SOMMAIRE

I. Introduction

Dans le cadre de notre seconde année du cycle Mathematique-Informatique à l'UFR, on nous est proposé un projet de 2 mois nous permettant de mettre en pratique nos connaissances et nos compétences pour construire un objet.

Notre groupe composé de:

- YANG Tianyi ,
- ARMAGAN Omer ,

saisi l'opportunité d'exploiter cet intérêt commun pour effectuer l'ébauche d'un projet universitaire aux responsables du cours Structures de donnees et algorithmes

- M. H-Wheeler Franck ,
- M. Morgenthaler Sebastien.

II. Gestion de projet

Pour constituer le développement du projet dès l'obtention de ce projet, le cours de Structures de donnees et algorithmes nous a assiste afin de effectuer certaines parties, toutefois, nous l'avons considèré difficile pour effectuer cette tache au niveau de gestion de temps car c'etait complexe de suivre tous les cours et a la fois terminer le projet. Malgre cela, nous avons fait de notre mieux pour acquerir des nouveaux competences et étant dans l'objectif de devenir developpeur, cette gestion est d'autant plus importante que le respect du délai, des coûts et de la performance est important dans la conception de creation d'objets.

Le principe de notre code est ainsi;

Au début la structure du programme:
Nous avons créé la structure de base de l'objet, avec une structure de sommet (vertex) et de face.

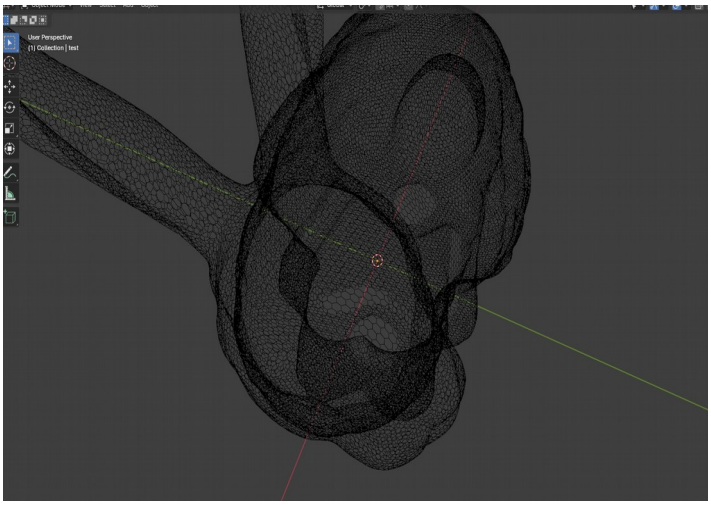
```
typedef struct vertex
{
   float a, b, c;
} Vertex;

typedef struct face
{
   int v1, v2, v3;
} Face;
```

À travers ces deux structures, nous pouvons facilement mettre en œuvre les opérations de lecture et d'écriture d'un fichier obj, avec les fonctions readObj() et writeObj() correspondantes dans le code.

Cependant, veuillez noter que la fonction writeOBJ() ici n'est pas correcte. Nous n'avons pas besoin de 'face', mais plutôt de 'ligne'.
La formule pour calculer la centroide est la suivante : un face contient trois vertex, et nous devons additionner respectivement les longueurs sur les axes *x*, *y* et *z* de ces trois vertex, puis diviser par trois.

À partir de la question 4.2, nous avons rencontré de nombreux problèmes.
Notre idée initiale etait;



```
typedef struct arete
{
    Vertex sommet1, sommet2;
} Arete;

typedef struct centroide
{
    Vertex centre;
     int faceA;
} Centroide;
```

C'est la structure Arete et centroide que nous avons conçue au début. Nous n'avons pas mis la face associe dans la structure 'arete', mais plutôt dans la structure 'centoide'. En raison de ces deux structures, nous avons conçu une fonction

*int facevoisin(Centoide c1, Centoide c2, Face *face)*

Ces fonctions initialement conçues peuvent être trouvées dans la section de commentaires située avant la fonction main() à la fin du code.
La fonction "facevoisin()" compare si deux faces ont deux sommets identiques en vérifiant s'il existe une arête commune, ce qui signifie que les deux faces sont voisines.

Ensuite, nous avons créé un programme de comparaison avec une complexité de n^2, à l'époque cette fonction n'avait pas encore la capacité de tri.Nous l'avons appelé triSelection() à l'époque, même s'il ne l'est pas vraiment(au niveau de la complexite)

La fonction triSelection(), en utilisant la fonction facevoisin() ci-dessus, commence à partir du premier élément du tableau stockant les centroïdes, compare le premier élément avec tous les éléments suivants pour déterminer s'ils sont des faces voisines. Si c'est le cas, une arête duale existe.
Ce programme fonctionne bien lorsque le volume de données est faible, mais devient super complex lorsque le volume de données est élevé.
Bien sûr, parce que ce programme s'est éloigné du sujet, nous avons abandonné cette idée.

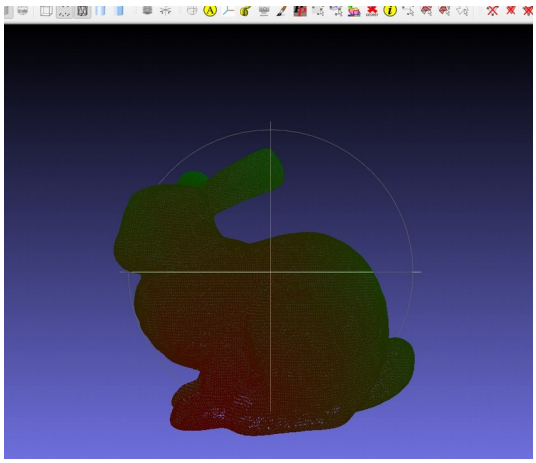
Recommencer à partir de 4.2:
la structure réaménagée:

```
typedef struct arete
{
    Vertex sommet1, sommet2;
    int faceA;
} Arete;
```

Suppression de faceA à l'intérieur de centroid et passe à la structure arete. Pour déterminer si deux aretes sont égaux, nous comparons si les sommets des deux aretes sont égaux. En comparant la hauteur du premier sommet, on compare les tailles des deux côtés.

Nous avons ensuite constaté d'importantes lacunes dans cette comparaison, mais à l'époque, nous avons quand même utilisé cette fonction pour effectuer le tri par sélection.
Après le tri par sélection, théoriquement, il est assez facile de repérer les arêtes identiques, car les arêtes identiques se trouvent nécessairement côte à côte dans le tableau.

Mais comme on l'a mentionné précédemment, cette comparaison regroupera en réalité les côtés de même hauteur du premier point, mais cela ne signifie pas nécessairement que ces deux côtés sont égaux. Ensuite, nous avons essayé plusieurs comparaisons, comme additionner les valeurs de deux sommets puis prendre la moyenne. Cependant, à la fin, nous avons constaté que toujours il existe des arêtes différentes mais qui sont adjacentes dans le tableau. Peut-être qu'utiliser les coordonnées des sommets comme paramètres de la structure arete n'est pas logique.



Final version:

```
typedef struct arete
{
    int num1, num2;
    int faceA;
} Arete;
```

En même temps que la génération des arêtes, attribuez un numéro à tous les sommets et stockez-les dans la structure des arêtes. Nous pouvons trier et comparer les arêtes en utilisant les numéros de sommet. Parce que les numéros de sommet sont uniques, le problème que nous avons rencontré précédemment ne se reproduira pas.

Mais lors du processus de recherche de aretedual, nous avons rencontré un problème. Parce que nous ne savons pas à l'avance combien de arêtes aretedual a, nous devons reallocer plusieurs fois lors de la demande de la taille du tableau.
Trop de reallocs augmentent l'instabilité et la complexité du programme, c'est pourquoi nous avons décidé de ne plus utiliser un tableau pour stocker aretedual, mais d'utiliser une liste chaînée.

```
typedef struct areted
{
    int f1, f2;
    struct areted *next;
} AreteD;
```

b)Les complexités

Complexite de la fonction triTas:

Cependant, la complexité précise peut varier en fonction de l'implémentation spécifique de *treeInsert()* et de la structure de données utilisée pour l'arbre AVL. Mais normalement, avec un arbre AVL équilibré, la complexité c'est logarithmique par rapport au nombre d'éléments déjà présents dans l'arbre, ce qui donne une complexité de O(n * log n) pour la fonction *triAVL()* dans le cas moyen. Donc la complexité de treeInsert() dans un cas moyen (pour AVL équilibré) serait en O(log n) car l'arbre AVL équilibré et des opérations d'insertion/rééquilibrage effectuées.

Complexite de la fonction triAVL:

La complexité change en fonction de *treeInsert()*. Mais dans le cas général, avec AVL équilibré, la complexité sera logarithmique par rapport au nombre d'éléments déjà présents dans l'arbre,
donc O(n * log n) pour la fonction *triAVL()* dans le cas moyen.

Complexite de la fonction triSelection:

Si la complexité de *sontEquivalentes()* est constante par rapport à la taille des arêtes, la complexité totale c'est O(n^2).
Si *sontEquivalentes()* a une complexité qui dépend de la taille des arêtes, la complexité totale serait plus élevée, mais cela dépendrait de cette fonction spécifique.