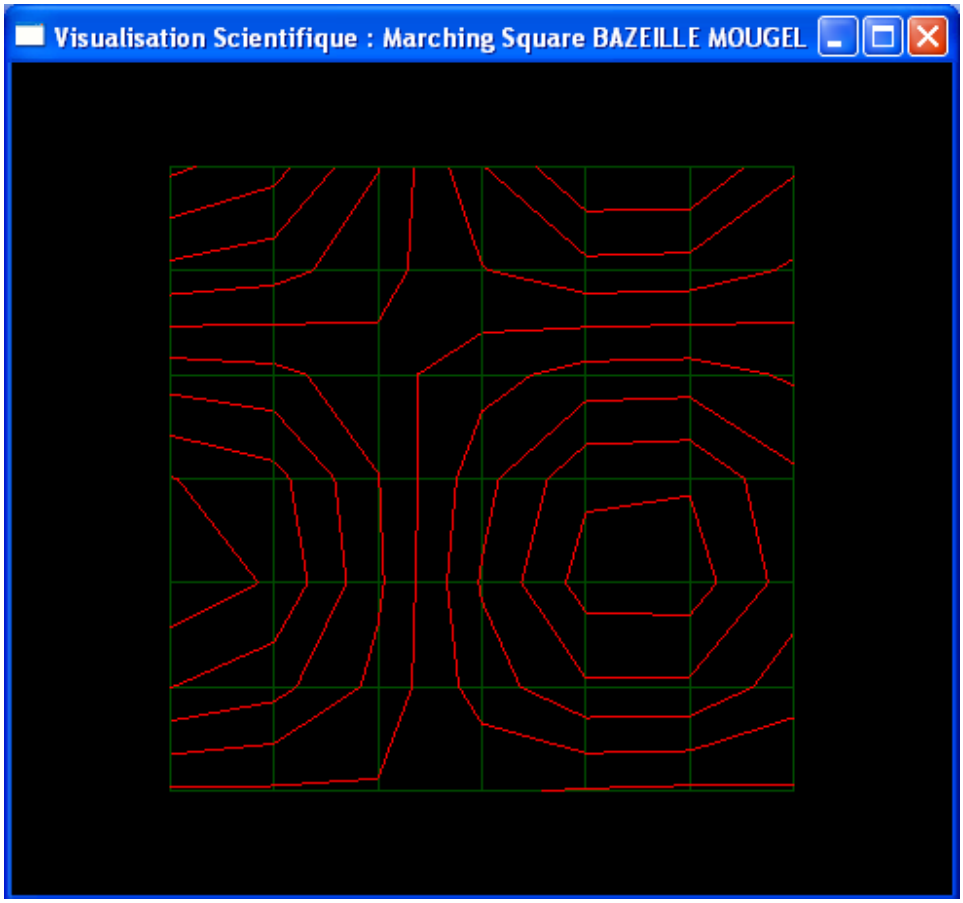
```
Point2D Interpol(double Isoval,Point2D p1,Point2D p2,double v1,double v2)
{
    double diff;
    Point2D p;

    if(fabs(v1-Isoval)<e) return p1;
    if(fabs(v2-Isoval)<e) return p2;
    if(fabs(v2-v1)<e)     return p1;

    /*calcul de l'interpolation 2D*/
    diff=(Isoval-v1)/(v2-v1);
    p.x=p1.x+diff*(p2.x-p1.x);
    p.y=p1.y+diff*(p2.y-p1.y);
    return p;
}

void traceSquare(double Isoval,Grille g )
{
    Point2D trace[4];
    int nbPoint=0;

    if(g.poids[0]>Isoval)
    {
        if(g.poids[1]<=Isoval)
        {
            trace[nbPoint++]=Interpol(Isoval,g.pt[0],g.pt[1],g.poids[0],g.poids[1]);
        }
        if(g.poids[3]<=Isoval)
        {
            trace[nbPoint++]=Interpol(Isoval,g.pt[0],g.pt[3],g.poids[0],g.poids[3]);
        }

    }
    if(g.poids[1]>Isoval)
    {
        if(g.poids[0]<=Isoval)
        {
            trace[nbPoint++]=Interpol(Isoval,g.pt[1],g.pt[0],g.poids[1],g.poids[0]);
        }
```

Ainsi que les fonctions suivantes :

- DrawLine .....(dessine une ligne entre deux points)
- DrawCarre ..... (dessine un carre)
- InitGrilleVal ..... (lecture des données dans le fichier data et stockage dans une matrice)
- InitGrille(initialise une grille avec les points et les poids associés lu dans la matrice précédente)
- Display ..... (fonction d'affichage)
- Reshape .....(fonction de redimensionnable de la fenêtre d'affichage)
- Main .....(fonction de gestion du programme).

Résultat obtenu.

Pour l'exemple d'affichage ci dessous nous avons dessiné les lignes de niveaux pour les isovaleurs : 2 , 2.2 , 2.4 , 2.6 , 2.8 , 3.0 , 3.2 , 3.4 , 3.6 et 3.8.



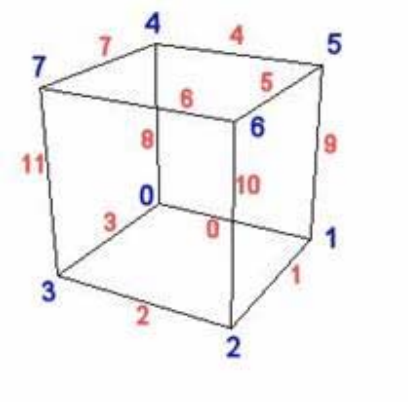
Marching Cube.

Algorithme.

L'algorithme du Marching Cubes a été inventé par Bill LORENSEN et Harvey CLINE. Il s'agit d'une méthode surfacique permettant d'extraire une surface équipotentielle (isosurface) d'un maillage structuré et uniforme 3D. Le Marching Cubes est une extension 3D de la technique du Marching Squares 2D vu dans la partie précédente.

