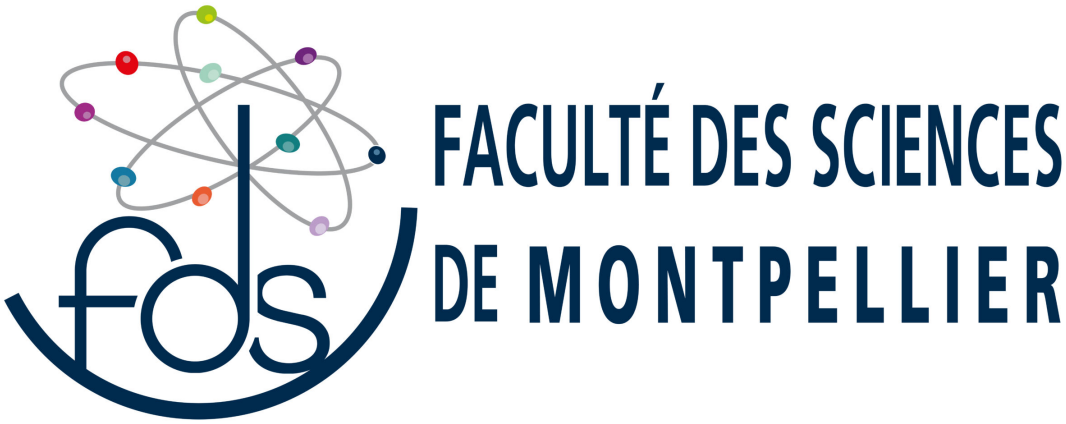
Université de Montpellier

M1 - IMAGINE - Modélisation et géométrie discrète
Compte Rendu TP5 - Simplification de maillages

Etudiant :
Guillaume Bataille

Encadrant :
Noura Faraj
Marc HARTLEY

Année 2022-2023



Sommaire

1. Contexte et Objectif

Contexte Lors du cours, nous avons abordé la simplification des maillages. Plusieurs méthodes sont à notre disposition pour le faire et les deux plus importantes sont la "Partition" et la "Décimation".

Objectif L'idée générale est de créer une classe maillage que nous allons enrichir avec des opérations de traitement de géométrie, ces processus seront activés par des touches du clavier. Tous les traitements effectués seront appliqués au maillage dans son état courant. F1 peut être utiliser pour voir la structure du maillage, n'hésitez pas à ajouter des options plus avancées de visualisation e.g., dessiner la normal, courbure etc... Pour mieux comprendre comment se comporte l'opérateur.

2. A - Boîte englobante

Pour pouvoir cacluler la boite englobante de notre mesh, on décide de parcourir tout les sommets de ce dernier et d'y dégager BBmin (le coin inférieur) et BBmax (le coin supérieur). Avec, et juste avec ses deux points, on est capable de créer notre boîte englobante de manière visuelle ou abstraite (juste la structure).
Voici les sommets calculées correspondant à la boîte englobante.

```
std::vector<Vec3> get_box(std::vector<Vec3> const & vertices, double e){
    std::vector<Vec3> res;
    res.clear();
    Vec3 bbmin = BBmin(vertices,e);
    Vec3 bbmax = BBmax(vertices,e);

    double xmin,ymin,zmin;
    double xmax,ymax,zmax;

    xmin = bbmin[0]; ymin = bbmin[1]; zmin = bbmin[2];
    xmax = bbmax[0]; ymax = bbmax[1]; zmax = bbmax[2];

    res.push_back(bbmin);

    res.push_back(Vec3(xmax,ymin,zmin));

    res.push_back(Vec3(xmin,ymin,zmax));

    res.push_back(Vec3(xmax,ymin,zmax));

    res.push_back(Vec3(xmin, ymax, zmin));

    res.push_back(Vec3(xmin, ymax, zmax));

    res.push_back(Vec3(xmax,ymax,zmin));

    res.push_back(bbmax);

    return res;

}

```

3. B - Grille de voxels

Maintenant qu'on a une boite englobante, on essaie de la partitionner afin d'avoir une grille 3D de voxels. Pour se faire, on découpe chaque côté en 'resolution' parties. Ce qui en 1D nous donne une graduation, en 2D une grille, et en 3D notre grille de voxel.

