Algoritmo del Convex Hull

Sergio Serusi
65041

1 Descrizione classe ConvexHullCore

La classe Convex Hull Core è la classe che contiene il cuore dell'algoritmo per il calcolo del Convex Hull.
La classe è composta dai seguenti metodi:

- ConvexHullCore(DrawableDcel *dcel), è il costruttore della classe, esso riceve in ingresso un puntatore alla Dcel e si occupa di creare il vettore che conterrà i punti della Dcel stessa;
- verifyEuleroProperty(), è un metodo che verifica se sia rispettata la proprietà della formula di Eulero. La formula di Eulero afferma che il numero di vertici presenti nella Dcel meno il numero di facce e sommato il numero di edge sia uguale a due. Restiruisce true se la proprietà è rispettata, false altrimenti;
- getVertex(), scorre tutti i vertici presenti nella Dcel e gli salva in un vettore (vertexS);
- isCoplanar(), controlla i primi quattro punti che andranno a formare il tetraedro iniziale di partenza, verificando se questi siano coplanari e quindi se questi quattro punti stiano sullo stesso piano. Per verificare se siano coplanari si calcola il determinante (usando Eigen come consigliato nelle slide del seminario) della matrice formata dai punti, se il determinante è uguale a zero allora i punti sono coplanari, quindi restituisce true, altrimenti false;
- executePermutation(), questo metodo viene eseguito per rendere l'input del problema randomizzato. Viene utilizzato un metodo della Standard Library, random shuffle, che dato in input l'iteratore del vettore, questo ne effettua una permutazione, in tempo lineare;
- setTetrahedron(), questo metodo viene eseguito per la creazione del tetraedro iniziale.
  Viene richiamato il metodo isNomalFaceTurnedTowardsThePoint() per verificare se i primi tre punti che andranno a costruire la prima faccia (del tringolo iniziale)

abbiano la normale rivolta verso il quarto punto, se così fosse, vengono invertici i punti uno e tre in modo da invertire la faccia.
Viene creato il trinagolo iniziare con la faccia e i suoi half edge, vengono settati tutti i paramentri relativi ai suoi half edge eccetto per i twin, che verranno settati nel metodo createNewFaces().
Una volta costruito il triangolo, si richiama il metodo createNewFaces() per creare le nuove tre facce del tetraedro, passando come input l'orizzonte del triangolo e il quarto vertice;

- createNewFaces(), riceve in ingresso l'orizzonte e il vertice su cui costruire le nuove facce.
  Il metodo, scorre la lista degli half edge dell'orizzonte, e per ognuno di essi crea una faccia e tre half edge. Dato un half edge dell'orizzonte, costruisco il primo half edge in modo che sia opposto all'half edge dell'orizzonte (che sarà il suo twin, e viceversa). I successivi half edge vengono costruiti collegando il nuovo vertice ricevuto in ingresso, e salvandoli in due vettori differenti, uno contiene l'half edge che è entrante sul vertice dell'horizon e l'altro gli half edge che sono uscendi dall'half edge dell'orizzonte. Grazie a questo, i twin vengono settati secondo la proprietà che il twin dell'half edge uscente sia l'half edge del vettore dei vettori entranti, di una posizione in più rispetto alla sua, quindi, uscente[i]->setTwin(entrante[(i+(dim-1))%dim]) e viceversa;

- isNormalFaceTurnedTowardsThePoint(), questo metodo si occupa di verificare se la faccia costruita dai primi tre punti abbia la normale rivolta verso il quarto punto. Questo metodo ci permette di evitare di invertire il senso degli half edge. Questo piccolo metodo è molto importante.
  Per verificare se la normale della faccia sia rivolta verso il punto, si verifica il calore del determinante, se negativo vuol dire che la normale è rivolta verso il punto;

- getHorizon(), il metodo riceve in ingresso un puntatore ad un set delle facce visibili dal punto corrente. Si scorrono gli half edge delle facce visibili e si verifica se il loro twin appartenga ad una faccia visibile, se ciò non fosse, vuol dire che si tratta di un half edge dell'orizzonte e lo inserisco in un set.
  Sucessivamente, mi servo di una mappa per ordinare gli half edge, una mappa map<Dcel::Vertex*, Dcel::HalfEdge*> dove inserisco per from vertice dell'half edge. Infine, partendo da un half edge dell'orizzonte, ordino tutti gli half edge dell'orizzonte;

- removeFacesVisibleByVertex(), rivece come input le facce visibili dal punto corrente e si occupa di eliminarle dalla dcel.
  Scorrendo l'elenco delle facce visibili, per ogni la scorro seguendo gli half edge e gli elimino, e decremento la cardinalità dei vertici cui tratta l'half edge, se i vertici che tratta hanno cardinalità zero, allora gli eliminerò dalla dcel. Una volta che ho seguito tutto il percorso della faccia, ne elimino la faccia stessa;

- findConvexHull() è il metodo principale della classe, si occupa di calcolare il convex hull del modello considerato. Questo metodo non ha bisogno di essere spiegato in quanto è palesemente lo stesso delle slide ma tradotto da pseudo-codice a codice c++.

