ALMA MATER STUDIORUM - UNIVERSITÁ DI BOLOGNA

CAMPUS DI CESENA

SCUOLA DI SCIENZE

CORSO DI LAUREA IN SCIENZE E TECNOLOGIE INFORMATICHE

Progettazione e realizzazione di un editor grafico per il rendering di scene tridimensionali mediante OpenGL

Relazione finale in

Metodi numerici per la grafica

Relatore

Prof.ssa Damiana Lazzaro

Presentata da

Lorenzo Bandini

II Sessione

Anno Accademico 2013-2014

Indice

Introduzione 3

Capitolo 1 - La Pipeline di Rendering 5

1.1 Architettura 5

1.2 Stage di Applicazione 7

1.3 Stage di Geometria 8

1.3.1 Model & View Transform 8

1.3.1.1 Model Transform 9

Trasformazioni geometriche 9

Trasformazione di traslazione 10

Trasformazione di rotazione 11

Coordinate omogenee 11

Trasformazioni 3D 14

1.3.1.2 View Transform 16

1.3.2 Vertex Shading 16

1.3.2.1 Modelli di Illuminazione 17

1.3.2.2 Shading 24

Constant Shading 24

Gouraud Shading 26

Phong Shading 27

1.3.3 Proiezione 27

1.3.4 Clipping 28

1.3.5 Screen Mapping 30

1.4 Stage di Rasterizzazione 31

1.4.1 Triangle Setup & Traversal 32

1.4.2 Pixel Shading 33

1.4.3 Merging 34

Capitolo 2 - OpenGL 37

2.1 Caratteristiche di OpenGL 37

2.2 GL, GLU, GLUT e paradigma ad eventi 39

2.3 GLUI 42

Capitolo 3 - Realizzazione dell'editor grafico 43

3.1 Panoramica generale 43

3.2 Interfaccia 44

3.3 Strutture dati utilizzate 45

3.4 Oggetti 49

3.4.1 Inserire oggetti nella scena 49

3.4.2 Primitive 3D 52

3.4.3 Oggetti estratti da file in formato Wavefront OBJ 54

3.4.3.1 Il formato Wavefront OBJ 54

3.4.3.2 Estrazione di dati da file OBJ 57

3.4.4 Superfici di rivoluzione 61

3.5 Trasformazioni su scena e oggetti 69

3.5.1 Picking 73

3.6 Eventi mouse e tastiera 76

3.7 Properties 79

3.8 Luci 81

Capitolo 4 - Esempi di applicazione 87

Bibliografia e Sitografia 91

Introduzione

Al giorno d'oggi l'utilizzo della grafica 3D è una costante nella vita quotidiana: l'industria pubblicitaria, cinematografica, automobilistica e videoludica (solo per citarne alcune) basano la maggior parte dei loro prodotti su scene e modelli creati artificialmente al computer, e questi prodotti ci circondano continuamente per le strade, in televisione o sui giornali. Avere a disposizione un buon editor grafico è quindi fondamentale alle aziende di questi settori per rendere i loro prodotti appetibili sul mercato, mostrando al pubblico animazioni sempre più innovative ed oggetti o personaggi sempre più realistici.

L'obiettivo di questo elaborato è quindi quello di progettare e realizzare un editor grafico per la modellazione di oggetti tridimensionali, di consentire ad un utente di costruire scene complesse curandone il rendering.

Il progetto è stato realizzato facendo uso del linguaggio C++ e degli strumenti messi a disposizione dalle librerie OpenGL.

La tesi è così organizzata:

- In primo luogo si introdurranno nel primo capitolo i vari stage della pipeline di rendering, e si entrerà nel dettaglio di ognuno di essi per comprendere i vari passi che portano dalla modellazione di un oggetto alla sua visualizzazione sullo schermo.
- In seguito, nel secondo capitolo ci si soffermerà sulla definizione delle librerie OpenGL e degli strumenti predisposti da quest'ultima per gestire ed influenzare il rendering di una scena tridimensionale.
- Infine il terzo capitolo è dedicato alla presentazione dell'editor grafico: saranno descritte in dettaglio tutte le sue componenti, le tecnologie utilizzate per svilupparlo, gli algoritmi utilizzati per gestire gli oggetti nella scena e la renderizzazione di quest'ultima sullo schermo. Verranno inoltre proposte alcune scene create con questo editor grafico, per fornirne un esempio di utilizzo.

Capitolo 1
La Pipeline di Rendering

In questo capitolo si discuteranno i differenti stati della pipeline di rendering. La funzione principale della pipeline è quella di generare, o meglio *renderizzare*, un'immagine bidimensionale data una telecamera virtuale, oggetti tridimensionali, sorgenti luminose, equazioni di shading, texture, ecc... La posizione e la forma degli oggetti nell'immagine sono determinati dalla loro geometria, dalle caratteristiche della scena e dal posizionamento della camera nella scena. L'aspetto degli oggetti è influenzato dalle proprietà dei materiali, dalle sorgenti luminose, textures e modelli di shading, [1],[2],[3],[4],[8].

1.1 Architettura

Come detto, la pipeline di rendering consiste in diversi stati. Il rendering real time è infatti diviso in *tre stage concettuali:*

- *applicazione;*
- *geometria;*
- *rasterizzazione*.

Questa struttura è il cuore della pipeline ed è condivisa da più o meno tutti i sistemi di rendering in real time. Ognuna di queste fasi può essere a sua volta una pipeline ed essere suddivisa in sotto-sezioni.

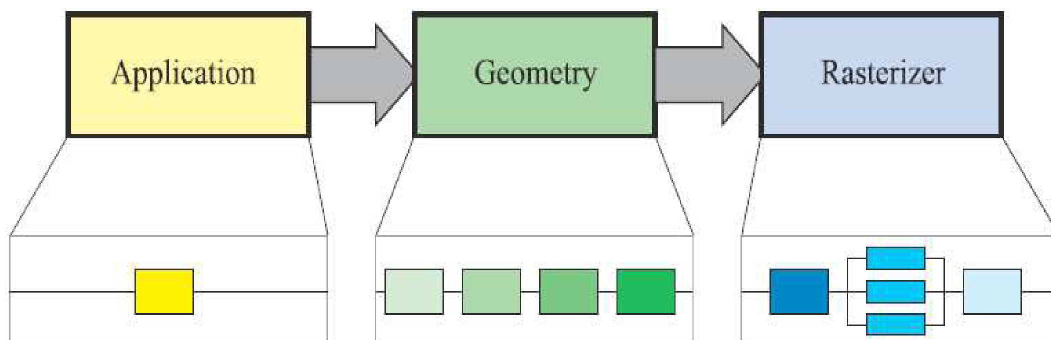

fig. 1.1 Schema della pipeline di rendering

Differenziamo ora fra stage concettuali (applicazione, geometria e rasterizzazione), stage funzionali e stage di pipeline.
Uno stage funzionale deve eseguire un task ben preciso ma non specifica in che modo eseguirlo all'interno della pipeline. Uno stage di pipeline, d'altro canto, viene eseguito simultaneamente con tutte le altre fasi di pipeline e può essere parallelizzata per migliorare la performance. Per esempio, è possibile dividere lo stage di geometria in cinque ulteriori fasi funzionali, ma è l'implementazione del sistema grafico che determina la sua divisione in stage di pipeline. Una implementazione potrebbe combinare due stage funzionali in uno di pipeline, mentre potrebbe dividere uno stage funzionale, più complesso a livello di calcolo, in più stage di pipeline, eventualmente parallelizzati.
Lo stage più lento della pipeline determina la *velocità di rendering*, cioè la velocità con cui le immagini sullo schermo vengono aggiornate. Questa velocità è espressa in *frame al secondo* (fps), che è il numero di immagini renderizzate per secondo. È possibile anche valutarla in *Hertz* (Hz), che è la semplice notazione 1/*secondi*, la frequenza di aggiornamento. Il tempo impiegato da un'applicazione per generare immagini molto spesso varia, dipende dalla complessità dei calcoli da eseguire per ogni frame. I frame al secondo sono usati per esprimere, oltre alle immagini generate per secondo, la performance media in un tempo di esecuzione. Dal momento che usiamo la pipeline, non basta sommare i tempi che impiegano i blocchi di dati che si vogliono renderizzare ad essere processati dalla pipeline. Questa è una conseguenza della costruzione della pipeline, che permette di eseguire task in parallelo.
Se potessimo localizzare il collo di bottiglia e misurare quanto tempo impiegano i dati a passare quello stage, allora potremmo calcolare la velocità di rendering. Assumiamo, per esempio, che il collo di bottiglia restituisce i risultati del calcolo in 20 ms (millisecondi); la velocità di rendering sarebbe $1/0.020 = 50$ Hz. Questo calcolo è vero solo se l'output del device ha la stessa frequenza di aggiornamento, altrimenti l'output

risulterà più lento. In contesti di pipeline utilizziamo il termine *throughput* invece della velocità di rendering.
Lo *stage di applicazione* viene guidato come si può intuire dall'applicazione che è sicuramente implementata su CPU generiche. Non è suddiviso in ulteriori pipeline, ma grazie ai diversi core presenti nelle CPU è possibile parallelizzarlo e processare più threads in parallelo per aumentarne prestazioni e efficienza. Alcuni dei task eseguiti dalle CPU sono il collision detection, input da tastiera e mouse, algoritmi di accelerazione globali, animazioni, simulazioni fisiche e molti altri, a seconda del tipo di applicazione processata. Lo *stage* successivo è quello *di geometria*, che esegue trasformazioni, proiezioni, illuminazione, clipping, ecc. su poligoni e vertici passati dalla fase precedente. In questo stage viene calcolato cosa deve essere disegnato sullo schermo, come deve essere disegnato e dove disegnarlo. Questa fase viene tipicamente eseguito in GPU che contengono molti core programmabili o su hardware ad hoc con funzioni fissate. Alla fine, lo *stage di rasterizzazione* renderizza un'immagine usando i dati generati dagli stage precedenti, assegnando ad ogni pixel il colore corretto durante il posizionamento. Lo stage di rasterizzazione si esegue completamente in GPU.
Alla fine di queste fasi, l'immagine può essere visualizzata a schermo.

1.2 Stage di Applicazione

Essendo implementato in CPU, lo sviluppatore ha pieno controllo su cosa avviene nello stage di applicazione, e può determinarne l'implementazione per migliorarne le performance. Le modifiche realizzate in questo stage, possono influenzare la performance delle fasi seguenti. Alla fine dello stage di applicazione, la geometria da renderizzare è data in pasto allo stage di geometria. Il task più importante di questa fase è quindi quello che permette di inviare le *primitive di rendering*, cioè, punti, linee e poligoni che eventualmente verranno disegnati sullo schermo (o su qualsiasi device di

output) allo stage successivo. Per migliorare le performance, questo stage è di solito eseguito in parallelo su più processori. Il processo più comunemente implementato è il *collision detection*. Dopo che una collisione fra due oggetti è stata trovata, il risultato deve essere generato e comunicato agli oggetti interessati. Lo stage di applicazione si occupa anche di prendere input da device esterni, come tastiere, mouse, giroscopi, ecc. In base a questo input, possono essere eseguite molte azioni diverse.

1.3 Stage di Geometria

fig. 1.2 Schema delle fasi dello stage di geometria

Lo stage di geometria è responsabile della maggior parte delle operazioni sui vertici e sui poligoni. È suddiviso in ulteriori stage: *Model & View Transform*, *Vertex Shading*, *Proiezione*, *Clipping*, e *Screen Mapping*. Notare ancora, che in base all'implementazione, questi stage possono essere o meno equivalenti agli stage di pipeline. In alcuni casi, un numero consecutivo di stage funzionali formano un singolo stage di pipeline (che concorrono in parallelo con gli altri stage di pipeline) oppure, in altri casi uno stage funzionale viene suddiviso in tanti altri stage di pipeline.

1.3.1 Model & View Transform

Nella fase di *Model & View Transform* vengono applicate trasformazioni geometriche e di vista agli oggetti presenti nella scena. In questa sezione quindi parleremo del passaggio dal sistema di riferimento dell'oggetto alla descrizione di quest'ultimo in coordinate di camera, soffermandoci sul concetto di trasformazione geometrica e di trasformazione di vista.

1.3.1.1 Model Transform

In questa fase dello stage di geometria vengono applicate le trasformazioni ai vertici e alle normali della geometria passata dallo stage di applicazione. L'oggetto viene quindi messo nella scena con la desiderata posizione, scalatura e orientamento. Ogni oggetto, a partire dal proprio sistema di riferimento (*object space*), viene trasformato opportunamente in un sistema di riferimento comune (*world space*) per andare a far parte della scena finale.

Trasformazioni geometriche

Le trasformazioni geometriche sono lo strumento che consente di manipolare punti e vettori all'interno del mondo dell'applicazione grafica. Queste trasformazioni sono funzioni che mappano un punto (vettore) in un altro punto (vettore).
La trasformazione di una primitiva geometrica si riduce alla trasformazione dei punti caratteristici (vertici) che la identificano nel rispetto della connettività originale. Questo grazie al fatto che trattiamo di trasformazioni affini.

Le trasformazioni geometriche affini sono trasformazioni lineari

$$f(aP + bQ) = \mathrm{a}f(P) + bf(Q)$$

che preservano:

- *collinearità* : punti di una linea giacciono ancora su di una linea dopo la trasformazione;
- *rapporto tra le distanze* : punto medio di un segmento rimane il punto medio di un segmento anche dopo la trasformazione.

Le trasformazioni geometriche permettono di traslare, ruotare, scalare o deformare oggetti che siano stati modellati nel loro spazio di coordinate del modello permettendo di istanziarli con attributi (posizione, orientamento, fattori di scala) diversi nello spazio di coordinate del mondo (*permettono il passaggio dal sistema di coordinate locali al sistema di coordinate del mondo*) .

Le trasformazioni geometriche di base sono:

- Traslazione;
- Scalatura;
- Rotazione.

Trasformazione di traslazione

Traslare una primitiva geometrica nel piano significa muovere ogni suo punto P(x,y) di d_x unità lungo l'asse X e di d_y unità lungo quello delle Y fino a raggiungere la nuova posizione P(x',y') dove:

$$x' = x + d_x \ , \ y' = y + d_y$$

In notazione matriciale:

$$P = \begin{bmatrix} x \\ y \end{bmatrix}; \qquad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}; \qquad T = \begin{bmatrix} d_x \\ d_y \end{bmatrix};$$

$$P' = P + T$$

con T vettore traslazione.

Trasformazione di scalatura

Scelto un punto C (punto fisso) di riferimento, scalare una primitiva geometrica significa riposizionare rispetto a C tutti i suoi punti in accordo ai fattori di scala s_x lungo l'asse X e s_y lungo l'asse Y scelti.
Se il punto fisso è l'origine O degli assi, la trasformazione di P in P' si ottiene con

$$x' = s_x \cdot x$$
$$y' = s_y \cdot y$$

in notazione matriciale ciò si esprime come

$$P' = S \cdot P$$

dove

$$P = \begin{bmatrix} x \\ y \end{bmatrix}; \qquad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}; \qquad S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix};$$

S premoltiplica P, in quanto quest'ultimo è definito come vettore colonna.

Trasformazione di rotazione

Fissato un punto C (*pivot*) di riferimento ed un verso di rotazione, ruotare una primitiva geometrica attorno a C significa muovere tutti i suoi punti nel verso assegnato in maniera che si conservi, per ognuno di essi, la distanza da C; una rotazione di Θ attorno all'origine O degli assi è definita come

$$x' = x \cdot \cos\Theta - y \cdot \sin\Theta$$
$$y' = x \cdot \sin\Theta + y \cdot \cos\Theta$$

in notazione matriciale si avrà

$$P' = R \cdot P$$

dove

$$P = \begin{bmatrix} x \\ y \end{bmatrix}; \qquad P' = \begin{bmatrix} x' \\ y' \end{bmatrix};$$

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix};$$

Coordinate omogenee

Le trasformazioni geometriche possono essere applicate in sequenza (la nostra applicazione ne è un esempio); nel caso di presenza di somme di vettori, a causa di traslazioni, e di moltiplicazioni, scalature e rotazioni, si ha una concatenazione disomogenea di trasformazioni.
Per risolvere tale problematica si sono introdotte le coordinate omogenee, per mezzo delle quali un punto P(x,y) risulta espresso come P(x_h,y_h,w) dove

$$x = x_h / w \,;\; y = y_h / w \text{ ; con } w \neq 0;$$

Due punti di coordinate (x,y,w) e (x',y',w') rappresentano lo stesso punto del piano se e solo se le coordinate dell'uno sono multiple delle rispettive coordinate dell'altro; almeno uno dei valori x, y, o w deve essere diverso da 0. Quando $w = 1$ (forma canonica) coordinate omogenee e cartesiane vengono a coincidere. Con ($x,y,w \neq 0$) si indicano punti, con ($x,y,0$) si rappresentano vettori.
Possiamo quindi riscrivere nella notazione in coordinate omogenee le trasformazioni di base sopracitate:

- *Trasformazione di Traslazione*

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- *Trasformazione di Scalatura*

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- *Trasformazione di Rotazione*

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

La rappresentazione in coordinate omogenee permette una concatenazione omogenea di trasformazioni. L'ordine di concatenazione è molto importante perché le trasformazioni geometriche, pur essendo associative, non sono commutative. Come esempio, volendo applicare, nell'ordine, le trasformazioni T1, T2, T3 e T4 occorre calcolare la trasformazione composizione T come

$$T = T_4 \cdot T_3 \cdot T_2 \cdot T_1$$

Trasformazioni 3D

Se tutte le trasformazioni nel piano possono essere rappresentate per mezzo delle coordinate omogenee da matrici 3×3, le trasformazioni nello spazio possono essere rappresentate da matrici 4×4. Nello spazio un punto in coordinate omogenee è rappresentato da un quadrupla

$$(x, y, z, w)$$

Le trasformazioni geometriche viste in precedenza nello spazio diventano:

- *Traslazione* la matrice di traslazione in uno spazio tridimensionale è una semplice estensione della matrice 2D

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- *Scalatura* la matrice di scalatura in uno spazio tridimensionale è una semplice estensione della matrice 2D

$$T(d_x, d_y, d_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- *Rotazione* più complicato è il caso della rotazione attorno ad un asse qualsiasi nello spazio tridimensionale. Ci viene in aiuto la proprietà che ogni rotazione 3D si può ottenere come composizione di rotazioni attorno ai tre assi principali

- *Rotazione attorno asse X*

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & \\ 0 & sin\theta & cos\theta & \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- *Rotazione attorno asse Y*

$$R = \begin{bmatrix} cos\theta & 0 & -sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- *Rotazione attorno asse Z*

$$R = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Per ricondursi al caso generale è sufficiente operare nel seguente modo:

1. Traslare l'oggetto in modo tale che l'asse di rotazione passi per l'origine *O* degli assi;
2. Ruotare l'oggetto in modo tale che l'asse di rotazione venga a coincidere con uno degli assi principali;
3. Ruotare l'oggetto nel verso e della quantità angolare richiesta;
4. Applicare la rotazione inversa al passo 2;
5. Applicare la traslazione inversa al passo 1.

1.3.1.2 View Transform

In questa fase vengono applicate le *trasformazioni di vista*. Definire la trasformazione di vista vuol dire definire il volume che contiene gli oggetti visibili (*volume di vista*). Il volume è un poliedro che si può definire individuando un rettangolo che ne è la sezione e le quattro rette che passano attraverso i vertici del rettangolo.

La trasformazione di vista, dopo aver definito la camera virtuale e il suo volume di vista, la posiziona nell'origine, orientandola lungo l'asse z. Dopo l'applicazione di queste trasformazioni, si dice che la geometria è descritta in *coordinate di camera* (*camera coordinates*).