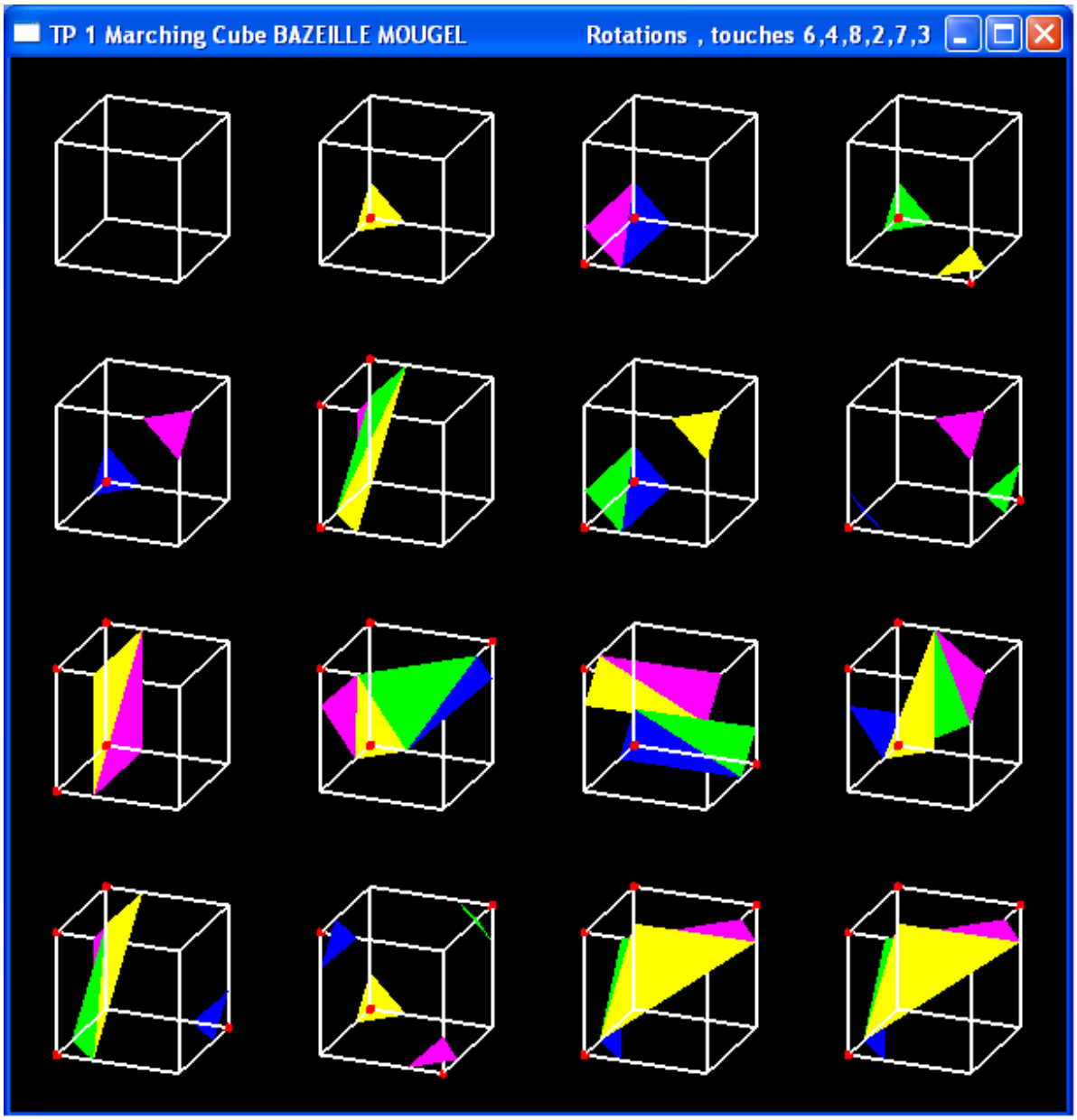
Le principe est de calculer, les différentes configurations que peut prendre l'isosurface dans un élément de volume, selon la répartition des intersections de l'isosurface sur les arrêtes de cet élément de volume.

Pour uniformiser les expressions, le code et les calculs nous utiliserons la numérotation des sommets et arêtes suivantes :



Dans notre cas, l'élément de volume est un cube, nous avons donc 8 sommets, 12 arêtes et chaque sommet peut prendre 2 états.

Cela donne donc 2^8 = 256 configurations possibles, qui sont autant de façon pour une surface d'intersecté les arêtes du volume. Chaque configuration correspond à un ensemble de facettes tracées à l'intérieur du volume, mais la présence de nombreux cas symétriques permet de se ramener à 16 configurations de base.



Les 16 cas de base dans l'algorithme du Marching Cubes

Chaque sommet en rouge indique que son isovaleur est inférieure à l'isosurface.

```
#ifndef __DEFINE_MARCHING_CUBES_H__
#define __DEFINE_MARCHING_CUBES_H__

const int TableDesArretes[256]={
0x0 , 0x109, 0x203, 0x30a, 0x406, 0x50f, 0x605, 0x70c,
0x80c, 0x905, 0xa0f, 0xb06, 0xc0a, 0xd03, 0xe09, 0xf00,
0x190, 0x99 , 0x393, 0x29a, 0x596, 0x49f, 0x795, 0x69c,
0x99c, 0x895, 0xb9f, 0xa96, 0xd9a, 0xc93, 0xf99, 0xe90,
0x230, 0x339, 0x33 , 0x13a, 0x636, 0x73f, 0x435, 0x53c,
                    ... etc ...

const int TableDesTriangles[256][16] =
{{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

                    ... etc ...
```

Programmation.

Les Structures utilisées :

- Point3D qui contient 3 entiers , les coordonnées dans l'espace d'un point.
- Grille qui représente un cube et contient 8 Point3D et leurs 8 poids associés.
- Triangle qui contient 3 Point3D.

Les fonctions utilisées :

- Fonction Interpolation Linéaire (Comme dans Marching Square mais en 3D)
- Fonction IndexCube

Cette fonction permet de générer un octet dont chaque bit représente l'état d'un des sommets du cube. Si la poids du sommet est supérieur a l'isovaleur le bit est a 1 sinon il est a 0.
Cet index cube est ensuite utilisé par la table des arêtes.
TableDesAretes[IndexCube] indique le numéro des arêtes intersectées par l'isosurface.

