Projet de Reconstruction et géométrie algorithmique

Reconstruction de surface à partir de données sparses

Kévin Polisano

11 mars 2013

1 Introduction

Objectif. Le projet a pour but de reconstruire une surface à partir de points 3D en entrée, par la méthode des « Moving Least Square ».

Le travail à réaliser est composé des étapes suivantes :

1. Calculer des normales non orientées
2. Calculer l'arbre couvrant de poids minimum
3. Réorienter des normales
4. Déterminer une fonction $f(x, y, z)$ en tout point
5. Calculer la surface isovaleur $f(x, y, z) = 0$
6. Calculer les normales (finales) de la surface

Le programme a été réalisé en `C++` via la librairie QGLViewer pour la visualisation en 3D. J'en décris l'installation dans le fichier `README.txt` puisque celle-ci est souvent source de problèmes. On compile via les commandes :

```
qmake-qt4
make
```

Le programme s'exécute alors simplement par :

```
./pointsToSurface data/fichier_de_donnees
```

Toutes les figures du rapport sont ainsi reproduisibles en lançant le programme sur le jeu de données considéré.

2 Implémentation

Dans le header `PointsToSurface.h` j'ai ajouté les attributs et méthodes suivants :

```
double coefLissage;       // coefficient de lissage
unsigned int nbVoisins;   // nombre de voisins consideres pour les
   normales
double margeBoundingBox; // marge autour de la boite englobante
double isoValeur;         // isovaleur de la fonction implicite
double padx;              // pas utilise sur l'axe (ox)
double pady;              // pas utilise sur l'axe (oy)
double padz;              // pas utilise sur l'axe (oz)

//distance max entre 2 points voisins (pour choisir rayon r
   convenablement)
double    _maxDist;
//donne les indices des noeuds voisins au noeud i dans le graphe
   _acm
l_int noeudsVoisins(unsigned int i);
void afficheTriangles(v_Triangle3D & V);
void afficheParam(std::string name);
```

Les attributs sont initialisés lors de la lecture des données, voir annexe.

Dans les `#define` de `PointsToSurface.cpp` j'ai permis à l'utilisateur de modifier lui-même la valeur de ces attributs, qui influent fortement sur la qualité de la surface reconstruite. Par défaut j'ai expérimenté et ai donc affecté les « bonnes » valeurs, mode automatique (AUTO 1).

Pour illustrer chacune des étapes je ferai une capture d'écran du résultat à partir du fichier de données `donnees2.1242.pno.txt`, c'est-à-dire à partir des point 3D de la Figure 1 on souhaite donc reconstruire une surface approximant une sphère.

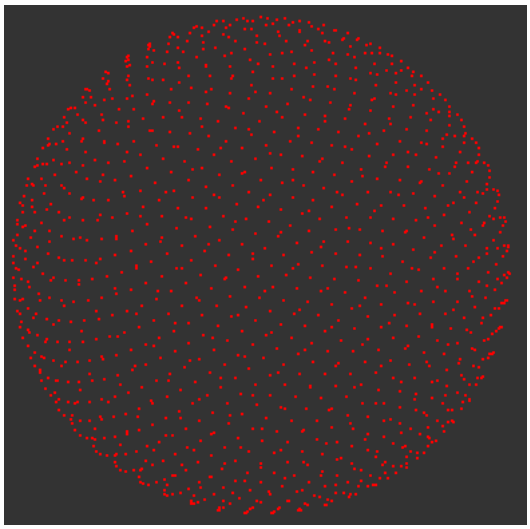


Figure 1 – Données en entrée de 1242 points répartis sur une sphère

2.1 Calculer des normales non orientées

```
void PointsToSurface::computeNonOrientedNormals() {
 unsigned int k = nbVoisins;
 // pour tous les points de l'espace
 for(unsigned int i=0;i<_points.size();++i) {
   // 1. on determine les k-voisins
   v_Point3D pts = kneighborhoodPoints(_points[i],_points,k);
   // 2. on determine le barycentre B des points de pts
   Point3D B(0,0,0);
   for (unsigned int j=0;j<k;++j) {
     B += pts[j];
   }
   B /= pts.size();
   // 3. On forme la liste des points pts-B
   for (unsigned int j=0;j<k;++j) {
     pts[j] -= B;
   }
   // On cree la matrice M 3x3
   double M[3][3];
   for (unsigned int ii=0;ii<3;ii++) {
     for (unsigned int jj=ii;jj<3;jj++) {
   M[ii][jj] = 0;
   for (unsigned int kk=0;kk<k;kk++) {
     double temp[3];
     temp[0] = pts[kk].x;
     temp[1] = pts[kk].y;
     temp[2] = pts[kk].z;
     M[ii][jj] += temp[ii]*temp[jj];
   }
     }
   }
   // on calcule le repere local associe
   Point3D u(0,0,0);
   Point3D v(0,0,0);
   Point3D n(0,0,0);

   calcul_repere_vecteurs_propres(M[0][0],M[0][1],M[0][2],
                  M[1][1],M[1][2],M[2][2],
                  u,v,n);
   // on ajoute la normale a la liste
   _noNormals.push_back(n);
 }
}
```

Le calcul des normales requiert une analyse en composante principale sur le voisinage local de chaque point. Dans le pratique nous prenons en général comme nombre de voisins `nbVoisins=5`. Remarquez qu'on ne remplit que la partie triangulaire supérieure de la matrice M 3×3 puisque celle-ci est symétrique. On constate sur la Figure 2 qu'on obtient bien les normales en chaque point, à ceci près qu'elles ne sont pas toutes orientées dans le même sens.