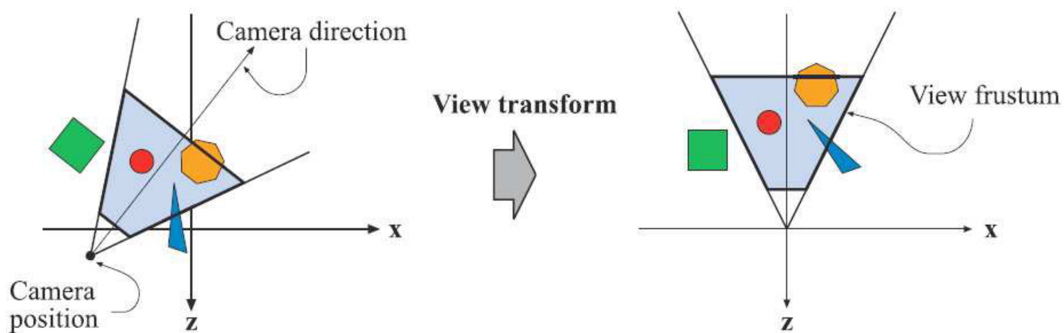

fig. 1.3 Applicazione della trasformazione di vista

Questa operazione ha lo scopo di facilitare le operazioni di proiezione e clipping.

1.3.2 Vertex Shading

In questa sezione, attraverso i *modelli di shading*, vengono applicati *modelli di illuminazione* che cercano di simulare l'interazione tra luce e materiali dei modelli della scena. In grafica real-time vengono utilizzati normalmente metodi *locali,* che calcolano il colore di un vertice considerando il materiale dell'oggetto, la posizione del vertice, la sua normale e la posizione della luce.

I *modelli di illuminazione* quindi descrivono i fattori che determinano il colore di una superficie in un determinato punto, tramite le interazioni tra le luci e le superfici e tenendo conto delle proprietà delle superfici e della natura della radiazione luminosa incidente. L'uso di un modello di illuminazione è necessario per ottenere una rappresentazione realistica delle superfici tridimensionali.
I *modelli di shading* invece determinano come viene applicato il modello di illuminazione e quali argomenti prende in input per determinare il colore di un punto sulla superficie. Il risultato dello shading del vertice viene poi mandato allo stage di rasterizzazione per essere interpolato.

1.3.2.1 Modelli di Illuminazione

Ci sono tre differenti classi di modelli di illuminazione: *luce ambientale, riflessione diffusa* e *riflessione speculare*. Questi tre modelli messi assieme danno luogo al modello di illuminazione di Phong.
In termini matematici un modello di illuminazione può essere espresso mediante una *equazione di illuminazione*, che descrive come ogni punto dell'oggetto sia illuminato in funzione della sua posizione nello spazio.
Le equazioni di illuminazione esposte in questo capitolo riescono solo a simulare il comportamento di materiali opachi e non di materiali trasparenti o semitrasparenti.
Il modello di illuminazione più semplice, ma anche il meno realistico, è quello in cui ogni oggetto è dotato di una propria intensità luminosa, senza che vi siano fonti esterne di illuminazione. Il risultato è quello di avere un mondo di sagome monocromatiche (a meno che i singoli poligoni di un poliedro non abbiano diversi colori).
Questo modello elementare può essere descritto dall'equazione

$$I = k_i$$

dove I è l'intensità risultante e k_i è la luminosità intrinseca dell'oggetto. Non essendoci termini dipendenti dalla posizione del punto si può calcolare I una sola volta per tutto l'oggetto.

Nella realtà inoltre, alcuni degli oggetti che non sono illuminati direttamente dalla luce non appaiono completamente neri. Queste parti sono illuminate dalla *illuminazione globale*, cioè dalla luce riflessa dall'ambiente circostante. Questa luce viene approssimata da una luce costante detta *luce ambientale.*

Questo tipo di componente viene simulata supponendo che l'oggetto sia illuminato da una sorgente di luce diffusa e non direzionale, prodotto del riflesso della luce sulle molteplici superfici presenti nell'ambiente.

Se supponiamo che la luce dell'ambiente colpisce ugualmente tutte le superfici da tutte le direzioni, allora l'equazione dell'illuminazione diventa:

$$I = I_a k_a$$

dove I_a è l'intensità della luce dell'ambiente, supposta costante per tutti gli oggetti. La quantità di luce dell'ambiente riflessa dalla superficie dell'oggetto è determinata da k_a, coefficiente di riflessione ambientale, che varia tra 0 ed 1. Il coefficiente di riflessione ambientale è una proprietà che caratterizza il materiale di cui la superficie è fatta.

fig. 1.4 Cambio di illuminazione incrementando ka

Come si può notare però gli oggetti illuminati da sola luce ambientale sono ancora uniformemente illuminati su tutta la loro superficie.

Supponiamo ora di posizionare nella scena una sorgente luminosa puntiforme (*point light source*) i cui raggi sono emessi uniformemente in tutte le direzioni. In questo caso la luminosità di ogni singolo punto dipenderà dalla sua distanza dalla sorgente luminosa e dalla direzione in cui i raggi incidono rispetto alla superficie. La *riflessione diffusa*, detta anche *riflessione Lambertiana*, è caratteristica dei materiali opachi, tipo il gesso. Queste superfici appaiono ugualmente luminose da qualunque punto di vista vengano osservate: non modificano la loro apparenza al variare del punto di vista poiché riflettono la luce uniformemente in tutte le direzioni. La luminosità dipende solo dall'angolo Θ formato dalla direzione del raggio luminoso (L) e la normale alla superficie nel punto di incidenza (N).

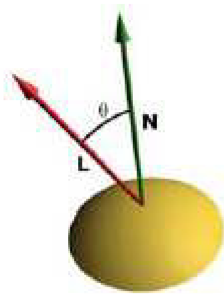

fig.1.5 Raggio luminoso e normale di una superficie curva

Un raggio di luce che intercetta una superficie ricopre un'area la cui grandezza è inversamente proporzionale al coseno dell'angolo Θ che il raggio forma con N.

Sia *d* la superficie, perpendicolare ad N, di un'area differenziale intorno al punto di incidenza, la superficie colpita da un raggio incidente è *d* /cosΘ.

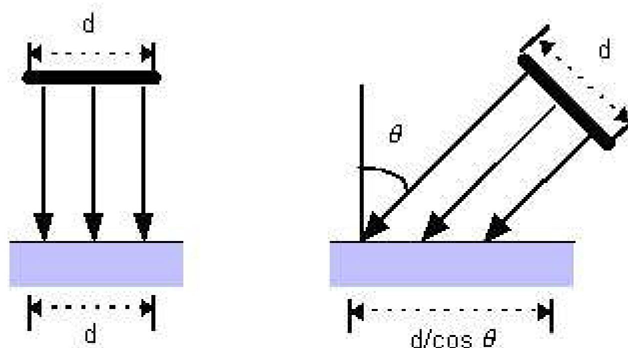

fig. 1.6 schemi differenti di superfici colpite da raggi di incidenza con differente angolatura

La porzione di luce vista dall'osservatore è data dalla legge di Lambert, secondo la quale:

"*la quantità di luce riflessa da una porzione d di superficie verso l'osservatore è direttamente proporzionale al coseno dell'angolo tra N e la direzione di vista*"

Ma poiché la quantità di superficie vista dall'osservatore è inversamente proporzionale al coseno del medesimo angolo, questi due fattori si cancellano e l'intensità luminosa rimane determinata solo in funzione del coseno dell'angolo Θ. Per esempio, all'aumentare dell'angolo di vista, l'osservatore vede un'area maggiore di superficie, ma la quantità di luce riflessa a quell'angolo per unità di area di superficie è proporzionalmente minore.

Così, per le superfici lambertiane la quantità di luce vista dall'osservatore è indipendente dalla posizione dell'osservatore ed è proporzionale al cos Θ, angolo di incidenza della luce. L'equazione dell'illuminazione diffusa è quindi:

$$I = I_p k_d \cos(\Theta)$$

dove I_p è l'intensità della sorgente luminosa puntiforme, k_d è il coefficiente di riflessione diffusiva ed è una costante che varia tra 0 ed 1 e varia da un materiale all'altro e Θ ha un valore compreso tra 0° a 90° (un punto della superficie non è illuminato da sorgenti che stanno dietro di esso).

Inoltre, se i vettori N ed L sono stati normalizzati, l'equazione dell'illuminazione diventa:

$$I = I_p k_d \overline{\mathrm{N}} \cdot \overline{\mathrm{L}}$$

Se una sorgente di luce puntiforme è sufficientemente distante dagli oggetti, essa forma lo stesso angolo con tutte le superfici che condividono la stessa normale. In questo caso la luce è detta *sorgente luminosa direzionale* ed L è costante per la sorgente luminosa. Con la riflessione diffusa, la parte della

superficie non illuminata dalla luce è nera. E' come se l'oggetto fosse illuminato da una torcia. Aggiungendo il termine dell'illuminazione ambientale all'equazione dell'illuminazione della riflessione diffusa abbiamo:

$$I = I_a\, k_a + I_p\, k_d\, (\bar{\mathrm{N}} \cdot \bar{\mathrm{L}})$$

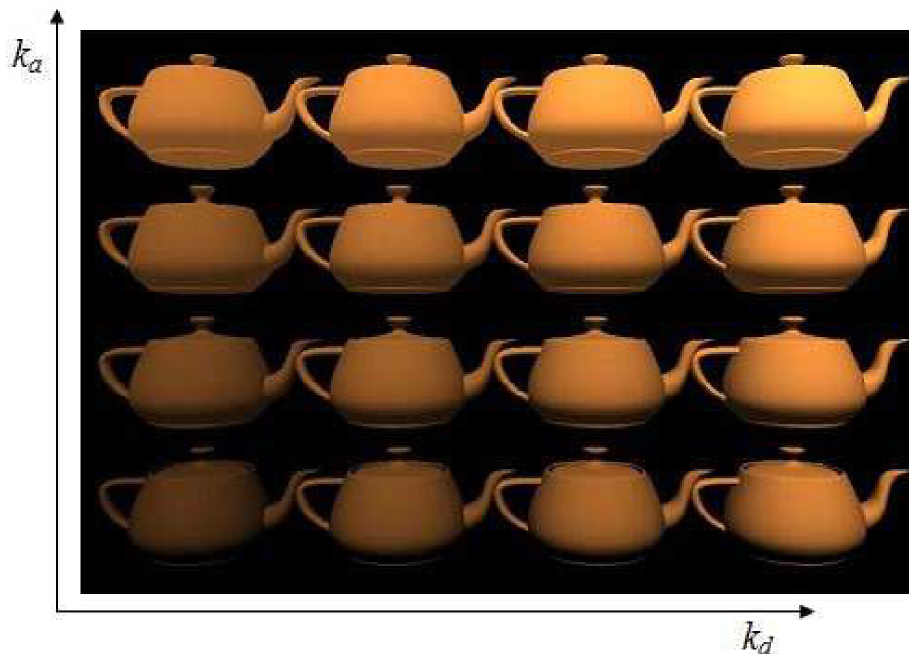

fig. 1.7 Variazione di illuminazione di una teiera all'aumentare di ka e kd

Per tener conto dell'attenuazione dell'intensità dell'illuminazione all'aumentare della distanza, si introduce inoltre un fattore di attenuazione, f_{att}, inversamente proporzionale alla distanza della sorgente di luce dalla superficie:

$$I = I_a\, k_a + f_{att}\, I_p\, k_d\, (\bar{\mathrm{N}} \cdot \bar{\mathrm{L}})$$

Una tipica formulazione del fattore di attenuazione abbastanza empirica è:

$$f_{att} = min\left(\frac{1}{c1 + c2\, d_L + c3\, d_L^2}, 1\right)$$

Dove le costanti *c1*, *c2* e *c3* sono definite dall'utente ed associate alla sorgente luminosa e d_L è la distanza dalla sorgente luminosa.

Se invece la superficie di un oggetto non è completamente opaca, la luce non viene riflessa ugualmente in tutte le direzioni. Data una superficie totalmente lucida, come uno specchio, la luce viene riflessa nella direzione di riflessione R che, geometricamente, non è altro che L (direzione di incidenza della luce) riflessa rispetto ad N (normale alla superficie)

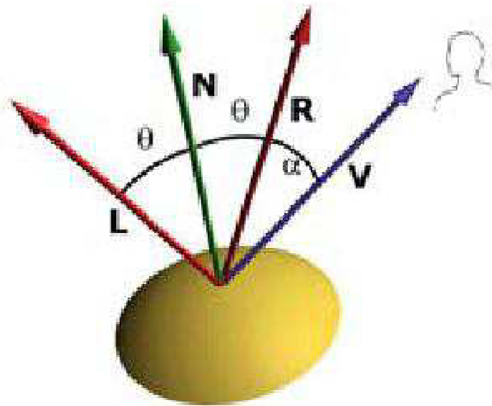

fig. 1.8 riflessione della luce su di una superficie non opaca

L'osservatore può vederla solo se la direzione di vista è allineata con la riflessione, cioè solo se $\alpha = 0$.

Phong ha però anche sviluppato un modello di illuminazione per riflettori non perfetti, come per esempio un oggetto di plastica o di cera. Il modello assume che si abbia riflessione massima per $\alpha = 0$ e che essa decada rapidamente all'aumentare di α. Un tale decadimento viene approssimato da $cos^n\alpha$, dove *n* viene detto *coefficiente di riflessione speculare del materiale*. Il valore di n può variare tra 1 e valori molto alti (anche superiori al 100), secondo il tipo di materiale che si vuole simulare. Una superficie a specchio sarebbe teoricamente rappresentata da $n = \infty$.

All' aumentare di *n* la luce riflessa si concentra in una regione sempre più stretta centrata sull'angolo di riflessione. I valori di *a* compresi nell'intervallo [100, 500] corrispondono approssimativamente alle superfici

metalliche. I valori inferiori a 100 corrispondono ai materiali che mostrano un'ampia zona di massima lucentezza.

L'equazione dell'illuminazione quindi diventa:

$$I = I_a\, k_a + f_{att}\, I_p\, [k_d \cos\Theta + k_s \cos^n \alpha]$$

dove k_s è il coefficiente di riflessione speculare, che varia tra 0 ed 1 e dipende dal particolare materiale.

Se i vettori R e V sono normalizzati, l'equazione si può scrivere come:

$$I = I_a\, k_a + I_p\, k_d\, (\bar{N} \cdot \bar{L}) + I_p\, k_s\, (\bar{R} \cdot \bar{V})^n$$

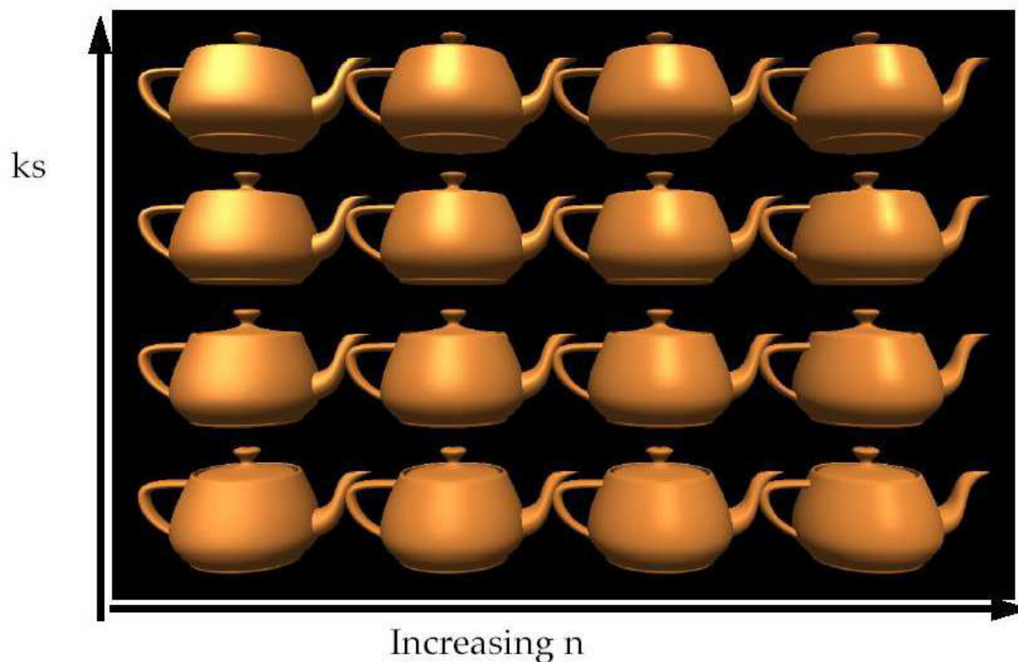

fig. 1.9 Variazione dell'illuminazione al variare di ks e ka

Le caratteristiche fondamentali del modello di Illuminazione di Phong sono: le sorgenti luminose puntiformi, le componenti speculari e diffusiva modellate solo in modo locale e la componente dell'ambiente modellata come costante, senza tener conto delle inter-riflessioni tra gli oggetti della scena della radiazione luminosa.

Il colore degli oggetti viene definito definendo opportunamente i coefficienti di riflessione diffusa e ambientale. Bisogna considerare inoltre

tre equazioni dell'illuminazione, una per ogni componente del modello del colore considerato.
Nel caso nella scena vi sia più di una luce, basta sommare i termini per ogni sorgente luminosa. Così se abbiamo *m* sorgenti luminose l'equazione dell'illuminazione diventa:

$$I = I_a k_a + \sum_{1 \leq j \leq m} I_{pj} [k_d (\bar{N} \cdot \bar{L}_j) + I_p k_s (\bar{R}_j \cdot \bar{V})^n]$$

1.3.2.2 SHADING

Come detto in precedenza, con il termine *shading* si intende il processo volto a determinare il colore di tutti i pixel che ricoprono una superficie usando un modello di illuminazione.
Il metodo più semplice consiste nel determinare per ogni pixel la superficie visibile, calcolare la normale alla superficie in quel punto ed in seguito valutare l'intensità della luce ed il colore usando un modello di illuminazione. Questo modello di shading però è molto costoso in quanto questa operazione è svolta per ogni singolo pixel.
Siccome generalmente gli oggetti complessi sono rappresentati da mesh poligonali, introduciamo i *modelli di shading*, meno dispendiosi del modello descritto in precedenza, in quanto forniscono una tecnica per determinare i colori di tutti i pixel che ricoprono una superficie, usando un appropriato modello di illuminazione, in maniera più efficiente.

Constant Shading

Il modello di shading più semplice per un poligono è il *constant shading,* noto anche come *flat shading*. Questo approccio applica un modello di illuminazione per determinare un singolo valore di intensità dell'equazione

dell'illuminazione una volta per ogni poligono. Questo approccio è valido se sono vere diverse ipotesi:

– La sorgente luminosa è posta all'infinito, così N·L è costante per ogni faccia del poligono.
– L'osservatore è posto all'infinito così L·V è costante per ogni faccia del poligono.
– Il poligono rappresenta la superficie da modellare e non è un'approssimazione di una superficie curva.

Se una delle prime due ipotesi non è verificata, se vogliamo utilizzare lo shading costante è necessario un metodo per determinare un valore singolo per ognuno degli L e V. Per esempio, i valori possono essere calcolati per il centro del poligono, o per il primo vertice del poligono.

Il constant shading è una tecnica di shading molto veloce perché richiede pochi calcoli; in compenso però si ha un risultato visivo poco soddisfacente, in quanto lascia visibile la suddivisione tra i poligoni, senza rendere nell'immagine l'andamento della superficie approssimata geometricamente dalla mesh di poligoni, fornendo una vista grossolana della scena.

fig. 1.10 Due esempi di constant shading

Inoltre se lo shading viene fatto indipendente su ogni poligono (sia esso costante o interpolato) si ha comunque una netta visibilità, non voluta, dei bordi tra due poligoni adiacenti, causati dalla brusca variazione della normale alla superficie. Si

incombe quindi nel cosiddetto effetto *Mach banding;* questo effetto è quello per cui un oggetto messo vicino ad uno più chiaro risulta più scuro e messo vicino ad uno più scuro risulta più chiaro. Per ovviare a questo inconveniente si sono sviluppati dei metodi di shading che tengono conto delle informazioni date da poligoni adiacenti.

Gouraud Shading

Nel *Gouraud shading* si tiene conto della geometria effettiva che si sta visualizzando: se la griglia di poligoni rappresenta una superficie curva, per ogni vertice della griglia non si utilizza la normale al poligono, ma la normale alla superficie. In questo modo il calcolo dello shading produce lo stesso valore su entrambi i lati di poligoni che hanno bordi in comune rendendo lo shading complessivo privo di salti.
Il metodo richiede che sia nota la normale alla superficie che si approssima in ogni vertice. Se non è disponibile, la si approssima con la media delle normali ai poligoni che condividono il vertice.
Dal punto di vista implementativo il Gouraud shading è mediamente efficiente, poiché l'equazione di illuminazione va calcolata una volta sola per vertice. Per poter individuare i vettori normali necessari per poter calcolare la normale nei vertici, occorre una struttura dati che rappresenti l'intera mesh di poligoni.

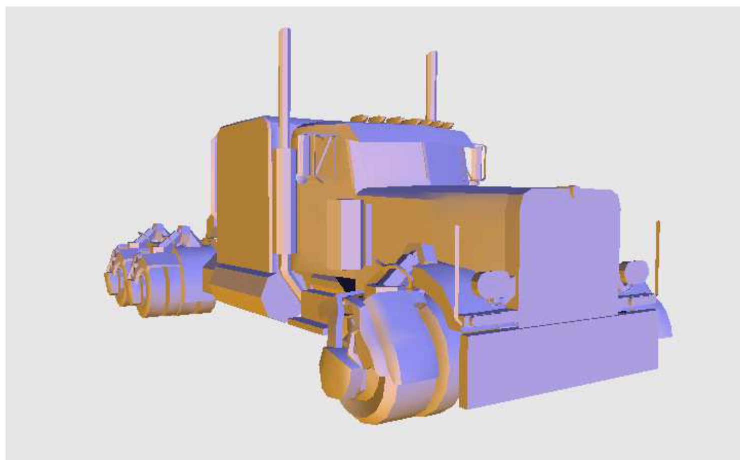

fig. 1.11 Esempio di Gouraud Shading

Il risultato visivo prodotto dal Gouraud shading è perlopiù quello desiderato, in quanto non è visibile la suddivisione in mesh dei poligoni, e si ottiene una rappresentazione liscia e senza discontinuità della superficie approssimata.

Phong Shading

Nel *Phong shading* le normali nei vertici vengono calcolate nella stessa maniera con cui si calcolavano nel Gouraud Shading, ma il calcolo dello shading dei pixel del poligono viene effettuato usando normali calcolate interpolando linearmente (e poi rinormalizzando) all'interno dello spazio delle normali. Il Phong shading risulta particolarmente realistico quando si vogliano rappresentare superfici dotate di un alto coefficiente di riflessione speculare. Si può intuire però che risulti più costoso in termini computazionali rispetto al Gouraud shading, in quanto si interpolano vettori e non valori interi e l'equazione di illuminazione viene calcolata per ogni pixel.

fig. 1.12 Esempio di Phong Shading e differenze con lo Shading di Gouraud

1.3.3 Proiezione

Nello *stage di proiezione* il volume di vista viene trasformato in un volume di vista canonico. Il volume di vista di partenza cambia a seconda del tipo di proiezione utilizzata:

- se la proiezione è *prospettica*, il volume di vista è un tronco di piramide a base rettangolare (*frustum*);
- se la proiezione è *ortografica,* il volume di vista è un parallelepipedo.

Questo volume di vista viene quindi normalizzato, ovvero viene trasformato in un cubo unitario con estremi in (-1,-1,-1) e (1,1,1) , chiamato appunto *volume di vista canonico.*

Disegnando poi le primitive ignorando la Z si ottiene l'immagine da mandare al display. Le informazione sulla profondità vengono però salvate in un buffer apposito, lo *z-buffer*.

Dopo l'applicazione della proiezione, si dice che la geometria è descritta in *coordinate dispositivo normalizzate* (*normalized device coordinates*).

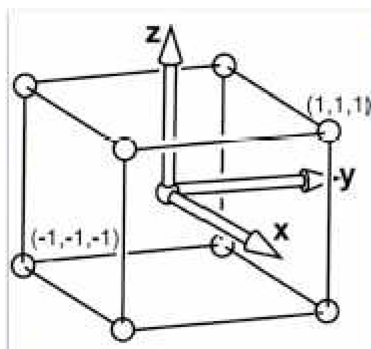

fig. 1.13 Volume di vista canonico

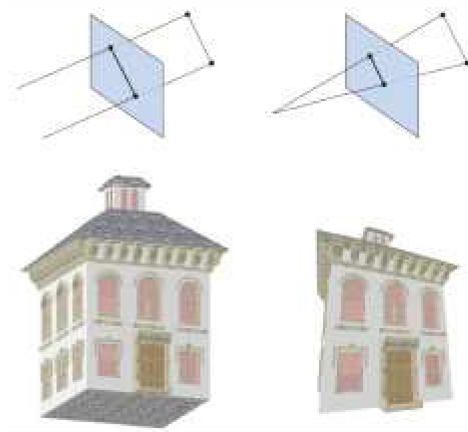

fig. 1.14 Differenze tra proiezione prospettica e ortografica

1.3.4 Clipping

Siccome solo le primitive dentro il volume di vista canonico vengono mandate alla fase di rasterizzazione, quelle che sono parzialmente all'interno del volume di vista devono essere modificate. Si utilizza quindi un algoritmo di *clipping*, che valuta quale porzione di figura è esterna al volume di vista canonico e la elimina.

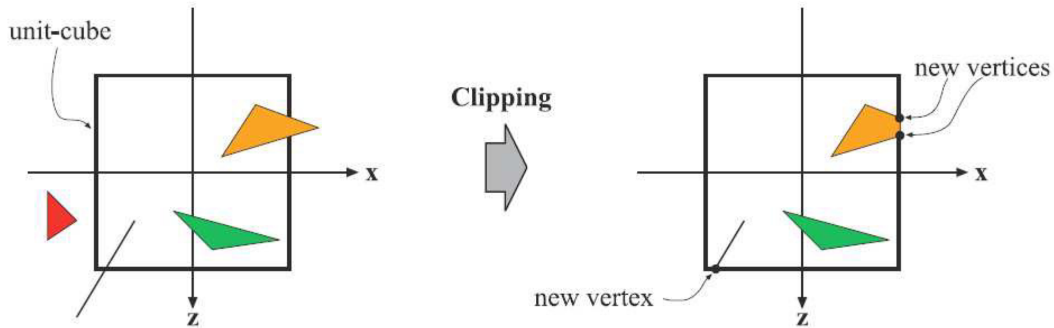

fig. 1.15 Applicazione di un algoritmo di clipping

Uno degli algoritmi di clipping più usati è quello di Cohen-Suthrland. Questo algoritmo ha come idea di base quella che il rettangolo di clipping suddivide il piano di proiezione in 9 regioni. Ad ogni regione viene quindi associato un codice numerico binario di 4 cifre:

- bit 1: sopra edge alto $y > y_{max}$
- bit 2: sotto edge basso $y < y_{min}$
- bit 3: a destra edge destro $x > x_{max}$
- bit 4: a sinistra edge sinistro $x < x_{min}$

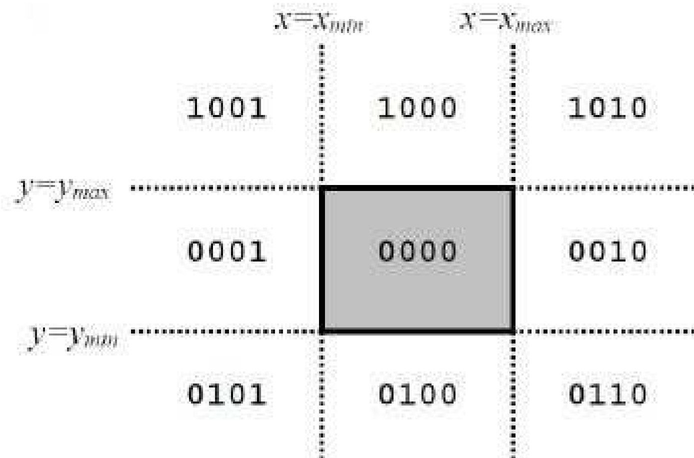

fig. 1.16 Schema dell'algoritmo di Cohen-Suthrland

Il clipping di un segmento quindi prevede la codifica (e confronto) dei suoi estremi sulla base delle regioni di appartenenza.

- Se il codice di entrambi gli estremi è 0000 (OR logico tra gli estremi da risultato nullo) allora si può banalmente decidere che il segmento è interno al rettangolo di clipping;

- Se l'operazione di AND logico tra i codici dei due estremi restituisce risultato non nullo allora il segmento è esterno al rettangolo di clipping:
- Se l'operazione di AND logico è nullo:
    1. Si individua l'intersezione tra il segmento e la retta relativa al primo bit discordante tra i codici;
    2. L'estremo col bit a 1 viene sostituito dal nuovo vertice (l'intersezione trovata al passo 1);
    3. Si itera il procedimento sugli altri bit discordanti. Ad ogni iterazione si controlla l'eventuale terminazione del processo(OR logico nullo).

1.3.5 Screen Mapping

L'ultima fase dello stage di geometria è lo *screen mapping*. Questa fase consiste nel mappare le coordinate tridimensionali (*x,y,z*) dei vertici nel cubo di lato unitario in coordinate bidimensionali (*x',y'*) della finestra sullo schermo. Queste nuove coordinate vengono dette *coordinate schermo* (*screen coordinates*). Le coordinate schermo insieme alle coordinate z (non modificate e salvate nello z-buffer) sono dette *coordinate finestra* (*window coordinates*).

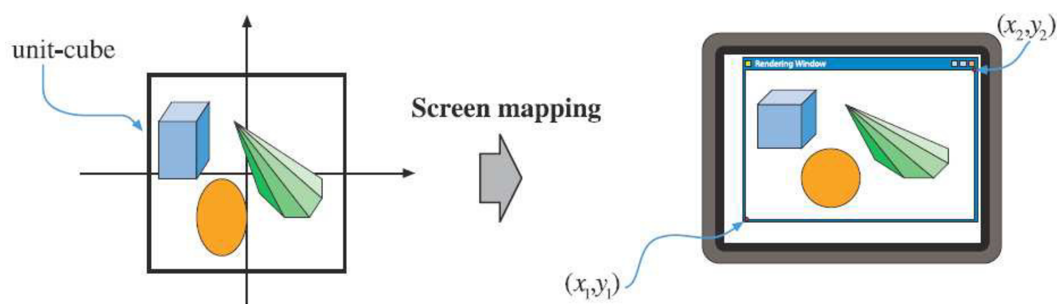

fig. 1.17 Screen Mapping

1.4 Stage di Rasterizzazione

Dati i vertici trasformati e proiettati con associati i dati di shading passati dallo *stage di geometria*, l'obiettivo dello *stage di rasterizzazione* è quello di calcolare e impostare il colore di ogni pixel coperto dall'oggetto. Questo processo si chiama *rasterizzazione* o *scan conversion*, che è la conversione da vertici 2D nello spazio dello schermo, ognuno con un valore su z (valore di profondità) e varie informazioni di shading associate ad ogni vertice, in pixel sullo schermo.
Nella pipeline di rendering, la rasterizzazione e affidata ai *fragment o pixel shader*, utili a determinare se le proprietà di colore di un pixel possono essere propagate ad un intero *frammento* di linea di scansione, in modo tale da decidere se poter applicare gli algoritmi coinvolti in parallelo a pixel appartenenti ad uno stesso frammento. Oltre alle tecniche legate alla determinazione di trama, ombreggiatura e trasparenza, lo stage di rasterizzazione si occupa di :

- *drawing*, ovvero il tracciamento di primitive bidimensionali all'interno di una griglia regolare di pixel che hanno distanza unitaria in ascissa e ordinata. L'algoritmo di drawing scelto deve quindi individuare le coordinate dei pixel che giacciono sulla linea ideale o che sono il più vicino possibile ad essa, in modo tale da approssimare al meglio il segmento.
- *filling*, cioè il riempimento di una figura chiusa con un colore. Ad ogni scan-line, identificati i pixel che appartengono al poligono in esame, l'algoritmo di filling sfrutta le informazioni trovate per aggiornare incrementalmente le intersezioni e fare il filling sulla scan-line successiva.
- *antialiasing*, cioè la riduzione dell'effetto di sotto-campionamento dell'immagine generata dovuta alla risoluzione spaziale fissa data dalla dimensione del pixel. L'algoritmo di antialiasing quindi permette di

ridimensionare l'effetto 'seghettatura' dei profili di un'immagine, colorando i pixel in maniera tale da ingannare l'occhio, facendo sembrare i profili continui e non seghettati.

Similmente allo stage di geometria, questo stage è diviso in vari stage funzionali: *triangle setup*, *triangle traversal*, *pixel shading* e *merging*.

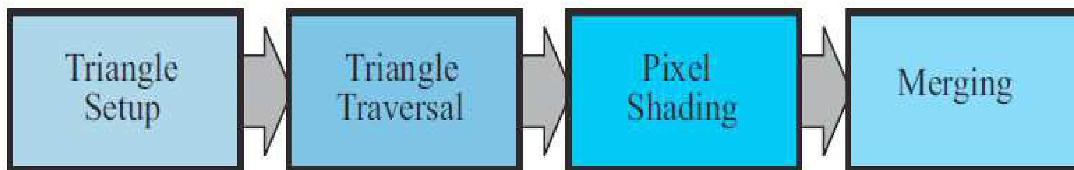

fig. 1.18 Fasi dello stage di Rasterizzazione

1.4.1 Triangle Setup & Traversal

I moderni hardware grafici sono ottimizzati per un tipo di primitiva: i triangoli. I triangoli sono l'entità geometrica che rappresenta al meglio una approssimazione lineare di una qualsiasi superficie che si vuole renderizzare, in quanto sono sempre convessi e risiedono sempre su di un singolo piano. Inoltre l'interpolazione di parametri è facilmente definibile per un triangolo, ed è più veloce e facile in termini computazionali.
Nella fase di *triangle setup* quindi i vertici arrivati dallo stage di geometria vengono collegati tra loro per ottenere dei triangoli, in modo tale però da mantenere la forma degli oggetti da renderizzare. Una volta creati i triangoli, viene applicato il *backface culling*, ovvero l'eliminazione delle superfici nascoste non visibili dalla vista specificata. I triangoli che rimasti dopo il backface culling vengono mandati allo stage successivo, il *triangle traversal*.
Nella fase di triangle traversal vengono identificati i pixel all'interno di ogni triangolo, e per ognuno viene creato un fragment. I pixel cosiddetti interni sono quelli che hanno il proprio centro all'interno del triangolo, e per identificarli vengono usate diverse strategie, come l'utilizzo di *bounding*

box, o l'uso di scansioni lineari per-pixel come il *backtrack traversal* o lo *zigzag traversal.*

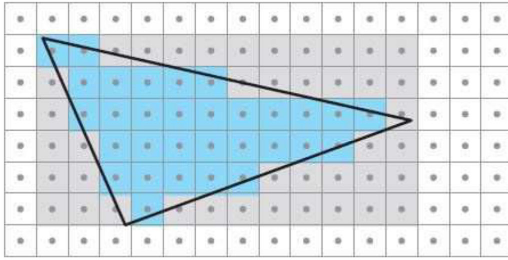

fig. 1.19 Bounding box

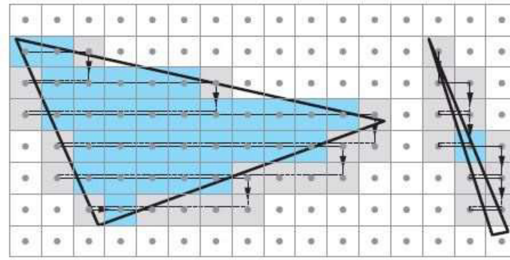

fig. 1.20 Backtrack traversal

Ogni fragment contiene i dati relativi ai vertici che sono stati interpolati su tutta la superficie, come la posizione del pixel sullo schermo, il valore di profondità dello z-buffer, il colore del vertice, le coordinate delle texture e i vettori delle normali in quel punto. Questi fragment vengono poi inviati allo stage di pixel shading, dove ne verrà determinato il colore.

1.4.2 Pixel Shading

Nello stage di *pixel shading* vengono effettuati i calcoli sui fragment per determinarne il colore. In questa fase vengono applicate texture, modificati i colori a seconda della presenza o meno dell'effetto nebbia, di luci, ombre o di trasparenze.

Fa parte dello stage di pixel shading anche il *fragment testing*, una tecnica di ottimizzazione che permette di scartare i fragments non visibili nell'immagine e di evitare un'eccessiva scrittura di dati per il rendering finale. Ogni fragment quindi è testato e confrontato con una serie di parametri, e se il fragment fallisce il test, quest'ultimo viene scartato.

I test che deve passare sono:

- *Alpha Test*, nel quale viene analizzata la componente alpha (trasparenza) del colore del fragment. Se questo valore è inferiore ad una certa soglia, il fragment viene scartato. Questo previene il

caricamento da parte del pixel shader di pixel non visibili in quanto trasparenti.

- *Stencil Test*, dove uno stencil buffer (delle stesse dimensioni del frame buffer) viene utilizzato per mascherare alcune aree del frame buffer. Viene controllato quindi se alla posizione del fragment corrisponde nello stencil buffer un permesso di scrittura. Se è così il fragment può essere disegnato, altrimenti viene scartato. Questo test è utile quando vengono proposte dissolvenze e decalcomanie.

1.4.3 Merging

La responsabilità dello stage di *merge* è quella di combinare il colore dei fragment ottenuto dagli stage precedenti con il colore mantenuto nel color buffer. Il risultato sarà quindi il colore finale del pixel da visualizzare sullo schermo. Questo stage è anche responsabile di risolvere il problema della visibilità. Ciò significa che quando tutta la scena è stata renderizzata, il color buffer deve contenere i colori delle primitive che sono visibili nella scena dal punto di vista della camera. Su molti hardware grafici, il problema si risolve con il *depth testing* (anche noto come *Z-testing*). Il depth testing utilizza un *depth buffer* (*Z-buffer*), della stessa dimensione e forma di un color buffer e per ogni pixel memorizza il valore su z dalla camera alla primitiva corrente più vicina. Significa che quando una primitiva viene renderizzata in un certo pixel, il valore su z di quella primitiva in quel pixel viene confrontato con il contenuto dello Z-buffer nello stesso pixel:

- Se il nuovo valore su z è più piccolo del valore nello Z-buffer, allora la primitiva che stiamo renderizzando è più vicina alla camera della primitiva che era prima più vicina alla camera in quel pixel. Allora, il valore su z e il colore di quel pixel vengono aggiornati con il valore su z e il colore della primitiva che stiamo renderizzando.

- Altrimenti, se il valore su z è più grande del valore nello Z-buffer, il color buffer e lo Z-buffer vengono lasciati inalterati.

L'algoritmo di Z-buffering è molto semplice, ha una complessità computazione convergente a O(n) (dove n è il numero delle primitive da renderizzare) e funziona con ogni primitiva il cui valore di z su ogni vertice rilevante può essere calcolato. Da notare anche che questo algoritmo permette di renderizzare le primitive in qualsiasi ordine, che è un'altro motivo della sua popolarità. Comunque, primitive parzialmente trasparenti non possono essere renderizzate in qualsiasi ordine, ma devono essere renderizzate dopo ogni primitiva completamente opaca e in ordine back-to-front. Questa è la più grande debolezza del depth testing.

Il *frame buffer* generalmente consiste nell'insieme di tutti i buffer di un sistema, ma a volte è usato per intendere solo il color buffer e lo Z-buffer insieme. Nel 1990, Haeberli e Akeley presentarono un'altro complemento al frame buffer, chiamato *accumulation buffer*. In questo buffer, le immagini vengono accumulate usando un set di operatori. Per esempio, una sequenza di immagini che mostrano un'animazione può essere accumulata e viene generato il *motion blur* come media delle immagini nel buffer. Altri effetti che possono essere generati sono la profondità di campo, antialiasing, soft shadows, ecc.

Quando le primitive hanno raggiunto e passato lo stage di rasterizzazione, quelle che sono visibili dal punto di vista della camera vengono disegnate sullo schermo. Lo schermo quindi mostra il contenuto del color buffer. Per evitare che l'occhio umano veda le primitive man mano che vengono disegnate sullo schermo è stato introdotto il *double buffering*. Questa tecnica renderizza l'immagine in un *back buffer* off-screen, e una volta renderizzata tutta l'immagine, il contenuto del back buffer viene scambiato con quello del *front buffer* che aveva precedentemente mostrato l'immagine sullo schermo.

Capitolo 2
OpenGL

OpenGL (***Open G****raphics* ***L****ibrary*) è una specifica che definisce una API (Application Programming Interface) per più linguaggi e per più piattaforme per scrivere applicazioni che producono computer grafica 2D e 3D. L'interfaccia consiste in circa 250 diverse chiamate di funzione che si possono usare per disegnare complesse scene tridimensionali a partire da semplici primitive. Sviluppato nel 1992 dalla Silicon Graphics Inc., è ampiamente usato nell'industria dei videogiochi, per applicazioni di CAD e di realtà virtuale. È lo standard di fatto per la computer grafica 3D in ambiente Unix, [4],[5],[7],[9],[10],[11].

2.1 Caratteristiche di OpenGL

A livello più basso OpenGL è una specifica, ovvero si tratta semplicemente di un documento che descrive un insieme di funzioni ed il comportamento preciso che queste devono avere. Da questa specifica, i produttori di hardware creano implementazioni, ovvero librerie di funzioni create rispettando quanto riportato sulla specifica OpenGL, facendo uso dell'accelerazione hardware ove possibile. I produttori devono comunque superare dei test specifici per poter fregiare i loro prodotti della qualifica di implementazioni OpenGL. Esistono implementazioni efficienti di OpenGL (che sfruttano in modo più o meno completo le GPU) per Microsoft Windows, Linux, molte piattaforme Unix, Playstation 3 e Mac OS.
OpenGL permette di nascondere la complessità di interfacciamento con acceleratori 3D differenti, offrendo al programmatore una API unica ed uniforme, e permette inoltre di mascherare le capacità offerte dai diversi acceleratori 3D, richiedendo che tutte le implementazioni supportino

completamente l'insieme di funzioni OpenGL, ricorrendo ad un'emulazione software se necessario.

Il compito principale di OpenGL è quello di ricevere primitive come punti, linee e poligoni, e di convertirli in pixel (*rasterizzazione*). Ciò è realizzato attraverso una pipeline grafica nota come *OpenGL state machine*. La maggior parte dei comandi OpenGL forniscono primitive alla pipeline grafica o istruiscono la pipeline su come elaborarle.

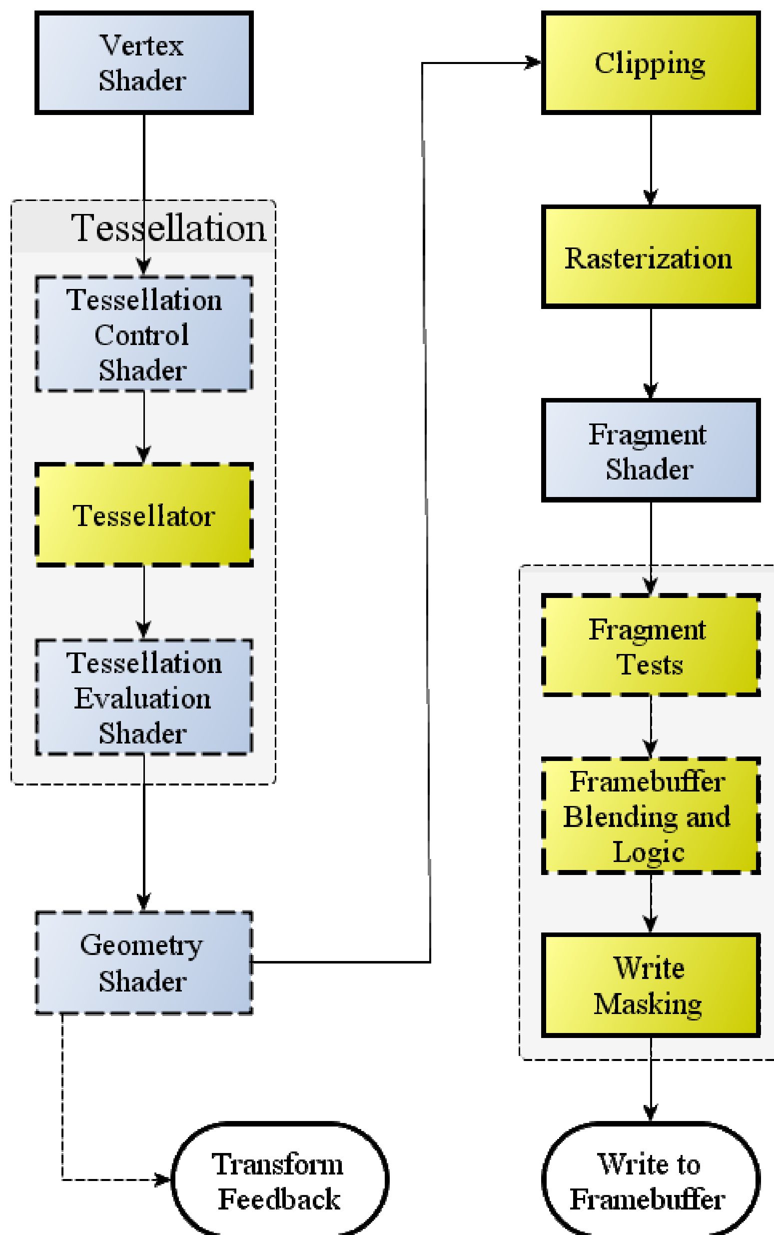

fig. 2.1 Pipeline di rendering OpenGL

Gli stage di questa pipeline, anche se in ordine differente, sono gli stessi della pipeline di rendering vista nel capitolo precedente. Ciò significa che attraverso le istruzioni OpenGL è possibile modificare qualsiasi aspetto della pipeline, permettendo una notevole personalizzazione. La natura di OpenGL obbliga quindi i programmatori ad avere una buona conoscenza della pipeline grafica stessa, ma al contempo lascia una certa libertà per implementare complessi algoritmi di rendering.
OpenGL è una API procedurale che opera a basso livello, che richiede al programmatore i passi precisi per disegnare una scena. Questo approccio si pone in contrasto con le API descrittive ad alto livello le quali, operando su struttura dati ad albero (*scene graph*), richiedono al programmatore solo una descrizione generica della scena, occupandosi dei dettagli più complessi del rendering. Nel corso degli anni le specifiche di OpenGL si sono evolute, permettendo agli utenti di supportare funzioni texture, funzioni di shadowing, programmare gli shader attraverso il linguaggio ***GLSL*** (Open**GL** **S**hading **L**anguage) e altre features che hanno portato OpenGL dalla versione 1.0 all'attuale 4.5.

2.2 GL, GLU, GLUT e paradigma ad eventi

I nomi delle funzioni OpenGL iniziano tutti con le lettere *gl*, e sono usualmente memorizzate in una libreria chiamata ***GL*** (***G**raphic **L**ibrary*). Queste istruzioni comprendono:

- glBegin() & glEnd(), che consentono di delimitare i vertici di una primitiva o di un insieme di primitive;
- glClear(), che permette di riportare il buffer ai valori di default;
- glClearColor(), che consente di specificare i valori di default del color buffer;
- glColor(), che permettono di settare un colore per le primitive;
- glEnable() & glDisable(), utilizzate per attivare o disabilitare alcune capacità di OpenGL;

- glFlush(), che forza l'esecuzione di istruzioni OpenGL in un tempo finito;
- glFrustum(), che se utilizzata moltiplica la matrice corrente per la matrice prospettica;
- glLight(), utilizzata per gestire le luci;
- glLoadIdentity(), che rimpiazza la matrice corrente con la matrice identità;
- glMatrixMode(), che specifica quale matrice è quella corrente;
- glNormal(), che permette di settare il vettore di normali di una primitiva;
- glPointSize(), che consente di determinare la dimensione di un punto;
- glPushMatrix() & glPopMatrix(), che effettuano il push e il pop delle matrici nello stack;
- glRotate(), glScale(), glTranslate(), usate per imprimere alle primitive trasformazioni geometriche;
- glVertex(), utilizzata per specificare un vertice;
- glViewport(), usata per settare la viewport.

La libreria GL inoltre definisce dei nuovi tipi di dato, come GLint e GLfloat, utilizzate per unificare la definizione di variabile intera e di float su sistemi operativi e architetture diverse.
La libreria ***GLU*** (***G****raphics* ***U****tility* ***L****ibrary*) invece contiene funzioni che utilizzano le funzioni base di OpenGL per fornire routine di disegno grafico di un livello più alto. In queste funzioni troviamo quelle per il mapping tra le screen coordinates e le world coordinates, funzioni per generare texture, gestire curve NURBS, istruzioni per settare il volume di vista e la posizione della camera, oltre a funzioni per gestire gli errori inviati dalle istruzioni OpenGL. Inoltre permette di gestire primitive non presenti nella libreria GL, come sfere, cilindri e dischi. Le funzioni della libreria GLU sono facilmente riconoscibili in quanto come prefisso riportano le lettere *glu*,

come ad esempio la funzione gluOrtho2D(), che definisce una matrice ortografica di proiezione bidimensionale.

L'*Open***GL** ***U****tility* ***T****oolkit* (***GLUT***) è una libreria a supporto di OpenGL che consente di gestire le interfacce e gli eventi di I/O provenienti dal sistema operativo. A differenza delle librerie viste prima, GLUT lavora con il *paradigma ad eventi*. Ogni volta che un evento accade (pressione di un tasto sulla tastiera, movimento del mouse ecc.) il sistema operativo manda un messaggio all'applicazione, intercettato da GLUT attraverso specifiche istruzioni, le quali scatenano delle chiamate a funzioni di callback definite dal programmatore. Sta quindi al programmatore creare delle funzioni per la gestione degli eventi, e registrarle con chiamate alle API GLUT. Le funzioni di questa libreria iniziano tutte con il prefisso *glut*, e comprendono:

- funzioni per la definizione di finestre, come glutWindowSize(), glutWindowPosition() e glutCreateWindow();
- funzioni per l'inizializzazione e la creazione di oggetti tridimensionali, come glutWireSphere() e glutSolidCube();
- funzioni per la gestione di eventi di input. Queste sono tra le funzioni più importanti di un programma OpenGL, e comprendono la gestione degli eventi mouse e tastiera (glutMouseFunc() e glutKeyboardFunc()), la visualizzazione del contenuto di una finestra e l'evento di ridimensionamento di quest'ultima (glutDisplayFunc() e glutReshapeFunc()), e la gestione del movimento del mouse sulla finestra(glutMotionFunc()). Per tutte queste funzioni, se richiamate in un programma, va specificata una funzione che ne gestisca l'evento di callback;
- funzioni per la gestione di eventi in background, come la glutIdleFunc();
- altre funzioni rilevanti in un programma che utilizza la libreria GLUT, come la funzione glutPostRedisplay(), che impone alla finestra corrente di essere ridisegnata, o la funzione glutMainLoop(), che

consente al programma di entrare in un loop nel quale verrà controllato ripetutamente il verificarsi degli eventi gestiti nel programma.

2.3 GLUI

Le librerie viste prima non consentono di inserire nella finestra controller come bottoni e checkbox. Questa lacuna è stata colmata introducendo la libreria ***GLUI*** (*Open**GL** **U**ser **I**nterface **L**ibrary*), una libreria costruita per integrare e interagire con le GLUT. Utilizzando le GLUI è possibile definire bottoni, checkbox, radiobutton, textbox editabili, spinner, scrollbar, listbox e molti altri strumenti utili alle applicazioni. GLUI inoltre non contiene codice *system-dependent*, ovvero funzionerà in egual modo su macchine Windows, Mac o Linux. E' inoltre stata studiato per semplificare la programmazione, consentendo di inserire elementi utilizzando solo una riga di codice. Qui sotto è riportata una finestra con tutti i possibili strumenti offerti dalla libreria:

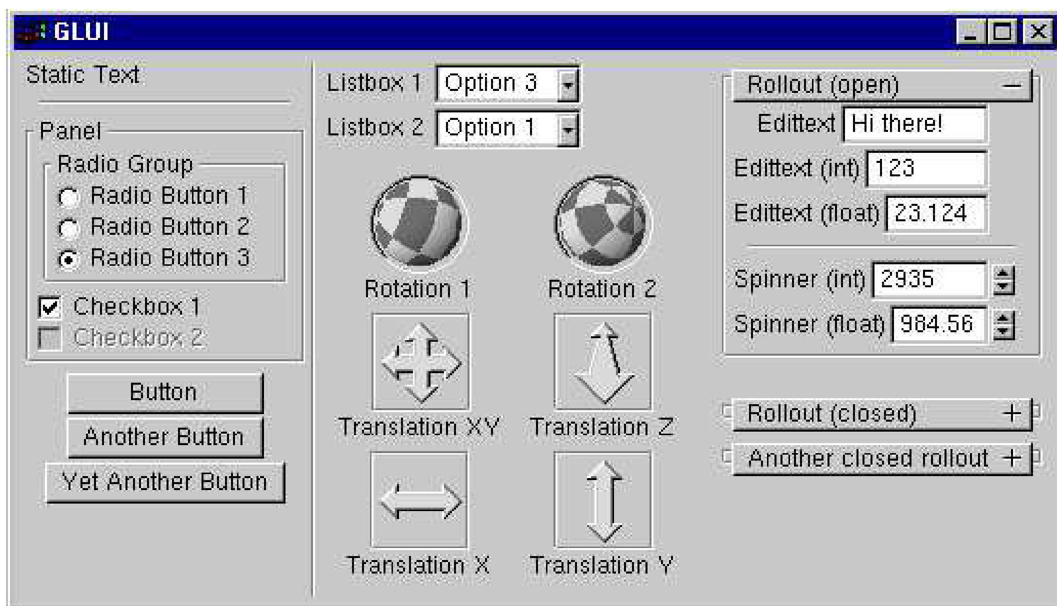

fig. 2.2 Strumenti GLUI

Capitolo 3
Realizzazione dell'editor grafico

In questo capitolo descriveremo il programma sviluppato, soffermandoci sulla progettazione dell'interfaccia e sulle varie funzionalità messe a disposizione e implementate attraverso le librerie OpenGL. Descriveremo anche qualche dettaglio tecnico inerente alle funzionalità, come le specifiche del formato Wavefront OBJ. Nel seguente capitolo vedremo quindi come sviluppare i vari stage della pipeline di rendering con l'ausilio delle istruzioni OpenGL, [6],[9],[10],[11].

3.1 Panoramica generale

L'applicativo si presenta come uno strumento per la manipolazione di primitive 3D, con un'interfaccia intuitiva e con la possibilità di caricare e manipolare files in formato Wavefront OBJ, del quale tratteremo le specifiche in seguito. Come piattaforma di sviluppo è stato usato MS Visual Studio 2010, distribuito da Microsoft, scelto per la semplicità di utilizzo e per la disponibilità di strumenti volti ad aiutare il programmatore durante lo sviluppo, come l'ottimo debugger. Visual Studio è inoltre gratuito per gli studenti del corso, un altro aspetto non da sottovalutare in ambito di sviluppo. Il progetto si è sviluppato sfruttando il linguaggio C++, che mette a disposizione librerie e costrutti molto utili alla programmazione sequenziale. Utilizzando il C++ e OpenGL è possibile quindi richiamare funzioni e operare a basso livello, diminuendo i tempi di renderizzazione delle scene, rendendo più gradevole la user experience.
La versione dell'applicativo allo stato attuale permette di inserire nella scena vari oggetti 3D contemporaneamente, di importare file in formato Wavefront OBJ, di inserire superfici di rivoluzione data una curva definita dall'utente e di interagire con questi oggetti, modificandone la posizione, il

colore e le dimensioni. Permette inoltre di interagire con le luci presenti nella scena, cambiandone colore e posizione.

3.2 Interfaccia

L'interfaccia dell'applicato può essere suddivisa in 3 componenti: la *view* (scena),la *bottom subwindow* (sottofinestra inferiore) e la *right subwindow* (sottofinestra di destra).

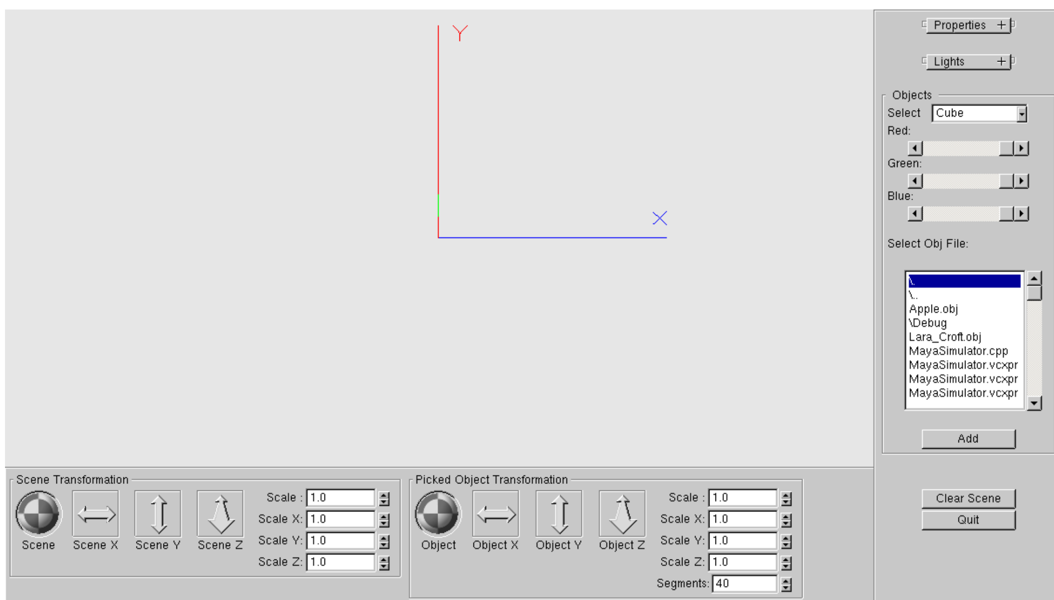

fig. 3.1 Interfaccia dell'editor

La *view* è ben visibile al centro dell'interfaccia e contiene al suo interno gli assi cartesiani x, y e z posti al centro della scena. In basso troviamo la *bottom subwindow,* che contiene gli strumenti per le trasformazioni sulla scena e quelli per le trasformazioni sugli oggetti. Le trasformazioni possibili sono la rotazione, la traslazione e la scalatura. Per i singoli oggetti è inoltre possibile decidere il numero di segmenti dei quali si compongono.

A destra è presenta invece la *right subwindow*, nella quale sono presenti i controlli per le luci, gli strumenti per l'inserimento di oggetti nella scena, una sezione *properties* dalla quale è possibile impostare alcune proprietà della scena, un tasto *Clear Scene* che consente di riportare la scena allo

stato iniziale ed un tasto *Quit* che permette di chiudere il programma. L'interfaccia è quindi abbastanza intuitiva e facile da utilizzare, e permette di gestire l'intera scena con pochi semplici tasti.

3.3 Strutture Dati utilizzate

A supporto delle istruzioni OpenGL, nell'applicativo vengono utilizzate strutture dati volte a semplificare la gestione degli oggetti nella scena e le loro trasformazioni. Le due strutture dati più importanti sono la `struct Object_In_List` e la lista `list< Object_In_List >Object_List`. Queste due strutture permettono al programma di gestire più oggetti contemporaneamente su di una singola scena, e per ognuno di essi le proprie trasformazioni(traslazione, rotazione, scalatura) oltre che altri importanti aspetti. Vediamo ora in dettaglio la composizione della `struct Object_In_List:`

```
typedef struct Object_In_List
{
	int object_type;
	int picking_value;
	float transf_matrix[16];
	float translation_vector[3];
	float scale_vector[4];
	string name;
	int objseq;
	int splineseq;
	bool alive;
	int segment;
	GLfloat color[4];
}
Object_In_List;
```

Il valore di `object_type` consente di determinare il tipo dell'oggetto(cubo, sfera, cono ecc.), e a seconda del tipo all'oggetto viene attribuito un nome, inserito come stringa nell'attributo `name`(es. SPHERE 1, CUBE 1). Ad ogni singolo oggetto inoltre vengono legate le informazioni relative alle sue trasformazioni: traslazione, inserite nel vettore `translation_vector[3]`, contenente rispettivamente nelle varie celle il valore della traslazione in x,y e z; rotazione, inserite nella matrice `transf_matrix[16]`; scalatura, inserite nel vettore `scale_vector[4]`, contenente rispettivamente il valore di scalatura globale, quello della scalatura in x, in y e in z. Di un oggetto vengono immagazzinate anche le informazioni relative al colore, mantenute nel vettore `color[4]`(R,G,B,A), al valore di picking, inserite nell'attributo `picking_value`, al numero di segmenti che compongono l'oggetto (attributo `segment`) e, in caso di file in formato Wavefront OBJ o di superfici di rivoluzione, due attributi che mi permettano di identificare la loro posizione nei rispettivi vettori, chiamati `objseq` e `splineseq`.

La lista di oggetti invece è implementata utilizzando la libreria `<list>` messa a disposizione dal C++. In questo modo è possibile gestirla tramite metodi già implementati nella libreria, risparmiando una notevole quantità di codice. La lista `Object_List` è quindi una lista di `Object_In_List`: per inserirvi un oggetto è necessario quindi istanziare un oggetto `Object_In_List`, inserire nei vari attributi i differenti valori e aggiungere il nuovo oggetto alla lista attraverso in metodo `push_back()`, che inserisce l'elemento in coda alla lista.

Altre strutture dati utilizzate dal programma sono quelle utilizzate per la gestione degli oggetti estratti da file OBJ e quelle utilizzate per gestire le superfici di rivoluzione. Nel primo caso vengono utilizzate strutture create appositamente per gestire questi tipi di file. Vengono infatti create `struct` per l'archiviazione delle informazioni relative a vertici, textures, normali e facce, parti fondamentali del formato sopra indicato.

```
typedef struct Position
{
        double x;
        double y;
        double z;
}
Position;
typedef struct Texel
{
        double u;
        double v;
}
Texel;
typedef struct Normal
{
        double x;
        double y;
        double z;
}
Normal;
```

```
typedef struct Face
{
      int position1[3];
      int texel1[3];
      int normal1[3];
}
Face;
typedef struct Model
{
   vector<Position>Vertices;
   vector<Texel> Textures;
   vector<Normal> Normals;
   vector<Face> Faces;
}
Model;
```

Come si può notare oltre alle strutture citate prima ne è presente un'altra (`Model`), che racchiude in sé tutte le informazioni necessarie alla visualizzazione di un singolo modello esportato da un file. Viene poi istanziato anche un vettore di `struct model` che permette all'utente di caricare più file in formato Wavefront OBJ, consentendo così di comporre una scena con oggetti complessi. Questo vettore ha dimensione fissata a 10 unità, in quanto un maggior numero di file OBJ caricati contemporaneamente provocherebbe un consistente rallentamento dell'applicativo.

Nel caso delle superfici di rivoluzione, invece, vengono utilizzate strutture dati per conservare informazioni relative ai punti del poligono di controllo definito dall'utente, ai punti ricavati dall'algoritmo di De Boor per la

visualizzazione della curva sulla schermo e ai punti ottenuti come rotazione sull'asse y.

```
typedef struct GLFloatPoint
{
   GLfloat x, y, z;
}
typedef struct SplineObject
{
    GLFloatPoint vertex[SEGMENT_LIMIT+1][SEGMENT_LIMIT+1];
    int spline_seg;
}
SplineObject;
typedef struct Faccia
{
    GLFloatPoint vertex[3];
}
Faccia;
```

Sono state quindi create 3 strutture dati: la prima utilizzata per immagazzinare i dati relativi ai vertici, contenente le informazioni relative alla loro posizione; la seconda impiegata per il salvataggio delle informazioni relative alla superficie di rivoluzione, ovvero i vertici e i segmenti di cui è composta; la terza invece viene utilizzata per gestire le facce triangolari che comporranno la superficie. A supporto di queste strutture sono stati istanziati vettori che contengono i vertici di controllo, i punti della curva e i punti ottenuti dalla rotazione di questi ultimi, oltre ad un vettore per la conservazione degli oggetti `SplineObject` e uno per la conservazione delle facce che li compongono.

3.4 Oggetti

In questa sezione verrà esposto il procedimento per inserire oggetti nella scena e gli strumenti messi a disposizione dalle librerie OpenGL utilizzati per svolgere questo compito. Verranno inoltre descritti i tipi di oggetti inseribili nella scena, analizzeremo i files in formato Wavefront OBJ, l'estrazione delle informazioni da essi e l'inserimento di quest'ultimi nella scena. Verranno inoltre descritti il procedimento e gli algoritmi per l'inserimento di superfici di rotazione a partire da profili NURBS definiti dall'utente.

3.4.1 Inserire oggetti nella scena

Per inserire un oggetto nella scena è necessario prima selezionarlo dalla *listbox*, strumento messo a disposizione dalla libreria GLUI che consente di selezionare un elemento presente in una lista e di impostare una variabile a seconda dell'elemento scelto. In questa lista sono presenti tutti i tipi di oggetto inseribili nella nostra scena:

```
/* creo e inizializzo un oggetto GLUI_Listbox e lo aggiungo al pannello object_panel, gli assegno la stringa "Select   " che verrà visualizzata sul lato sinistro della listbox e gli passo una variabile intera alla quale verrà assegnato un valore diverso a seconda dell'elemento selezionato */

GLUI_Listbox *listbox =
glui->add_listbox_to_panel(
      object_panel,
      "Select   ",
      &WhichObject);
```

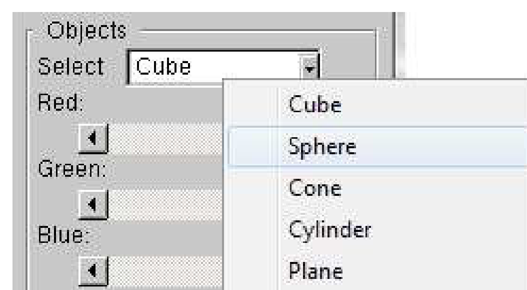

fig. 3.2 Listbox per l'inserimento di oggetti

```
/* aggiungo elementi alla listbox, dandogli il valore che
verrà assegnato alla variabile WhichObject e la stringa da
visualizzare nella lista */
listbox->add_item(CUBE,"Cube");
listbox->add_item(SPHERE,"Sphere");
listbox->add_item(CONE,"Cone");
ecc...
```

Di un oggetto oltre che il tipo è necessario specificarne anche un colore. Per farlo nell'applicativo vengono utilizzati delle *scrollbar*, una per ogni livello di colore (rosso, verde, blu).

```
/* creo un oggetto GLUI_Scrollbar */
GLUI_Scrollbar *sbo;

/* scrivo un testo sopra la scrollbar per distinguere il
colore che verrà modificato */
new GLUI_StaticText(object_panel, "Red:");

/* inizializzo la scrollbar */
sbo = new GLUI_Scrollbar(
object_panel, /* pannello nel quale
inserire la scrollbar */
"Red", /* nome della scrollbar */
GLUI_SCROLL_HORIZONTAL, /* orientamento della scrollbar */
&object_color[0], /* variabile modificata */
);

/* setto il limiti della variabile da 0 a 1 */
sbo->set_float_limits(0, 1);
```

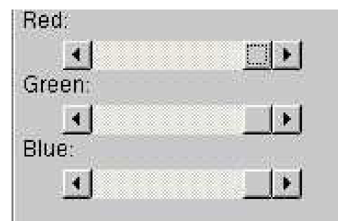

fig. 3.3 Scrollbar per definire il colore di un oggetto

Il vettore `object_color[]` verrà poi opportunamente inserito come attributo `color` dell'oggetto inserito in lista.

A questo punto, dopo aver selezionato un elemento ed averne scelto il colore è necessario premere il pulsante *Add,* la cui funzione di controllo provvede ad aggiungere un oggetto del tipo e del colore selezionato alla lista di oggetti che verranno visualizzati nella view. Il pulsante *Add* ha quindi questa struttura:

```
GLUI_Button *button = glui->add_button_to_panel(
object_panel, /* pannello nel quale inserirlo */
"Add", /* stringa da visualizzare sul bottone */
ADD_ID, /* ID da passare alla funzione di controllo */
(GLUI_Update_CB)listbox_cb /* funzione di controllo */
);
```

La funzione `listbox_cb()` permette di inserire un oggetto nella lista `Object_List`, predisposta per mantenere tutti gli oggetti inseriti dall'utente. Nella funzione quindi viene istanziato un nuovo elemento `Object_In_List`, ai quali attributi `color`, `object_type` e `name` vengono assegnati rispettivamente il colore definito dall'utente, il tipo di oggetto inserito e un nome, dipendente anch'esso dal tipo di oggetto. Gli altri attributi vengono inizializzati con il valore di default, fatta eccezione per il valore dell'attributo `picking_value`, al quale viene assegnato un valore incrementato ad ogni inserimento di un nuovo oggetto (vedi Capitolo 3.5). A questo punto, dopo l'inizializzazione il nuovo elemento viene inserito in lista. Ad ogni refresh della schermata quindi si scorre la lista di oggetti, e per ogni elemento presente vengono effettuati una serie di controlli sugli attributi, che permettono di determinare quale tipo di oggetto visualizzare, chiamando le opportune funzioni che vedremo in seguito. I tipi di oggetti inseribili nella scena sono:

- *primitive 3D;*
- *oggetti estratti da files in formato Wavefront OBJ;*
- *superfici di rivoluzione ottenute da curve NURBS definite dall'utente.*

3.4.2 Primitive 3D

Le *primitive 3D* sono i solidi geometrici predefiniti in OpenGL. Le primitive standard sono quelle di uso comune come la sfera, il cubo, il cono e altre:

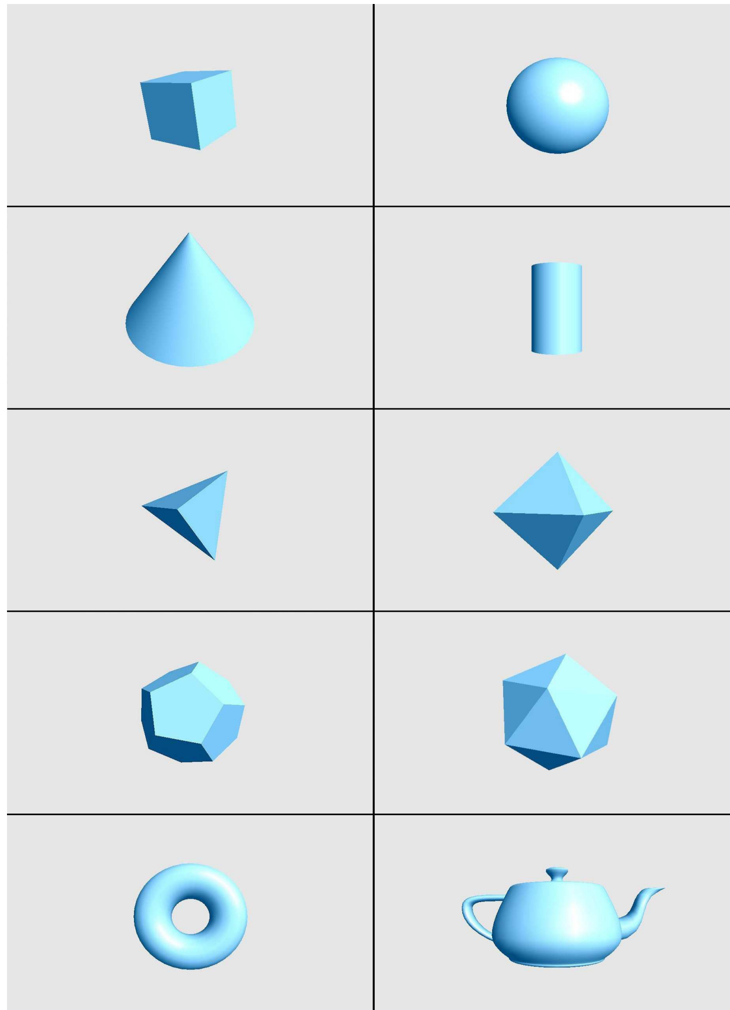

fig. 3.4 Primitive 3D nella scena

OpenGL rende molto facile e intuitivo l'inserimento di queste primitive nella scena. Per inserirle, infatti, basta una sola istruzione, nella quale si deve specificare il tipo di primitiva che si vuole visualizzare e alcuni parametri se richiesti. Per esempio per inserire un cubo è necessario scrivere questa istruzione:

```
glutSolidCube(float size); /* cubo solido */
glutWireCube(float size);  /* cubo in wireframe */
/* size è la dimensione del cubo */
```

Analogamente si agisce per gli altri oggetti, sostituendo a *Cube* il nome dell'oggetto da inserire (es. *Sphere, Cone, Tetrahedron, Teapot* ecc.). Eccezione fatta per il cilindro, che richiede un maggior numero di istruzioni. Per inserirlo, infatti, è necessario prima definire un oggetto di tipo *GLUquadricObj* e inizializzarlo. Poi è necessario definire lo stile con cui verrà disegnato (wireframe o solido). Infine si deve procedere con la funzione gluCylinder():

```
/* definisco un oggetto di tipo GLUquadricObj */
GLUquadricObj *cilindro;

/* inizializzo l'oggetto */
cilindro = gluNewQuadric();

/* definisco lo stile(GLU_FILL solido,GLU_LINE wireframe) */
gluQuadricDrawStyle(cilindro,GLU_FILL);
gluCylinder(
     GLUquadricObj * obj, oggetto inizializzato */
     GLdouble baseRadius, /* raggio cilindro a z=0 */
     GLdouble topRadius, /* raggio cilindro a z=height */
     GLdouble height, /* altezza cilindro */
     GLint slices, /* numero segmenti orizzontali */
     GLint stacks /* numero segmenti verticali  */
          );
```

E' possibile inoltre inserire nella scena un piano, ottenuto come trasformazione geometrica di scalatura sull'asse y di un cubo opportunamente inizializzato:

```
glScalef(1, 0.002, 1);
glutSolidCube(1);
```

3.4.3 Oggetti estratti da file in formato Wavefront OBJ

In questa parte parleremo dei file in formato Wavefront OBJ, della loro composizione e di come è possibile estrarli e inserire nella nostra scena. Partiremo quindi con una introduzione al formato, e ne spiegheremo in dettaglio la composizione. In seguito verrà esposto l'algoritmo per l'estrazione delle informazioni necessarie alla visualizzazione degli oggetti presenti in questi file, e le metodologie utilizzate per la loro renderizzazione nella scena, [6].

3.4.3.1 Il formato Wavefront OBJ

Un file object (obj) rappresenta un formato sviluppato da Wavefront Technologies usato per definire la geometria a altre proprietà di oggetti grafici. Tramite questo formato possono essere elencate tutte le informazioni per la definizione di linee, poligoni, curve e superfici freeform. Le linee e i poligoni sono descritti in termini dei loro vertici mentre curve e superfici sono definite tramite speciali punti di controllo e altri parametri che dipendono dal tipo di curva (Bezier, B-Spline, ecc.). Spesso un file obj è utilizzato per l'interscambio di oggetti grafici tra diverse piattaforme di visualizzazione.

Keywords

La struttura di un file obj è molto semplice. Innanzitutto non è necessario includere un header all'inizio del file. Spesso il file inizia con un breve commento contenente informazioni sull'oggetto definito nel file. I commenti (in linea) su un file obj sono individuati dal carattere "#".
Ogni riga inizia con una keyword seguita dai dati di riferimento per tale keyword. Ecco a seguire una lista delle più importanti keywords che possono essere trovate in un file obj.

Informazioni sui Vertici

– **v** *x y z w* Definisce un vertice geometrico con le rispettive coordinate lungo le tre dimensioni (x, y, z). I vertici elencati nel file obj sono numerati in ordine di apparizione. Ciò significa che il primo vertice sarà etichettato "1", il secondo "2" e così via. Questa indicizzazione sarà utile, come vedremo nella prossima sezione, per la combinazione dei vertici per costituire un oggetto. Il quarto parametro "w" rappresenta una coordinata omogenea utile nel caso in cui bisogna rappresentare curve o superfici. Se omesso, il valore di default per "w" è 1.0. Tutti i parametri sono rappresentati in formato float.

– **vt** *u v w* Definisce come avviene la mappatura UV di una texture. "u, v, w" sono numeri float compresi tra 0 e 1 che indicano dove mappare la texture lungo le tre direzioni. Se la texture è controllata lungo una sola direzione orizzontale allora deve essere specificato solo "u". In questo caso gli altri due parametri assumono il valore di default 0.

– **vn** *i j k* Specifica le componenti i, j e k della normale al vertice di riferimento.

Elementi grafici

Una volta che abbiamo definito i vertici, tramite differenti *elementi grafici* possiamo mettere in relazione queste informazioni per costituire l'oggetto desiderato. Possiamo definire così facce, linee o singoli punti nello spazio. I vertici elencati nel file obj sono indicizzati per ordine di apparizione nell'elenco.

– **p** *v1 v2 v3...* Specifica un punto nello spazio. "v1" è la posizione del primo punto (corrispondente al primo vertice elencato nel file). Ovviamente ogni punto richiede un vertice, e possono essere rappresentati molti punti su una singola riga (Es: **p** *v1 v2 .... vN*).

- **l** *v1/vt1 v2/vt2...* Specifica una linea composta da un minimo di 2 punti. Ogni "v#" è il vertice di un punto in una linea. Accanto all'indice del vertice può essere presente l'informazione sulla mappatura di una texture. L'informazione sul vertice e sulla texture è separata da uno "/" (senza includere spazi). Una linea può essere definita senza includere informazione sulla texture (Es: **l** *v1 v2*).
- **f** *v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3...* Specifica una faccia composta da almeno 3 vertici. Accanto a ciascun vertice possono essere elencate le informazioni su texture e normali. L'informazione sulla texture (relativa a un determinato vertice "vt" elencato precedentemente) precede sempre l'informazione sulle normali. Se vengono incluse le informazioni su texture e normali per un vertice, devono essere incluse in tutti i vertici che costituiscono la faccia. Nel caso in cui abbiamo solo le informazioni sulla posizione dei vertici e le corrispondenti informazioni sulle normali possiamo combinare assieme queste informazioni con la seguente sintassi: **f** *1//1 2//2 3//3 4//4* (faccia composta da 4 vertici con informazione sulle normali). La più semplice definizione di una faccia è costituita da tre vertici senza ulteriori informazioni (Es: **f** *v1 v2 v3*).

Raggruppamento

Attraverso il raggruppamento possono essere realizzate collezioni di oggetti (linee, facce, vertici, ecc.) che rendono più semplice le future manipolazioni su un modello.

- **g** *name1 name2...* Definisce il nome del gruppo di appartenenza per un oggetto. Questa keyword è utile per organizzare collezioni di elementi e semplificare la manipolazione del modello. Un oggetto può appartenere a più gruppi contemporaneamente. Se viene omesso il gruppo per un oggetto, quest'ultimo assume il valore "default".

- **s** *group_member* Questo parametro indica gli elementi sulle quali le normali sono interpolate per attribuire all'elemento un'apparenza "smooth". Il valore di default è "off" o "0"; in questo caso non viene adoperato nessun valore di smoothing.
- **o** *object_name* Parametro opzionale per dare un nome all'oggetto definito subito dopo nel file obj. Tutti gli elementi di raggruppamento vengono applicati a tutti gli oggetti sottostanti nel file fino a quando un nuovo gruppo viene dichiarato.

Attributi di Display/Render

Indicano una serie di attribuiti che vengono assegnati agli oggetti durante la fase di renderizzazione.

- **usemtl** *material_name* Definisce il nome del materiale da assegnare all'oggetto. Il materiale deve essere definito separatamente in un file con estensione mtl.
- **mtllib** *filename1 filename2...* Specifica il nome del file mtl contenente i materiali. Possono essere definiti più file mtl contemporaneamente.

3.4.3.2 Estrazione di dati da file OBJ

Ricavare un algoritmo per l'estrazione di dati da file in formato Wavefront OBJ risulta abbastanza intuitivo. Innanzitutto si deve procedere con la selezione di un file: per fare ciò nell'editor è stata utilizzata una componente delle GLUI chiamata `GLUI_FileBrowser`.

Questa componente si presenta come una finestra che contiene una lista di file che l'utente può selezionare, attivando una funzione di callback.

```
GLUI_FileBrowser(
GLUI_Node *parent, /* pannello che contiene l'oggetto */
const char *name, /* stringa stampata sopra il file browser */
```

```
int frame_type, /* stile del file browser */
int ID, /* ID da passare alla funzione di callback */
GLUI_CB callback /* funzione di callback */
);
```

Nel nostro caso la funzione di callback è `control_cb(int control)`, nella quale viene controllato se la variabile `control` assume il valore `GET_FILE`, ID del file browser. In questo caso la il percorso del file viene salvato nella variabile `file_name`, inviata alla funzione predisposta all'estrazione dei dati (`extractOBJdata()`). Prima di inviare la stringa alla funzione però è necessario controllare se il file è veramente in formato OBJ. Semplicemente quindi si controllano le ultime tre lettere della stringa, e se non corrispondono alle lettere o, b, e j la lettura del file viene annullata, in quanto non porterebbe alcun risultato.

Come detto prima la funzione `extractOBJdata()` è quella responsabile della lettura del file e dell'estrazione delle informazioni contenute in esso. In questa funzione verrà quindi aperto ed esaminato il file passatogli dalla funzione di callback. La semplicità della struttura del file permette di determinare con facilità le informazioni utili eseguendo una scansione sequenziale del file riga per riga. Per ogni riga si analizzano i primi caratteri (keywords), determinando il tipo di dato che si andrà a leggere. A questo punto a seconda del tipo di dato viene operata una differente sequenza di istruzioni, che permetteranno di ottenere i valori utili alla visualizzazione dell'oggetto. Per esempio per estrarre le informazioni relative ai vertici verranno ricercati nella riga tre valori numerici separati da spazi, corrispondenti alle coordinate x, y, e z. Per le facce invece si ricercheranno tre sequenze di valori interi (nel caso di facce triangolari) separati da "/". Queste informazioni verranno quindi salvate in appositi vettori di strutture dati viste in precedenza, che conterranno tutti i dati relativi a vertici, normali, texture e facce. A questo punto tutti questi dati verranno riuniti in un'unica struttura di tipo `Model`, che verrà inserita nel vettore `OBJFiles[]`. L'indice del vettore nel quale verrà salvato il modello sarà

copiato nell'attributo `objseq` del nuovo oggetto inserito in lista. In questo modo ad ogni iterazione della lista, in corrispondenza di un oggetto di tipo `OBJFILE`, verrà lanciata una funzione che mi permetterà di stampare a video tutte le facce del modello contenuto all'indice `objseq` del vettore `OBJFiles[]`. Per avere uniformità nella visualizzazione, l'algoritmo proposto funziona solo con file contenenti facce triangolari. La visualizzazione di un modello sarà quindi effettuata renderizzando tutte le facce di cui è composto. Per fare ciò si scorrono tutte le facce contenute nel modello posto all'indice `index` del vettore. Per ogni faccia quindi verrà disegnato un triangolo con i tre vertici appartenenti ad essa, dando così vita alle varie facce del modello:

```
for(unsigned i=0; i< OBJFiles[index].Faces.size() ; i++)
{
glBegin(GL_TRIANGLES);
//Primo Vertice
glNormal3f(
            OBJFiles[index].Normals.at(
                  OBJFiles[index].Faces.at(i).normal1[0]-1
                                        ).x,
            OBJFiles[index].Normals.at(
                  OBJFiles[index].Faces.at(i).normal1[0]-1
                                        ).y,
            OBJFiles[index].Normals.at(
                  OBJFiles[index].Faces.at(i).normal1[0]-1
                                        ).z
            );
glVertex3f(
            OBJFiles[index].Vertices.at(
                  OBJFiles[index].Faces.at(i).position1[0]-1
                                        ).x
            OBJFiles[index].Vertices.at(
                  OBJFiles[index].Faces.at(i).position1[0]-1
```

```
                                        ).y
            OBJFiles[index].Vertices.at(
                  OBJFiles[index].Faces.at(i).position1[0]-1
                                        ).z
            );
//Secondo Vertice
glNormal3f(
            OBJFiles[index].Normals.at(
                  OBJFiles[index].Faces.at(i).normal1[1]-1
                                        ).x,
            OBJFiles[index].Normals.at(
                  OBJFiles[index].Faces.at(i).normal1[1]-1
                                        ).y,
            OBJFiles[index].Normals.at(
                  OBJFiles[index].Faces.at(i).normal1[1]-1
                                        ).z
            );
glVertex3f(
            OBJFiles[index].Vertices.at(
                  OBJFiles[index].Faces.at(i).position1[1]-1
                                        ).x
            OBJFiles[index].Vertices.at(
                  OBJFiles[index].Faces.at(i).position1[1]-1
                                        ).y
            OBJFiles[index].Vertices.at(
                  OBJFiles[index].Faces.at(i).position1[1]-1
                                        ).z
            );
//Terzo Vertice
ecc..
glEnd();
}
```

Ad ogni vertice inoltre vengono applicate le normali attraverso la funzione `glNormal3f(GLfloat normalx, GLfloat normaly, GLfloat`

`normalz)`. In questo modo il modello verrà illuminato utilizzando il modello di shading di Gouraud.
Questa routine è contenuta nella funzione `DrawObjFile(int index)`, richiamata nel momento in cui l'attributo `object_type` di un oggetto della lista risulti essere uguale alla costante `OBJFILE`. Il valore di `index` viene impostato uguale al valore contenuto nell'attributo `objseq` dell'oggetto.

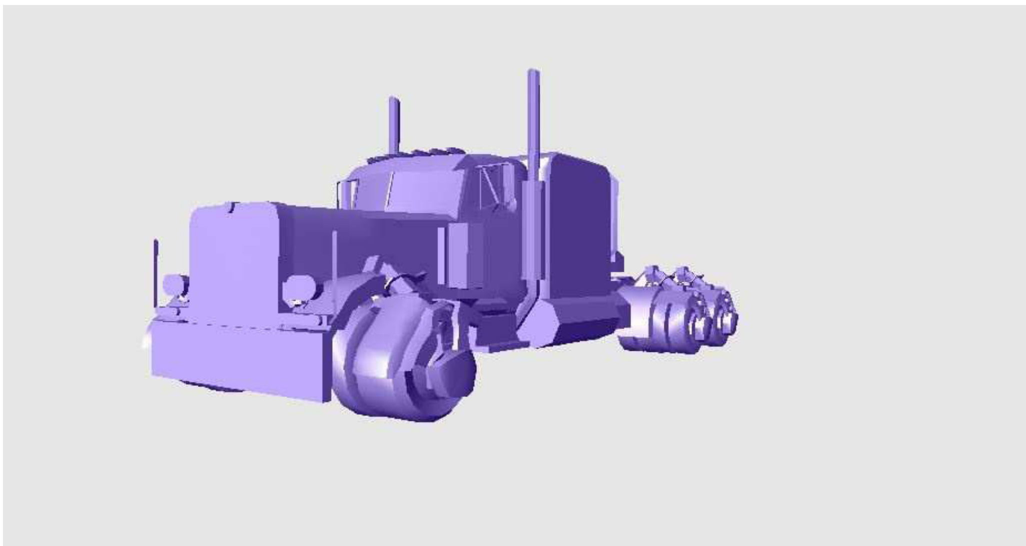

fig. 3.5 File obj caricato nella scena raffigurante un camion

3.4.4 Superfici di rivoluzione

Per inserire nella scena una superficie di rivoluzione è necessario prima definire un profilo NURBS. L'utente quindi, attraverso la definizione di un poligono di controllo, definirà una curva nello spazio, che ruotata individuerà i vertici della superficie. In seguito questi punti verranno uniti in facce triangolari, dando vita ad una superficie di rivoluzione. Per fare ciò, selezionata la voce Revolution Surf dalla listbox e premuto il pulsante Add, si aprirà all'utente una nuova finestra, nella quale sarà possibile definire i vertici di controllo della curva:

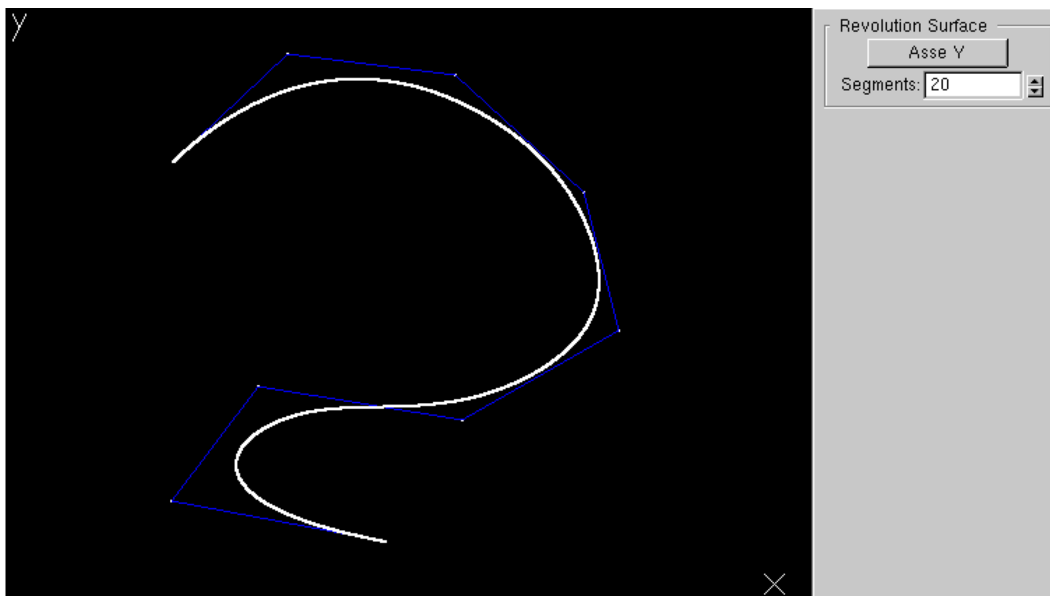

fig. 3.6 Finestra per l'inserimento di profili NURBS

La nuova finestra è composta di due parti: la view principale, sulla quale è possibile definire il poligono di controllo, e un right subwindow, dalla quale è possibile decidere il numero di segmenti che comporranno la superficie utilizzando uno spinner, ed è possibile avviare la rotazione attorno l'asse y. Per poterla aprire, la nuova finestra va prima creata, invocando il comando `glutCreateWindow()`. A questo punto, come per la finestra principale, è necessario definire le funzioni per il controllo degli eventi. In questo caso sarà necessario definire la funzione di controllo per il mouse, oltre a quella per la gestione del display. La funzione myMouse2(int button, int state, int x, int y) è quella richiamata al verificarsi di un evento mouse. In questa funzione vengono gestiti i vertici di controllo definiti dall'utente, immagazzinati in un vettore `Punti[]` e collegati tra loro attraverso una linea blu per definire il poligono di controllo. Per determinare un vertice in un punto è necessario cliccarvi sopra utilizzando il tasto sinistro del mouse, mentre con il tasto centrale si torna indietro di un punto, in modo tale da poter modificare i vertici già definiti. La funzione `display2()`, scelta per la gestione dell'evento display della nuova finestra, consente di visualizzare il poligono di controllo e la curva ricavata da quest'ultimo, utilizzando l'algoritmo di `De Boor()` per la determinazione dei punti della curva. La creazione della finestra e la dichiarazione delle funzioni di controllo

avviene nella funzione di callback del pulsante Add, che prima di avviare la routine per l'inserimento di una superficie controlla che la costante ricevuta in ingresso equivalga al valore `SPLINE`, indicato come ID predefinito per le superfici di rivoluzione.

Una volta che l'utente ha determinato i vertici e ha scelto il numero di segmenti, premendo il pulsante `Asse Y` si avvierà l'algoritmo per la determinazione dei vertici della superficie di controllo, racchiuso nella funzione `dotsRevolution()`:

```
SplineObject ob;
for(int i = 0; i <= spline_segments; i++)
{
for(int j = 0; j <= spline_segments; j++)
 {
   float r = curveDots[i*(10000/spline_segments)].x;
   float theta = 2.0 * 3.1415926 * j / spline_segments;
   ob.vertex[j][i].x = r * cosf(theta);
   ob.vertex[j][i].y=curveDots[i*(10000/spline_segments)].y;
   ob.vertex[j][i].z = r * sinf(theta);
 }
}
return ob;
```

Questo algoritmo prende `spline_segments` (numero di segmenti) punti equidistanti delle curva ottenuta in precedenza e li ruota attorno all'asse y disegnando altri `spline_segments` punti, che diventeranno i vertici della nuova superficie. Questi vertici vengono poi opportunamente salvati in un elemento `ob` di tipo `SplineObject`, che verrà restituito alla funzione chiamante. La funzione chiamante è `revolution()`, messa a controllo del pulsante `Asse Y`. L'oggetto ottenuto dalla funzione `dotsRevolution()` viene inserito in un vettore `SplineObject Splines[]` in posizione `index`, che verrà salvata nell'attributo `splineseq` dell'oggetto inserito in

lista. In questo modo, al momento della visualizzazione su schermo, sarà possibile risalire alla superficie utilizzando l'indice del vettore nel quale è contenuta. In questa funzione, oltre a quella vista in precedenza, vengono invocate altre funzioni, la `crea_Facce()` e la `crea_Normali()`. La prima viene utilizzata per determinare i vertici che comporranno le facce triangolari della superficie. La seconda viene utilizzata per determinare le normali alle facce create.

```
void crea_Facce(int index)
{
list<Faccia> Facce;
Faccia face;
int a,b,c,d;
for(int i = 0; i < Splines[index].spline_seg; i++)
 {
   a = i;
   b = i+1;
   for(int j = 0; j < Splines[index].spline_seg; j++)
     {
        c = j;
        d = j+1;
        if(j == Splines[index].spline_seg-1)
          {
             d=0;
          }
        face.vertex[0].x = Splines[index].vertex[c][a].x;
        face.vertex[0].y = Splines[index].vertex[c][a].y;
        face.vertex[0].z = Splines[index].vertex[c][a].z;
        face.vertex[1].x = Splines[index].vertex[c][b].x;
        face.vertex[1].y = Splines[index].vertex[c][b].y;
        face.vertex[1].z = Splines[index].vertex[c][b].z;
        face.vertex[2].x = Splines[index].vertex[d][a].x;
        face.vertex[2].y = Splines[index].vertex[d][a].y;
```

```
            face.vertex[2].z = Splines[index].vertex[d][a].z;
            Facce.push_back(face);
            face.vertex[0].x = Splines[index].vertex[c][b].x;
            face.vertex[0].y = Splines[index].vertex[c][b].y;
            face.vertex[0].z = Splines[index].vertex[c][b].z;
            face.vertex[1].x = Splines[index].vertex[d][b].x;
            face.vertex[1].y = Splines[index].vertex[d][b].y;
            face.vertex[1].z = Splines[index].vertex[d][b].z;
            face.vertex[2].x = Splines[index].vertex[d][a].x;
            face.vertex[2].y = Splines[index].vertex[d][a].y;
            face.vertex[2].z = Splines[index].vertex[d][a].z;
            Facce.push_back(face);
        }
 }
face_list[index] = Facce;
}
```

Come si può notare la funzione mette insieme i vertici in gruppi di 3 e li salva in una struttura `Faccia`. Tutte queste facce poi vengono messe in una lista inserita in un vettore `face_list[]`, utilizzata in seguito per la visualizzazione delle facce su schermo. L'indice al quale la lista viene salvata in questo vettore corrisponde all'indice del vettore nel quale viene salvato l'oggetto `SplineObject` determinato prima. In questo modo, al momento di operare la stampa delle primitive su schermo, si itererà sulla lista di facce contenute nel vettore `face_list[]`, utilizzando come indice il valore dell'attributo `splineseq`.

La funzione `crea_Normali()` utilizza le facce triangolari calcolate con la funzione precedente per calcolare le normali alle facce, da applicare poi durante la renderizzazione dell'oggetto:

```
void crea_Normali(int index)
{
vector<Normal> normali;
Normal normal;
list<Faccia>::iterator p;

for (p = face_list[index].begin(); p !=
face_list[index].end(); p++)
    {
     normal.x = ((*p).vertex[1].y - (*p).vertex[0].y) *
                ((*p).vertex[2].z - (*p).vertex[0].z) -
                ((*p).vertex[1].z - (*p).vertex[0].z) *
                ((*p).vertex[2].y - (*p).vertex[0].y);
     normal.y = ((*p).vertex[1].z - (*p).vertex[0].z) *
                ((*p).vertex[2].x - (*p).vertex[0].x) -
                ((*p).vertex[1].x - (*p).vertex[0].x) *
                ((*p).vertex[2].z - (*p).vertex[0].z);
     normal.z = ((*p).vertex[1].x - (*p).vertex[0].x) *
                ((*p).vertex[2].y - (*p).vertex[0].y) -
                ((*p).vertex[1].y - (*p).vertex[0].y) *
                ((*p).vertex[2].x - (*p).vertex[0].x);
     normali.push_back(normal);
     }
     normal_list[index] = normali;
}
```

L'algoritmo proposto permette di calcolare le normali alle facce, e di aggiungerle al vettore di normali `normal_list[]`. Utilizzando poi il valore dell'attributo `splineseq`, analogamente alle facce sarà possibile risalire alle normali e applicarle nel rendering finale. Durante lo scorrimento della lista di oggetti quindi, in corrispondenza di quelli con valore dell'attributo `object_type` equivalente a `SPLINE`, verrà lanciata la

funzione `drawRevolutionSurface(int index)`, che preso in ingresso l'indice al quale sono allocate le facce e le normali provvederà a stampare a schermo le facce triangolari, dando forma alla superficie di rivoluzione:

```
list<Faccia>::iterator p;
for (p = face_list[index].begin();
      p != face_list[index].end();
      p++)
{
glBegin(GL_TRIANGLES);
glNormal3f(
              -normal_list[index].at(i).x,
              -normal_list[index].at(i).y,
              -normal_list[index].at(i).z
           );
glVertex3f(
              (*p).vertex[0].x,
              (*p).vertex[0].y,
              (*p).vertex[0].z
           );
glVertex3f(
              (*p).vertex[1].x,
              (*p).vertex[1].y,
              (*p).vertex[1].z
           );
glVertex3f(
              (*p).vertex[2].x,
              (*p).vertex[2].y,
              (*p).vertex[2].z
           );
glEnd();
}
i++;
```

La superficie ottenuta, a differenza degli oggetti di tipo OBJ, sarà illuminata utilizzando uno shading flat. Questa scelta è stata determinata dalla maggiore semplicità di calcolo delle normali alle facce rispetto a quelle ai vertici e dal voler mostrare concretamente le differenze che intercorrono tra i due tipi di shading.

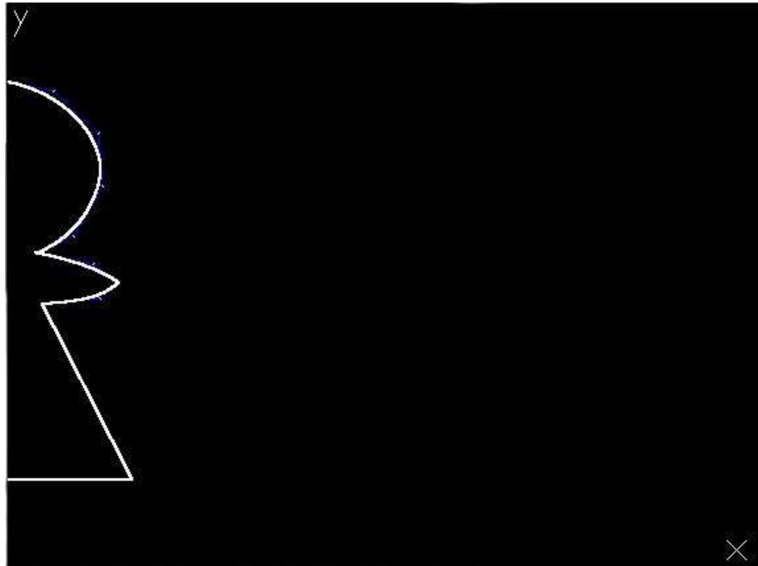

fig. 3.7 Profilo di un pedone di un scacchiera

fig. 3.8 Superficie ottenuta ruotando il profilo in figura 3.7

3.5 Trasformazioni su scena e oggetti

L'editor grafico rende disponibile all'utente la possibilità di manipolare la scena e gli oggetti presenti in essa applicando trasformazioni geometriche tridimensionali utilizzando gli strumenti contenuti nella bottom subwindow.

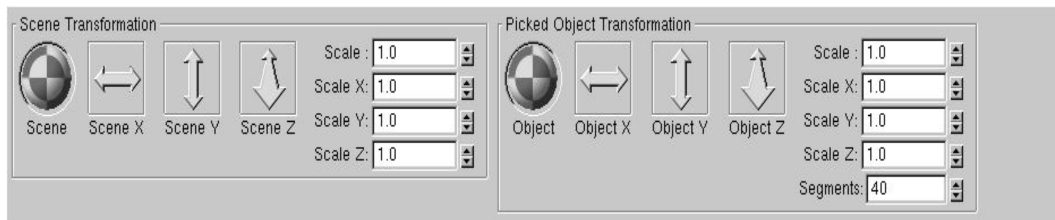

fig. 3.9 Bottom Subwindow

Come si può notare si ha una divisione in due pannelli separati, uno per la scena e l'altro per gli oggetti selezionati dall'utente. Entrambi contengono gli stessi strumenti, fatta eccezione per quello relativo ai segmenti, contenuto solo nel pannello Picked Object.

Analizzando gli strumenti messi a disposizione dell'utente troviamo un oggetto `GLUI_Rotation`, utilizzato per le trasformazioni di rotazione, tre oggetti `GLUI_Translation`, impiegati per le trasformazioni di traslazione sui tre assi, e 4 `GLUI_Spinner`, tramite i quali è possibile applicare alla scena e agli oggetti trasformazioni di scalatura.

Andando nel dettaglio, per applicare trasformazioni geometriche ad un oggetto o alla scena attraverso le istruzioni OpenGL esistono due diverse metodologie: la prima consiste nell'applicare all'oggetto una matrice di trasformazione, ottenuta come prodotto di varie matrici di trasformazione relative alle varie operazioni; la seconda invece consiste nell'utilizzo di funzioni specifiche di OpenGL, alle quali è necessario passare dei parametri che quantificano l'entità della trasformazione. Queste funzioni poi convertono i parametri in matrici di trasformazione, che verranno poi applicate agli oggetti e alla scena.

Nell'editor proposto vengono utilizzate entrambe le metodologie, in quanto gli strumenti utilizzati operano utilizzando diverse strutture dati. La prima

metodologia proposta viene utilizzata quando si deve imprimere alla scena o all'oggetto una rotazione, in quanto l'oggetto `GLUI_Rotation` opera utilizzando una matrice di trasformazione 4x4. Nello specifico un oggetto `GLUI_Rotation` permette di manipolare un controller a forma di sfera alla quale è applicata una texture a scacchiera. Il movimento di questa sfera permette di modificare la matrice, che verrà poi applicata alla scena o agli oggetti selezionati come matrice di trasformazione. Tenendo premuto il tasto CTRL durante la manipolazione della sfera verrà consentito solo il movimento orizzontale di quest'ultima, e premendo ALT quello verticale.

```
GLUI_Rotation *rotation = new GLUI_Rotation(
GLUI_Panel *panel, /* pannello nel quale va inserito */
char *name, /* stringa che compare sotto alla sfera */
float *matrix /* matrice modificata dalla sfera */
);
rotation->set_spin(1.0);
```

Per applicare poi la trasformazione è necessario richiamare la funzione `glMultMatrixf(matrix)` .

Per le altre trasformazioni, invece, vengono impiegate funzioni OpenGL create su misura per le trasformazioni. Nel caso della traslazione infatti viene utilizzata la funzione `glTranslatef(GLFloat x, GLFloat y, GLFloat z)`, che trasla l'oggetto di riferimento in x, y e z. Nel nostro applicativo l'intensità della traslazione è data dall'utilizzo dei `GLUI_Translation`. Questi controlli permettono all'utente di manipolare i valori di traslazione di x, y e z cliccando e trascinando le frecce sulla schermata. Tenendo premuto SHIFT il movimento viene fatto 100 volte più veloce, con CTRL 100 volte più lento.

```
/* inizializzo un elemento GLUI_Translation */
GLUI_Translation *translation;
translation = panel->add_translation_to_panel(
```

```
GLUI_Panel *panel, /* pannello nel quale aggiungerlo */
char *name, /* stringa da visualizzare sotto al cursore */
int trans_type, /* costante che indica l'asse sulla
                   quale operare la traslazione */
float *vector /* vettore dove salvare il valore della
                 traslazione */
);
/*velocità con la quale la traslazione viene incrementata*/
translation->set_speed(.005);

/* il valore di trans_type varia a seconda dell'asse: per
l'asse x si utilizzerà GLUI_TRANSLATION_X, per l'asse y
GLUI_TRANSLATION_Y e per quella z GLUI_TRANSLATION_Z */
```

A questo punto il vettore `vector` conterrà al suo interno i valori di traslazione in sui tre assi, e attraverso la funzione `glTranslatef (GLFloat x, GLFloat y, GLFloat z)` sarà possibile applicare la traslazione, sostituendo opportunamente a x, y e z i valori presenti nel vettore vector.

Nel caso della trasformazione di scalatura invece, per determinare i fattori di scalatura vengono utilizzati degli elementi `GLUI_Spinner`. Questi spinner sono textbox editabili affiancate da due frecce che permettono di incrementare il valore della variabile ad essi legate digitando direttamente il valore nella textbox oppure premendo le frecce, che incrementeranno o decrementeranno il valore a seconda del tempo per le quali sono premute. Premendo il tasto SHIFT e le frecce il valore di step muterà con un fattore 100, premendo CTRL muterà con un fattore di 1/100.

```
GLUI_Spinner *spinner = new GLUI_Spinner(
GLUI_Panel *panel, /* pannello nel quale aggiungerlo */
char *name, /* stringa da visualizzare sotto al cursore */
int data_type, /* costante che indica se si opera su
                  variabili di tipo intero o float */
```

```
float/int *scale /* variabile dove salvare il valore della
                    traslazione(il tipo dipende dal valore
                    di data_type) */
);
/* assegno i limiti alla variabile che verrà modificata */
spinner->set_float_limits(.01f, 4.0);

/* il valore di data_type può essere GLUI_SPINNER_INT o
GLUI_SPINNER_FLOAT, dipendente dal fatto che si voglia
lavorare con una variabile intera o una float */
```

Per applicare poi la scalatura è necessario utilizzare la funzione `glScalef(float scaleX, float scaleY, float scaleZ)`, che scalerà il nostro oggetto sugli assi x, y, e z rispettivamente per i valori delle variabili `scaleX`, `scaleY` e `scaleZ`.

Nel nostro programma si da all'utente la possibilità di scalare l'oggetto su tutti e tre gli assi contemporaneamente oppure singolarmente, utilizzando 4 diversi spinner. Ovviamente nel caso di una scalatura globale dell'oggetto, i valori di `scaleX`, `scaleY` e `scaleZ` coincideranno tutti con quello della variabile modificata dallo spinner. Negli altri casi, le variabili relative agli assi che non occorreranno in scalatura avranno valore settato a 1.

Le trasformazioni sulla scena vengono applicate richiamando le funzioni esposte sopra prima della visualizzazione di tutti gli oggetti presenti nella scena, in modo tale da essere applicate a tutto ciò che verrà renderizzato. Per applicare le trasformazioni ad un oggetto invece è necessario prima selezionarlo attraverso il click del pulsante sinistro del mouse sull'oggetto. La pressione del tasto sinistro del mouse comporta la selezione dell'elemento attraverso un meccanismo di selezione denominato *picking*.

3.5.1 Picking

Il picking è uno strumento che consente di determinare quali primitive contenute nel frame buffer intersecano il volume di vista definito in corrispondenza del cursore del mouse. Tutti le primitive che intersecano questo volume causano un *selection hit*. La lista delle primitive "colpite" viene restituita come un vettore di valori interi, *names*, e un buffer contenente i relativi dati, *hit record*, che corrisponde all'attuale contenuto del *name stack*. Lo stack viene costruito caricandovi i nomi attribuiti alle primitive mentre si è in *selection mode*. Attraverso questi nomi sarà quindi possibile determinare quali primitive sono state colpite, analizzando i valori presenti nel vettore names e nell'hit record.

Per utilizzare questo meccanismo sarà quindi necessario specificare l'array che verrà utilizzato come buffer, entrare in selection mode, inizializzare lo stack attribuendo i nomi alle primitive, definire il volume di vista per la selezione ed infine uscire dalla selection mode e controllare il contenuto dell'hit record. Ogni hit record consiste in quattro componenti:

- il numero di names presenti nel name stack quando l'evento di hit è accaduto;
- il minimo e il massimo valore di z delle coordinate finestra tra tutti i vertici delle primitive che intersecano il volume di vista;
- il contenuto del name stack al momento dell'evento di hit.

OpenGL rende disponibile alcune funzioni per gestire l'evento di picking:

- `glSelectBuffer(GLsizei size, GLuint *buffer);`
  Specifica l'array da usare per i dati restituiti dalla selezione. Il secondo argomento della funzione è un puntatore a un buffer di valori interi dove verranno inseriti i valori, mentre size è il massimo numero di valori immagazzinabili nell'array;

- `glRenderMode(GLenum mode);`

  Permette di cambiare da rendering a selection mode. La variabile mode può assumer i valori GL_RENDER e GL_SELECT. L'applicazione rimane nella modalità data fino a che non viene richiamata la stessa funzione con un'altra modalità. Nel caso in cui ci si trovi in selection mode, il valore di ritorno della funzione è il numero di selection hits, ovvero di primitivi colpite dal picking;

- `glInitNames();`

  inizializza lo stack;

- `glPushName(GLuint name);`

  Inserisce il nome `name` in cima allo stack. Questa operazione va sempre eseguita dopo l'inizializzazione dello stack, in quanto la funzione glLoadName() causerebbe un errore se chiamata su di uno stack vuoto;

- `glPopName();`

  Elimina l'elemento in cima alla lista;

- `glLoadName(GLuint name);`

  Sostituisce il primo elemento dello stack con quello specificato in `name`. Le chiamate a glPushName, Pop e Load sono ignorate se non ci si trova in selection mode;

- `gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble height, Glint viewport[4]);`

  Crea una matrice di proiezione che restringe l'area della selezione ad una porzione della viewport. Il centro della regione di picking è (x,y) nelle coordinate finestra, nel nostro caso corrispondono alla posizione del cursore; width e height definiscono le dimensioni della regione di

picking; viewport[] indica i contorni della viewport corrente, che possono essere ottenuti richiamando la funzione glGetIntegerv(GL_VIEWPORT, GLint *viewport).

Nel nostro editor l'evento di picking è gestito dalla funzione `gl_select(int x, int y)`, che richiamata alla pressione del tasto sinistro del mouse prende in ingresso le coordinate del cursore e restituisce in due variabili globali il numero di elementi colpiti e il buffer nel quale è contenuto l'hit record. A questo punto, ogni volta cha la funzione `DrawObjects(GLenum mode)`(funzione predisposta alla stampa a video delle primitive) viene richiamata, si deve controllare se `mode` ha valore `GL_SELECT` o `GL_RENDER`. Nel primo caso, prima di ogni primitiva, verrà richiamata la funzione `glLoadName(),` in modo tale da caricare l'elemento nello stack. Nel secondo caso invece si provvederà a determinare quali oggetti sono stati colpiti analizzando il buffer restituito dalla funzione `DrawObjects()`. Verrà quindi salvato il name dell'oggetto colpito (nel caso di più oggetti colpiti verrà selezionato solo l'ultimo inserito dall'utente in termini di tempo), e verrà confrontato con tutti i valori degli attributi `picking_value` (predisposto al mantenimento del name di un oggetti) dei vari oggetti presenti in lista. Nel caso in cui i due valori coincidessero, le trasformazioni operate nel pannello Picked Object Transformation verranno copiate nei rispettivi attributi del'oggetto, e applicate ad esso ad ogni scorrimento della lista di oggetti. In questo modo ogni oggetto avrà le sue trasformazioni esclusive, permettendo all'utente di creare scene complesse.

3.6 Eventi mouse e tastiera

Con "eventi mouse" si intendono tutte le possibili interazioni che un utente può avere con il mouse, come la pressione dei tasti (destro, sinistro, centrale) o il movimento del mouse stesso. Analogamente gli "eventi tastiera" sono tutte le possibili interazioni con la tastiera, ovvero la pressione dei tasti di quest'ultima.

L'applicativo in esame permette di utilizzare queste periferiche per interagire con la scena. Come visto in precedenza la pressione del tasto sinistro del mouse comporta l'avvio della funzione `gl_select()`, la funzione che controlla l'evento di picking. La pressione del tasto destro del mouse invece comporta l'apertura di un menù sullo schermo, dal quale è possibile selezionare una vista:

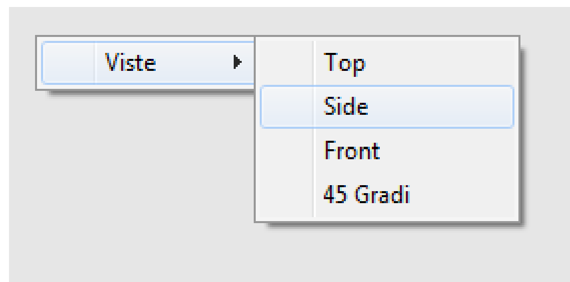

fig. 3.10 Menù per scegliere tra le viste disponibili

In questo modo selezionando una vista si innescherà un conseguente cambio di parametri, che permetterà all'utente di avere una vista frontale, dall'alto, laterale o a 45 gradi.

Permettere al programma di aprire un menù di questo tipo è abbastanza semplice. Innanzitutto si deve creare il menù e i vari sottomenù ad esso legato, e inserire come parametro la funzione che verrà richiamata al momento della selezione di una delle voci del menù:

```
/* creo il menù principale */
int id_MENU1=glutCreateMenu(menu1);
```

```
/* creo un sottomenù e gli aggiungo delle voci */
int id_MENU2=glutCreateMenu(menu1);

/* glutAddMenuEntry("Testo visualizzato nel menù",
   parametro passato alla funzione indicata) */

glutAddMenuEntry("Top", WIEW_TOP);
glutAddMenuEntry("Side", WIEW_SIDE);
ecc..

/* aggiungo al menù id_MENU1 il sottomenù id_MENU2
   sotto la voce "Viste" */
glutSetMenu(id_MENU1);
glutAddSubMenu("Viste",id_MENU2);
```

Conclusa la creazione si deve legare il menù ad un evento mouse, nel nostro caso la pressione del pulsante destro del mouse:

```
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

A questo punto nella funzione passata come parametro alla funzione `glutCreateMenu()` è necessario scrivere il codice che permetterà di cambiare vista a seconda della voce selezionata. Viene quindi controllato il parametro passato alla funzione di controllo, e a seconda del suo valore viene impostato il valore della variabile `obj_view`, responsabile del cambio di vista nella funzione `myGlutDisplay()`.

Tutti gli eventi mouse sono gestiti nella funzione `myGlutMouse(int button, int state, int x, int y)`, che presi in ingresso il bottone del mouse premuto, lo stato del bottone (`GLUT_UP` o `GLUT_DOWN`) e le coordinate del mouse al momento della pressione ne gestisce l'uso. Per permettere alla funzione di ricevere questi dati in ingresso è necessario passarla come parametro alla funzione `glutMouseFunc()`.

L'applicativo gestisce inoltre 3 eventi tastiera:

- Premendo la lettera "i" si richiama la funzione "idle" del programma. Per funzione idle si intende una funzione che opera in background e che permette una continua animazione nella finestra principale. Se attiva quindi, la funzione idle è continuamente richiamata quando l'applicativo non riceve eventi.

Nel nostro programma la funzione idle applica una rotazione continua alla scena, mostrandola a 360 gradi. In questo modo è possibile esaminare la scena da tutte le angolazioni in maniera semplice e veloce. Per fare ciò la funzione `myGlutIdle()` incrementa il valore della variabile `idleSpin`, che venendo applicata alla scena attraverso la funzione `glRotatef(idleSpin,0,1,0)` permette di far ruotare la scena. La funzione `myGlutIdle()` permette di attivare l'evento idle in quanto passata come parametro alla funzione `glutIdleFunc()` richiamata alla pressione del tasto "i" . Per fermare l'esecuzione della funzione idle è necessario premere nuovamente il tasto "i". La nuova pressione del tasto comporterà la chiamata della funzione `glutIdleFunc()` con parametro `NULL` che causerà l'arresto dell'idle.

- Premendo la lettera "d" invece si attiverà la funzione `deleteElement()`. Questa funzione consente all'utente di eliminare dalla scena l'oggetto selezionato attraverso la funzione di picking, discussa nel capitolo precedente. Entrando nel dettaglio la funzione `deleteElement()` non elimina l'oggetto selezionato dalla lista, ma ne setta l'attributo `alive` a `false`. In questo modo, durante il ciclo per la visualizzazione degli oggetti nella scena, gli oggetti con il valore `alive` settato a `false` non verranno visualizzati nella scena.

− L'ultimo evento tastiera si verifica alla pressione del tasto "q". Premendo questo tasto infatti sarà possibile terminare il programma. Questa stessa funzione è svolta anche dal tasto Quit come visto in precedenza.

Questi eventi sono gestiti nella funzione `myGlutKeyboard()` , predisposta per la ricezione di eventi tastiera perché passata come parametro alla funzione `glutKeyboardFunc()`, che come le funzioni `glutMouseFunc()` e `glutIdleFunc()` sono tra le funzioni principali di controllo di OpenGL.

3.7 Properties

La sezione *properties* è gestita nella right subwindow tramite l'utilizzo di un elemento chiamato "rollout", messo a disposizione dalla libreria GLUI e che permette di racchiudere ed aprire al momento del bisogno un insieme di elementi contenuti in esso:

```
/* creo un nuovo elemento GLUI_Panel e lo inizializzo come
un GLUI_Rollout, passando come parametri la finestra di
appartenenza (glui), come nome la stringa "Properties" e il
valore booleano false, che definisce l'elemento rollout
"chiuso" all'avvio del programma */

GLUI_Panel *obj_panel;
obj_panel = new GLUI_Rollout(glui, "Properties", false);
```

In esso sono racchiusi due "checkbox", elementi anch'essi messi a disposizione dalla libreria GLUI, e che consentono di impostare una variabile intera ad essi associati a valori 1 o 0 a seconda che la checkbox sia spuntata o meno. Queste due checkbox concedono all'utente la possibilità di decidere se visualizzare le figure in wireframe o in stato solido, e se visualizzare gli assi cartesiani o meno.

```
/* creo e inizializzo due nuovi elementi GLUI_Checkbox
aggiungendoli al rollout creato in precedenza. Ad ognuno
sono attribuiti anche un nome, una variabile sulla quale
operare, un ID e una funzione di controllo */
new GLUI_Checkbox(obj_panel, "Wireframe", &wireframe, 1,
control_cb);
new GLUI_Checkbox(obj_panel, "Show Axes", &show_axes, 1);
```

Ad ogni ciclo di visualizzazione della scena verrà quindi effettuato un controllo sul valore della variabile `wireframe`, che se impostata a valore 1 consentirà al programma di scegliere le istruzione per la visualizzazione delle primitive in wireframe come visto nella sezione relativa alle primitive 3D. Il controllo sulla variabile `show_axes` comporterà invece l'esecuzione o meno della funzione `DrawAxes()`, predisposta per la visualizzazione sulla scena degli assi cartesiani.

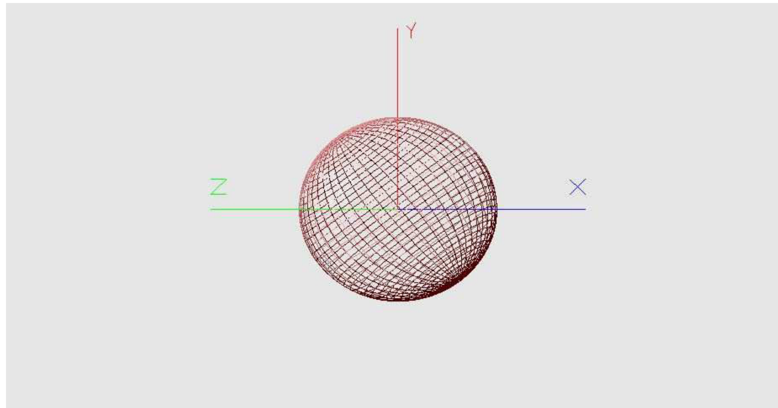

fig. 3.11 Scena con sfera con checkbox Wireframe e ShowAxes abilitate

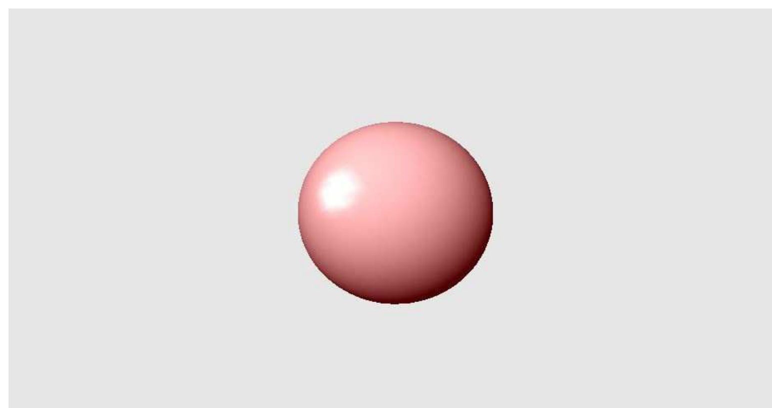

fig. 3.12 Scena con sfera con checkbox Wireframe e ShowAxes disabilitate

3.8 Luci

In questo applicativo all'utente è consentita la gestione di due fonti luminose attraverso il menù presente sulla destra. Prima di poter essere modificate le informazioni relative alle luci però, queste ultime devono essere inizializzate. Le istruzioni di inizializzazione delle luci sono racchiuse nella funzione `LightsSetting()`, opportunamente richiamata nel main del programma. Per poter inizializzare adeguatamente una luce è necessario definirne alcuni parametri, come la posizione, l'intensità luminosa ambientale e quella diffusiva.

Nel nostro programma è stato scelto di inizializzare le luci utilizzando solo questi parametri, in quanto non risultava necessario l'uso di più parametri(intensità speculare, attenuazione, distribuzione dell'intensità ecc.), lasciati quindi ai valori di default, riuscendo comunque ad ottenere un buon livello di illuminazione della scena.

Come prima cosa si procede con l'abilitare l'illuminazione della scena attraverso la funzione `glEnable(GLenum cap)`, dove *`cap`* specifica una costante che indica una capacità della libreria grafica. Nel nostro caso *`cap`* assume il valore `GL_LIGHTING`. Una volta abilitata l'illuminazione della scena è necessario definire delle fonti luminose. OpenGL al massimo riesce a gestire otto fonti luminose, ma nel nostro caso ne utilizzeremo solo due. Per attivare una fonte luminosa è necessario invocare ancora una volta la funzione `glEnable()`, e passarle come parametro una costante decisa dal programmatore, nel nostro caso `GL_LIGHT0` per la prima luce e `GL_LIGHT1` per la seconda. A questo punto è necessario definire i parametri per le fonti luminose, utilizzando la funzione `glLightfv()`:

```
void glLightfv(
      GLenum light, /* specifica una luce */
      GLenum pname, /* specifica un parametro della fonte
                       luminosa per la luce light */
```

```
        const GLfloat *params /*specifica un puntatore a un
        vettore di valori che verranno attribuiti al
        parametro pname della luce light */
);
```

Nell' editor quindi:

```
glEnable(GL_LIGHTING); /* abilito l'utilizzo di luci */

glEnable(GL_LIGHT0); /* abilito la luce GL_LIGHT0 */

/* setto il valore dell'intensità luminosa ambientale
(GL_AMBIENT) al valore del vettore light0_ambient, che
contiene i valori RGBA.*/
glLightfv(GL_LIGHT0, GL_AMBIENT, light0_ambient);

/* analogamente opero per l'intensità luminosa diffusiva */
glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);

/* setto il valore della posizione della luce (GL_POSITION)
al valore del vettore light0_position, che contiene i valori
x,y,z della posizione. */
glLightfv(GL_LIGHT0, GL_POSITION, light0_position);
/* opero similmente per la seconda luce */
glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
glLightfv(GL_LIGHT1, GL_POSITION, light1_position);

/* da notare la mancata chiamata alla funzione
glEnable(GL_LIGHT1), siccome all'avvio la seconda luce è
disabilitata. */
```

La parte relativa alle luci nel menù è racchiusa in un elemento rollout, inizialmente chiuso:

```
/* creo un nuovo elemento GLUI_Rollout e lo aggiungo alla
finestra glui, con nome "Lights" e inizialmente chiuso */
GLUI_Rollout *roll_lights = new GLUI_Rollout(glui,"Lights",
false);
/*  aggiungo  2  pannelli(uno  per  ogni  luce)  all'elemento
rollout creato in precedenza */
GLUI_Panel *light0 = new GLUI_Panel(roll_lights, "Light 1");
GLUI_Panel *light1 = new GLUI_Panel(roll_lights, "Light 2");
```

Come è possibile notare dall'immagine all'avvio dell'applicativo una luce risulta disabilitata, ma è possibile abilitarla cliccando sulla checkbox "Enabled". A questo punto anche tutti gli strumenti verranno attivati, in modo tale da poter interagire liberamente con entrambe le luci. Delle luci presenti sulla scena è possibile modificarne intensità, posizione e colore, oltre a poterle attivare o disattivare come detto sopra. Per poter interagire con questi aspetti sono stati utilizzati quattro diversi strumenti:

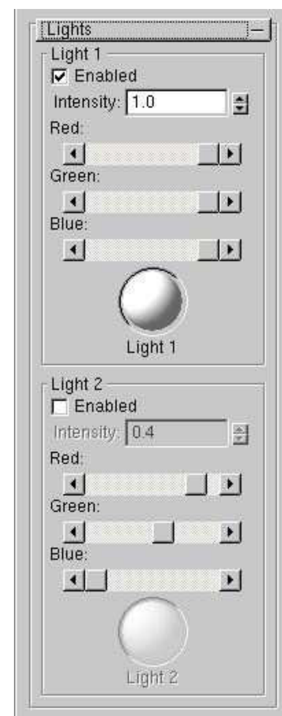

fig. 3.13 Rollout Lights

- Checkbox, che utilizza una variabile intera impostandone come valore 1 o 0, a seconda che la casella sia spuntata o meno. In questo modo è possibile abilitare o disabilitare la luce rispettiva alla checkbox spuntata:

  ```
  new GLUI_Checkbox(
    light0, /* pannello nel quale inserirlo */
    "Enabled",/* stringa affiancata alla checkbox */
    &light0_enabled, /* variabile intera modificata */
  ```

```
 LIGHT0_ENABLED_ID, /* parametro passato alla funzione
                             di callback */
 control_cb /* funzione di callback chiamata quando
                  viene selezionata */
           );
```

Nella funzione di callback vengono effettuati dei controlli che permettono all'applicativo di abilitare o disabilitare le luci a seconda del valore della variabile `light0_enabled`(in questo caso della prima luce), attraverso la chiamata del metodo `disable()` o del metodo `enable()`, messo a disposizione degli oggetti facenti parte della libreria glui;

– Spinner, che controlla l'intensità della luce operando su una variabile di tipo float. Lo spinner permette di modificare il valore al suo interno utilizzando le frecce che si trovano sulla destra oppure digitando direttamente il valore tramite tastiera:

```
/* creo un oggetto GLUI_Spinner */
GLUI_Spinner *light0_spinner;
/* inizializzo lo spinner */
light0_spinner = new GLUI_Spinner(
      light0, /* pannello nel quale inserire lo
                  spinner */
      "Intensity:", /* stringa vicino allo spinner */
      &light0_intensity, /* variabile modificata dallo
                              spinner */
LIGHT0_INTENSITY_ID, /* costante passata alla funzione
                           di callback */
control_cb /* funzione di callback */
                                   );
/* setto i limiti della variabile da 0 a 1 */
light0_spinner->set_float_limits(0.0, 1.0);
```

La variabile `light0_intensity` (nel caso della prima luce) viene modificata attraverso lo spinner, e opportunamente moltiplicata ai valori del vettore responsabile dell'intensità della luce diffusiva, in modo tale da ridurne o aumentarne l'intensità. Queste operazioni sono svolte nella funzione di callback `control_cb()`;

- Scrollbar, che consente di modificare i valori RGB dell'intensità luminosa ambientale delle luci, modificando i valori di un vettore. E' possibile muovere la scrollbar utilizzando le frecce poste ai lati di essa oppure spostando l'indicatore del livello con il mouse:

```
/* creo un oggetto GLUI_Scrollbar */
GLUI_Scrollbar *sb;
/* scrivo un testo sopra la scrollbar per distinguere il
colore che verrà modificato */
new GLUI_StaticText(light0,"Red:");
/* inizializzo la scrollbar */
sb = new GLUI_Scrollbar(
      light0, /* pannello nel quale inserire la
                 scrollbar */
      "Red", /* nome della scrollbar */
      GLUI_SCROLL_HORIZONTAL, /* orientamento della
                                 scrollbar */
      &light0_diffuse[0], /* variabile modificata */
      LIGHT0_INTENSITY_ID, /* costante passata alla
                              funzione di callback */
      control_cb /* funzione di callback */
);
/* setto il limiti della variabile da 0 a 1 */
sb->set_float_limits(0, 1);
```

Nella funzione di callback i parametri dell'intensità luminosa ambientale vengono opportunamente sostituiti con quelli ottenuti dal movimento delle scrollbar.

- Rotation, uno strumento che permette di registrare i movimenti operati su di una sfera e di immagazzinarli in una matrice, che opportunamente applicata come matrice di trasformazione consente di ruotare la fonte luminosa:

```
/* creo un oggetto GLUI_Rotation */
GLUI_Rotation *light0_rot;
/* inizializzo la sfera */
light0_rot = new GLUI_Rotation(
            light0, /* pannello nel quale va inserito */
            "Light 1", /* stringa che compare sotto alla
                          sfera */
            light0_rotation /* matrice di trasformazione
                               modificata dalla sfera */
                            );
/* setta il valore di smorzamento della sfera */
light1_rot->set_spin(.82);
```

La matrice di trasformazione ottenuta verrà applicata alla luce attraverso l'uso della funzione:

```
glMultMatrixf( light0_rotation);
```

Tutte queste specifiche sono mantenute nella funzione `GluiSetting()`. Nella scena le luci sono identificate da due sfere, posizionate negli stessi punti delle due sorgenti luminose. Il loro colore inoltre varia col variare dell'intensità luminosa ambientale delle luci, dando così all'utente un idea del colore delle due fonti luminose.

Capitolo 4
Esempi di applicazione

In questo ultimo capitolo verranno mostrati alcuni esempi di utilizzo del nostro editor grafico. In particolare verranno esaminate due scene create mediante l'editor, nelle quali verranno utilizzati tutti gli strumenti visti nei precedenti capitoli.

Nella prima scena si è voluto ricreare una piccola sala da pranzo con un tavolo, sul quale sono poggiati un vaso, una lampada e due figure geometriche. Appesi alle pareti poi sono stati applicati dei quadri di colori diversi. Per fare ciò si è partiti dalla composizione della stanza: due piani di colore bianco hanno consentito di creare le pareti della stanza, mentre un altro piano di colore marrone è stato utilizzato per creare il pavimento. Successivamente sono stati inseriti i quadri alle pareti, ottenuti da dei cubi scalati e colorati opportunamente. Il tavolo, il vaso e la lampada sono stati importati da file OBJ, e aggiunti alla scena dopo essere stati scalati, ruotati e traslati opportunamente. Infine sopra al tavolo sono stati aggiunti una piramide, ottenuta da un cono composto di quattro segmenti, e un icosaedro, colorandoli in maniera differente. Per illuminare la scena è stata utilizzata una sola luce di colore bianco opportunamente ruotata.

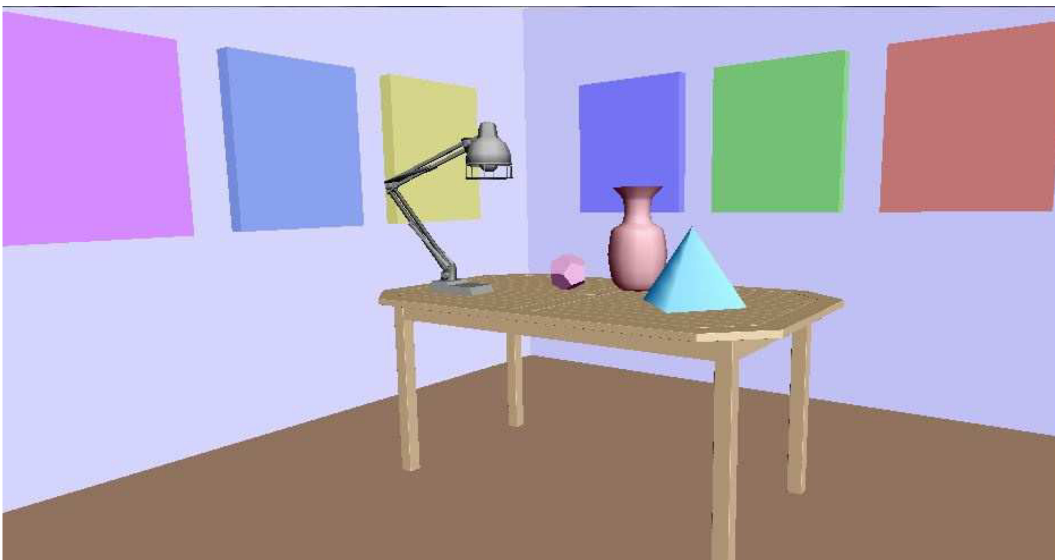

fig. 4.1 Sala da pranzo creata con l'editor grafico

Nella seconda scena invece è riportata una scena fantastica, nella quale sono riportati alcuni robot tratti da una film posti su dei piedistalli. Tutti i robot e sono stati estratti da file OBJ diversi. Uno dei piedistalli è stato creato utilizzando lo strumento per le superfici di rivoluzione.

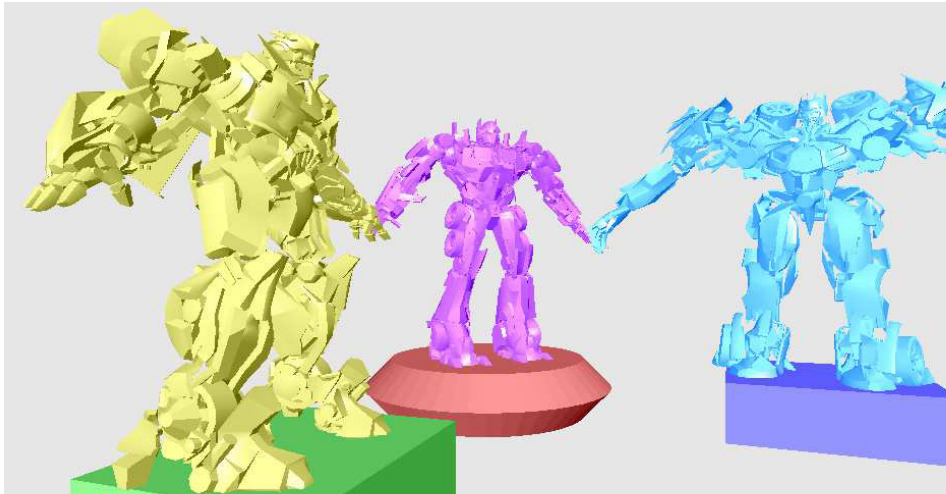

fig. 4.2 Scena completa con robot

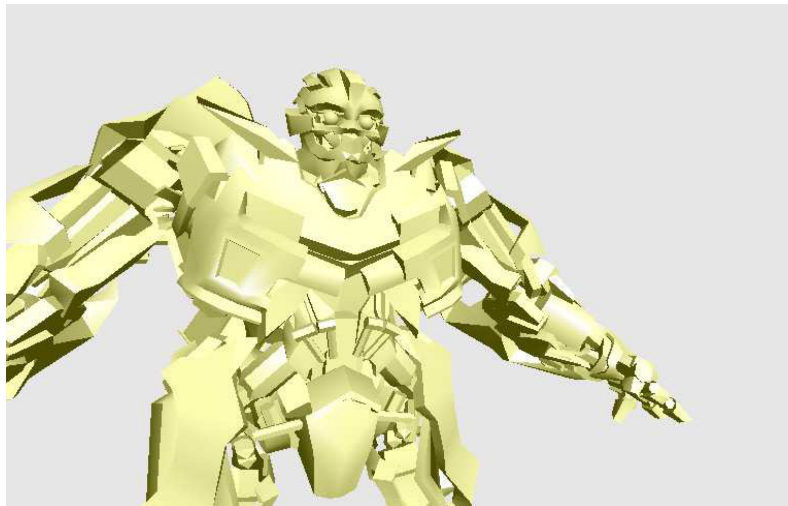

fig. 4.3 Dettaglio sulla struttura di uno dei robot

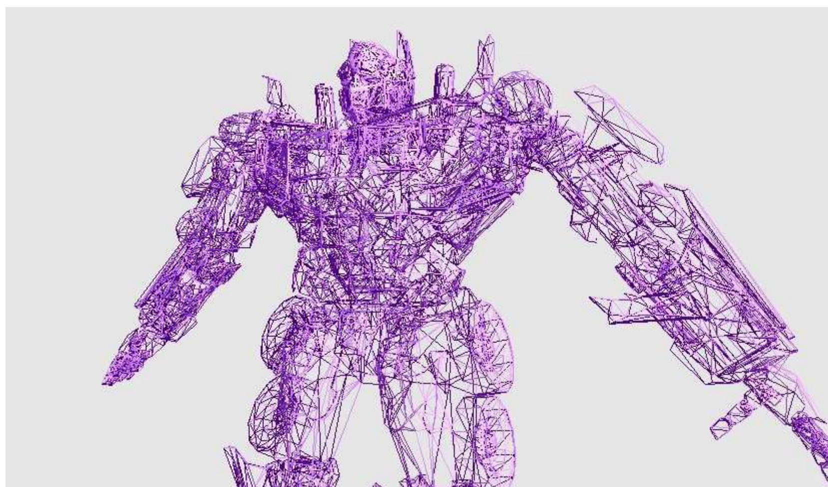

fig. 4.3 Dettaglio sulla struttura di uno dei robot in wireframe

In conclusione si può dire che l'esperienza come utente è gradevole, in quanto l'editor scorre fluidamente ed è particolarmente intuitivo da usare. Offre tanti strumenti per personalizzare la scena, e con la possibilità di importare oggetti da file OBJ e di definire superfici di rivoluzione viene concesso all'utente di definire scene complesse e ricche di oggetti diversi. Sicuramente necessita di qualche piccola modifica per essere considerato un editor grafico completo, come la possibilità di applicare textures agli oggetti o di attribuirvi un materiale. Inoltre non è possibile salvare le scene create ne ottenere degli screenshot da esse se non utilizzando altri strumenti messi a disposizione dal sistema operativo. Con gli strumenti a disposizione però si è raggiunto un buon livello di implementazione e l'esperienza risulta soddisfacente .

Bibliografia e Sitografia

[1] John F. Hughes, Andries Van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven K. Feiner, Kurt Akeley, *"Computer Graphics, Principles and Practice Third Edition"*, Addison-Wesley, Luglio 2013;

[2] Damiana Lazzaro, dispense didattiche del corso di "*Metodi numerici per la grafica*";

[3] Daniele Marini, dispense didattiche dal corso di "*Programmazione grafica e laboratorio*";

[4] Graham Sellers, Richard S. Wright, Nicholas Haemel, *"OpenGL Superbible Sixth Edition, Comprehensive Tutorial and Reference"*, Addison-Wesley, Luglio 2013;

[5] Dave Shreiner, Graham Sellers, John Kessenich, Bill Licea-Kane, *"OpenGL Programming Guide Eighth Edition, The Official Guide to Learning OpenGL, Version 4.3"*, Addison-Wesley, Marzo 2013;

[6] Giovanni Gallo, *"Formati OBJ e RAW"* in : http://www.dmi.unict.it/~gallo/materiale/cg/Formati%20OBJ%20e%20RAW.pdf, (ultimo aggiornamento 15/11/2011);

[7] Paul Rademacher, "*GLUI, a GLUT-Based User Interface Library*", in: https://www.cs.unc.edu/~rademach/glui/src/release/glui_manual_v2_beta.pdf, (ultimo aggiornamento 10/06/1999);

[8] Matthias Teschner, *"Image Processing and Computer Graphics, Rendering Pipeline"*, in: http://cg.informatik.uni-freiburg.de/course_notes/graphics_01_pipeline.pdf, (ultimo aggiornamento 24/10/2013).;

[9] "*OpenGL Programming Guide*" in : http://www.glprogramming.com/ ;

[10] "*Tutorials for modern OpenGL*" in : http://www.opengl-tutorial.org/ ;

[11] "*OpenGL, The Industry's Foundation for High Performance Graphics*", in: http://www.opengl.org/.