Comme il y a 12 arêtes différentes dans un cube les données contenues dans la table des arêtes sont sur 12 bits soit 3 caractères en hexadécimal.

```
Point3D Interpol(double Isoval,Point3D p1,Point3D p2,double v1,double v2)
{
    double diff;
    Point3D p;

    if(fabs(v1-Isoval)<e) return p1;
    if(fabs(v2-Isoval)<e) return p2;
    if(fabs(v2-v1)<e)     return p1;

    /*calcul de l'interpolation 3D*/
    diff=(Isoval-v1)/(v2-v1);
    p.x=p1.x+diff*(p2.x-p1.x);
    p.y=p1.y+diff*(p2.y-p1.y);
    p.z=p1.z+diff*(p2.z-p1.z);
    return p;
}


static unsigned char indexCube(Grille grille,double isoval)
{
    unsigned char index=0;

    if(grille.poids[0]<isoval)index|=1;
    if(grille.poids[1]<isoval)index|=2;
    if(grille.poids[2]<isoval)index|=4;
    if(grille.poids[3]<isoval)index|=8;
    if(grille.poids[4]<isoval)index|=16;
    if(grille.poids[5]<isoval)index|=32;
    if(grille.poids[6]<isoval)index|=64;
    if(grille.poids[7]<isoval)index|=128;
    return index;
}
```

- Fonction Calcul Points d'intersection

Sachant le numéro des arêtes qui sont intersectées on peut maintenant calculer la liste des points d'intersection entre l'isosurface et le cube élémentaire examiné.
Cette fonction fait donc appel a Interpolation Linéaire et remplie un tableau contenu l'ensemble des points d'intersection.

- Fonction Draw Marching Cubes

Cette fonction dessine l'ensemble des facettes sur le cube élémentaire examiné.
Elle fait appel a notre deuxième table vu précédemment appelé TableDesTriangles car cette contient des liste ordonnées des arêtes intersectées. Ces listes permettent de calculer les points d'intersection puis de générer des facettes triangulaires.

```
void drawMarchingCube(Grille cel, double isoVal)
{
    int i;
    int nbTriangles=nombreDeTriangles(cel,isoVal);
    unsigned char IndexCube=indexCube(cel,isoVal);
    Point3D *ListeDesPointsIntersction=pointsIntersection(IndexCube,cel,isoVal);
    Triangle3D triangles;

    for(i=0;TableDesTriangles[IndexCube][i]!=-1;i+=3)
    {
        triangles.pt[0] = ListeDesPointsIntersction[TableDesTriangles[IndexCube][i]];
        triangles.pt[1] = ListeDesPointsIntersction[TableDesTriangles[IndexCube][i+1]];
        triangles.pt[2] = ListeDesPointsIntersction[TableDesTriangles[IndexCube][i+2]];
        drawTriangle(triangles);
    }

}
```

Egalement les fonctions.

- DrawPoint qui dessine un point.
- InitPoids qui initialise les poids pour les 16 configurations de base du Marching cubes.
- InitCube pour l'initialisation des coordonnées d'un cube.
- DrawTriangle qui dessine une facette.
- DrawCube qui dessine un cube.
- Display fonction qui gère l'ensemble de l'affichage du programme.
- Clavier fonction qui gère l'ensemble des commandes clavier.
- Reshape fonction de redimensionnement de la fenêtre d'affichage.
- Et la fonction Main qui gère l'exécution du programme.

Affichage d'une sphère par l'algorithme du Marching Cubes.

Nous disposons d'un maillage 3D uniforme comportant des informations. Chaque élément de volume du maillage est parcouru. Pour chacun de ces volumes, les poids des sommets du cube élémentaire sont comparés à l'isovaleur. En fonction des ces tests, un index binaire (8 bits=1 octet) est créé (bit à 1 si la valeur du sommet est inférieure à l'isovaleur, 0 sinon).
A partir de cet index, on consulte la table qui indique l'état topologique de l'élément de volume (quelles arrêtes sont intersectées).
Pour chaque arête intersectée, on calcule par interpolation linéaire, les coordonnées du point d'intersections et ensuite il suffit de dessiner ou de stocker la facette.
L'ensemble du volume que l'on veut représenter est découpé en un nombre important de facettes.
En résumé :
Pour chaque volume du maillage :

- Créer un index (8 bits) contenant l'état de chaque sommet.
- Trouver les arrêtes intersectées et calculer les coordonnées d'intersection.
- Dessiner les facettes.

Résultat obtenu (exemple de la sphère).

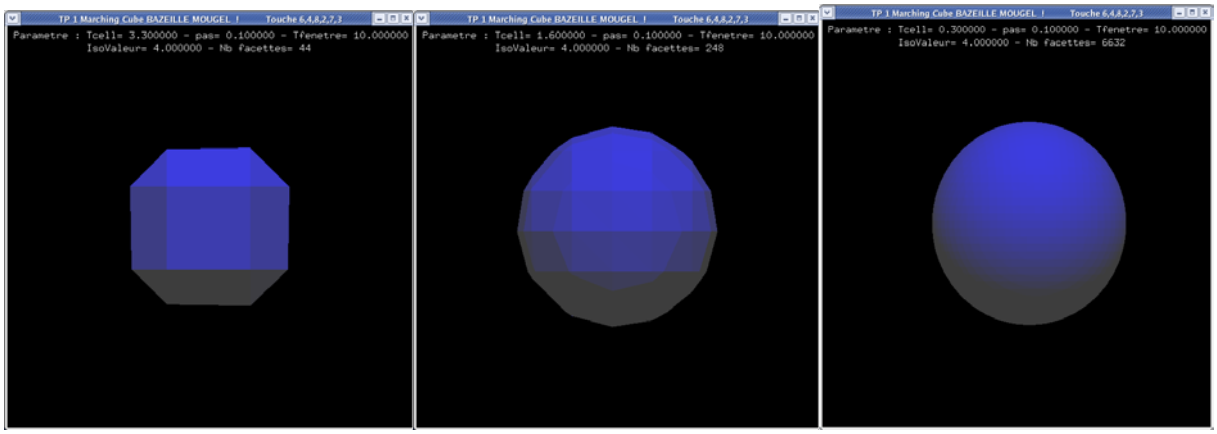
L'équation de sphère est un bon moyen de mettre en valeur l'influence de l'isovaleur de référence car l'isosurface affiché correspond à une sphère qui pour rayon l'isovaleur de référence (comme si toutes les sphères étaient imbriquées les unes dans les autres). Elle permet aussi de voir nettement l'influence de la taille du cube élémentaire.