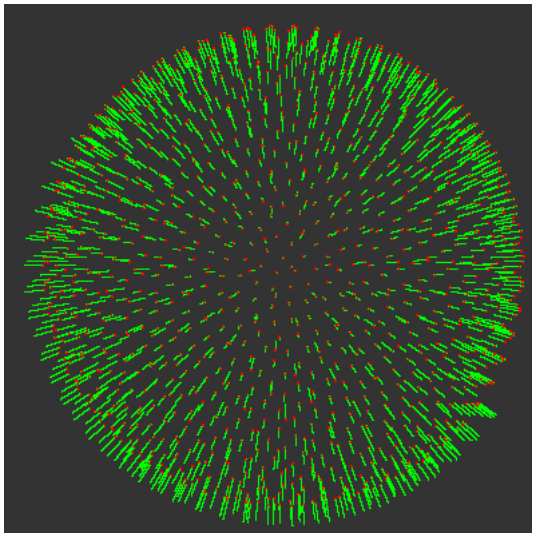


Figure 2 – Normales non orientées en chaque point

2.2 Calculer l'arbre couvrant de poids minimum

```
void PointsToSurface::computeMinimalSpanningTree() {
 unsigned int nbNoeuds = _points.size();
 Graphe * G = new Graphe(nbNoeuds); // graphe de proximite
 double r = 2*_maxDist; // rayon de recherche des voisins
 double d = 0; // distance entre pi et pj
 double v = 0; // valuation de l'arrete (i,j) = 1 - |<ni,nj>|
 for (unsigned int i=0;i<_points.size()-1;i++) {
   for (unsigned int j=i+1;j<_points.size();j++) {
     d = distance_(_points[i],_points[j]);
     if (d <= r) {
       v = 1 - produit_scalaire(_noNormals[i],_noNormals[j]);
       G->ajouter_arc(i,j,v);
     }
   }
 }
 _acm = G->arbre_couvrant_minimal();
}
```

De façon à être sûr que le graphe soit connexe (pas de points isolés), on prend comme rayon de recherche `r=2*_maxDist` où `_maxDist` est la plus grande distance séparant 2 points voisins. Cette variable est remplie lors de la lecture des données dans `readInputFile()`.

Par ailleurs puisque le graphe est non orienté, l'arc (i, j) est identique à (j, i) c'est pourquoi la deuxième boucle commence à $j = i + 1$. Si on représente le graphe on obtient sur la Figure 3 un arbre connexe passant par tous les points :

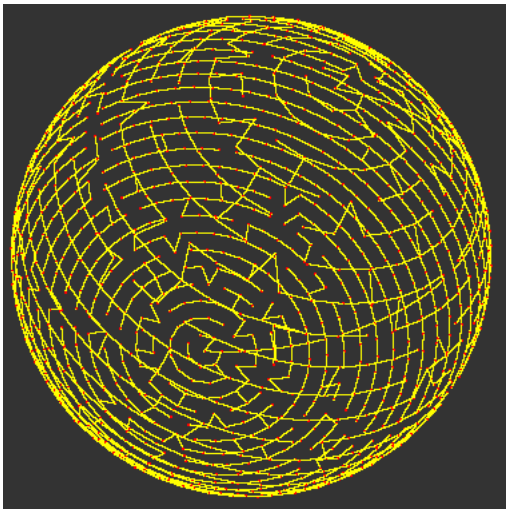


Figure 3 – Arbre couvrant de poids minimum

2.3 Réorienter des normales

```
l_int PointsToSurface::noeudsVoisins(unsigned int i) {
  l_int la = _acm.noeud(i).la; // liste des arretes partant du noeud i
  l_int voisins;
  Arc arcCourant;
  unsigned int n1,n2;
  // on parcourt les arretes
  for (list<unsigned int>::iterator it = la.begin(); it != la.end();++
     it) {
    arcCourant = _acm.arcs()[*it];
    n1 = arcCourant.n1;
    n2 = arcCourant.n2;
    if (i == n1)
      voisins.push_back(n2);
    else
      voisins.push_back(n1);
  }
  return voisins;
}
```

```
void PointsToSurface::computeOrientedNormals() {
  _oNormals =_noNormals;
  unsigned int nbNoeuds = _acm.noeuds().size(); // nombre de noeuds
     dans _acm
  stack<size_t> pile;                           // noeuds a traiter
  vector<bool> fait(nbNoeuds, false);           // noeuds traites

  // on commence par traiter le noeud 0
  pile.push(0);
  fait[0] = true;

  // pour chaque noeud non traite
```

```
  while (!pile.empty())
    {
      // on recupere le premier de la pile
      const size_t i = pile.top();
      pile.pop();

      // on recupere ses voisins
      l_int voisins = noeudsVoisins(i);
      // on recupere la normale au point correspondant au noeud traite
      const Point3D & N = _oNormals[i];

      // on ajoute les voisins a la pile des sommets a traiter.
      for (list<unsigned int>::iterator it = voisins.begin();
       it != voisins.end();++it) {
      const size_t voisinCour = *it;
      // s'il n'est pas deja traite
      if (!fait[voisinCour]) {
          // on l'ajoute
          pile.push(voisinCour);
          fait[voisinCour] = true;
          // si la normale associee est inversee, on la retourne.
          Point3D &Nv = _oNormals[voisinCour];
          if (produit_scalaire(N, Nv) < 0)
        Nv *= -1;
        }
    }
    }
}
```

Il était possible d'écrire cette méthode par des appels récursifs. Ceci dit lorsque l'on travaille avec de gros jeux de données, il est dangereux d'effectuer des traitements récursifs, dans le mesure où la taille de la pile dans laquelle sont stockées les appels de fonctions est bien plus petite que celle du tas. C'est pourquoi je préconise un traitement itératif dans le tas, au sein duquel je peux me permettre d'utiliser un conteneur de type pile (« stack ») qui va imiter ce qu'aurait fait automatiquement les appels de fonctions récursives. En effet la pile contient les indices des noeuds à traiter (on commence par le noeud 0, le père) et le vecteur de booléens `fait` de taille correspondant au nombre de noeuds est initialisé à faux. Dès qu'un noeud est traité on passe son indice dans `fait` à vrai. Le reste des commentaires est suffisamment parlant pour expliquer le fonctionnement du parcours en largeur de l'arbre. Sur la Figure 4 on illustre le fait que les normales ont bien été retournées (en bleues).

