Esercitazione 2 - scena con modelli 3D

Punto 1
I file (con estensione *.m*) forniti contengono l'insieme di vertici e facce necessari a disegnare i modelli tridimensionali (o mesh).

- vertici: *VERTEX vertexId coordX coordY coordZ*
- facce: *FACE faceId vertexId1 vertexId2 vertextId3*

Il caricamento dei modelli tridimensionali avviene nel metodo *init()* il quale:
- legge il file contenente il modello
- memorizza i vertici e le facce necessari per la visualizzazione del modello
- calcola le normali alle facce e ai vertici
- carica in GPU (modalità display list) i vertici, le facce e le normali.

Vettore normale ad una superficie
Una normale ad una superficie è un vettore di norma unitaria ortogonale a quella superficie.

Il modello tridimensionale è costituito da un insieme facce triangolari. Per calcolarne la normale è necessario:
- calcolare v1 come differenza tra i punti P e O (P - O)
- calcolare v2 come differenza tra i punti Q e O (Q - O)
- calcolare v3 attraverso il prodotto vettoriale tra v1 e v2 (v1 x v2)
- normalizzare v3

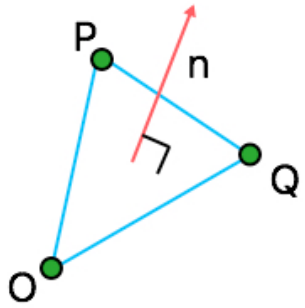

Figura 1 - Normale ad un superficie

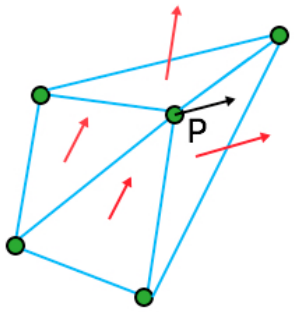

Figura 2 - Normale ad un vertice

Vettore normale ad un vertice
Per calcolare la normale ad un vertice è necessario calcolare le normali alle facce ad esso adiacenti, sommarle tra loro e infine normalizzare il vettore ottenuto.

Modalità display list
Modalità che permette la memorizzazione in GPU di un insieme di primitive grafiche per poterle invocare più rapidamente in futuro.

```
nameList = glGenLists(1);
glNewList(nameList, GL_COMPILE|GL_COMPILE_AND_EXECUTE);

// Primitive grafiche...

glEndList();
```

La prima istruzione assegna un identificativo numerico alla display list, *glNewList()* e *glEndList()* delimitano la definizione della display list.
OSSERVAZIONE: GL_COMPILE si utilizza quando si vuole memorizzare la display list ma non visualizzarla immediatamente, GL_COMPILE_AND_EXECUTE si utilizza quando si vuole memorizzare la display list e visualizzarla immediatamente.

Per visualizzare in qualsiasi momento la display list è necessario invocare la seguente primitiva:

```
glCallList(nameList);
```

Visualizzazione di superfici quadriche

La libreria GLUT fornisce una serie di superfici quadriche predefinite.

```
glutWireIcosahedron();
glutWireTeapot(1.0);
glutWireTorus(0.5, 1.0, 50, 100);
```

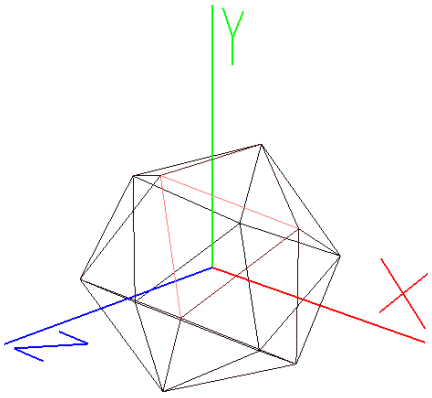

Figura 3 - Icosaedro

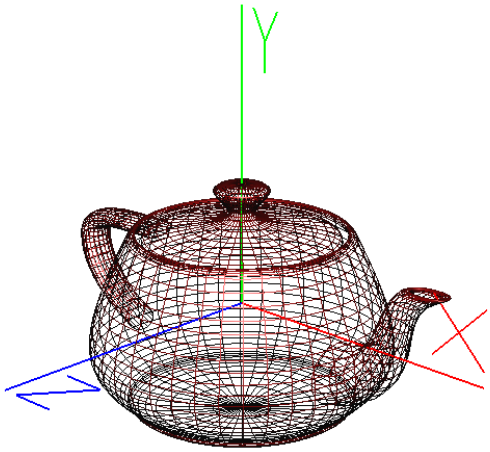

Figura 4 - Tazza di tè

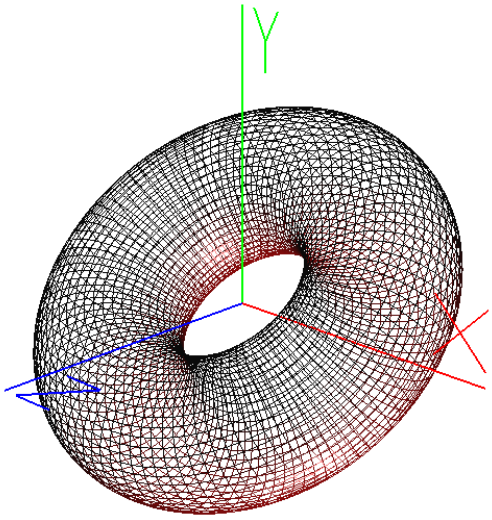

Figura 5 - Toro

Punto 2

Funzionalità dell'applicazione:

- *change eye point* (da implementare). Consente di modificare il punto di vista della camera.
- *change reference point* (già implementato). Consente di modificare il punto di fuoco della camera.
- *change up vector* (già implementato). Consente di modificare il vettore up della camera.
- *change light position* (già implementato). Consente di modificare la posizione della luce.
- *rotate model* (da implementare). Consente di ruotare la mesh rispetto al WCS (non utilizzato nel punto 3 dell'esercitazione).
- *zoom in/out* (da implementare). Consente di simulare un avvicinamento/allontanamento della visuale della camera. Si può realizzare modificando il parametro *field of view*; diminuendo il valore di quest'ultimo si effettua un'operazione di *zoom in*, aumentandolo si effettua un'operazione di *zoom out*.

- *projection* (da implementare). Consente di cambiare il tipo di prospettiva tra prospettica e ortogonale.

  A livello implementativo:

```
if (orthogonal == false)
    gluPerspective(fieldOfView, aspect, 1, 100);
    // aspect = WindowWidth / WindowHeight
else if (orthogonal == true)
    glOrtho(-2.0, 2.0, -2.0, 2.0, -100, 100);
```

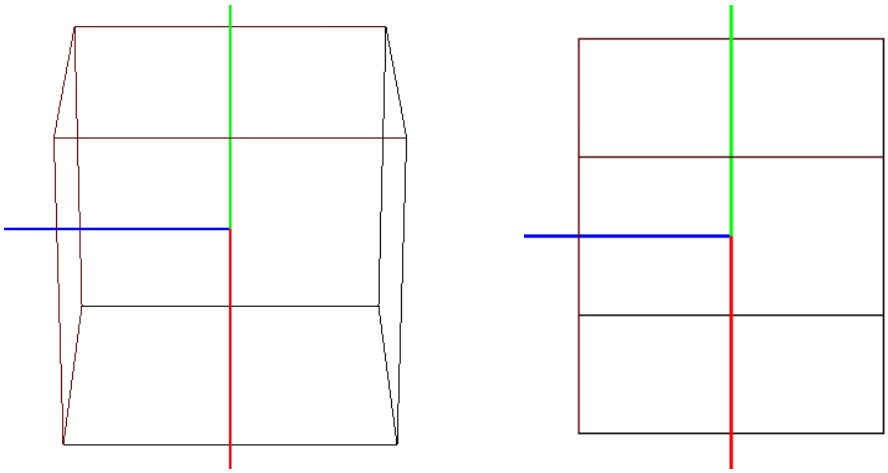

Figura 6 - A sinistra un cubo in prospettiva prospettica, a destra un cubo in prospettiva ortogonale

- *culling* (da implementare). È una tecnica che determina se uno dei poligoni dai quali è composta la mesh è visibile dall'angolo di visuale dell'inquadratura scelta. Se la normale al poligono in questione punta verso una direzione che si allontana dalla telecamera significa che quel determinato poligono non è visibile dall'inquadratura, e non necessita di essere raffigurato. Più precisamente se l'angolo formato dalla direzione della camera con la normale è minore di 90°, il poligono è visibile dall'inquadrature, altrimenti no.

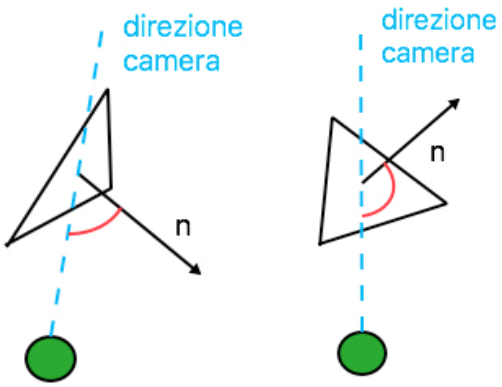

Figura 7 - A sinistra angolo < 90° e quindi visibile, a destra angolo > 90° e quindi non visibile

A livello implementativo:

```
if (culling == true) {
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);
} else if (culling == false)
    glDisable(GL_CULL_FACE);
```

OSSERVAZIONE: questo processo rende il rendering delle immagini più veloce ed efficiente, in quanto elimina tutte le parti non visibili e che quindi non devono essere elaborate.

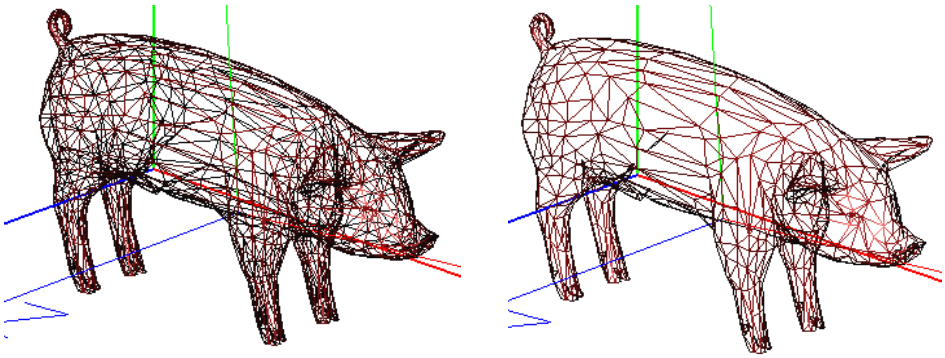

Figura 8 - A sinistra mesh senza culling, a destra mesh con culling

- *wireframe* (da implementare). Se l'opzione è abilitata vengono disegnati solamente i vertici che costituiscono la mesh (non la parte interna delle facce che costituiscono la mesh).
  OSSERVAZIONE: questo metodo richiede calcoli molto più semplici rispetto alla rappresentazione di superfici solide, ed è quindi considerevolmente più veloce.

  A livello implementativo:

```
if (wireframe == true)
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
else if (wireframe == false)
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

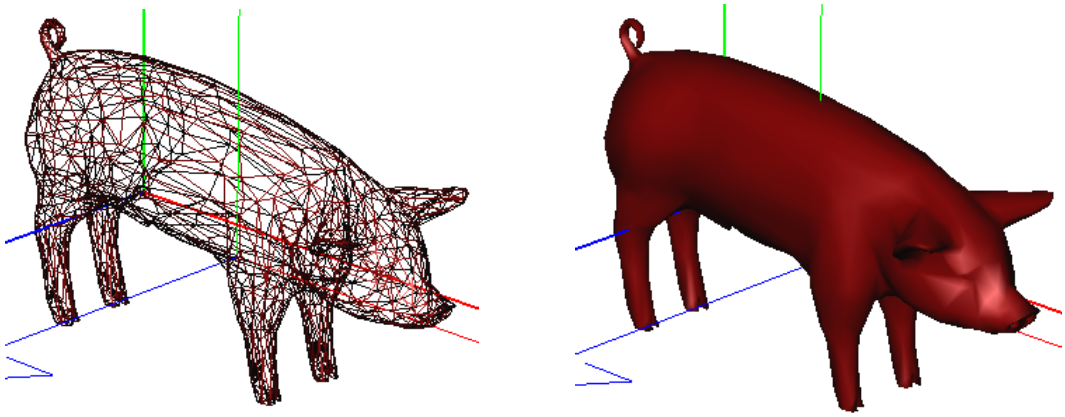

Figura 9 - A sinistra mesh con wireframe abilitato, a destra mesh senza wireframe

- *shading* (da implementare). Consente di modificare il tipo di ombreggiatura visualizzata sulle mesh. Per calcolare l'ombreggiatura si utilizzano le normali ai vertici calcolate nel punto 1 dell'esercitazione (anche in questo caso si utilizza l'angolo tra la direzione della camera e la normale - minore è l'angolo più il colore tende al bianco, maggiore è l'angolo più il colore tende al nero). Sono disponibili due modalità di ombreggiatura:
  - flat: calcola il colore basandosi sull'angolo tra direzione della camera e la normale ad un vertice e lo spalma su tutto triangolo.
  - smooth: calcola il colore di ogni vertice basandosi sull'angolo tra direzione della camera e la normale a ciascun vertice e interpola i risultati ottenuti per colorare il resto del triangolo.

A livello implementativo:

```
if (shading == true)
   glShadeModel(GL_SMOOTH);
else if (shading == false)
   glShadeModel(GL_FLAT);
```

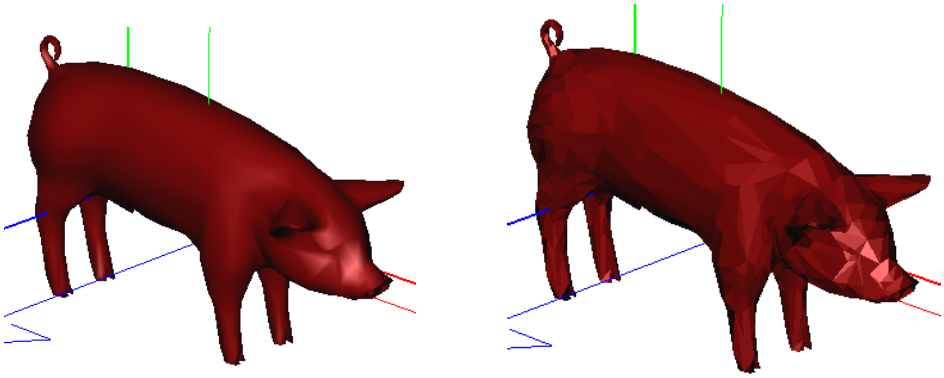

Figura 10 - A sinistra illuminazione di tipo smooth, a destra illuminazione di tipo flat

Sono necessarie infine due ulteriori istruzioni per attivare l'ombreggiatura sulle mesh:

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

- *materials* (già implementato). Consente di modificare il materiale associato agli oggetti (ottone, plastica rossa, smeraldo, ardesia).
- *track-ball* (parzialmente implementato). Consente di muoversi su una sfera immaginaria che racchiude il punto di fuoco della camera.
- *camera motion* (da implementare - tasto 's' della tastiera). Permette di far muovere la camera lungo un percorso descritto da una curva di Bézier mantenendo il punto di fuoco della camera fisso. N.B.: è necessario che la curva di Bézier sia un percorso chiuso.
- *print system status* (già implementato). Stampa a video i valori attuali delle variabili dell'applicazione (ad esempio se il culling è abilitato oppure no).
- *reset* (già implementato). Riporta l'applicazione allo stato iniziale.
- *quit* (già implementato). Consente di chiudere l'applicazione.

Punto 3

Per poter procedere con la risoluzione del punto 3 dell'esercitazione è necessario aver chiaro il funzionamento dello stack delle matrici di OpenGL.

Stack: tipo di dato astratto al quale si può accedere secondo la modalità LIFO (Last In First Out). In particolare:

- push aggiunge un nuovo elemento in cima allo stack
- pop rimuove l'elemento in cima allo stack

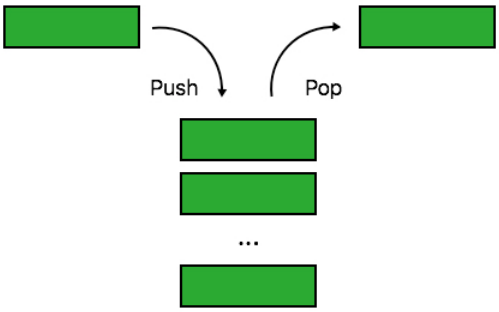

Figura 11 - Stack

Istruzioni fondamentali per la manipolazione dello stack in OpenGL:

- glPushMatrix() duplica la matrice attuale (elemento top dello stack) e la aggiunge allo stack
- glPopMatrix() rimuove la matrice attuale (elemento top dello stack) dallo stack
- glLoadIdentity() sostituisce la matrice attuale (elemento top dello stack) con la matrice identità
- glLoadMatrixf(m) sostituisce la matrice attuale (elemento top dello stack) con la matrice *m*
- glMultMatrixf(m) sostituisce la matrice attuale con la moltiplicazione tra la matrice attuale e la matrice *m*.
- glTranslatef(x, y, z) sostituisce la matrice attuale con la moltiplicazione tra la matrice attuale e la matrice di traslazione definita dai parametri *x*, *y* e *z*.
  Matrice di traslazione:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- glRotatef(angle, xAxis, yAxis, zAxis) sostituisce la matrice attuale con la moltiplicazione tra la matrice attuale e la matrice di rotazione definita dai parametri *angle*, *x*, *y* e *z*.
  Matrice di rotazione (c = cos(angle), s = sin(angle), ||(x, y, z)|| = 1):

$$\begin{pmatrix} x^2(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ yx(1-c)+zs & y^2(1-c)+c & yz(1-c)-xs & 0 \\ xz(1-c)-ys & yz(1-c)+xs & z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

  È inoltre possibile applicare una rotazione su un solo asse, in particolare:
  - per applicare una rotazione rispetto all'asse x => x = 1, y = 0, z = 0
  - per applicare una rotazione rispetto all'asse y => x = 0, y = 1, z = 0
  - per applicare una rotazione rispetto all'asse z => x = 0, y = 0, z = 1

N.B.: un vertice prima di essere disegnato viene moltiplicato per la matrice attuale sullo stack (vertexToDraw = M_{stack} • vertex). I nuovi punti ottenuti sono quelli effettivamente utilizzati per disegnare il vertice sulla finestra.

<u>Ordine di applicazioni delle trasformazioni</u>
Per applicare ad un oggetto le trasformazioni di traslazione e scala in questo ordine è necessario calcolare la matrice di trasformazione complessiva come $T = T_{SCALA} \bullet T_{TRASLAZIONE}$ e successivamente moltiplicare la matrice T ottenuta per i vertici dell'oggetto da trasformare ($vertexToDraw = T \bullet vertex$).
N.B.: la moltiplicazione tra matrici non gode della proprietà commutativa, pertanto l'applicazione delle trasformazioni in ordine inverso causa una trasformazione di scala e traslazione che produce un risultato diverso.

In quest'ottica per applicare le trasformazioni WCS e OCS agli oggetti sarà necessario applicare la seguente catena di trasformazioni:

$$T_{WCSn} \bullet \dots \bullet T_{WCS2} \bullet T_{WCS1} \bullet initialPositionObj_i \bullet T_{OCSn} \bullet \dots \bullet T_{OCS2} \bullet T_{OCS1} \bullet vertexObj_i$$

$$T_{WCS} \bullet initialPositionObj_i \bullet T_{OCS} \bullet vertexObj_i$$

dove:

- T_{WCS} sono le trasformazioni (di rotazione e traslazione) applicate dall'utente rispetto al WCS

- $initialPositionObj_i$ è la posizione iniziale del sistema di riferimento della i-esima mesh rispetto al WCS (sostanzialmente l'OCS dell'i-esima mesh)
- T_{OCS} sono le trasformazioni (di rotazione e traslazione) applicate dall'utente rispetto all'OCS
- $vertexObj_i$ sono i vertici della i-esima mesh da disegnare

A livello implementativo:

```
for (int i = 0; i < N_MESH; i++) {
        glPushMatrix();

        glMultMatrixf(matrixWCS[i]); // Twcs
        glMultMatrixf(matrix[i]); // initialPositionObj
        drawAxis(1.0, 1); // Assi OCS dell'i-esima mesh
        glMultMatrixf(matrixOCS[i]); // Tocs

        glCallList(nameList[i]); // vertexObj

        glPopMatrix();
}
```