Simplification de maillage

Ensimag 1A - Préparation au Projet C

1 Présentation

L'objet de cet exercice est de réaliser un ensemble de sous-programmes permettant de charger un maillage 3D polyédrique, le simplifier, et sauvegarder le résultat. Un maillage 3D polyédrique est un objet géométrique permettant de représenter numériquement la forme d'un objet, largement utilisé notamment pour la création de films d'animation, dans les jeux vidéos, ou encore pour la conception assistée par ordinateur. Il est en général constitué de trois types de primitives : sommets, arêtes et faces planes, en anglais Vertex, Edge, Face. La simplification de maillage consiste à produire, à partir d'un maillage « high-poly » avec un grand nombre de primitives, un maillage « low-poly » qui approche le premier avec un nombre plus réduit de primitives, en vue d'accélerer différentes tâches (typiquement le rendu graphique d'une image à partir du maillage). Par souci de simplicité, nous supposerons ici manipuler des maillages fermés (aucun bord ouvert, une arête est voisine d'exactement 2 faces) et aux faces uniquement triangulaires : c'est le cas pour les données fournies.

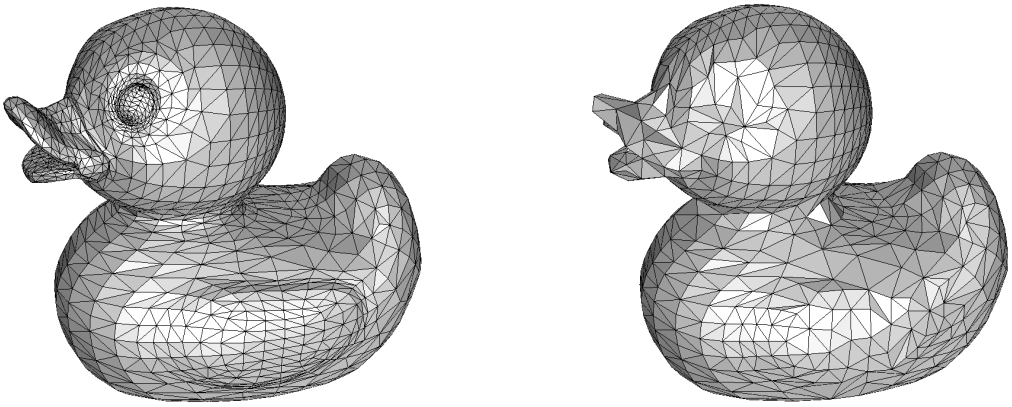

FIGURE 1 – Principe de la simplification de maillage : un maillage « high-poly » comportant 4212 faces, et un exemple de simplification à 2000 faces, conservant l'aspect général de l'objet.

Dans le cadre de ce mini-projet, nous étudierons une variante simplifiée d'un algorithme de simplification très populaire [2], reposant sur une opération élémentaire de décimation sur le maillage appellée *edge-collapse*, ou contraction d'arête, illustrée en figure 2. La contraction d'une arête orientée (u, v) consiste à supprimer l'arête du maillage en ramenant toutes les arêtes aboutissant à u vers v. L'opération entraîne la suppression du sommet u, et également la suppression des faces triangulairees à gauche et à droite de l'arête. Elle permet donc de passer d'un maillage cohérent à un autre maillage cohérent. L'algorithme de simplification à implémenter dans le cadre de ce projet consistera donc simplement à contracter successivement des arêtes du maillage selon un ordre donné (typiquement contracter les arêtes les plus courtes d'abord), jusqu'à atteindre la réduction désirée par l'utilisateur. Un module `collapse` vous est fourni dans un premier temps pour réaliser une contraction d'arête.

Le travail à effectuer dans le cadre du mini-projet répondra aux besoins fonctionnels suivants :

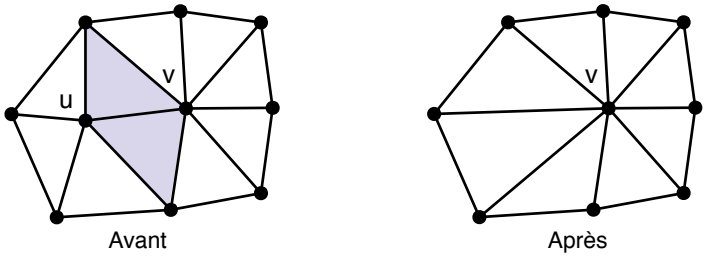

FIGURE 2 – Maillage avant et après contraction de l'arête (u, v)

— Utilisation et vérification de la structure de données imposée « half-edge »
— Lecture des arguments en ligne de commande : fichier source, destination, nombre ciblé de faces après décimation ; petit message d'aide en ligne si ces arguments sont absents.
— Chargement / Sauvegarde d'un maillage au format OFF.
— Sous-programme de simplification du maillage.