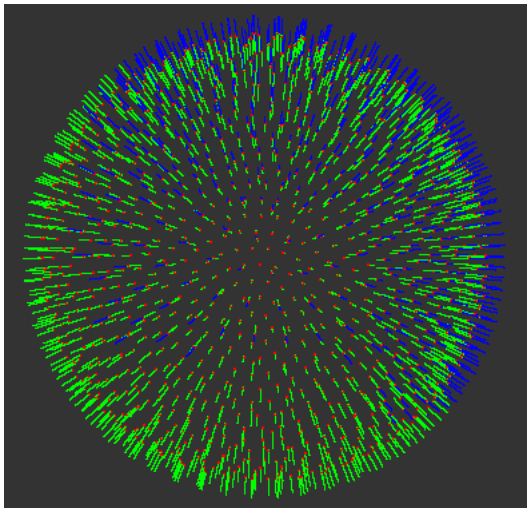


Figure 4 – Normales orientées

2.4 Déterminer une fonction $f(x, y, z)$ en tout point

```
double PointsToSurface::computeImplicitFunc(double x,double y,double
    z) {
  double resultat = 0;
  double poidsTotal = 0;
  double sigma = coefLissage;
  double xi,yi,zi,nx,ny,nz,wi;
  // on parcourt tous les points et on forme P-Pi = (xi,yi,zi)
  // et les normales aux points parcourus sont notees ni = (nx,ny,nz)
  for (unsigned int i=0;i<_points.size();i++) {
    xi = x - _points[i].x;
    yi = y - _points[i].y;
    zi = z - _points[i].z;
    nx = _oNormals[i].x;
    ny = _oNormals[i].y;
    nz = _oNormals[i].z;
    wi = exp(-(xi*xi+yi*yi+zi*zi)/(sigma*sigma));
    poidsTotal += wi;
    resultat += (nx*xi+ny*yi+nz*zi)*wi;
  }
  return resultat/poidsTotal;
}
```

RAS du point de vue de l'implémentation des MLS, c'est une application directe de la formule suivante calculant la distance moyenne aux plans :

$$f(P) = \frac{\sum_i n_i^T (P - P_i)\omega(\|P - P_i\|)}{\sum_i \omega(\|P - P_i\|)}$$

avec $\omega(x) = e^{-(x/\sigma)^2}$ où σ est le coefficient de lissage.

La visualisation de f sur la boundingBox donne bien le résultat escompté (Figure 5) :

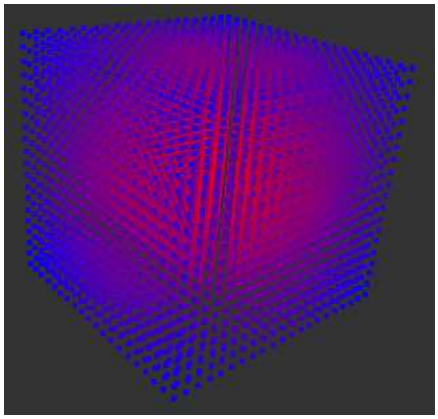


Figure 5 – Représentation de la fonction implicite

2.5 Calculer la surface isovaleur $f(x, y, z) = 0$

```
void PointsToSurface::computeMesh() {
 // creation de la grille 3D
 Point3D pmin = _boundingBox[0];
 Point3D pmax = _boundingBox[1];
 double m = margeBoundingBox; // marge
 double x_min = pmin.x-m, y_min = pmin.y-m, z_min = pmin.z-m;
 double x_max = pmax.x+m, y_max = pmax.y+m, z_max = pmax.z+m;
 unsigned int nx = padx;
 unsigned int ny = pady;
 unsigned int nz = padz;
 Grille3D grille = Grille3D(x_min,y_min,z_min,
                 x_max,y_max,z_max,
                 nx,ny,nz);

 // creation du tableau vf de la fonction aux sommets de la grille
 //  vf[i+DIM_X*(j+DIM_Y*k)] = f(G.x(i),G.y(j),G.z(k))
 unsigned int DIM_X = nx+1, DIM_Y = ny+1, DIM_Z = nz+1;
 double * vf = new double[DIM_X*DIM_Y*DIM_Z];
 bool * ind = new bool[DIM_X*DIM_Y*DIM_Z];
 for (unsigned int i=0; i<nx; i++) {
     for (unsigned int j=0; j<ny; j++) {
     for (unsigned int k=0; k<nz; k++) {
       vf[i+DIM_X*(j+DIM_Y*k)] = computeImplicitFunc(grille.x(i),
                               grille.y(j),
                               grille.z(k));
       ind[i+DIM_X*(j+DIM_Y*k)] = true;
       // pour exclure les bords de la bounding box
       if ((i == nx-1) || (j == ny-1) || (k == nz-1)) {
         ind[i+DIM_X*(j+DIM_Y*k)] = false;
       }

     }
     }
 }
```

```
  // creation des tetraedres via isovaleur de la fonction implicite 3D
  double iso_val = isoValeur;
  SurfaceIsovaleurGrille S = SurfaceIsovaleurGrille();
  S.surface_isovaleur(_surfacep,grille,vf,iso_val,ind);
}
```