3.1 Visuel pour avoir l'intuition

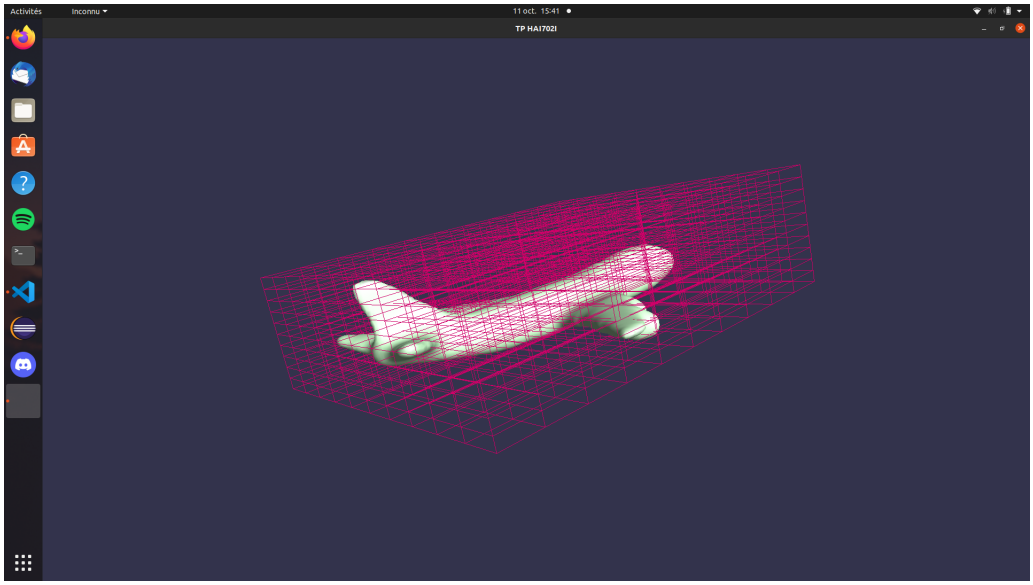


FIGURE 3.1 – Visuel de la grille de voxel 16x1616, joli mais inutile dans l'implémentation

3.2 Structure de la grille

Pour ma grille, je décide de faire un std : :vector de Représentant. Une instance représentant défini les propriétés de chaque case de ma grille de voxel : occurence des points se trouvant dans cette zone , moyenne des position des points de la zone , moyenne des normales des points dans la zone.

```
1  struct Representant {
2      Vec3 position; //Position du representant
3      Vec3 normal; //Normale du representant
4      unsigned int count = 0; // Compteur de combien il représente de vertices.
5  };
```

Rappel : Afin d'accéder à un élément (identifié par i,j et k) de ma grille qui est en 1D on peut y accéder via : i + (j*nx) +(k*nx*ny) avec (i,j,k) les coordonnées visés, nx le nombre de case sur une largeur X et ny le nombre de case sur une longueur y.

4. C - Utilisation de la grille de représentant

4.1 Création et remplissage de la grille

-initialisation de la grille à la taille resolution*resolution*resolution
-remplissage de la grille en parcourant tout les vertices

```
1
2  void fill_Grille(std::vector<Representant> & Gr,std::vector<Vec3> const & vertices, std::vector<Vec3> const & no
3
4  int idG; // L'id d'une case du vecteur 1D correspondant a un voxel de la grille 3D
5  Vec3 ijk;// Les coordonnées grille ijk correspondant aux coordonnées monde arrondie
6  Vec3 bbmin = BBmin(vertices,e);
7  Vec3 bbmax = BBmax(vertices,e);
8
9  Gr.clear();
10 init_Grille(Gr,resolution); // Initialisation de la grille de représentant
11 for (size_t i = 0; i < vertices.size(); i++ ){ // Pour tout les sommets i de mon mesh
12     ijk = get_ijk_from_xyz(vertices[i],bbmin,bbmax,resolution);
13     idG = get_id_grille(ijk,resolution);
14     Gr[idG].position += vertices[i];
15     Gr[idG].normal += normals[i] ;
16     Gr[idG].count ++;
17
18 }
19 };
20
```