2 Descrizione classe ConflictGraph

La classe Conflict Graph è la classe che contiene lo sviluppo del conflict graph, che ci permette di portarci dietro lo stato dell'algoritmo e quindi passare da una complessità O(n*n) a O(n*log n).
Il conflict graph per mantenere le informazioni si serve di due inportanti strutture dati, v_conflict è una mappa map<Dcel::Face*, std::set<Dcel::Vertex*>*>, invece f_conflict è una mappa map<Dcel::Vertex*, std::set<Dcel::Face*>*>. La struttura v_conflict, associa ad una faccia un set di vertici in conflitto con essa, f_conflict, associa ad un vertice un set ti facce in conflitto con il vertice.
La classe è composta dai seguenti metodi:

- ConflictGraph(), è il costruttore della classe, si occupa di salvare il puntatore della dcel e del vettore dei punti (vertexS);
- initializeCG(), è il metodo che si occupa di inizializzare il conflict graph. Partendo dal quinto (posizione 4) punto del vettore si occupa di verificare quali facce vedono i punti e quindi sono in conflitto. Se la normale della faccia considerata è rivolta verso il punto considerato, allora inserisco il punto e la faccia sia in f_conflict che in v_conflict;
- isVisible(), riceve come input una faccia ed un vertice e si occupa di verificare se la normale della faccia sia rivolta verso il punto, e quindi se il punto vede la faccia;
- addFaceToVertex(), rivece in ingresso un vertice vertex ed una faccia face, inserisce nella mappa f_conflict alla posizione vertex, la faccia considerata;
- addVertexToFace(), come il metodo precedente, ma inserisce nella mappa v_conflict;
- deleteFaces() riceve in ingresso le facce da eliminare dal Conflict graph. Questo metodo elimina i riferimenti alle facce dei punti, e poi elimina le facce stesse. questo metodo è stato sviluppato in modo ottimizzato, non verifica su tutti i set delle facce associati ad ogni vertice se la faccia è presente e allora la elimina, ma solo sui vertici che siamo sicuri che abbiano come faccia in conflitto la faccia considerata;
- deleteVertex() riceve in ingresso il vertice da eliminare. Questo metodo è duale al precedente;
- getFacesVisibleByVertex() riceve in ingresso un vertice vertex. Questo metodo restituisce un set di facce che sono in conflitto e quindi visibili dal vertice vertex;

- getVertexVisibleByFace() riceve in ingresso una faccia face. Questo metodo restituisce un set di vertici che sono in conflitto e quindi visibili dalla faccia face;
- updateCG() riceve in ingresso una faccia ed un set di vertici. Questo metodo so occupa di aggiornare il conflict graph, data la faccia ricevuta in ingresso, questa è in conflitto con il set di vertici (anche essi ricevuti in ingresso) e quindi vengono aggiornati v_conflict ed f_conflict;
- getVertexMapToControlForTheNewFace(), questo metodo riceve in ingresso l'orizzonte e restituisce una mappa che associa ad ogni half edge dell'orizzonte, i set di vertici in conflitto con la faccia sua e del suo twin. Questo metodo è importante perchè ci permette di aggiornare il conflict graph in modo ottimale, secondo la proprietà che data una nuova faccia f che si costruisce sull'half edge dell'orizzonte hE, la faccia f può essere in conflitto solo con i vertici presenti nella faccia di hE e del suo twin.

3 Flusso di Esecuzione dell'algoritmo

Il flusso di esecuzione sostanzialmente constiste nei passi dell'algoritmo stesso e quindi si tratta di una descrizione stessa del metodo findConvexHull() della classe ConvexHullCore.
Si inizia chiamando il metodo getVertexs() dove ci salva i vertici della dcel in un vettore, ne eseguiamo la permutazione di questi punti per avere l'input randomizzato. Eseguiamo il metodo reset sulla dcel, in modo da resettarla perchè questa conterra i vertici, le facce e gli half edge del convex hull. Sucessivamente viene creato il conflict grah e viene inzializzato.
Per ogni vertice del vettore che contiene i vertici, partendo dal quinto, si verifica se il vertice corrente ha delle facce in conflitto, se non ne ha vuol dire che è interno al convex hull e quindi non facciamo nulla se non eliminare il punto stesso dal conflict graph, se invece ha delle facce in conflitto vuol dire che questo punto è fuori. Si aggiunge questo punto alla dcel, si recupera l'orizzonte delle facce che vede il punto corrente, su cui costruire poi le nuove facce, si eliminano le facce visibli dal conflict graph e si rimuovono dalla dcel. Sucessivamente si creano le nuove facce, per ogni faccia si aggiorna il conflict graph. Infine si elimina il punto corrente dal conflict graph.
L'algoritmo termina quando tutti i punti sono stati verificati.