On créé une grille 3D légèrement plus grande que les dimensions de la boundingBox, puisque la MLS est rappelons le une approximation. C'est le rôle de la variable `marge`. Ayant eu également quelques soucis avec les bords de la grille (puisque aux frontières la fonction implicite peut valoir numériquement 0, du fait des poids $\omega(\|P - P_i\|)$, et donc être apparenté à un élément de surface), j'ai utilisé le vecteur de booléens `ind` pour exclure les indices correspondants au bord, le problème fut alors réglé. Le résultat du marching cube générant la surface $f(x, y, z) = 0$ est donné Figure 6 :

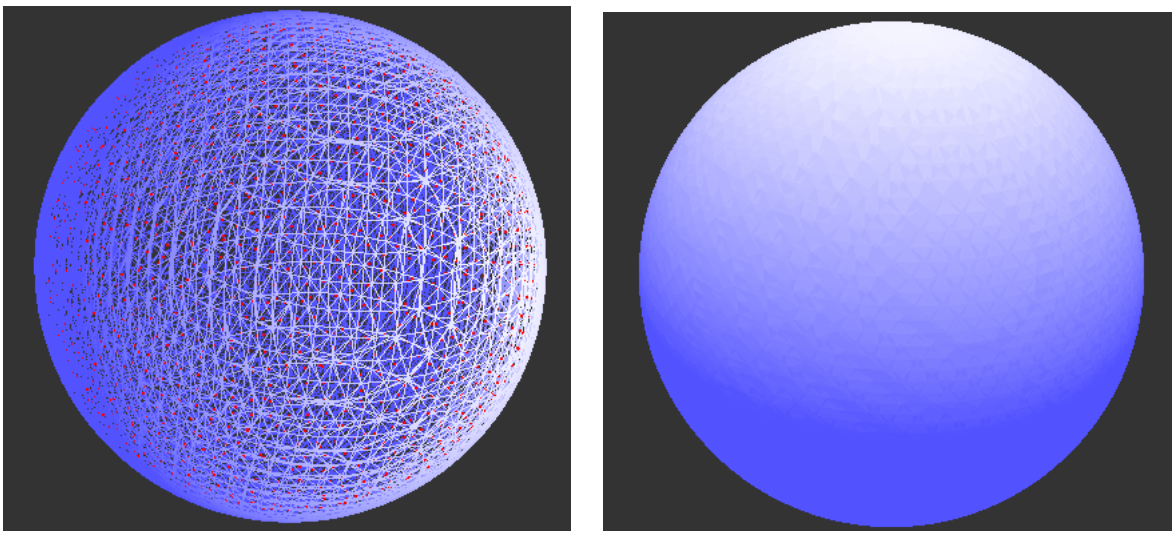


Figure 6 – Surface obtenue par marching cube sur la fonction implicite f

2.6 Calculer les normales (finales) de la surface

```
void PointsToSurface::computeNormalsFromImplicitFunc() {
 double e = 0.01;
 double xi,yi,zi,nx,ny,nz;
 // on parcourt tous les points de la SURFACE (donc les triangles)
 for (unsigned int i=0;i<_surfacep.size();i++) {
   // pour chaque triangle on recupere les trois points
   v_Point3D face;
   face.push_back(_surfacep[i].S0);
   face.push_back(_surfacep[i].S1);
   face.push_back(_surfacep[i].S2);
   // on reforme un triangle dont les points sont les nouvelles
      normales
   Triangle3D normales;
```

```
    for (unsigned int j=0;j<face.size();j++) {
      xi = face[j].x;
      yi = face[j].y;
      zi = face[j].z;
      // calcul des nouvelles normales via le gradient de la fct
         implicite
      nx = computeImplicitFunc(xi-e,yi,zi)-computeImplicitFunc(xi+e,yi
         ,zi);
      ny = computeImplicitFunc(xi,yi-e,zi)-computeImplicitFunc(xi,yi+e
         ,zi);
      nz = computeImplicitFunc(xi,yi,zi-e)-computeImplicitFunc(xi,yi,
         zi+e);
      Point3D Pn = normalise(Point3D(nx,ny,nz));
      if (j == 0) {
    normales.S0 = Pn;
      } else if (j == 1) {
    normales.S1 = Pn;
      } else {
    normales.S2 = Pn;
      }
    }
    // on ajoute le triangle a _surfacen
    _surfacen.push_back(normales);
  }
}
```

On voyait sur la Figure 6 la présence des triangles formant la surface. Cela provient du fait qu'on représente la surface munie des normales aux points 3D, qui ne sont pas confondues avec les réelles normales de la surface reconstruite dans la mesure où encore une fois celle-ci n'est qu'une approximation. Il faut donc recalculer les normales à la surface obtenue $f(x, y, z) = 0$, ce que nous faisons en calculant un gradient approché sur un déplacement infinitésimal ($e = 0.01$), ce qui est la partie la plus coûteuse puisque cela nécessite 6 appels à la fonction f pour chaque normale, autrement dit 6 Moving Least Square. Les nouvelles normales obtenues, le résultat apparait alors sur la Figure 7 beaucoup plus lisse :

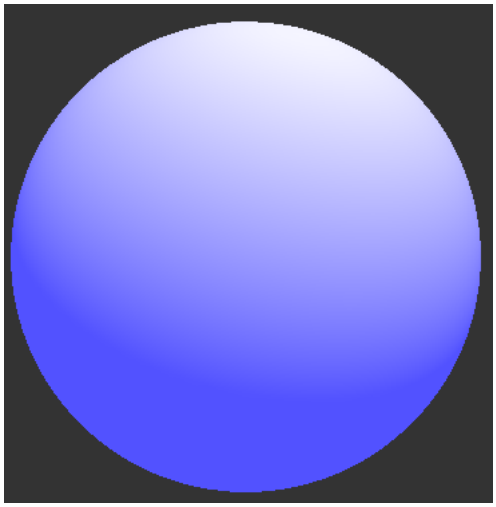


Figure 7 – Représentation finale de la surface munie des bonnes normales

3 Résultats

Nous donnons enfin ci-après sur la Figure 14 et Figure 15 le résultat de la reconstruction pour les autres jeux de données.