5. D - Ré-indexage et élimination

Une fois qu'on a tout gérer du côté vertices et normales, on peut passer au réindexage des triangles afin de les simplifier comme convenu. Les triangles, confondu seront bien évidemment pas pris en compte ou "éliminé". On parcours donc tout nos triangles et via get_ijk_from_xyz on recupère les ijk du voxel contenant chaque sommet du triangle courant. Si deux de ses sommets s'applatissent sur le représentant, alors on ne le prends pas en compte. Sinon on le récupère pour notre maillage simplifié.

```
void re_index_triangles(const std::vector<Vec3> & vertices, const std::vector< Triangle > & triangles, std::vect
std::cout<<"in index triangle" << triangles.size()<< std::endl;
unsigned int v0,v1,v2,r0,r1,r2; // Les représentants et vertices corrsepondant au sommets des triangles
int idG; // L'id d'une case du vecteur 1D correspondant a un voxel de la grille 3D
Vec3 ijk;// Les coordonnées grille ijk correspondant aux coordonnées monde arrondie
Vec3 bbmin = BBmin(vertices,e);
Vec3 bbmax = BBmax(vertices,e);
new_triangles.clear();

for (size_t i = 0; i < triangles.size();i++){ // Pour tout les triangles, on recup les 3 id_vertices et on recup
    v0 = triangles[i][0];
    v1 = triangles[i][1];
    v2 = triangles[i][2];

    r0 = get_id_grille(get_ijk_from_xyz(vertices[v0],bbmin,bbmax,resolution),resolution);
    r1 = get_id_grille(get_ijk_from_xyz(vertices[v1],bbmin,bbmax,resolution),resolution);
    r2 = get_id_grille(get_ijk_from_xyz(vertices[v2],bbmin,bbmax,resolution),resolution);


    if(r0 != r1 && r1 != r2 && r2 != r0){
        Triangle t (r0,r1,r2);
        new_triangles.push_back(Triangle(r0,r1,r2));

    }
}
};

```

Voici pour mieux comprendre mes fonctions ge_tid_ijk et get_id_grid :

```

// Fonction qui retourne les coordonnées grille ijk a partir de coordonnées xyz
Vec3 get_ijk_from_xyz(Vec3 pos_world, Vec3 bbmin, Vec3 bbmax, int resol){
    Vec3 res;
    double x,y,z;
    int i,j,k;
    double xl, yl, zl, dx, dy, dz;
```

```
8
9       // Position xyz
10      x = pos_world[0];
11      y = pos_world[1];
12      z = pos_world[2];
13
14      //Longueur d'un des coté x y z de la boite englobante
15      xl = bbmax[0] - bbmin[0];
16      yl = bbmax[1] - bbmin[1];
17      zl = bbmax[2] - bbmin[2];
18
19      // Une portion sur chaque longueur
20      dx = xl/resol;
21      dy = yl/resol;
22      dz = zl/resol;
23
24      // Formule pour récupéré la position i j et k de la ou le x y z tombe
25      i = (x-bbmin[0]) / dx;
26      j = (y-bbmin[1]) / dy;
27      k = (z-bbmin[2]) / dz;
28      res = Vec3(i,j,k);
29      return res;
30  }
31
32
33  // Fonction qui a partir de coordonées grille ijk retourne l'indice de la strucutre grille choisie.
34  int get_id_grille(Vec3 ijk, int resol){
35      int i,j,k, nx, ny;
36      i = ijk[0];
37      j = ijk[1];
38      k = ijk[2];
39      nx = resol;
40      ny = resol;
41      return i + (j*nx) +(k*nx*ny);
42  }
43
44
```

