Rapport du projet modélisation géométrique

Abou Edou Florian, Glass Benjamin

December 2024

1 Introduction

Le projet ci-joint se donnait l'objectif d'implémenter une méthode efficace afin de simplifier un maillage. La difficulté de ce procédé est de trouver un critère de sélection sur les sommets et les arêtes à contracter pour un rendu satisfaisant. L'algorithme de simplification proposé par Hyun Soo Kim, Han Kyun Choi, et Kwan H. Lee a pour qualité de proposer une contraction qui est moins coûteuse en mémoire que la méthode QEM, avec une bonne conservation de la géométrie qualitativement parlant.

2 Rappel de l'algorithme

Il consiste à calculer le coût de la contraction (nous parlerons également par la suite de "collapse") de chaque arête et de le placer dans un tas-min, avec comme racine le coût de contraction le moins élevé. À chaque itération, on effectue la contraction la moins coûteuse en se basant sur le tas créé précédemment. On parcourt ensuite le tas et on met à jour les erreurs des arêtes dont une extrémité est incidente à l'arête précédemment contractée, puis on supprime les arêtes incidentes au nœud effacé. On met également à jour les arêtes qui sont reliées à une arête adjacente à l'arête supprimée. En effet, les vecteurs unitaires utilisés pour calculer l'erreur ne sont plus les mêmes.

On continue ainsi jusqu'à atteindre un seuil au-delà duquel on juge les contractions trop coûteuses pour être effectuées. Une fois ce seuil atteint, on arrête, et on affiche le rendu. Afin d'uniformiser nos tests, notre seuil sera le coût médian de contraction d'une arête.

3 Présentation des outils utilisés

L'algorithme prend en entrée un maillage représenté sous la forme d'un fichier .off. Pour pouvoir manipuler les données du fichier de manière efficace, nous avons choisi d'utiliser la librairie OpenMesh offrant une structure de données efficace à manipuler (mesh qui implémente la structure halfedge) avec des fonctions de transformation de maillage préimplémentées. Notre programme est décomposé en 5 modules et un fichier main:

- `acquisition.py` permettant de lire un fichier .off et de le retranscrire en une instance mesh grâce à la fonction `read_trimesh`.

- `affiche_mesh.py` permettant d'afficher le maillage contenu dans la structure de données mesh sur une fenêtre graphique. Cette fonction utilise les iterateurs mesh.vertices() renvoyant les vertices du maillage mesh , mesh.faces() renvoyant les faces contenu dans mesh et mesh. fv() renvoyant les vertices d'une face.

- `mesh_manipulation.py` contient des fonctions permettant de parcourir et d'initialiser une couleur à chaque sommet. Le fait original est que la couleur est déterminée selon la courbure du sommet dans le maillage. Plus la courbure est élevé plus le point est rouge. On utilise les fonctions mesh.vih(v) qui parcours les demi-arêtes e incidente au sommet v à colorer que l'on place en argument de mesh.face_handle(e) pour obtenir la face f auquelle elle appartient. On récupère également la demi-arête opposée à e grâce à la fonction mesh.handle_opposite_halfedge et on obtient également la face auquelle elle appartient. On obtient les normales aux 2 faces en utilisant la fonction mesh.calc_face_normal(f).
  Le module contient également des fonctions plus génériques comme la fonction get_neighbors prenant en argument un voisin v et enumérant grâce à la fonction mesh.vv ses voisins.

- `error.py` contient les fonctions permettant de calculer l'erreur de simplification. La fonction get_rgb contient la méthode mesh.color(v) qui permet de récupérer la couleur d'un vertex v. La fonction Q(mesh,v0,v1) permet de calculer l'erreur quadratique de la suppression de l'arête [v0,v1] en parcourant les faces adjacentes au vecteur. La fonction I(mesh,v) permet de calculer l'importance visuelle d'un vertex v. La fonction C(mesh,v0,v1) permet de calculer l'erreur de collapse de l'arête [v0,v1] Enfin la fonction error(mesh,v0,v1) permet d'obtenir l'erreur de simplification totale affectée à une arête orienté e1, égale au produit des fonctions de coût auxquelles on a ajouté 1 à chaque composante.