Les paramètres utilisés sont les suivants :

	nbVoisins	margeBoundingBox	coefLissage	Figure
"data/donnees1.160.pno.txt"	5	0.5	0.3	Figure 7
"data/donnees2.1242.pno.txt"	5	0.5	0.3	Figure 7
"data/donnees3.1682.pno.txt"	5	0.5	0.3	Figure 14(a)
"data/selle.441.pts.txt"	5	0.5	0.3	Figure 14(b)
"data/club71.16864.pts.txt"	12	0.15	0.1	Figure 14(c)
"data/cactus.3337.pts.txt"	50	0.2	0.05	Figure 15(a)
"data/cat10.10000.pts.txt"	250	0.2	0.03	Figure 15(c)
"data/mannequin.12772.pts.txt"	250	0.2	0.03	Figure 15(b)

Figure 8 – Paramètres optimaux spécifiques à chaque type de données

remarque 1 : pour les deux derniers fichiers j'ai du modifier la ligne `camera()->setSceneCenter` en `camera()->setSceneCenter(qglviewer::Vec(0,0,0))` dans `viewer.init()` de façon à pouvoir visualiser les points. Je ne saurais expliquer cet étrange comportement...

remarque 2 : une utilisation de `valgrind` met en évidence beaucoup de fuites mémoires sur le squelette d'origine, qui semble provenir de la librairie QGLViewer elle-même.

Discussion de l'influence des paramètres

• *Le nombre de voisins*

Premièrement ce que l'on remarque est que pour les 4 premières lignes du tableau un voisinage de 5 voisins seulement est utilisé, en effet que ce soit les sphères, le « vase » ou la selle toutes ces figures possèdent localement une forte courbure donc quelques voisins suffisent à déterminer le plan moyen puis la normale. Tandis que dans les cas plus pathologiques, typiquement comme c'est le cas pour le cactus, les points sont parfaitement alignés sur la longueur et relativement espacés sur la largeur, ainsi les 5 voisins (même 12 !) sont alignés donc ne permettent pas déterminer un plan correct ! C'est pourquoi par précaution 50 voisins ont été considérés.

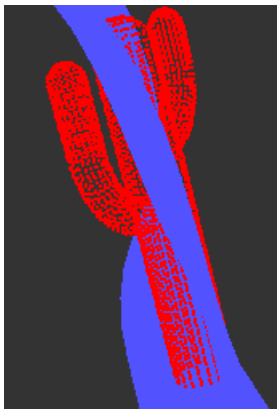


Figure 9 – Le cactus avec prise en compte de 12 voisins seulement

• *Le paramètre de lissage*

Un paramètre ayant encore davantage d'impact sur la qualité de la surface reconstruire est le paramètre σ des MLS. Ce n'est pas flagrant sur les 4 premiers fichiers de données, en revanche ça le devient pour les suivants. En effet un σ légèrement trop grand donne lieu à des résultats grossiers comme ci-dessous :

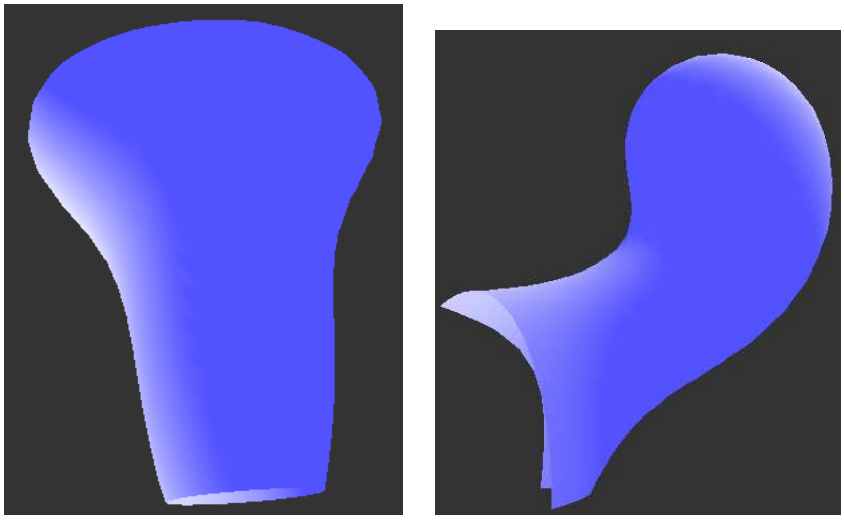


Figure 10 – Le cactus et le club reconstruit avec $\sigma = 0.3$ trop grand

• *La marge*

La marge ajoutée autour de la BoundingBox est également à ajuster pour éviter les trous ou au contraire les « interpolations » indésirables (à droite sur la Figure 13 un « pic » est créé au sommet du vase).

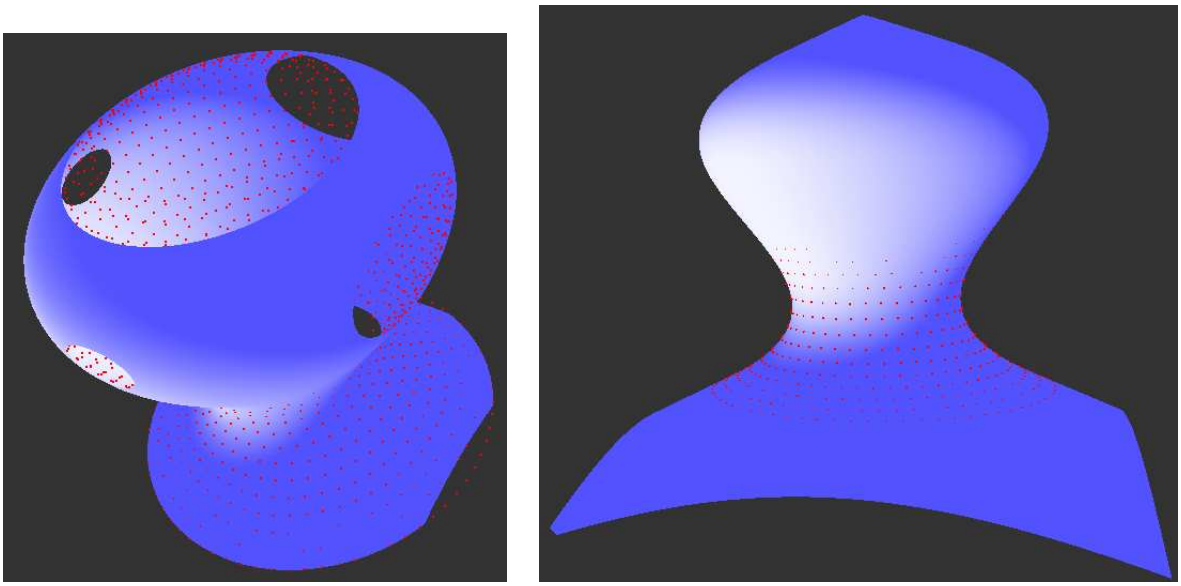


Figure 11 – Une marge nulle (à gauche) et une marge de 1 (à droite)

• *Les pas d'échantillonage*

Les trois directions sont discrétisées de manière équiréparties avec le même pas, mais on pourrait facilement adapter le pas en tenant compte des longueurs réelles des 3 dimensions de la BoundingBox. Le paramètre d'échantillonage est important car plus celui-ci est petit et plus les détails seront mis en valeurs, la surface apparaîtra également plus lisse. Le revers de la médaille est que la complexité en terme de calculs y est très sensible. Ainsi les 4 premiers fichiers sont traités avec un pas de 50 tandis que les suivants sont traités avec un pas de 20 car ils possèdent un plus grand nombre de points. Sauf le chat et le mannequin qui nécessite un pas de 70.

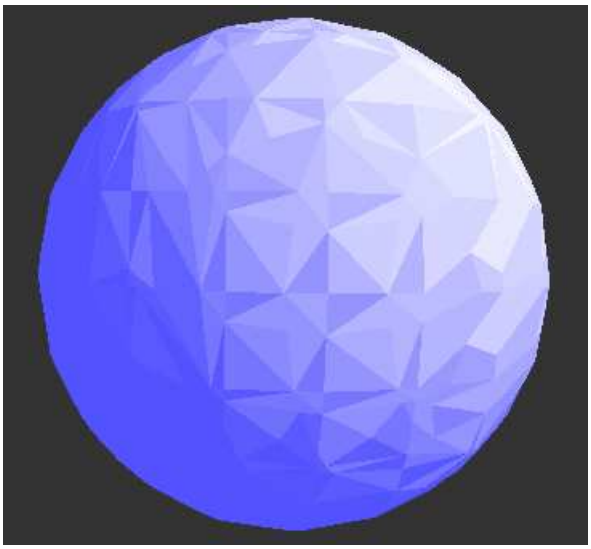


Figure 12 – Echantillonage trop faible avec un pas de 10

remarque : comme signalé dans la partie implémentation, nous avons eu recours à un vecteur de booléens `ind` de manière à exclure le traitement des bords, dans le cas contraire les artefacts dont on parlait peuvent survenir comme sur la figure ci-dessous.

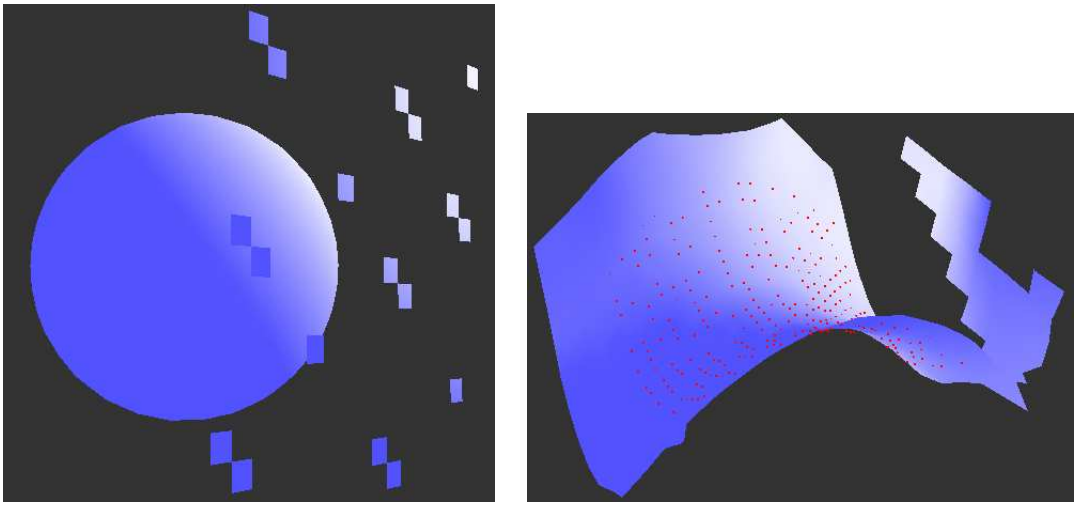
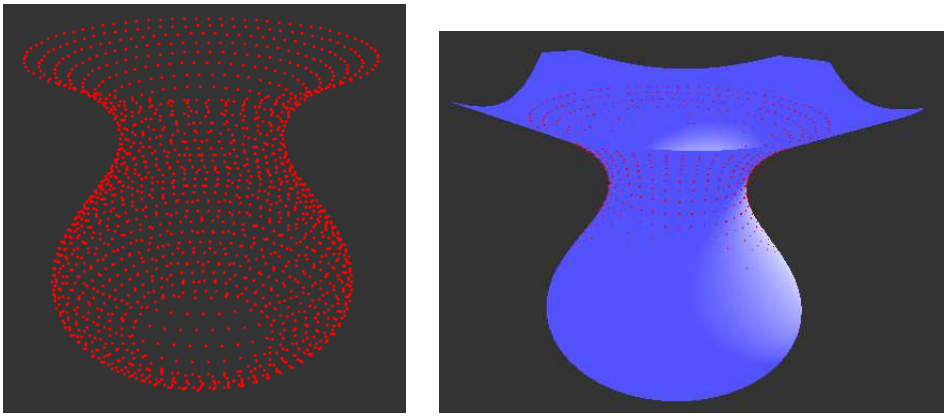
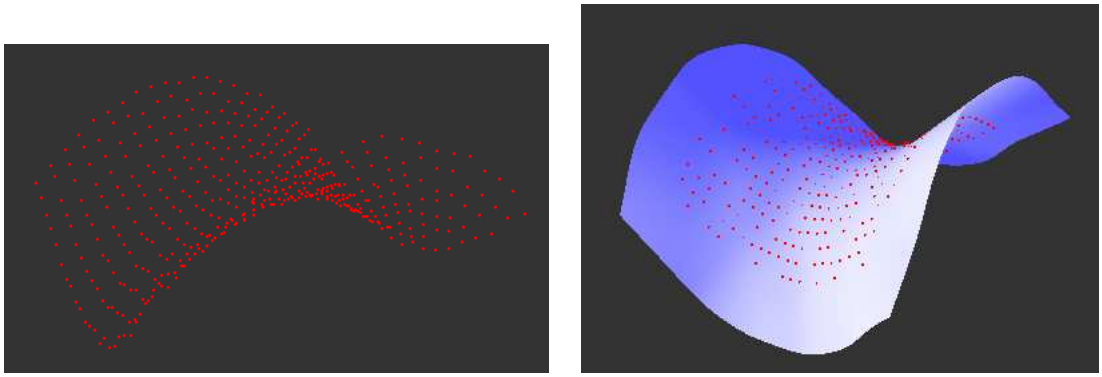


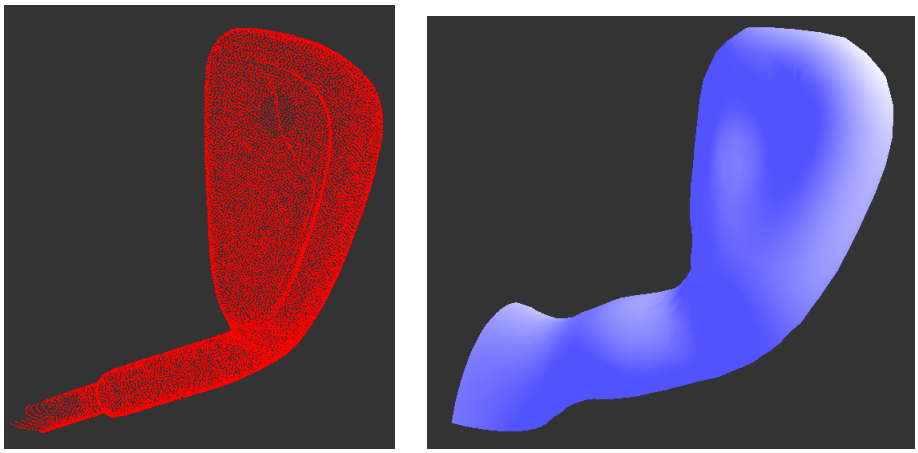
Figure 13 – Artefacts provenant du traitement des bords



(a) "data/donnees3.1682.pno.txt"

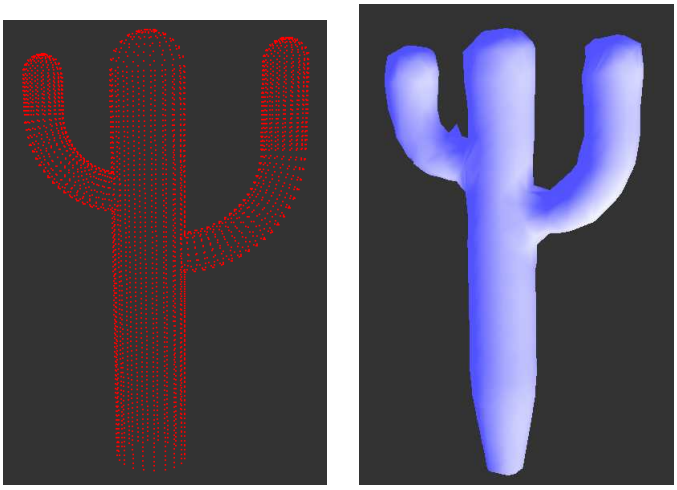


(b) "data/selle.441.pts.txt"