L'équation de la sphère est la suivante $x^2 + y^2 + z^2 = r^2$

On utilise dans ce programme des fichiers Lumière et Calcul de Normal pour un meilleur rendu d'affichage.

Ci dessous 3 exemples d'une sphère par le Marching Cubes avec :

1. Taille de cellule élémentaire 3.3 , sur taille fenêtre de 10 > 44 facettes.
2. Taille de cellule élémentaire 1.6 , sur taille fenêtre de 10 > 248 facettes.
3. Taille de cellule élémentaire 0.3 , sur taille fenêtre de 10 > 6632 facettes.



1 2 3

Marching Tetrahedra.

Algorithme.

L'algorithme du Marching Tetrahedra dérive de l'algorithme du Marching Cubes. Le principe est de diviser le cube élémentaire en six tétraèdres. Chaque tétraèdre se comporte alors comme un volume élémentaire, les propriétés et valeurs des sommets et des arêtes sont inchangées, et ce volume peut être intersecté par l'isosurface.

Comme pour le Marching Cube, il faut trouver les différentes configurations que peut prendre l'isosurface dans un élément de volume, selon la répartition des intersections de l'isosurface sur les arrêtes du volume.

Dans ce cas, le volume élémentaire est un tétraèdre, il y a donc 4 sommets, 6 arêtes et chaque sommet peut prendre 2 états. Cela donne ainsi 16 configurations possibles d'intersections entre le volume et l'isosurface. On peut se ramener facilement à huit configurations grâce aux cas symétriques.

Avec cette méthode le nombre de facettes triangulaires augmente considérablement. Pour le Marching Cubes il y a au maximum 5 facettes par cube élémentaire alors que pour le Marching Tetrahedra il y a au maximum 6*2 = 12 facettes par cube élémentaire.

De plus, cette méthode contrairement au Marching Cubes peut entraîner l'existence de bosses artificielles sur l'isosurface car l'interpolation est faite sur la diagonale d'une face et non sur la face elle même.

L'algorithme est plus simple que le précèdent car aucune table n'est à créer ou à consulter.

Chaque élément de volume du maillage est parcouru. Pour chacun de ces volumes et pour chaque tétraèdre, les valeurs des sommets sont comparées à l'isovaleur. En fonction des ces tests, un index binaire (4 bits) est créé (bit à 1 si la valeur du sommet est inférieure à l'isovaleur, 0 sinon).

Pour chaque arête intersectée, on calcule par interpolation linéaire, les coordonnées du point d'intersections et ensuite il suffit de dessiner ou de stocker la facette.

En résumé :

Pour chaque volume du maillage et pour chaque tétraèdre :

- Créer un index (4 bits) contenant l'état de chaque sommet.
- Trouver les arêtes intersectées
- Calculer les coordonnées d'intersection dans le tétraèdre.
- Dessiner les facettes.

Programmation.

Les structures utilisées sont les même que pour le Marching Cubes c'est a dire les structures Point3D,Grille,Triangle et nous avons rajouter la structure Tetrahedre qui contient 4 Poin3D et de leurs valeurs associées. Nous utilisons aussi la table des sommets suivante :

static int tableDesSommets [6][4] ={{0,2,3,7},{0,6,1,4},{5,6,1,4},{0,2,6,7},{0,4,6,7},{0,6,1,2}};

Les fonctions utilisées :

- Interpolation Linéaire c'est la même fonction que pour Marching Cubes.
- IndexTetra , cette fonction retourne un index contenant l'état de chaque sommet du tetrahedre.
- Point d'Intersection , cette fonction calcule l'ensemble des points d'intersection selon l'IndexTetra calculé précédemment.

```
unsigned char indexTetrahedre(Tetrahedre tetra,double isoval)
{
    unsigned char index=0;

    if(*(tetra.v0)<isoval)index|=1;
    if(*(tetra.v1)<isoval)index|=2;
    if(*(tetra.v2)<isoval)index|=4;
    if(*(tetra.v3)<isoval)index|=8;
    return index;
}

/*Calcul les points d'intersection*/
int pointsIntersection(unsigned char indexTetrahedre,Tetrahedre tetra,double isoval,
                       Triangle3D **triangles)
{
    (*triangles)=NULL;

    switch (indexTetrahedre)
    {
        /* 0 facette concerne*/
        case 0x00 :
        case 0x0f :
            return 0;
        break;

        /* 1 facette concerne*/
        case 0x01 :
        case 0x0e :
            (*triangles)=(Triangle3D*)malloc(sizeof(Triangle3D));

            (*triangles)->pt[0]=Interpol(isoval,*tetra.p0,*tetra.p1,*tetra.v0,*tetra.v1);
            (*triangles)->pt[1]=Interpol(isoval,*tetra.p0,*tetra.p2,*tetra.v0,*tetra.v2);
            (*triangles)->pt[2]=Interpol(isoval,*tetra.p0,*tetra.p3,*tetra.v0,*tetra.v3);
            return 1;
        break;

        case 0x02 :
        case 0x0d :
            (*triangles)=(Triangle3D*)malloc(sizeof(Triangle3D));
```

- DrawMarchingTetra. Cette fonction remplie et dessine un tetrahedre.

```
void drawMarchingTetrahedra(Grille cel, double isoVal,int indicetetra)
{
    int j;
    int nbTriangles;
    Triangle3D *triangles=NULL;
    Tetrahedre tetra;

    {
        /*initialisation du tetrahedre*/
        tetra.p0=&cel.pt[tableDesSommets[indicetetra][0]];
        tetra.p1=&cel.pt[tableDesSommets[indicetetra][1]];
        tetra.p2=&cel.pt[tableDesSommets[indicetetra][2]];
        tetra.p3=&cel.pt[tableDesSommets[indicetetra][3]];

        tetra.v0=&cel.poids[tableDesSommets[indicetetra][0]];
        tetra.v1=&cel.poids[tableDesSommets[indicetetra][1]];
        tetra.v2=&cel.poids[tableDesSommets[indicetetra][2]];
        tetra.v3=&cel.poids[tableDesSommets[indicetetra][3]];

        drawTetra(tableDesSommets[indicetetra][0],tableDesSommets[indicetetra][1],
                  tableDesSommets[indicetetra][2],tableDesSommets[indicetetra][3]);

        nbTriangles=pointsIntersection(indexTetrahedre(tetra,isoVal),tetra,isoVal,&triangles

        printf("nb %d\n",nbTriangles);
        for(j=0;j<nbTriangles;j++)
        {
            drawTriangle(triangles[j]);
        }
        free(triangles);
    }
}
```

Comme dans Marching Cubes nous disposons de nombreuses procédures d'affichage.

