1

R3 : $<, >, <=, >=$
ordre lexicographie sur les pair est deja dans la stl

1.1 Multimaps

(Requires a less-than comparison function.)

$O(3n_T)$: jede Kante jedes Dreiecks als Schlüssel
$|K|$ $O(log(n_T))$: Zugriff auf Container fur jede Kante
$\Rightarrow O(n_T log(n_T))$
speichern in 2dim Array : Zugriff in konstanter Zeit

unordered map(hash tables) has constant time performance on all operations provided no collisions occur. When collisions occur, traversal of a linked list containing all elements of the same bucket (those that hash to the same value) is necessary, and in the worst case, there is only one bucket ; hence O(n)

1.2 Listen

$O(3n_T)$, push front() : jede Kante jedes Dreiecks einfügen
$O(NlogN)$, container size N : sort() : definiere hierfür $<$ für R3 (Nach ersten und dann nach zweitem Element sortieren etc)

doppelt verlinkt ! und dann mit Kante und Sommet prev und next von Listenelement !
konstanter Zugriff mit Iterator !

$O(3n_T)$: pop front und speichern in 2 dim array

Alternative für 2D array für konstanten Zugriff ?
Daten könnten direkt in 2D array gespeichert werden, Listen und Maps überflüssig

1.3 aire

orientierter Flächeninhalt

$$(a_i, b_i, p) = (a_i \times b_i) \cdot p = \det(a_i, b_i, p)$$

bei det Vektoren in Zeilen

2 Première tentative de documentation

3 Le projet

3.1 La classe template T3 et la classe Triangle

3.2 La classe maillage

3.3 Trouver le triangle adjacent

setAdjacencyViaMultimap

Le but de cette méthode est l'initialisation des membres *neighbor1, neighbor2, neighbor3* de la classe triangle. Les membres sont décrits pas leurs position dans la liste (ARRAY) des triangles.
L'initialisation est réalisé en utilisant le container *multimaps*. Pour chaque triangle (a_1, a_2, a_3) on ajoute trois éléments au multimap où les arêtes $\{a_i, a_j\}$, $i, j \in \{1, 2, 3\},\ i \neq j$ présentent les clés et le numéro de triangle est le valeur appliqué. Par conséquent, l'initialisation se produit dans $O(3n_T)$ et si deux triangles (t_1, t_2) sont adjacents par l'arête $\{a, b\}$, les éléments $(\{a, b\}, t_1)$, $\{a, b\}, t_2)$ ont les mêmes clés. Les triangles suivants sont trouvés par la procédure suivante :

1. On parcourt sur l'entière multimap en sauvegardant l'indice de triangle $t_1 = (a_1, a_2, a_3)$ associé à l'arête $\{a_i, a_j\}$ récente et écrasant l'élément $(\{a_i, a_j\}, t)$ de la map. La complexité est $O(3n_T)$.
2. On cherche si le clé $\{a_i, a_j\}$ est associé à un autre triangle t_2. Dans ce cas les triangles t_1, t_2 sont adjacents. L'opération de recherche se passe selon la recherche dans un multimap dans $O(logn_T)$.
3. On met t_2 comme voisin $k \in \{1, 2, 3\}$ si le sommet de t_1 au position k n'est pas un sommet de t_2 et vice versa. Cet opération est réalisé dans un temps constant.

setAdjacencyViaList

As in setAdjacencyViaMultimap, this method sets the members, describing the neighbors, for each triangle in the considered mesh. In the same manner as before, the members are described using their position in the triangle array.
The idea is to store each triangle, in varying order, three times in a list. For the triangle t with the points (a_1, a_2, a_3), the triangles $(a_1, a_2, t), (a_1, a_3, t)$ and (a_2, a_3, t) are appended to the list. The creation of this list has complexity $O(3n_T)$.
After the creation, the list is sorted in lexicographic order. The complexity for sorting a list with n_T elements is $O(n_T logn_T)$.
Due to the lexicographical ordering, the list allows now to define adjacencies between triangles. Two consecutive triangles of the list are considered. The first two entries of the triangles are compared. If they are equal, the triangles share an edge and can be defined as adjacent. $(a, b, t_1), (a, b, t_2)$
complexity $O(3n_T)$
In summary, the method setAdjacencyViaList has complexity $O(n_T logn_T)$.

3.4 L'algorithme promenade

3.5 La visualisation avec gnuplot