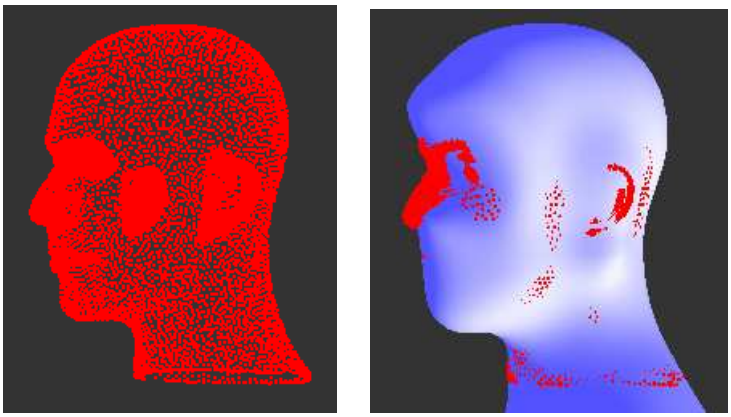


(c) "data/club71.16864.pts.txt"

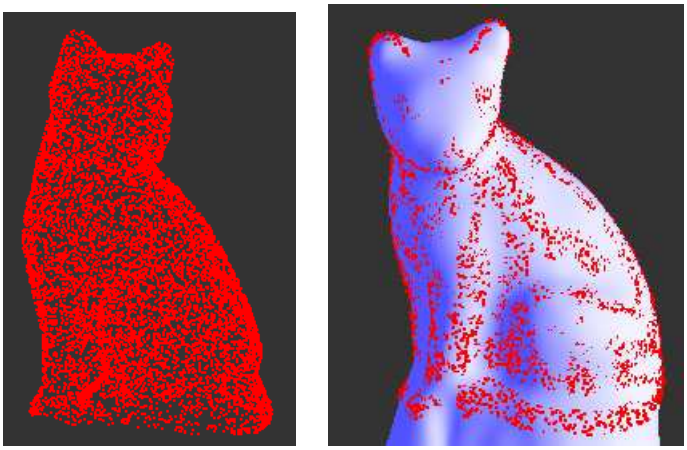
Figure 14 – Reconstruction des surfaces



(a) "data/cactus.3337.pts.txt"



(b) "data/mannequin.12772.pts.txt"



(c) "data/cat10.10000.pts.txt "

Figure 15 – Reconstruction des surfaces

4 Annexe

```
bool PointsToSurface::readInputFile(const QString &filename) {

 _points.clear();

 unsigned int nS;
 double r, xx, yy, zz;

 std::string str = filename.toStdString();
 const char * name = filename.toStdString().c_str();
 std::ifstream file(name);
 cout << "Lecture du fichier " << str << "..." << endl;
 file >> nS;
 file >> r;
 _points.resize(nS);
 for (unsigned int i = 0; i < nS; i++)
   {
     file >> xx;
     file >> yy;
     file >> zz;
     if (file.good() || file.eof()) {
   _points[i].x = xx;
   _points[i].y = yy;
   _points[i].z = zz;
     }
   }

 boite_englobante(_points,_boundingBox[0],_boundingBox[1]);

 // mean of the min distance between 2 points
 _meanDist = 0.0;
 _maxDist = DBL_MIN;
 double temp = 0;
 for(unsigned int i=0;i<_points.size();++i) {
   v_Point3D pts = kneighborhoodPoints(_points[i],_points,2);
   temp = distance_(pts[0],pts[1]);
   _meanDist = _meanDist+temp;
   if (temp > _maxDist)
     _maxDist = temp;
 }
 _meanDist = _meanDist/(double)_points.size();

 // fermeture du fichier
 file.close();
 cout << "Chargement des donnees termine..." << endl;

 // Determination des parametres adequats en fonction des donnees
    entrees
 std::string data1 = "data/donnees1.160.pno.txt";
 std::string data2 = "data/donnees2.1242.pno.txt";
 std::string data3 = "data/donnees3.1682.pno.txt";
```

```
  std::string cactus = "data/cactus.3337.pts.txt";
  std::string club = "data/club71.16864.pts.txt";
  std::string cat = "data/cat10.10000.pts.txt";
  std::string mannequin = "data/mannequin.12772.pts.txt";
  std::string selle = "data/selle.441.pts.txt";

  // Mode automatique, parametres optimaux relatifs a chaque fichier
  if (AUTO) {
    // par defaut
    nbVoisins = 5;
    isoValeur = 0;
    coefLissage = 0.4;
    margeBoundingBox = 0.2;
    padx = 20;
    pady = 20;
    padz = 20;
    // personnalises
    if ((str == data1) || (str == data2) || (str == data3) || (str ==
       selle))
      {
      coefLissage = 0.3;
      margeBoundingBox = 0.5;
      padx = 50;
      pady = 50;
      padz = 50;
    } else if (str == club) {
      nbVoisins = 12;
      coefLissage = 0.1;
      margeBoundingBox = 0.15;
    } else if (str == cactus) {
      nbVoisins = 50;
      coefLissage = 0.05;
      margeBoundingBox = 0.2;
    } else if ((str == mannequin) || (str == cat)) {
      nbVoisins = 12;
      coefLissage = 0.4;
      margeBoundingBox = 0.2;
    }
  } else { // Mode manuel, les parametres sont initialises dans les #
     define
     coefLissage = LISSAGE;
     margeBoundingBox = MARGE;
     nbVoisins = NB_VOISINS;
     isoValeur = ISO_VAL;
     padx = PAS;
     pady = PAS;
     padz = PAS;
  }

  afficheParam(str);

  return true;
}
```