Oggetti e funzioni utili

L'applicazione è interamente basata sulla libreria VCG, una libreria grafica di cui non fidarsi (lo dico per il vostro bene).
Non ci si aspetta che impariate ad usare tutta la libreria, ne si vuole che perdiate troppo tempo a farlo, per cui farò una lista delle cose essenziali che potrebbero servirvi, partendo dalla loro inclusione e arrivando al loro utilizzo.

Per prima cosa vediamo alcuni degli oggetti principali che sicuramente andrete a maneggiare.

Punti

Partiamo dai punti. Un punto è esattamente quello che vi aspettate: un insieme di 3 valori (x, y, z) dove questi valori indicano un punto o un vettore nello spazio cartesiano.
Un punto è un oggetto che viene dichiarato così:

CGPoint <nome punto>;

Il tipo CGPoint è identico al tipo **vcg::Point3<double>** (o **vcg::Point3d**). Tutte le funzioni vcg che richiedono come parametri punti o normali accettano tranquillamente il tipo CGPoint, per non abbiate paura.

Se per esempio abbiamo un oggetto **P** di tipo CGPoint, allora:

P[i]	vi restituisce l'i-esimo valore del punto, con i che va da 0 a 2. **I valori sono double.**
P.X()	vi restituisce la x del punto. Equivale a P[0].
P.Y()	vi restituisce la y del punto. Equivale a P[1].
P.Z()	vi restituisce la z del punto. Equivale a P[2].
P.Norm()	vi restituisce la norma di P.
P.Normalize();	normalizza P.

Sono inoltre definite alcune operazioni su questi oggetti. Se **A** e **B** sono oggetti di tipo CGPoint, mentre **k** è uno scalare, allora:

P = A + B;	P è il vettore somma di A e B. P = (Ax +Bx, Ay + By, Az + Bz). Lo stesso effetto è ottenuto facendo P[0] = A[0] + B[0]; P[1] = ...
P = -P;	Inverte P. P = (-Px, -Py, -Pz).
P = A * k;	P è il risultato del prodotto scalare tra A e k. P = (Ax * k, Ay * k, Az * k).
P = A ^ B;	P è il risultato del wedge product di A e B. P è il vettore ortogonale ad A e B.
A == B	vi restituisce true se i due punti sono uguali. Sono definiti anche gli altri operatori booleani, ma non ho avuto la sfortuna di testarli.

Ci sono poi funzioni che lavorano con i punti. Tra queste, quelle di cui potreste aver bisogno sono:

vcg::Distance(CGPoint a, CGPoint b)	che restituisce la distanza tra 2 punti (double)
vcg::Angle(CGPoint a, CGPoint b)	che restituisce l'angolo tra 2 vettori (double)

Se poi avete bisogno di ruotare un vettore vi servirà definire la matrice di rotazione. Per fare questo dovete (se non presente tra gli oggetti del namespace vcg) includere la classe delle matrici 3x3:

#include <vcg/math/matrix33.h>

Ora potete dichiarare un oggetto **vcg::Matrix33<double>** (o **vcg::Matrix33d**). Questo oggetto è come se fosse un array 3x3, per cui potete riferirvi ad un elemento con usando gli indici:

```
…
vcg::Matrix33d rot;
…
rot[0][0] = …          //elemento della riga 0 e della colonna 0
rot[1][2] = …          //elemento della riga 1 e della colonna 2
…
```

Per una rotazione usate:

rot.SetRotateDeg(<*valore*> *, asse* **)**
oppure
rot.SetRotateRad(<*valore*> *, asse* **)**

a seconda che vogliate una rotazione in gradi o radianti, attorno al **generico** vettore *asse*. Una volta settata la matrice potete usare

P = rot * A;

che assegnerà a P il risultato della rotazione di A.

Vertex

Il vertice non è altro che una evoluzione del tipo CGPoint. Viene definito come tipo **CGVertex** ed è composto da alcune caratteristiche aggiuntive rispetto al CGPoint. Innanzi tutto un vertice ha sia una posizione nello spazio che una normale, poi un colore e tutta una serie di flag più o meno utili. Viene dichiarato così:

CGVertex <nome vertice>;

Se abbiamo un vertice **V** di tipo CGVertex:

V.P()	vi restituisce un CGPoint, che indica la sua posizione nello spazio
V.N()	vi restituisce un CGPoint, che indica la sua normale
V.C()	vi restituisce il colore come oggetto vcg::Color4b(). È un vettore contenente I valori RGBA del colore.
V.SetS()	setta il vertice come selezionato.
V.ClearS()	setta il vertice come non selezionato.
V.IsS()	restituisce true se il vertice è selezionato.
V.SetV()	setta il vertice come visitato.
V.ClearV()	setta il vertice come non visitato.
V.IsV()	restituisce true se il vertice è visitato.