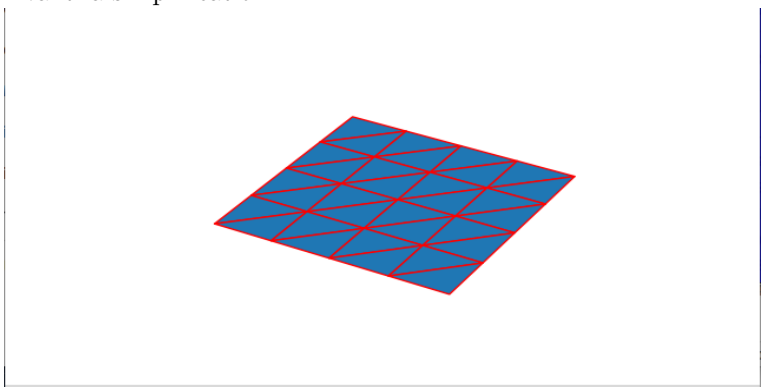
- `error_heap.py` contenant les différentes méthodes permettant de manipuler un tas qui va nous permettre de ranger par ordre de priorité les arêtes à simplifier selon l'erreur la plus petite. On utilise dans la plupart des cas des methodes préimplémentées du module heapq , en particulier heappush pour insérer une arête, heappop pour supprimer l'arête dont le coût de suppression est minimum.

- `main.py` qui contient l'implémentation de l'algorithme de simplification précisée plus haut.

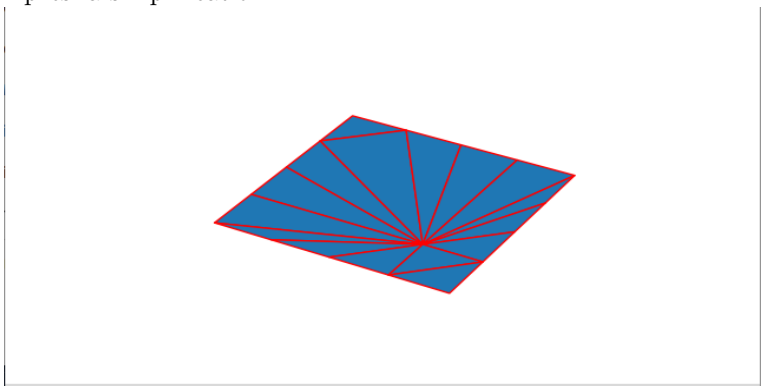
4 Résultats

Un jeu de test a été mis en place de façon à tester et évaluer l'efficacité de l'algorithme par rapport à un nombre croissant d'arêtes. On obtient les résultats suivants, consultables également avec MeshLab dans le dossier `src/main`:

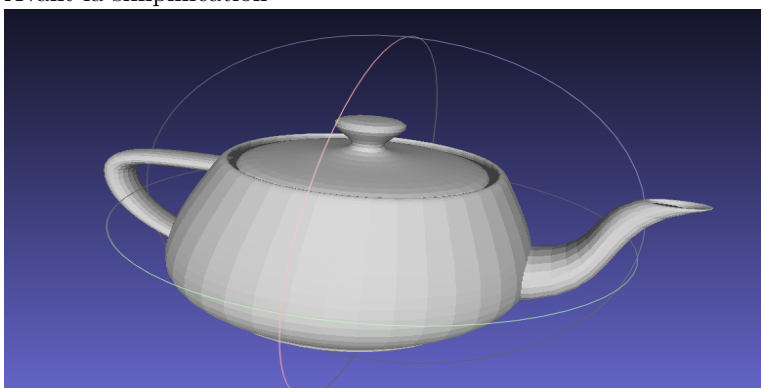
- plan55.off
  Avant la simplification



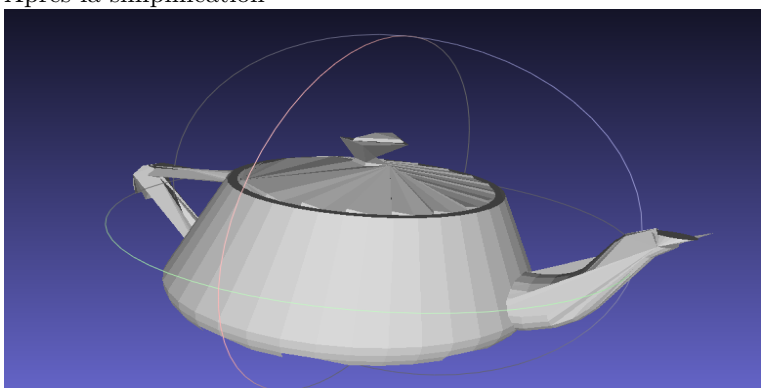
  Après la simplification



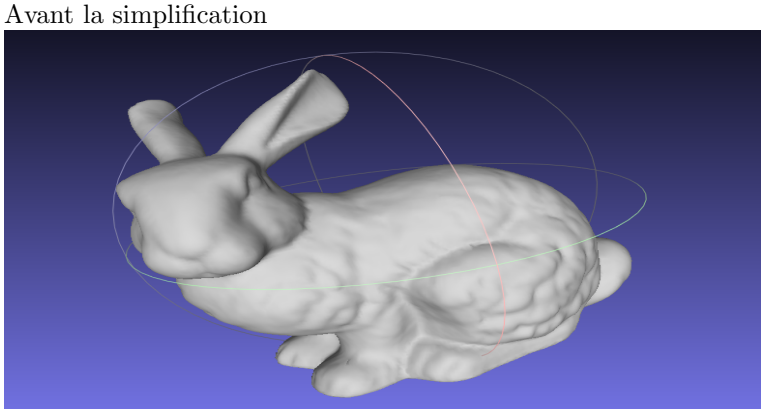
- teapot.off
  Avant la simplification



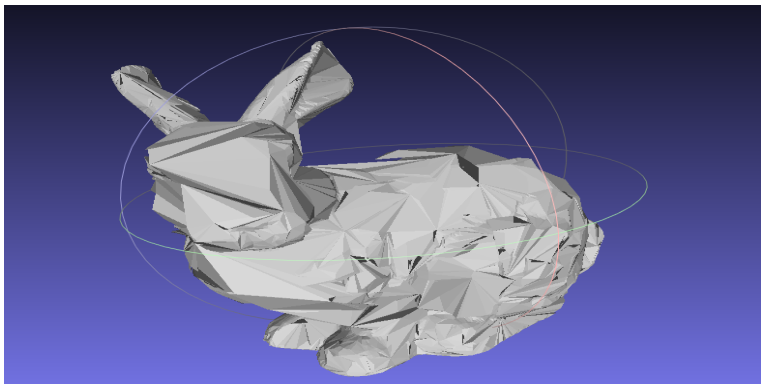
  Après la simplification



- bunny.off
  Avant la simplification



  Après la simplification



performance de la simplification (nombre d'edge)				
modèle	avant simplification	après simplification	ratio de simplification (%)	temps d'exécution de la simplification (s)
plan55.off	56	32	42.9	0.03
teapot.off	9565	1258	86.8	27.33
bunny.off	104288	13835	86.7	276.54

Finalement, notre algorithme effectue une simplification efficace en terme d'arêtes et satisfaisant visuellement parlant, mais le temps d'exécution peut être amélioré. Des pistes d'améliorations seraient par exemple le passage du code dans un langage plus performant, comme le C++, ou encore d'optimiser le calcul des quadrics, qui est le calcul le plus coûteux de notre programme.

Algorithm 1: Mesh simplification with vertex color algorithm

Data: mesh $mesh$, heap $heap$, threshold $threshold$
Result: mesh simplifié
while $!heap.isEmpty()$ **do**
  $error, halfedge \leftarrow heap.pop()$;
  if $error > threshold$ **then**
    $break$;
  end
  if $collapsePossible(halfedge)$ **then**
    $collapse(mesh, halfedge)$;
    $updateHeap(heap, halfedge)$;
  end
end
return $mesh$;

Algorithm 2: Error

Data: mesh M, le point de départ de l'arrête v_0 et le point d'arrivée de l'arrête v_1
Result: erreur de la contraction v_0 à v_1
$Erreur \leftarrow (1 + Q(M, v_0, v_1)) * (1 + I(M, v_0)) * (1 + C(M, v_0, v_1))$;
return $Erreur$