BatTirocinio

Ripasso C++

- Sintassi generale
- Analogie con C e Java
- Progetti Qt

Cinolib

Strutturazione delle classi riguardanti le mesh, la geometria e IO.

Gerarchia classi mesh:

```
    * abstract mesh
      * abstract polygonmesh
        * polygonmesh
        * quadmesh
        * trimesh
      * abstract polyhedralmesh
        * hexmesh
        * polyhedralmesh
        * tetmesh
```

Attributi principali delle mesh:

- vertici -> salvati come vec3d
- poligoni -> salvati come vettori di uint, dove gli uint rappresentano l'indice dei vertice
- (edge) -> salvati in un vettore di uint, calcolati implicitamente dai vertici

Trasformazioni geometriche

Le trasformazione geometriche sono state approcciate inizialmente modificando “manualmente” vertici e poligoni, nonostante la mesh abbiano già i metodi implementati per queste trasformazioni, con lo scopo di capire in che modo sono salvati i dati in cinolib. La risoluzione di questi problemi è stata implementata su mesh di superficie, ma il procedimento è analogo per le mesh volumetriche.

Traslazione

> A traslazione è un movimento rigido della mesh nello spazio tridimensionale, in modo tale che ogni retta passante tra due vertici qualsiasi mantenga la stessa direzione prima e dopo il movimento.

Matematicamente rappresentiamo la traslazione o come vettore di 3 elementi o come ultima colonna della matrice di trasformazione 4x4, a livello di codice lo rappresentiamo come vec3d.

```
Matrice 4x4       Vettore     Risultante
o o o x          x       vx+x
o o o y          y       vy+y
o o o z          z       vz+z
o o o o
```

Possibili approcci per effettuare la traslazione:

Soluzione già implementata: cinolib prevede già un metodo pr le mesh di traslazione a cui basta passare un vec3d rappresentante la traslazione .

```
void translate(const vec3d &delta);
```

Approccio ad alto livello: è stato implementato l’overload dell’operatore + tra vec3d, quindi basta fare un ciclo in cui si somma ad ogni vertice il vettore di traslazione.

```
for(auto &v : mesh.vector_verts())
  v += traslazione;
```

Approccio a basso livello: i vec3d (vertici) hanno un metodo set che permette di impostare i valori delle coordinate, si può quindi utilizzare questo approccio:

```
for(auto &v : mesh.vector_verts())
  v.set(v.x()+traslazione.x(), v.y()+traslazione.y(),
v.z()+traslazione.z());
```

Si sarebbe potuto anche usare un approccio ancora più basso andando a modificare uno per uno gli attributi della classe, ma questi sono privati, quindi questo approcci non è attuabile.

Scalatura

> La scalatura è una trasformazione che prevede l'aumento della distanza proporzionale tra tutti i vertici della mesh lungo i tre assi.
> Se le proporzioni vengono mantenute, quindi il fattore di scalatura è uguale su tutti e tre gli assi, si parla di scalatura uniforme (o isotropica), mentre se abbiamo un fattore di scalatura diverso tra i diversi assi parliamo di scalatura non uniforme (o anisotropica).

Matematicamente possiamo rappresentare anch'essa come vettore di 3 elementi, il quale verrà poi moltiplicato per il vettore delle coordinate del punto da scalare, oppure come diagonale della matrice di trasformazione 3x3 o 4x4.

```
Matrice 4x4     Matrice 3x3       Vettore     Risultante
x o o o       x o o         x       vx·x
o y o o       o y o         y       vy·y
o o z o       o o z         z       vz·z
o o o o
```

Approcci per la scalatura:
Soluzione già implementata: cinolib prevede già un metodo per le mesh di scalatura isotropica a cui basta dare un fattore di scala.

```
void  scale(const double scale_factor);
```

Approccio generale: si può implementare una scalatura grazie ad una matrice di trasformazione 3x3 e alla funzione tranform da applicare ai vec3d (vertici), in questo modo possiamo effettuare sia una scalatura isotropica che anisotropica, mentre il metodo scale della cinolib prevede solo scalature isotropiche.

```
double scale[3][3] = {{s_x,0,0},
                      {0,s_y,0},
                      {0,0,s_z}};
for(auto &v : mesh.vector_verts())
  transform(v,scale);
```

Rotazione

> La rotazione è una trasformazione del piano o dello spazio euclideo che sposta gli oggetti in modo rigido e che lascia fisso un punto (nel caso di un piano) o una retta (nel caso dello spazio). Il punto o retta che rimangono fissi nella trasfomazione prendono il nome rispettivamente di centro e asse della rotazione.
>
> Gli elementi della rotazione sono:
>
> - il verso
> - l'ampiezza di rotazione
> - il centro di rotazione

Matematicamente la rappresentazione delle rotazioni può avvenirte in vari modi: matrici 3x3, angoli di elusero, asse+angolo, quaternioni.

La matrice di trasfomazine varia in base all'asse in cui vogliamo ruotare:

$$Asse\ x \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \quad Asse\ y \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \quad Asse\ z \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Approcci per la rotazione:
Soluzione già implementata: cinolib prevede una funzione di rotazione che utilizza il metodo asse + angolo, in cui l'asse è rappresentato da un vec3d, mentre l'angolo da un double.

```
void  rotate(const vec3d & axis, const double angle);
```

Approccio con le matrici: si può implementare la rotazione in una maniera un po' più "rustica" grazie all'ausilio delle matrici 3x3 e l'ausilio della funzione tranform da applicare ai vec3d (vertici). NB: il risultato cambia in base all'ordine in cui si applicano le matrici di trasfomazine, non si tratta di un operazione commutativa

```
//rotazione asse x
double angle = 0;
double rotate[3][3] = { {1,         0,           0 },
                        {0, cos(angle), -sin(angle)},
                        {0, sin(angle),  cos(angle)}};

transform(v,rotate);
```

```
//rotazione asse y
double angle = 0;
double rotate[3][3] = { { cos(angle),  0, sin(angle)},
                        {          0,  1,         0 },
                        { -sin(angle), 0, cos(angle)}};

transform(v,rotate);
```

```
//rotazione asse z
double angle = 0;
double rotate[3][3] = { {cos(angle),    -sin(angle),    0},
                        {sin(angle),    cos(angle),     0},
                        {0,             0,              1}};

transform(v,rotate);
```

Fusione mesh

L'obiettivo è fondere due mesh in una unica, non tenendo conto della possible compenetrazine di queste due.
I problemi da risolvere sono:

- aggiunta dei vertici alla mesh
- aggiunta dei poligoni alla mesh (in particolare i poligoni della seconda mesh)
- aggiunta degli edge (in particolare della seconda mesh, ma in realtà è fatto in automatico nel momento dell'aggiunta dei poligoni)

Come primo passo o si usa una mesh vuota di partenza, in cui si copia la prima mesh, oppure si utilizza la prima mesh direttamente come base.
Una volta preparata la "base di partenza" inizia la parte critica, cioè quella di unione.

Per ottenere il nostro risutlato andremo ad utilizzare il metodi `.vert_add(vec3d vert)` e `.poly_add(std::vector<uint> poly)` che prendono rispettivamente un vertice e un vettore di indici di vertice.

Il primo problema si presenta dopo l'aggiunta dei vertici della seconda mesh, quando dobbiamo aggiungere i poligoni, per cui dobbiamo ricavare l'indice: ci salviamo in numero di vertici della prima mesh (offset) e lo aggiungiamo all'indice di ogni vertice in ogni poligono.
Dopo questa operazione possiamo aggiungere i poligoni.

```
//calcolo offset
uint offset = mesh1.num_verts();

//aggiunta vertici
for(vec3d &v: mesh2.vector_verts()){
    mesh1.vert_add(v);
}

//aggiunta poligoni
for(vector<uint> &v: mesh2.vector_polys()){
  //aggiunta offset indici mesh2
  for(uint &x: v) {
        x += offset;
  }
  mesh1.poly_add(v);
}
```

Questo approccio funziona solo con le mesh di superficie, in quanto nel vettore di vettori restituito da `.vector_polys()` gli uint rappresentano gli indici dei vertici che compongono il poligono, mentre con le mesh volumetriche rappresentano gli indici delle facce che compongono il poliedro

Per questo motivo andiamo a correggere l'algoritmo riportato qui sopra, in modo che funzioni sia con le mesh di superficie che con quelle volumetriche:

```
//calcolo offset e vettore ausiliario
uint offset = m.num_verts();
std::vector<uint> vertPolys;

//aggiunta dei vertici
for(auto &v : m2.vector_verts()){
  m.vert_add(v);
}

//aggiunta dei poligoni
for(uint i=0; i<m2.num_polys(); i++) {
  vertPolys = m2.poly_verts_id(i);

  for(auto &v : vertPolys)
    v += offset;

  m.poly_add(vertPolys);
}
```

Da notare che `.poly_add()` nel caso delle mesh volumetriche prende un vettore di uint, dove questi ultimi rappresentano i vertici del poliedro (come succede con le mesh di superficie), ma il funzionamento è differente, quanto non solo verranno aggiunti gli edge ma anche le facce automaticamente e saranno gli indici di queste ultime a comporre il vettore dei poligoni (vedi anche parte evidenziata sui sopra).

Progetto tesi: operatori compressione e decompressione (ho inventato i nomi)

Nel progetto di tesi si andranno a sviluppare due operatori, uno inverso all'altro.

Il primo operatore permette, data una hexmesh, di selezionare un livello, cioè tutti gli esaedri "complanari" (il termine è un po' improprio ma rende l'idea) ed eliminarli, dopo averli eliminati si devono ricongiungere le due parti di mesh rimaste sconnesse a causa dell'eliminazione del livelli di esaedri.

L'operatore opposto invece permette sempre di selezionare un livello, ma, anziché togliere il "piano" di esaedri, si va ad aggiungere un "foglio" di esaedri e quindi si va a "pompare" una hexmesh aggiungendo un livello.