Lors de ce travail, vous serez amenés à étudier et à manipuler les constructions suivantes du langage C et de son environnement : structures et pointeurs, lecture d'arguments en ligne de commande, pointeurs de fonction, allocation dynamique / libération de la mémoire, manipulation de fichiers, assertions, débogage d'algorithmes utilisant une structure avancée : gdb, ddd, valgrind.

2 Structure de donnée imposée et vérification

Pour manipuler un maillage dans ce programme, vous utiliserez une structure de donnée classique et répandue : la structure de *half-edge*, ou demi-arête, qui permet de stocker efficacement toute relation d'adjacence et d'incidence sur le maillage. Celle-ci se présente de la manière suivante :

```
/* definition d'une demi-arete (half-edge) */
struct half_edge {
    struct half_edge *next;          /* prochain half-edge dans la face */
    struct half_edge *prev;          /* predecesseur du half-edge dans la face */
    struct vertex    *vertex;        /* sommet vers lequel pointe ce half-edge */
    struct face      *face;          /* face a laquelle appartient ce half-edge */
    struct half_edge *pair;          /* half-edge jumeau */
    /* donnees supplementaires possibles ici */
};

/* definition d'un sommet (vertex) */
struct vertex {
    float x, y, z;                   /* coordonnees du sommet */
    struct half_edge *leaving_edge;  /* une demi-arete partant de ce sommet */
};

/* definition d'une face (=polygone) */
struct face {
    struct half_edge *edges;         /* un element de la liste des demi-aretes */
};
```

Cette structure de donnée repose sur la notion de demi-arête *orientée*, qui possède presque toutes les incidences sur le maillage. Chaque arête du maillage initial est représentée par deux demi-arêtes qui sont appariées (u, v) et (v, u), chacune possédant un pointeur `pair` sur l'autre. Une demi-arête possède un pointeur `face` sur la face immédiatement à sa gauche, et sur son sommet d'arrivée (`vertex`). Elle stocke

également deux pointeurs `prev` et `next` vers les demi-arêtes précédentes et suivantes de cette même face (double chaînage). Avec cette structure, une face peut simplement être définie avec un pointeur sur une demi-arête de la face, le reste des arêtes étant visitables en suivant le chaînage des demi-arêtes. De même, un sommet ne stocke que ses coordonnées, et un pointeur `leaving_edge` sur l'une des arêtes partant du sommet, les autres étant visitables en suivant le chaînage (si `e` est une demi-arête partant du sommet, la suivante est `e->pair->next`). Similairement, le sommet origine d'une arête est accessible via `e->pair->vertex`. Ces relations sont illustrées en figure 3 :

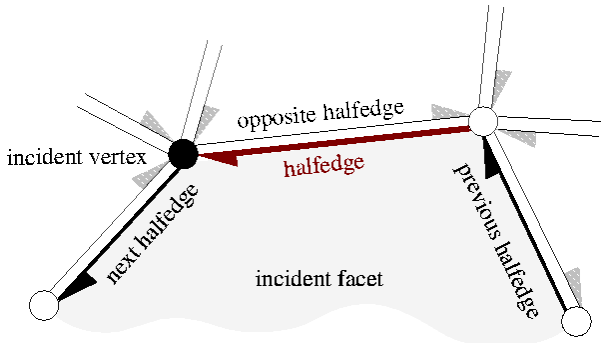

FIGURE 3 – Demi-arête et relations d'incidence stockées.

Pour encapsuler et stocker l'ensemble des primitives d'un même maillage, on utilisera simplement une structure de trois listes, en utilisant le module `list` fourni :

```
struct mesh {
    struct list *vertices;
    struct list *faces;
    struct list *hedges;
};
```

Pour simplification, le module `list` encapsule les allocations et désallocations dynamiques de cellules mémoires, via les fonctions d'insertions et de retrait d'éléments dans la liste. Il vous faut néanmoins maintenir des allocations et pointeurs cohérents dans le maillage. Notez qu'une liste manipule des pointeurs de donnée génériques de type `void*`, permettant de l'utiliser pour n'importe quel type de primitive, la taille des données manipulées étant spécifiée à la création d'une liste.