Résultat obtenu (exemple de la sphère).

Comme dans Marching Cubes on utilise dans ce programme des fichiers Lumière et Calcul de Normal pour un meilleur rendu d'affichage.

Ci dessous 3 exemples d'une sphère par le Marching Tetra avec :

1. Taille de cellule élémentaire 3.3 , sur taille fenêtre de 10 > 114 facettes.
2. Taille de cellule élémentaire 1.6 , sur taille fenêtre de 10 > 248 facettes.
3. Taille de cellule élémentaire 0.3 , sur taille fenêtre de 10 > 19920 facettes.

En comparaison, le Marching Cubes faisait respectivement 44 , 248 , 6632 avec les mêmes tailles pour de cellule élémentaire et la fenêtre.

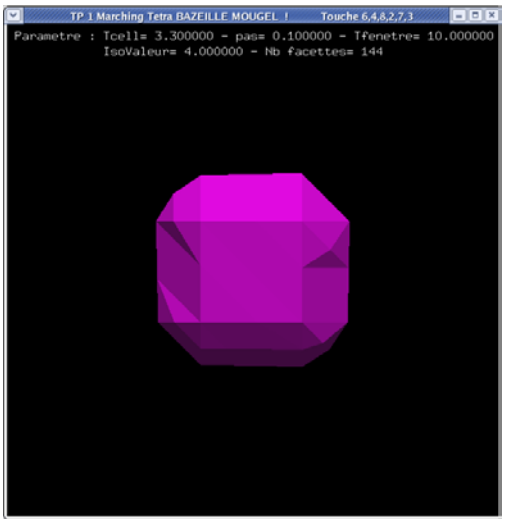


figure 1

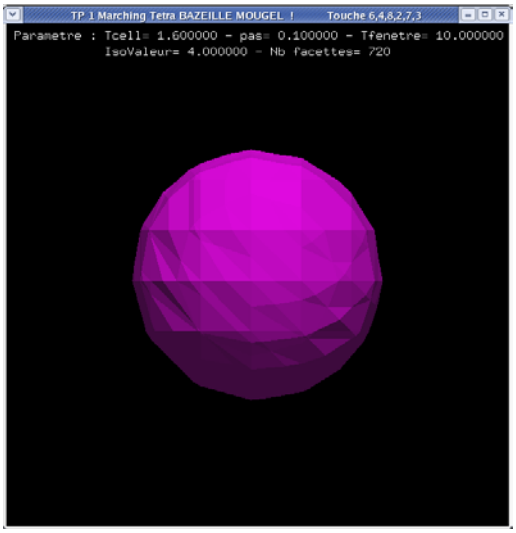


figure 2

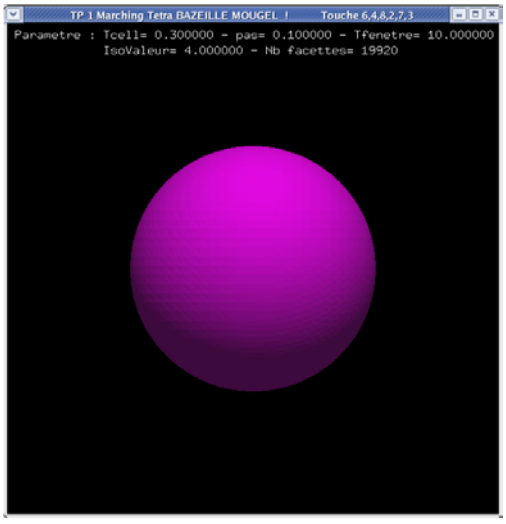


figure 3

Exemple d'utilisation supplémentaires.

Avec fonctions mathématiques.

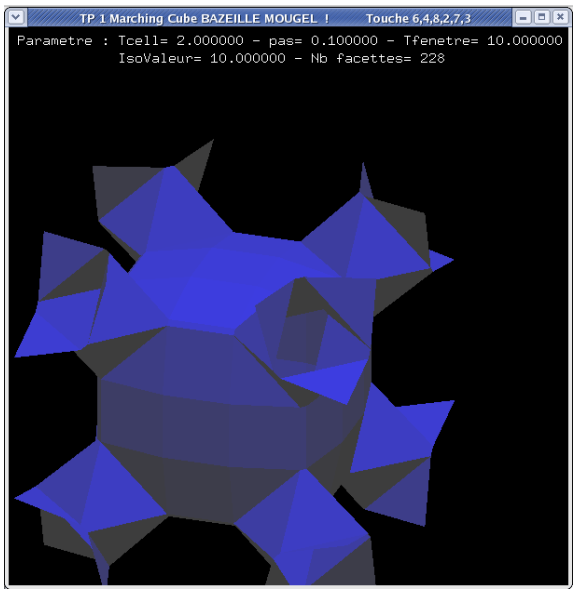
Nous avons déjà pu vérifier la validité des deux algorithmes en générant une figure de bases , la sphère , mais nous allons maintenant les tester équations un peu plus complexes.

Chromosome :

Formule :

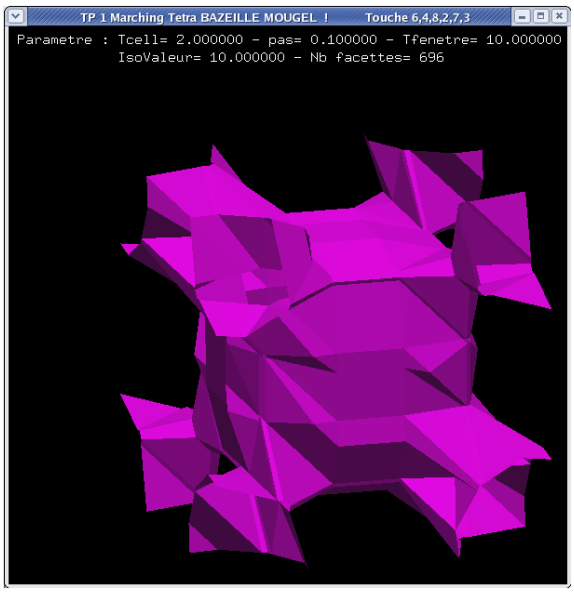
$(x^2 + y^2 + z^2) * (((\cos(x) / x)^2) + ((\cos(y) / y)^2) + ((\cos(y) / y)^2)) = r^2$

Cas du cube



Nombre de facette = 229

Cas du Tetrahedra



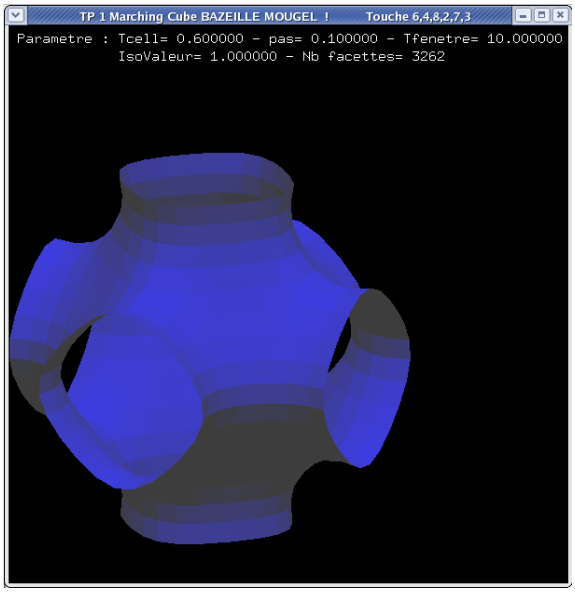
Nombre de facette = 696

Croix :

Formule:

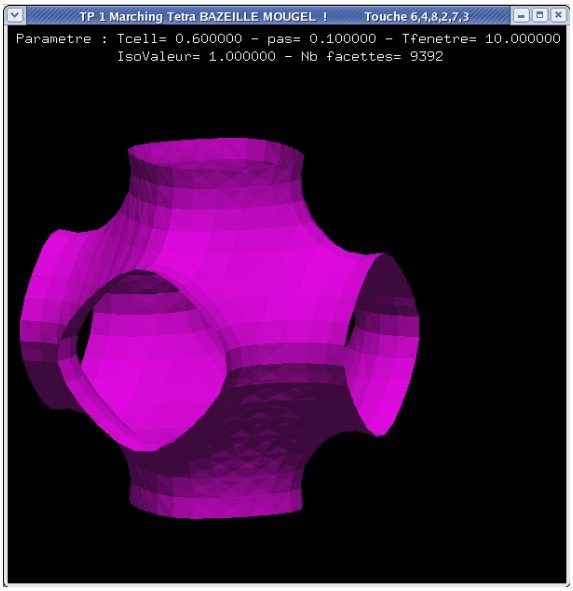
$$\sin(x)/x + \sin(y)/y + \sin(z)/z = r^2$$

Cas du cube



Nombre de facette = 3262

Cas du Tetrahedra



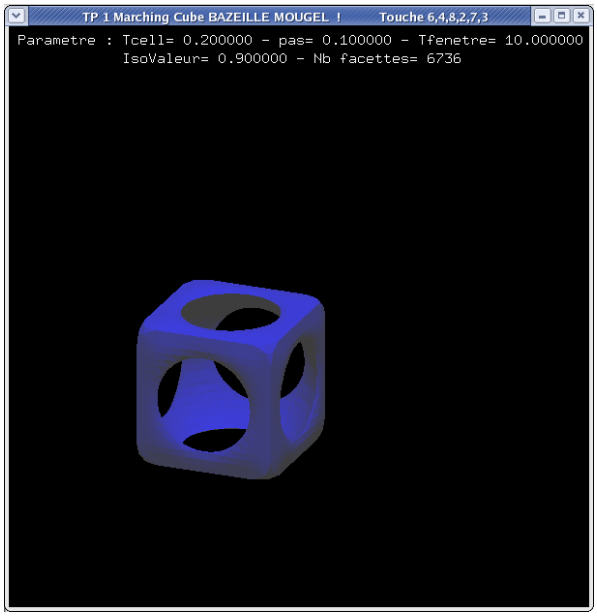
Nombre de facette = 9392

Cube Percé :

Formule:

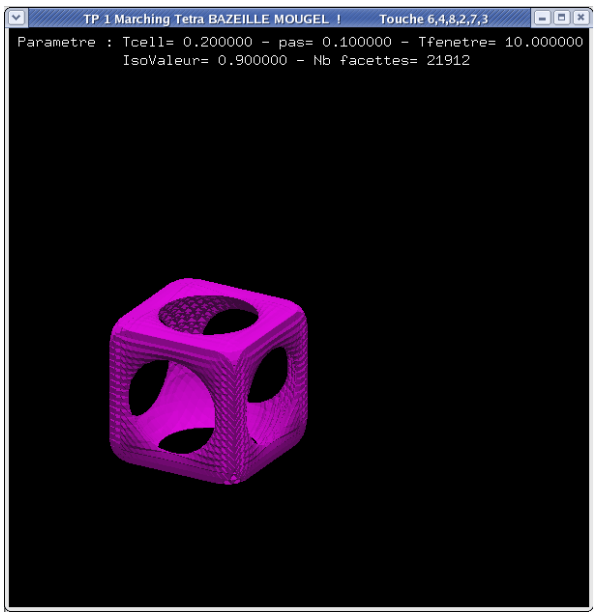
1 + [(1/2)^2 * (x^2 + y^2 + z^2)] ^(-6) - [(1/2.3)^8 * (x^8 + y^8 + z^8)] ^(6)

Cas du cube



Nombre de facette = 6736

Cas du Tetrahedra



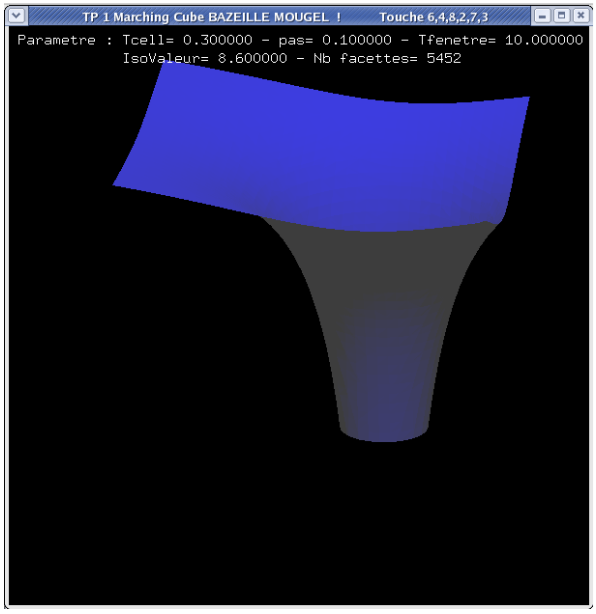
Nombre de facette = 21912

Trou noir :

Formule :

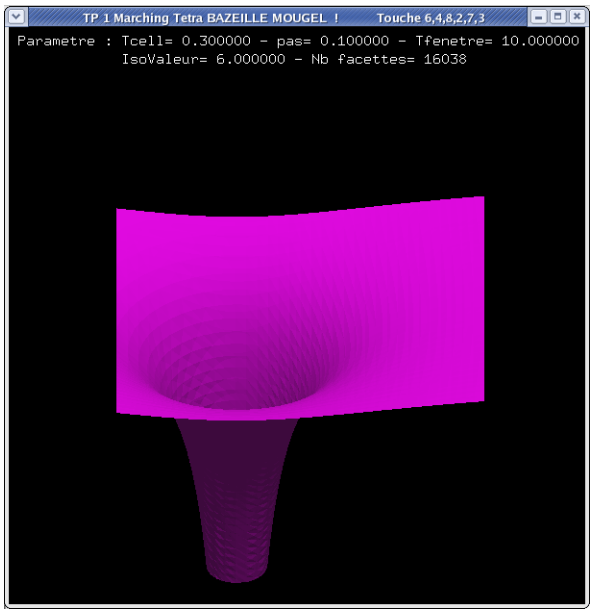
$(y^2 + z^2) * a * x^2 = r^2$

Cas du cube



Nombre de facette = 5452

Cas du Tetrahedra



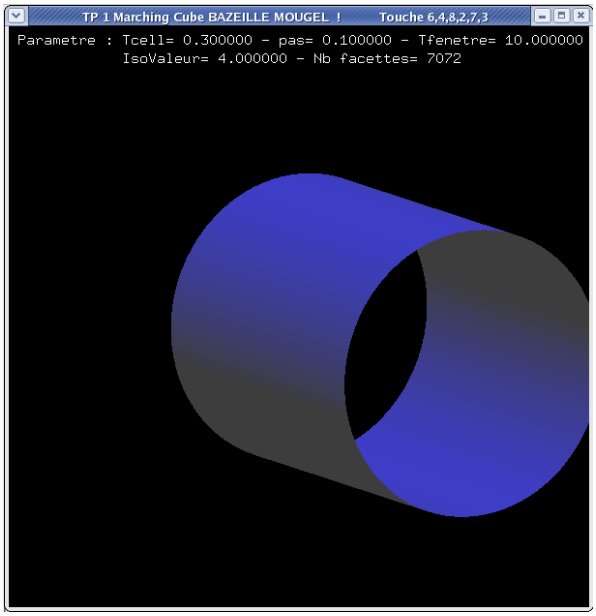
Nombre de facette = 16038

Tube :

Formule :

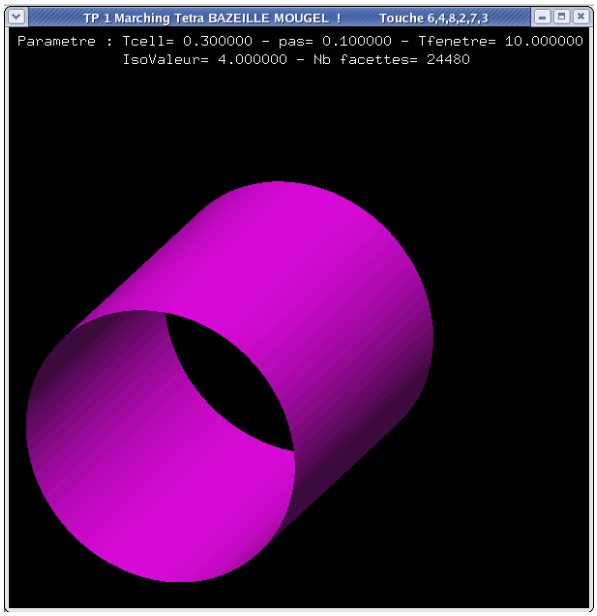
$x^2 + y^2 = r^2$

Cas du cube



Nombre de facette = 7072

Cas du Tetrahedra



Nombre de facette = 24480

Avec un fichier contenant un maillage complet (exemple du crane).

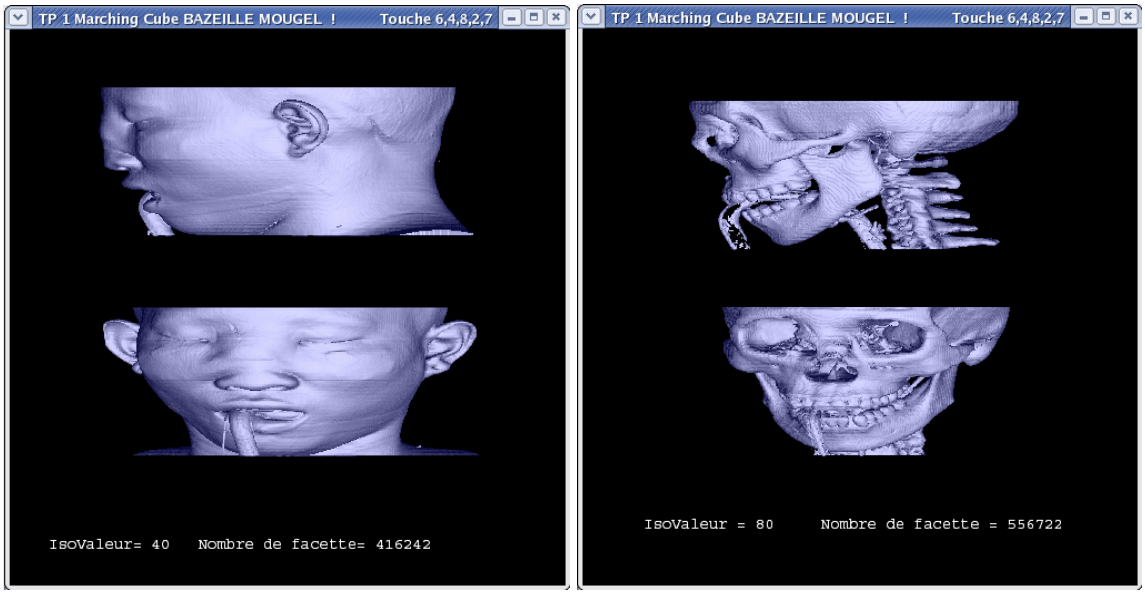
Nous disposons dans ce cas de jeux de données réels d'imagerie médicale, constitué de d'un fichier contenant l'ensemble d'un maillage 3D : l'un représente la tête d'un humain.

Ces fichiers de données sont en plus très volumineux. Et organisé de la maniére suivante :

- Chaque fichier représente une couche du crâne.(coordonnée en z)
- Dans chaque fichier on trouve les coordonnées en x et y
- Chaque donnée étant codé sur 2 octet (16 bits)
- Le 1er octet contient les bits de poids faible et le 2eme les bits de poids fort

Nous avons donc placer ces données dans un tableau de 256 x 256 x 94 qui représente l'espace dans lequel on déplace le cube de visualisation du marching cubes

En effet, après visualisation par l'algorithme du Marching Cube on visionne la tête vue de l'extérieur pour une isovaleur de 40(constitué de plus 400 000 facette) et plus on augmente l'isovaleur plus on pénètre dans le crâne. Le crâne est visible pour une isovaleur de 80 (le crâne est constitué de plus 500 000 facette)



Conclusion.

Comparatif entre Marching Cubes et Marching Tetrahedra.

Nous avons pu constater que chaque algorithme possédait ses points forts et ses points faibles.
Le Marching Cubes est moins gourmand en ressource (temps de calcul ou mémoire) car il limite le nombre de facettes. Cependant son approximation de l'isosurface est moins précise. Il présente aussi le problème des cas ambigus qu'il est nécessaire de résoudre pour ne pas avoir de trous dans l'isosurface.

En revanche, en ce qui concerne le rendu obtenu il est beaucoup plus agréable à l'œil car il semble plus doux que celui obtenu avec le Marching Tetrahedra.
L'algorithme du Marching Tetrahedra génère en effet des creux et des bosses artificielles sur l'isosurface. (Cet effet de bosses est dû à la tesselation du cube élémentaire par des tétraèdres qui implique une approximation de l'isosurface sur les diagonales des faces du cube élémentaire.)

Malgré cet effet visuel « indésirable », la précision de représentation de l'isosurface est bien meilleure dans le cas du Marching Tetrahedra. Concernant l'évolution du nombre de facettes, en fonction de la taille du cube élémentaire, nous avons remarquer que l'algorithme du Marching Tetrahedra produit en moyenne 3 fois plus plus de facettes que le Marching Cubes.

Avantages et inconvénients.

Marching Cubes :

+ Le nombre de triangle est peu important.
+ Le temps de calcul est court.
- L'isosurface est moins proche de la réalité visuellement .
- La table des configurations utilisées est trop vaste.

Marching Tetrahedra :

+ Le cas des ambiguïtés ne se pose pas.
+ L'isosurface est plus précise.
+ Le nombre des configurations utilisées est infime.
- Le temps de calcul est plus long.
- Les ressources machine utilisées sont plus importantes.

Ainsi chacun pourra comprendre que ces algorithmes ont des caractéristiques qui leurs sont propres, bien que par ailleurs ils soient assez semblables. Cependant l'algorithme du Marching Tetrahedra nous semble des plus efficace en toutes circonstances, bien que très fréquemment au vu de la quantité de données à traiter, c'est l'algorithme du marching cubes, voir celui du marching triangle.

Visualisation Scientifique.

La mise en œuvre des algorithmes du Marching cubes et du Marching Tetrahedra nous a permis de prendre conscience de certains problèmes liés au domaine de la visualisation scientifique et aux algorithmes utilisés.

Les deux algorithmes étudiés sont très gourmand en temps de calcul ou en espace mémoire.

Il est important également, dans le choix de l'algorithme, de savoir quels résultats on souhaite obtenir : Pour l'aspect visuel, il serait préférable d'utiliser le Marching Cubes car le rendu est plus doux et lisse (il n'y a pas d'effet de bosses). Pour la précision, le Marching Tetrahedra s'impose car l'approximation de l'isosurface est de meilleure qualité.

Les choix de la taille du volume d'exploration et de la taille du cube élémentaire sont également très importants car ces paramètres déterminent le temps de calculs des coordonnées des facettes de l'isosurface. Il est donc important de connaître la taille minimale que peut prendre le volume d'exploration (volume le plus serré autour de l'objet à représenter) pour pouvoir, en conséquence, ajuster la taille du cube élémentaire et ainsi gagner en précision.

Il serait intéressant maintenant d'essayer des algorithmes tels que le Dividing Cubes, pour pouvoir comparer les performances aux niveaux du temps de calcul , de l'occupation mémoire, de la précision des données représentées, et de la qualité du rendu.

ANNEXE :

Les Codes Sources

Le Marching Square

Le Marching Cubes

Le Marching Tetrahedra