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 contracter de manière optimale 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 soit peu coûteuse en mémoire, avec un rendu agréable et tenant compte de la coloration des sommets.

2 Rappel de l'algorithme

L'algorithme présenté à la fin de la feuille. Il consiste à calculer l'erreur de la contraction de chaque arête et de la placer dans une file de priorité (classée selon les erreurs croissantes). À chaque itération, on pioche une arête de la file et on effectue une contraction de l'arête (edge collapse) en supprimant le nœud où part l'arête. On parcourt ensuite la liste et on met à jour les erreurs des arêtes dont une extrémité est incidente à l'arête précédemment contractée et on supprime celles incidentes au nœud effacé. On met également à jour les arêtes qui sont reliés à un arête adjacente à l'arête supprimée car les vecteurs unitaires utilisés pour calculer l'erreur ne sont plus les mêmes.

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. Le module créé est décomposé en 5 fichiers python:

- 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 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 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 supression 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 total affectée à une arête orienté e1 egale au produit des fonctions de coût auxquelles on a rajouté 1 à chaque composante.

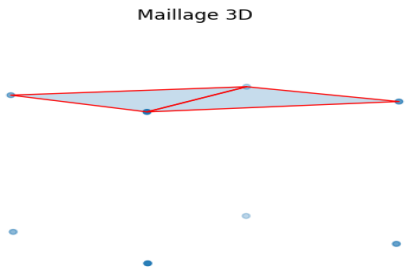
- Error_heap contenant les différentes méthodes permettant de manipuler un tas qui va nous permettre de ranger par ordre de priorité les arêtes à simplifier selon l'erreur la plus petite. On utilise dans la pupart des cas des methodes préimplémentée du module heapq , en particulier heappush pour insérer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

- Main qui contient l'implémentation de l'algorithme de simplification précisée plus haut. Pour permettre de simplifier l'algorithme, la structure de donnée a été modifiée : la liste a été changée en un dictionnaire de tas dict rangé selon les vertices v de telle sorte que le tas d'indice v contient toutes les arêtes dont v est l'une des extrémité.

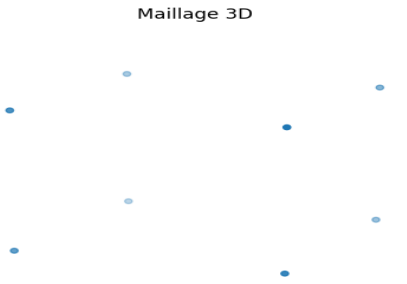
4 Resultats

Un jeu de test a été mise en place de façon à tester et évaluer l'efficacité de l'algorithme par rapport à un nombre croissant de noeud. On obtient le résulat suivant:

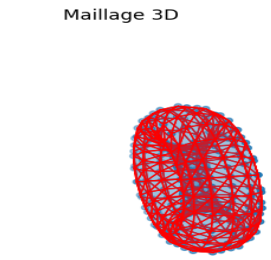
- my_file.off
  Avant la simplification



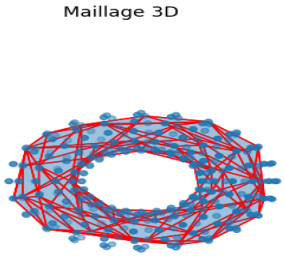
Après la simplification



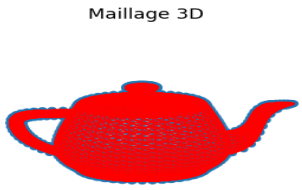
- torus.off
  Avant la simplification



  Après la simplification



- teapot.off
  Avant la simplification



  Après la simplification

performance de la simplification (nombre d'edge)		
modèle	avant simplification	après simplification
my_file.off	18	0
torus.off	5184	1134
teapot.off	5680	9

Algorithm 1: Mesh simplification with vertex color algorithm

Data: mesh M
Result: mesh M simplifié
$Heap \leftarrow initialiseHeap()$;
for $edge \in M.Halfedges$ **do**
  $point\_deb \leftarrow M.point\_from\_Halfedge(edge)$;
  $point\_fin \leftarrow M.point\_to\_Halfedge(edge)$;
  $error \leftarrow error(M, point\_deb, point\_fin)$;
  $Heap \leftarrow Heap.push(error, point\_deb, point\_fin, edge)$;
end
while $!Heap$ **do**
  $error, point\_deb, point\_fin, edge \leftarrow Heap.pop()$;
  $point\_detruit \leftarrow point\_deb$;
  $point\_invariant \leftarrow point\_fin$;
  $M \leftarrow M.collapse(edge)$;
  $Heap2 \leftarrow initialiseHeap()$;
  while $!Heap$ **do**
    $erreur, vertex_{debut}, vertex_{fin}, edge_{init} \leftarrow Heap.pop()$;
    if
     $vertex_{debut} == point\_invariant || vertex_{fin} == point\_invariant$
     then
      $error \leftarrow error(M, vertex_{debut}, vertex_{fin})$;
      $Heap2 \leftarrow$
       $Heap2.push(error, vertex_{debut}, vertex_{fin}, edge_{retr})$;
    end
  end
  $Heap \leftarrow Heap2$
end
return M

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 $Erreur$