Ecrivez une fonction vérifiant la cohérence d'une structure de maillage, permettant d'identifier le cas échéant une primitive incohérente. Notez que pour un maillage fermé, la structure ne doit contenir aucun pointeur `NULL`. Pour vérifier la bonne construction d'un maillage, on pourra aussi vérifier s'il est triangulaire, la *fermeture* du maillage (chaque demi-arête possède-t-elle une paire ?), ou encore la réciprocité des liens entre primitives : les demi-arêtes chaînées dans une face pointent-elles vers la bonne face ? les demi-arêtes partant d'un sommet ont-elles bien pour origine ce sommet ? le double-chaînage des arêtes d'une face est-il cohérent ? D'autres critères sont possibles. Vous utiliserez cette fonction lors des étapes suivantes (`assert`...)

3 Lecture et écriture de fichiers OFF

Les maillages manipulés seront lus et écrits au format OFF, un format ASCII simple et répandu, visualisable avec l'outil `geomview` installé sur les machines de l'Ensimag. Des exemples de tels maillages vous sont fournis pour vos tests dans l'archive du projet. Sous sa forme la plus simple, le format est le suivant :

```
OFF                          # en-tête OFF
NVert  NFace  0              # nb de sommets, faces
x  y  z                      # coordonnées du sommet 0
...
x  y  z                      # coordonnées du sommet NVert-1
Nv  v[0] v[1] ... v[Nv-1]    # face 0: Nv sommets dans la face
...                          #   (v est un index de sommet dans 0..NVert-1)
Nv  v[0] v[1] ... v[Nv-1]    # face NFace-1
```

c'est à dire une entête OFF, suivi du nombre de primitives, de la liste des coordonnées des sommets et de la liste des faces. Chacune des faces comporte le nombre de sommets et une liste d'indices de sommets définissant la face.

Implémentez le chargement et la sauvegarde d'un maillage à partir d'un fichier OFF.

- Au cours du chargement et sauvegarde d'un OFF, on a besoin de faire la correspondance entre un `vertex` et son indice dans la liste. Utilisez la fonction `list_get_index` du module `list`.
- Le chargement doit créer les différentes primitives et fournir un maillage cohérent. Notamment les chaînages des faces et tous les différents pointeurs doivent être correctement initialisés.
- La plupart des chaînages et primitives peuvent être créés directement à partir des informations lues : la lecture de chaque nouveau sommet d'une face dans le OFF correspond à une création de demi-arête. Néanmoins l'initialisation du champ `pair` pose un problème particulier car il faut établir la correspondance d'une demi-arête et sa jumelle, ce qui nécessite une recherche parmi les demi-arêtes déjà créées (l'incidence entre faces n'est pas explicitement stockée dans le fichier OFF). Il peut être utile de créer par sommet au chargement une liste provisoire non ordonnée des demi-arêtes quittant le sommet. Au moment de la lecture d'un nouveau sommet et la création de la demi-arête (u, v) correspondante, il est alors possible de chercher sa jumelle `e` dans la liste du sommet d'arrivée, qui vérifiera `e->vertex==u` .
- Il est conseillé de déboguer sur de petits maillages dans un premier temps (SMALL-SPHERE, OCTAHEDRON).

4 Simplification

Implémentez la simplification par contractions successives. On pourra affecter une priorité de décimation à chaque demi-arête sous la forme d'un champ `float` ajouté dans `half_edge`, **après** les champs existants. Cette priorité pourra être calculée directement avec la norme de l'arête et gérée en traitant la liste d'arêtes du maillage comme une file de priorité. Par souci de simplicité, on pourra implémenter cette file en triant **en place** la liste d'arêtes d'un maillage, ce qui permet de gérer en une seule et même opération la suppression d'une arête du maillage et sa disparition de la file de priorité. Notez que les fonctions de tri de liste fournies prennent en argument un pointeur vers une fonction fournissant la relation d'ordre entre deux données de la liste.

- énumérer les demi-arêtes pour les contracter successivement jusqu'à atteindre le nombre de faces cible indiqué par l'utilisateur. Un module binaire est fourni pour la contraction d'une arête du maillage (`collapse.h/.o`) et fait l'objet d'une extension (section 5). Il est important de n'ajouter de nouveaux champs qu'après ceux existants dans les structures `half_edge`, `vertex`, `face` et `mesh`, pour préserver la compatibilité binaire du module.
- attention, chaque contraction d'arête modifie la norme des arêtes dans le voisinage immédiat de (u, v), et donc nécessite un réordonnancement de ces arêtes dans la liste triée.
- Certaines contractions peuvent être identifiées comme invalides par le module `collapse` (voir les explications en section 5). Attention à bien gérer celles-ci dans la file.

5 Contraction d'arête (extension)

Cette partie donne les éléments permettant de remplacer le module `collapse` fourni par votre propre module. Avant de contracter une arête (u, v) (figure 2), il est nécessaire de vérifier si la contraction envisagée est *valide*, c'est à dire qu'elle n'engendre pas un maillage incohérent [1]. Pour cela, il suffit de vérifier que l'intersection des 1-voisinages de u et de v correspondent à exactement 2 sommets distincts [1] :

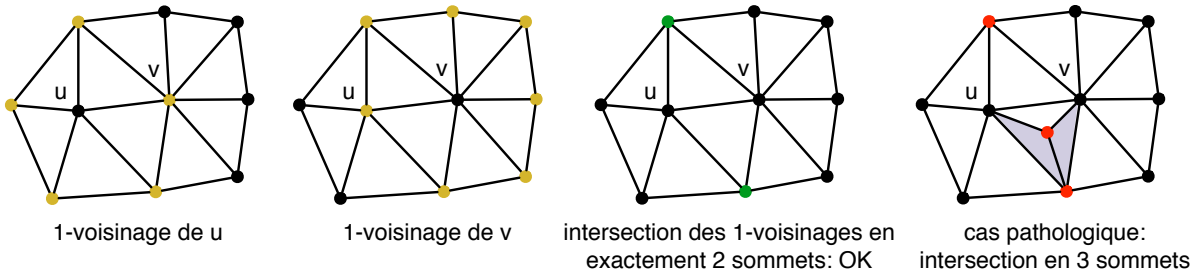

FIGURE 4 – Validité d'une contraction (condition du lien, démontrée en [1]). Dans le cas pathologique montré ici, la contraction de la demi-arête (u,v) entraînerait l'applatissement des 2 triangles colorés l'un sur l'autre, engendrant un maillage incorrect.

Ecrivez une fonction énumérant et renvoyant le *1-voisinage* d'un sommet sous forme d'une liste.

Implémentez la contraction d'arête. Celle-ci doit assurer la mise à jour de tout chaînage référençant le sommet u à éliminer. Attention aux pointeurs invalides pointant sur une structure entre-temps désallouée ! Utilisez judicieusement la programmation défensive et les assertions.

- Faire la contraction seulement si celle-ci est valide.
- L'arête (u, v), sa jumelle et les triangles adjacents doivent disparaître des chaînages des primitives adjacentes, en particulier de tout champ `leaving_edge` et `edges` de tout sommet ou face.
- Toute demi-arête pointant initialement vers u doit pointer vers v.
- Après avoir mis à jour les pointeurs, supprimez/désallouez le sommet u, l'arête (u, v) et sa jumelle, et les faces adjacentes à (u, v) du maillage.

6 Autres extensions possibles

- exécutez votre code de simplification sur le plus gros maillage (COW.OFF), et profilez votre code pour identifier la partie la plus gourmande en temps de calcul. Vous vous documenterez sur l'outil `gprof`. Optimisez votre code. Il pourra être utile de distinguer 2 modes de compilation DEBUG et RELEASE, l'un non optimisé mais avec les tests de débogage, l'autre sans tests de débogage (ajouter -DNDEBUG pour désactiver les assertions) et avec options de compilation optimisée.
- OFF permet de définir des couleurs par sommet et face du maillage (cherchez la documentation de référence). Gérez les couleurs possibles dans vos structures et utilisez-les pour visualiser les priorités de décimation et déboguer.
- définir et tester d'autres critères de qualité de simplification que la longueur des arêtes.

1. Le 1-voisinage d'un sommet est l'ensemble des sommets voisins distants d'exactement une arête sur le maillage.

Références

[1] Tamal K. Dey, Herbert Edelsbrunner, Sumanta Guha, and Dmitry V. Nekhayev. Topology preserving edge contraction. *Publ. Inst. Math. (Beograd) (N.S)*, 66 :23–45, 1998.

[2] Hugues Hoppe. Progressive Meshes. pages 99–108, New York, 1996. ACM Press / ACM SIGGRAPH.

7 Annexe