---

6. E - Normalisation des représentation

J'ai décidé de normaliser dans la fonction simplify que vous allez voir ci dessous. Afin de normaliser les positions, je divise les positions par le nombre d'occurence reçu de chaque représentant et pour la normale je normalise.

```
1
2  void simplify (
3      const std::vector<Vec3> & vertices,
4      const std::vector< Triangle > & triangles,
5      const std::vector< Vec3 > & normals,
6      std::vector <Representant> &Gr,
7      double e,
8      unsigned int resolution,
9      std::vector< Triangle > & new_triangles
10     ){
11 std::cout<< "Simplify called "<<std::endl;
12     fill_Grille(Gr,vertices,normals,e,resolution);
13     re_index_triangles(vertices,triangles,Gr,e,resolution,new_triangles);
14     for (size_t i =0 ; i < Gr.size(); i++){
15
16         // NORMALISATION //
17         if (Gr[i].count>1){ // Si il y'a au moins deux valeurs dans la grille
18             Gr[i].position = Gr[i].position/Gr[i].count; // Normalisation des position par division de count
19             //std::cout<<" count : "<< i << " -> " << Gr[i].count <<" "<<Gr.size()<<std::endl;
20             Gr[i].normal.normalize(); // Normalisation de la normale via la methode normalize
21
22         }
23     }
24 std::cout<< "Simplify end "<<std::endl;
25
26 };
27
```

7. Visuel de simplification

Comme j'ai du mal à modifier le mesh courant afin de le simplifier, j'affiche ici les vertices obtenus via la grille de représentant à différentes résolution :

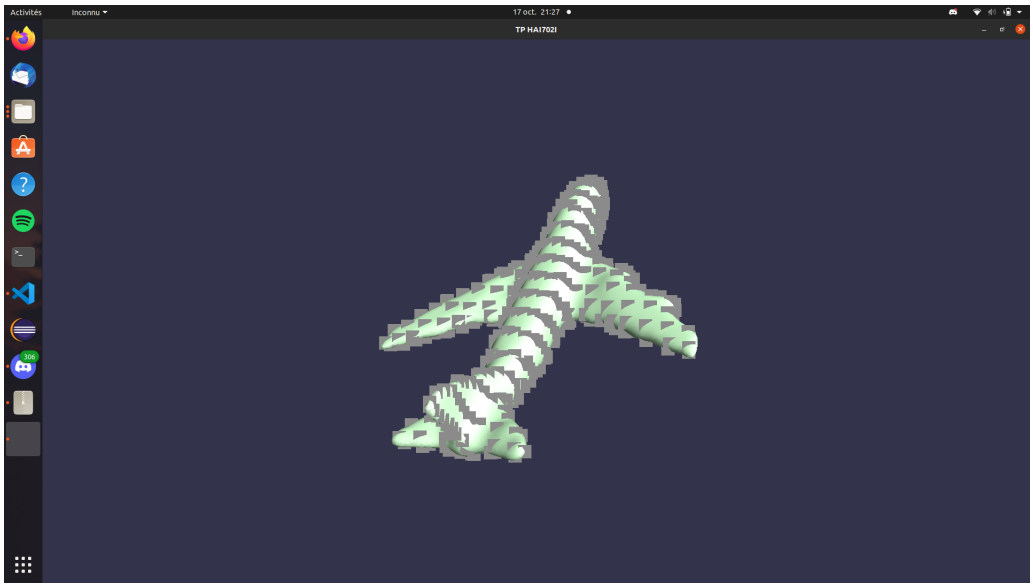


FIGURE 7.1 – Simplification des vertices avec une grille 16*16*16

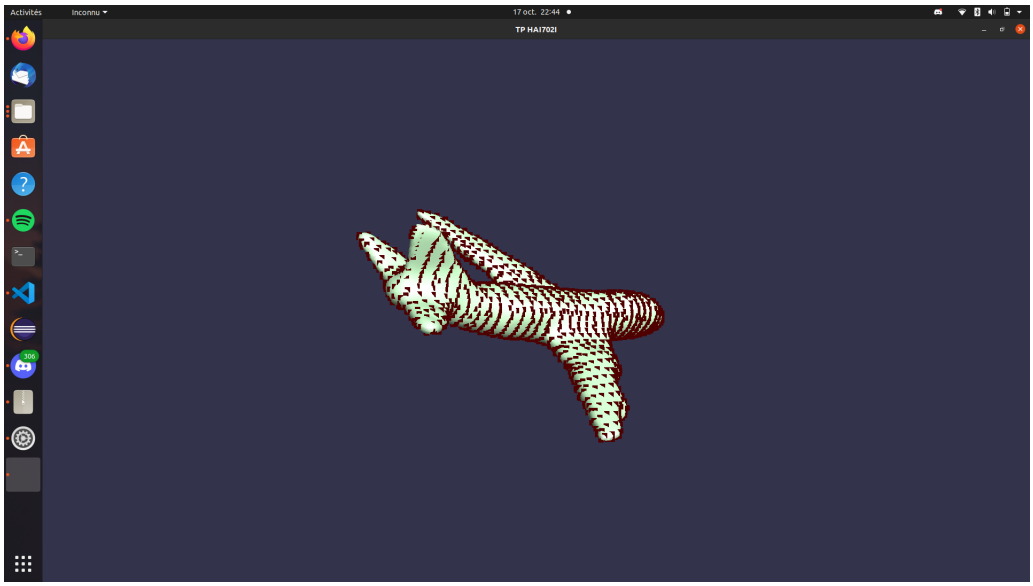


FIGURE 7.2 – Simplification des vertices avec une grille 32*32*32

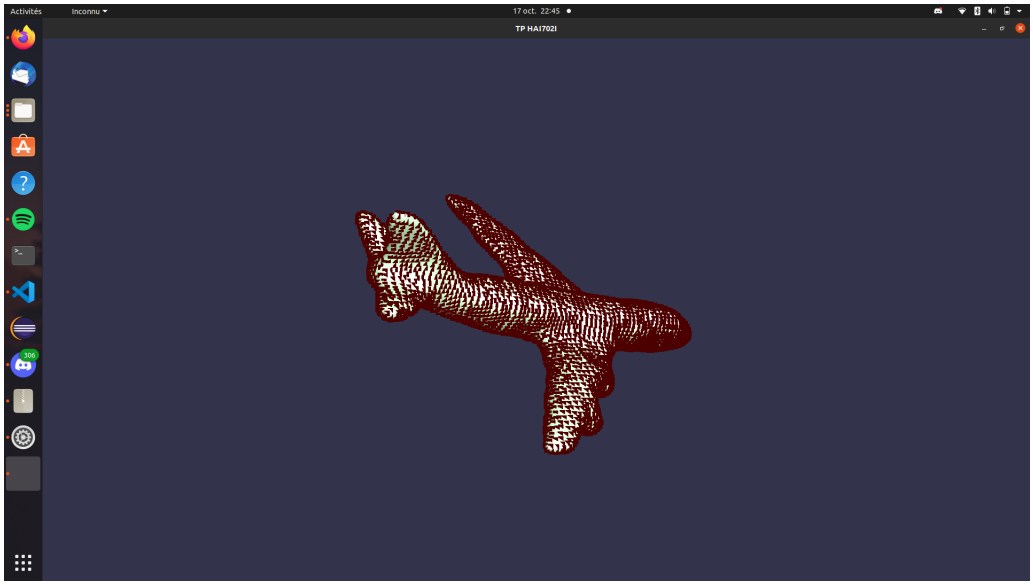


FIGURE 7.3 – Simplification des vertices avec une grille 64*64*64

8. Bonus - Mesure d'erreur quadratique

FIGURE 8.1 – Legende

[1]