Questi dovrebbero i metodi salienti dei vertici. Le operazioni sono molto banali:

V.P() = A;	assegna A come posizione al vertice V
V.N() = A;	assegna A come normale al vertice V
V.C() = C;	assegna il colore C al vertice V (non credo che lo userete).

A queste cose ne vanno aggiunte altre che però metto nella sezione delle Mesh.

Face

Le facce non sono altro che un insieme di 3 vertici, di una normale, di un colore e di vari flag. Una faccia è definita come tipo CGFace. Se **F** è la nostra CGFace, allora:

F.P(i)	vi restituisce le coordinate dell'i-esimo vertice (un CGPoint), con i che va da 0 a 2.
F.V(i)	vi restituisce l'i-esimo vertice (un CGVertex), con i che va da 0 a 2.
F.N()	vi restituisce la normale della faccia (un CGPoint).
F.C()	vi restituisce il colore della faccia (come per i Vertex).
F.FFp(i)	vi restituisce un puntatore all'i-esima faccia adiacente a F (la faccia ha un edge in comune con F).
F.SetS()	setta la faccia come selezionata.
F.ClearS()	setta la faccia come non selezionata.
F.IsS()	restituisce true se la faccia è selezionata.
F.SetV()	setta la faccia come visitata.
F.ClearV()	setta la faccia come non visitata.
F.IsV()	restituisce true se la faccia è visitata.

Ci sono poi come per i vertici le operazioni di assegnazione diretta di P(i), N() e C().
Di particolare interesse può essere la funzione **vcg::Barycenter(<faccia>)** che restituisce il baricentro di una faccia (un CGPoint).

Mesh

Come sapete (o dovreste sapere) una mesh non è altro che un insieme di vertici, edge e triangoli. In VCG non esiste il concetto di edge nel senso stretto del termine (non esiste un oggetto di tipo Edge), per cui le mesh sono formate solo da liste di triangoli e vertici.
Una mesh verrà definita come:

CGMesh <nome mesh>;

dove <nome mesh> lo decidete voi. Nel caso del convex hull ho già impostato il nome a **conv_hull**, ma se non vi piace cambiate pure.

Gli attributi essenziali che dovete conoscere di questo tipo di oggetto sono pochi.
Se per esempio abbiamo un oggetto **M** di tipo CGMesh, allora:

M.vn	vi restituisce il numero dei vertici presenti nella mesh. È un valore intero.
M.vert[i]	vi restituisce l'i-esimo vertice. Il vertice è un oggetto di tipo CGVertex.
M.fn	vi restituisce il numero delle facce presenti nella mesh. È un valore intero.
M.face[i]	vi restituisce l'i-esima faccia. La faccia è un oggetto di tipo CGFace.
M.bbox	vi restituisce il bounding box della mesh sotto forma di oggetto vcg::Box3.

Diciamo che le caratteristiche salienti della mesh si riassumono qua.

I vertici e le facce nella mesh, dovrebbero essere organizzati su dei vettori (ossia std::vector), per cui M.vert.begin() restituisce l'iteratore alla prima posizione del vettore, mentre M.vert.end() alla fine. Le stesse cose valgono per le facce

ATTENZIONE: il metodo end() punta alla posizione immediatamente fuori dal vettore. Un vettore di lunghezza N ha gli elementi che vanno da 0 a N-1. end() punta alla posizione N.

Attraverso la classe CGMesh potete poi dichiarare alcuni tipi di dato, tra cui:

CGMesh::VertexPointer	puntatore a vertice
CGMesh::FacePointer	puntatore a faccia
CGMesh::VertexIterator	iteratore per il vettore dei vertici (o per un vettore di vertici)
CGMesh::FaceIterator	iteratore per il vettore delle facce (o per un vettore di facce)

Resta da spiegare come aggiungere vertici e facce a una mesh. Per fare queste due operazioni si usano:

vcg::tri::Allocator<CGMesh>::AddVertices(<*mesh*>, <*numero di vertici da inserire*>);
vcg::tri::Allocator<CGMesh>::AddFaces(<*mesh*>, <*numero di facce da inserire*>);

Queste due funzioni prealloccano sulla mesh <*mesh*> il numero scelto di vertici o facce. Per esempio:

```
vcg::tri::Allocator<CGMesh>::AddVertices(conv_hull, 3);
vcg::tri::Allocator<CGMesh>::AddFaces(conv_hull, 1);
```

allocca lo spazio per 3 vertici e una faccia nella mesh conv_hull.

Dopo questa operazione preliminare si passa al popolamento vero e proprio, andando effettivamente a creare gli oggetti.
Si vanno a creare 3 puntatori (seguendo l'esempio) a vertice

```
CGMesh::VertexPointer ivp[3];
```

Si crea un iteratore a vertice e lo si inizializza alla posizione iniziale del vettore dei vertici della mesh

```
CGMesh::VertexIterator vi = conv_hull.vert.begin();
```

Per ognuno dei 3 vertici si procede con

```
for(int i = 0; i < 3; i++)
{
   ivp[i] = &*vi;              //Assegna al riferimento ivp[i] il vertice vi della lista (&*vi =
                               indirizzo del valore puntato da vi)
   (*vi).P() = CGPoint(x, y, z);   //Posiziona il vertice nel punto (x,y,z)
   (*vi).N() = CGPoint(a, b, c);   //Assegna al vertice la normale (a,b,c);
   ++vi;                       //Incrementa la posizione nella lista dei vertici
}
```

Fatto questo i nostri vertici sono stati creati e piazzati nella mesh. È sufficiente definire la posizione del punto, tralasciando la sua normale.
Successivamente andiamo a creare la faccia, partendo con la dichiarazione di un iteratore di faccia

```
CGMesh::FaceIterator fi = d.face.begin();
```

per poi assagnare alla faccia i suoi 3 vertici, e la sua (eventuale) normale

```
(*fi).V(0) = ivp[0];        //Assegna il primo vertice
(*fi).V(1) = ivp[1];        //Assegna il secondo vertice
```

(*fi).V(2) = ivp[2]; //Assegna il terzo vertice
(*fi).N() = CGPoint(a,b,c); //Assegna alla faccia la normale (a,b,c)

Questa era la serie di operazioni necessarie all'aggiunta di un triangolo ad una mesh vuota. È consigliabile che calcoliate il convex hull prima e poi andiate ad aggiungere le facce tutte in una volta.

Bounding Box

Un bounding box viene definito in VCG come un oggetto di tipo Box3 (vcg::Box3) composto semplicemente dal suo punto di minimo e dal suo punto di massimo. L'inclusione da fare per poter utilizzare I box (nel caso non sia già possibile utilizzarli) è la seguente:

#include <vcg/space/box3.h>

Ora potete istanziare oggetti di tipo Box3 ed utilizzare le loro funzioni. Se **b** è l'oggetto di tipo Box3 le sue funzioni principali sono le seguenti:

b.min	è il CGPoint del minimo del bounding box. Modificando questo punto si modifica il box.
b.max	è il CGPoint del massimo del bounding box. Modificando questo punto si modifica il box.
b.Center()	restituisce il CGPoint del centro del bounding box.
b.Diag()	restituisce la lunghezza della diagonale del bounding box. Il tipo del valore di ritorno è dipendente dal tipo del bounding box.
b.DimX()	restituisce la lunghezza in x del boundign box.
b.DimY()	come sopra, ma per la y
b.DimZ()	come sopra, ma per la z
b.setNull()	setta il box come un box nullo. Un box nullo è un punto.
b.Add(CGPoint)	modifica il bounding box in accordo con il punto passato come parametro. Se il punto è all'interno del bounding box non succede nulla. Se è esterno allora verrano modificati di conseguenza il min e il max del bounding box.
b.Add(Box3)	modifica il bounding box in accordo con il box passato come parametro. Se il box è all'interno del bounding box non succede nulla. Se è esterno allora verrano modificati di conseguenza il min e il max del bounding box.
b.IsIn(CGPoint)	restituisce un booleano. True se il punto è interno al bounding box, False altrimenti. Gli estremi del box sono compresi.
b.IsInEx(CGPoint)	restituisce un booleano. True se il punto è interno al bounding box, False altrimenti. Il max del box non è compreso.
b.Collide(Box3)	resituisce True se i due box si intersecano.
b.P(i)	restituisce l'i-esimo vertice del box. I vertici sono di tipo CGPoint e i va da 0 a 7.
b.Volume()	restituisce il volume del box.

Raggi

Una classe interessante è quella relativa hai raggi. I raggi non sono altro che una coppia (CGPoint p, CGPoint n) che descrive la retta che parte da p e viaggia in direzione n. Equivale alla rappresentazione parametrica della retta dove il parametro t è maggiore o uguale a zero.

Intersezioni

La gestione delle intersezioni è gestitibile attraverso la classe Intersection che offre metodi di calcolo delle intersezioni tra oggetti di natura diversa offerti dalla libreria VCG. Tutte le funzioni di calcolo delle intersezioni restituiscono valori booleani. Una parte di queste però possiede dei parametri di output. La include da aggiungere al codice è:

#include <vcg/space/intersection3.h>

Le intersezioni sono semplici metodi, non oggetti. Non c'è bisogno di istanziare nessun tipo di oggetto “Intersezione”. I metodi messi a disposizione sono questi: