Università di Pisa
Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

# Studio e realizzazione di moduli software per la visualizzazione e l'archiviazione di modelli tridimensionali ad elevata complessità

Tesi di
Gasparello Paolo Simone

Relatori:
Prof. Ancilotti Paolo .....

Prof. Avizzano Carlo Alberto .....

Dott. Ing. Rossi Fabio .....

Dott. Ing. Tecchia Franco .....

Sessione di Laurea 9 maggio 2007
Anno Accademico 2005-2006

*A tutti i miei familiari ed amici*

## Ringraziamenti

*Giunto al termine di questo percorso desidero ringraziare tutte le persone che hanno reso possibile lo svolgimento del mio lavoro di tesi, a partire dal prof. Massimo Bergamasco che mi ha permesso di svolgere questa tesi presso il laboratorio PERCRO della Scuola Superiore S. Anna.*

*Ringrazio tutti i componenti del laboratorio, gli ing. Franco, Davide, Sandro e Marcello per la disponibilità e la gentilezza con cui hanno saputo consigliarmi e rispondere a tutte le mie domande. Un ringraziamento particolare va all'ing. Fabio Rossi, che mi ha introdotto nel mondo della computer grafica ed ha ricoperto un ruolo fondamentale nel mio percorso formativo.*

*Ringrazio i miei genitori, che si sono adattati alla mia vita priva di orari... e tutti i miei amici e amiche che mi sono stati vicini e sono riusciti a sopportare i miei lunghi monologhi.*

*Ringrazio i miei compagni di facoltà e tutte le persone che hanno creduto in me fino in fondo.*

# Indice

# Elenco delle figure

# Elenco delle tabelle

# Capitolo 1

# Introduzione

## 1.1 Visualizzazione di Ambienti Virtuali

### 1.1.1 Realtà Virtuale

La *realtà virtuale* è un'esperienza immersiva multisensoriale. L'idea di fondo è di costruire dei mondi che non esistono realmente ma che forniscano sensazioni e stimoli in gradi di rendere verosimile ogni possibile situazione immaginaria, tanto da poter far credere a chi vive l'esperienza virtuale di essere parte integrante di quel mondo.

Affinché questo accada, un mondo di realtà virtuale deve rispondere a tutte le leggi della fisica alle quali siamo abituati da sempre e con le quali conviviamo quotidianamente. Un sistema di realtà virtuale, dovendo assolvere a funzioni quali la percezione dei movimenti dell'utente, il loro controllo e la risposta sensoriale, necessita delle seguenti tecnologie:

- rappresentazione grafica in tempo reale
- display stereoscopico
- rilevamento della posizione e della traiettoria di parti del corpo umano
- suono olofonico
- ritorno di forza e di sensazioni tattili

- sintesi ed analisi vocale

Per realizzare questi obiettivi, occorrono workstation capaci di eseguire in tempo reale tutti i calcoli necessaria produrre la risposta all'azione del soggetto, senza ritardi avvertibili da quest'ultimo. In particolare, l'elaborazione grafica 3D consente la creazione dell'ambiente virtuale e la visione stereoscopica fa sì che l'utente si senta immerso in tale ambiente, riuscendo a percepire la tridimensionalità degli oggetti che lo circondano.

Secondo Sheridan (1992) i fattori che caratterizzano la presenza in ambienti virtuali sono tre:

1. La quantità di informazione sensoriale diretta all'operatore
2. La possibilità di modificare i sensori umani nell'ambiente virtuale
3. La possibilità di controllare l'ambiente

La quantità di informazione che stimola le facoltà sensoriali dell'utente deve rendere più realistica possibile l'immersione nell'ambiente virtuale in cui egli si trova, coinvolgendolo completamente. Ad esempio le immagini in bianco e nero forniscono molte meno informazioni di quelle a colori, così come un'immagine ad alta definizione risulta più realistica della stessa in bassa risoluzione.

Il secondo fattore caratterizza la possibilità, da parte dell'operatore, di poter volgere l'attenzione ad eventi che lo circondano, come ad esempio inseguire con lo sguardo un oggetto in movimento o muovere la testa nella direzione di un suono.

L'interazione con l'ambiente è fortemente correlata con la possibilità di poterlo controllare. La manipolazione degli oggetti è fondamentale per la veridicità delle sensazioni virtuali: l'esplorazione, lo spostamento e la manipolazione degli oggetti sono azioni frequenti nella vita quotidiana ed è quindi fondamentale che siano riprodotti nell'Ambiente Virtuale.

Si può perciò parlare di Presenza Virtuale solo quando queste tre componenti vengono riprodotte in maniera ottimale. L'obiettivo è, allora, quello di mettere a punto sistemi di interazione fisica per il controllo dell'ambiente.

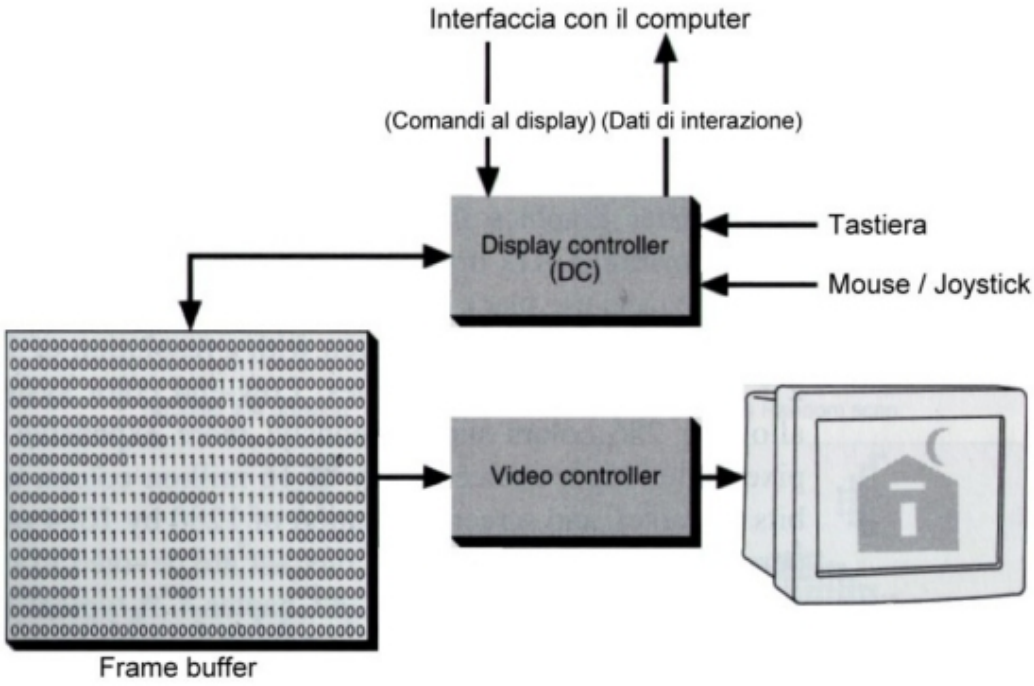



Figura 1.1: Architettura di un *raster display*

Il laboratorio PERCRO della Scuola S. Anna di Pisa opera in quest'ambito, usando tre approcci di sviluppo:

1. Sviluppo di sistemi di interfaccia e per il ritorno di forza e tattile
2. Sviluppo di rendering grafico ad alta definizione in tempo reale
3. Problematiche relative alla modellazione

### 1.1.2 Computer Graphics

Sotto il nome di *computer graphics* rientrano numerose discipline e tecniche apparentemente non molto correlate, che hanno spesso in comune il solo mezzo (il computer) con il quale vengono realizzate: si pensi alle interfacce grafiche di programmi e sistemi operativi, ai grafici scientifici e statistici, al CAD, ai sistemi multimediali, alle simulazioni.

Il substrato comune è costituito dal sistema video, un complesso insieme di controllori e interfacce che gestiscono la visualizzazione di oggetti e for-

me realizzate con svariate tecniche; tale sistema è comunemente identificato come *raster display* (Foley *e altri*, 1994). Un'immagine sul raster display è costituita dal *raster*, cioè un insieme di linee orizzontali dette *scanlines*, ognuna delle quali è formata da una riga di *pixel*. Per ciascun *pixel* il video controller invia un raggio luminoso la cui intensità varia a seconda del suo valore[1].

L'immagine generata è dunque composta da una matrice di pixel. Tale matrice, denominata *pixmap*, viene poi memorizzata in uno spazio di memoria centrale dedicato chiamato *frame buffer*. I raster display hanno nel tempo soppiantato la visualizzazione di tipo vettoriale a causa dei loro bassi costi e della possibilità di visualizzare immagini a colori, ma continuano a soffrire degli svantaggi legati ad una discretizzazione dell'immagine, approssimazione che causa effetti di scalettatura legati agli errori di campionamento[2].

### 1.1.3 La grafica tridimensionale

Rispetto alla grafica bidimensionale, nella grafica 3D si aggiunge un grado di complessità. La terza dimensione aggiunta introduce un insieme di problemi dovuti al fatto che i dispositivi video sono per loro natura 2D.

Un aiuto concettuale può essere quello di pensare ad una fotocamera mobile che, posta in un determinato punto dello spazio[3], realizza un'immagine bidimensionale, cioè un *fotogramma* o *frame* dell'ambiente o dell'oggetto 3D. Nella realtà la telecamera è un programma che produce l'immagine sullo schermo, e l'ambiente o l'oggetto 3D è un insieme di dati costituiti da punti, linee e superfici.

La creazione di ciascun *frame* passa attraverso l'attuazione di una serie di passi che riguardano:

1. Specifica del *tipo di proiezione*: nel seguito saranno considerate in particolare le due proiezioni più importanti, la proiezione *prospettica* e la

[1] in realtà nei sistemi a colori vi sono tre raggi, uno per ogni colore primario: *rosso*, *verde*, *blu*

[2] in teoria dei segnali questi effetti sono chiamati *aliasing*

[3] detto *punto di vista* oppure *viewpoint*

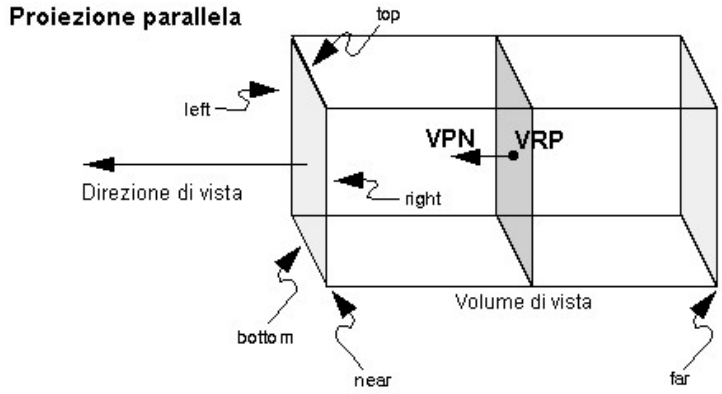



Figura 1.2: Proiezione *ortogonale*

proiezione *ortogonale*[4]. Nel primo caso la distanza tra il centro della proiezione ed il piano di vista è finita, nel secondo caso il centro della proiezione va all'infinito, tanto che è improprio parlare di centro della proiezione: si parla piuttosto di *direzione* di proiezione.

2. Specifica dei parametri di vista, e cioè: le coordinate del modello nel sistema di riferimento solidale al modello stesso (*object coordinates*) e la posizione dell'osservatore e del piano di vista (o *viewplane*[5]). Di norma si usano due sistemi di coordinate: le *object coordinates* ossia le coordinate solidali all'oggetto e le *eye coordinates*, che sono le coordinate dell'oggetto in un sistema dmi riferimento solidale all'osservatore.

3. *Clipping*, ovvero l'eliminazione di quella porzione della scena 3D che non rientra entro i confini della finestra grafica. Viene definito a tale scopo un volume di vista (*view volume*) differente a seconda del tipo di proiezione, sul quale vengono poi applicati gli algoritmi di clipping.

4. *Proiezione e visualizzazione*: il contenuto della proiezione del volume di vista sul piano di proiezione viene visualizzato su una zona rettangolare dello schermo, detta *viewport*.

[4]o proiezione *parallela*
[5]la superficie sulla quale l'immagine viene proiettata

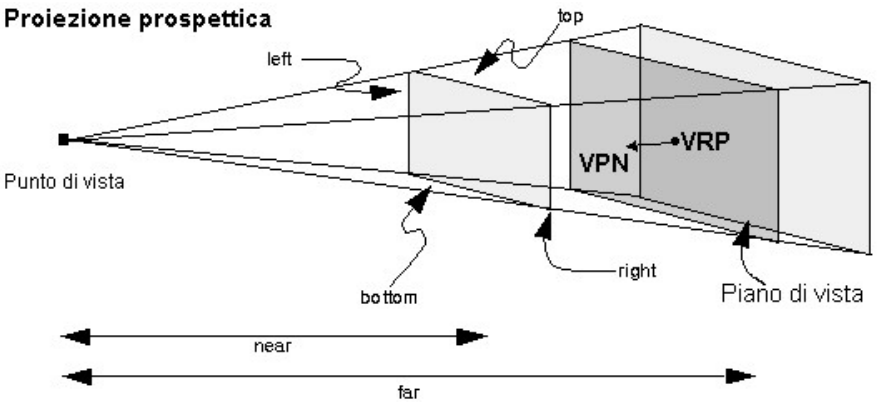



Figura 1.3: Proiezione *prospettica*

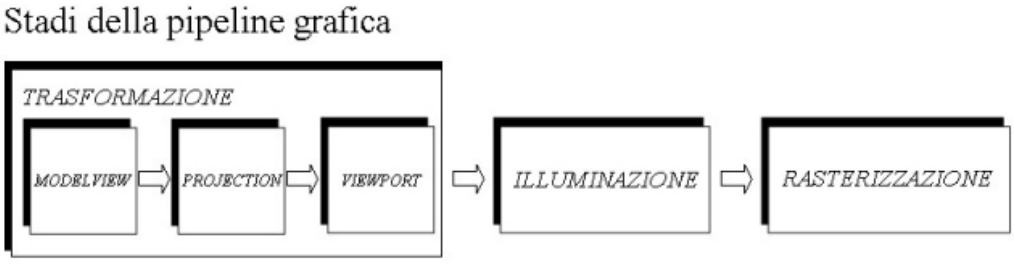



Figura 1.4: La *pipeline grafica*

Il piano di vista è definito mediante un punto giacente su di esso detto *VRP*[6] ed una normale al piano denominata *VPN*[7]; il volume di vista è la porzione della scena visibile dall'osservatore in base al tipo di proiezione utilizzata ed eventualmente all'apertura del campo di vista. Nel caso di proiezione prospettica il volume di vista è una piramide semi-infinita con vertice nell'osservatore, nel caso di proiezione parallela è un parallelepipedo infinito. Per rendere finito il volume di vista si ricorre a due ulteriori piani di *clipping*, i quali specificano una distanza minima ed una distanza massima oltre la quale la geometria non è più visibile; tali piani prendono rispettivamente il nome di *front* o *near plane* e di *back* o *far plane*.

La catena di trasformazioni che i dati geometrici devono attraversare affinché la rappresentazione matematica diventi un'immagine *rendered* prende

[6] *View Reference Point*
[7] *View Plane Normal*

il nome di *pipeline grafica* (figura 1.4 nella pagina precedente). Essa può essere suddivisa in tre stadi: *trasformazione*, *illuminazione* e *rasterizzazione*.

Lo stadio di *trasformazione* è a sua volta suddiviso in altri tre sottostadi. Avendo come input le coordinate di un vertice dell'oggetto nel sistema di coordinate solidali all'oggetto stesso, si passa al sistema di coordinate solidali all'osservatore mediante la trasformazione[8] di *modelviewing*. Tale trasformazione può essere considerata come l'azione di posizionare la camera e l'oggetto nella scena (figura 1.5 nella pagina successiva). Segue poi la trasformazione di *projection*, che definisce il volume di vista ed esclude le porzioni non visibili della scena. Infine nella trasformazione di *viewport* le coordinate 3D vengono convertite in *window coordinates*, ovvero in coordinate bidimensionali indicanti la posizione del punto sullo schermo. Le *window coordinates* sono, in effetti, tre. La terza coordinata, la $z$, è utilizzata per risolvere correttamente la visibilità degli oggetti ed è memorizzata in un'apposita area di memoria: lo *Z-buffer*. Questo perché più punti 3D possono essere mappati nello stesso punto sullo schermo, ma solamente quello più vicino all'osservatore deve risultare effettivamente visibile.

Lo stadio di illuminazione implementa la corretta determinazione del colore di ogni *pixel* sullo schermo, basandosi su informazioni relative ad i materiali e sul tipo e posizione delle sorgenti di luce nella scena. Quello che viene calcolato è un contributo di colore che viene poi combinato con altri contributi derivanti dalle eventuali informazioni sulle *textures*.

Lo stadio di *rasterizzazione*, infine, gestisce tutte le operazioni effettuate sul singolo *pixel*, cioè operazioni 2D come il texture mapping, l'alpha blending e lo Z-buffering.

Gli acceleratori grafici di ultima generazione permettono di gestire tutte le fasi sopra descritte della pipeline grafica, a differenza dei modelli antecedenti che erano in grado di gestire la sola fase di *rasterizzazione*, lasciando alla CPU il carico delle operazioni di *trasformazione* ed *illuminazione*. Per questo motivo, con le ultime schede grafiche è nato il concetto di *GPU*[9] dal momento

[8] ovvero una moltiplicazione matriciale

[9] *Graphic Processor Unit*

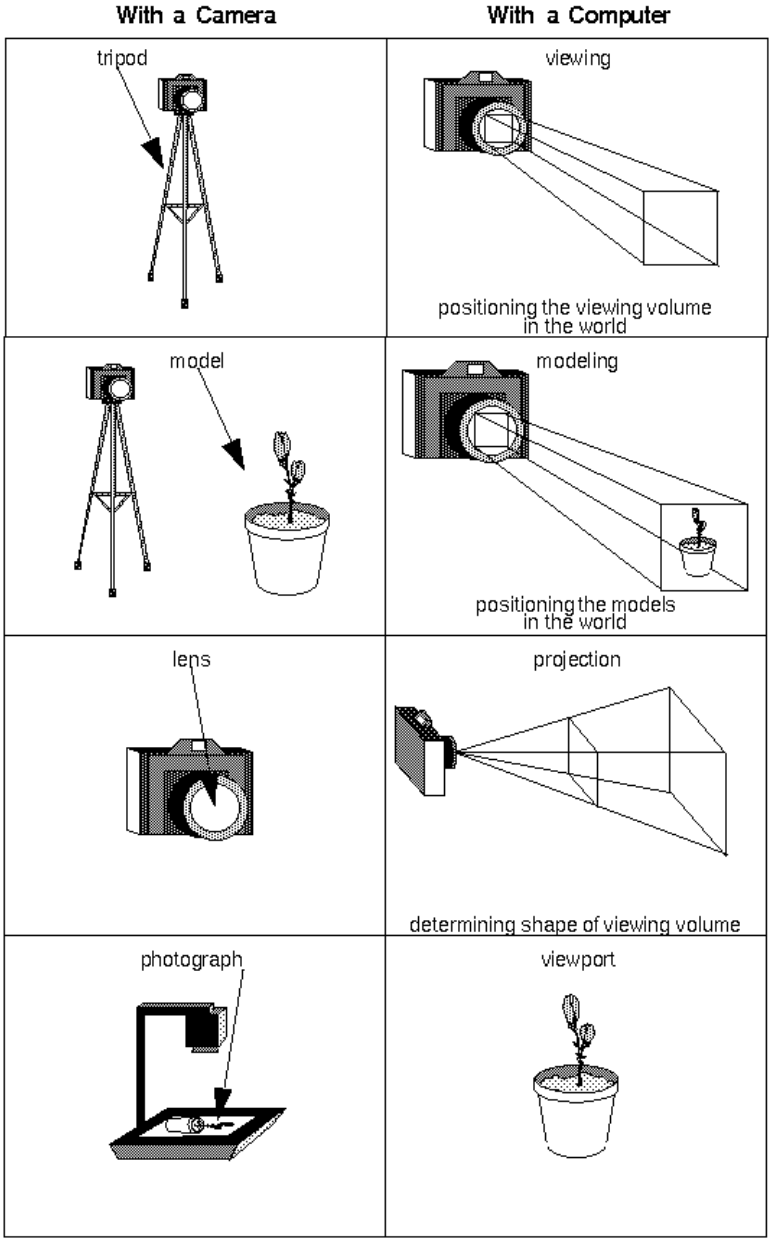



Figura 1.5: Analogia tra una fotocamera e la pipeline grafica

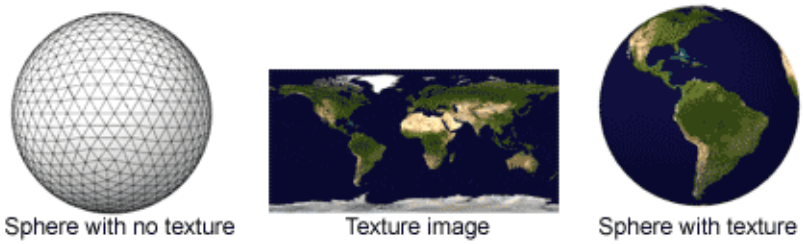



Figura 1.6: Applicazione di una *texture*

che tali dispositivi possono essere considerati come dei veri e propri processori dedicati.

Il modello di illuminazione usato nelle moderne pipeline grafiche è quello di Phong (1975), divenuto uno standard nella computer grafica per il buon compromesso tra qualità finale e costo computazionale. Secondo tale modello, la riflessione della radiazione luminosa viene modellata in termini di tre componenti additive: *diffusa*, *speculare* ed *ambiente*, ognuna definita sia per le sorgenti luminose che per i materiali della scena, assumendo in questo caso il significato di riflessività.

- *Luce ambiente*: proviene ed è riflessa in tutte le direzioni dello stesso modo. Per tale motivo colpisce ogni punto con la stessa intensità indipendentemente dalla sua normale. Non è utile per creare effetti 3D.

- *Luce diffusa*: proviene da una direzione particolare ed è riflessa in tutte le direzioni. L'intensità con cui illumina un punto dipende dall'angolo di incidenza tra la normale in tal punto e la direzione di provenienza della luce. Questa componente determina gli effetti 3D.

- *Luce speculare*: proviene da una direzione particolare e viene riflessa in una direzione particolare. Serve a creare l'effetto luccicante tipico dei materiali metallici.

Per ogni primitiva deve essere specificato un materiale, i cui parametri possono essere applicati uniformemente alla superficie della primitiva, o modulati da una tessitura o *texture* (figura 1.6).

La *texture* è una matrice rettangolare di dati, ad esempio colore e luminanza o colore e *componente alpha*[10]. I singoli dati della matrice vengono detti *texels*. In altre parole, si tratta di un'immagine bidimensionale le cui informazioni vengono associate (*mapped*) alle primitive geometriche, con le quali condividono tutte le trasformazioni cui tali primitive sono soggette. Il vantaggio dell'uso delle textures è rappresentato dal fatto che, indipendentemente dalla complessità della geometria che sostituiscono, necessitano di un tempo costante per la resa grafica. Combinate con la geometria, le textures sono in grado di migliorare notevolmente il grado apparente di dettaglio di un modello, senza aumentare il numero di primitive geometriche utilizzate[11].

Nella grafica 3D i modelli sono solitamente rappresentati usando *mesh di triangoli*[12]. Dal punto di vista geometrico una mesh è una superficie lineare a tratti, formata da facce triangolari unite tra loro. Agli elementi di una mesh possono essere associati, oltre che informazioni geometriche, anche informazioni di altro genere che possono essere suddivise in due categorie: gli *attributi scalari* come le normali o le coordinate delle textures, e gli *attributi discreti*, ad esempio il materiale.

### 1.1.4 OpenGL e grafica interattiva

Affinché un'applicazione possa accedere alla pipeline grafica è necessaria una libreria od API[13] che si occupi del compito di gestire l'hardware sottostante, sollevando in tal modo l'applicazione dal colloquio diretto con i *driver* del dispositivo.

Nell'ambito della visualizzazione grafica esistono varie librerie preposte a questo scopo. Alcune di esse sono legate direttamente ad uno specifico tipo di hardware[14] e sono configurate per funzionare solo con tale prodotto.

[10] l'*alpha channel* è un'area di memoria contenente informazioni usate per effetti di trasparenza o traslucenza

[11] si veda a tal scopo la tecnica del *bump mapping* (Kilgard, 2000) e del *normal mapping* (Cignoni *e altri*, 1998a)

[12] vedi la sezione 1.3 nella pagina 19

[13] *Application Programming Interface*

[14] ne è un esempio le *Glide* di 3dFX

Il codice scritto usando queste librerie non è adattabile, in questo caso, ad altre piattaforme.

Altre librerie sono invece legate ad uno specifico sistema operativo. Un esempio è costituito dalle librerie *DirectX*, realizzate da Microsoft. Il codice è in questo caso indipendente dall'hardware usato, ma non è portabile su piattaforme differenti, come Unix ad esempio.

Infine vi sono alcune librerie indipendenti sia dall'hardware che dalla piattaforma. Una di queste è la libreria *OpenGL*[15], introdotta dalla Silicon Graphics e della quale esiste un'implementazione per tutti i maggiori sistemi operativi esistenti. La larga diffusione di questa libreria ha fatto in modo che la grande maggioranza dell'attuale hardware grafico supporti direttamente da hardware le funzionalità che OpenGL mette a disposizione.

OpenGL offre una serie di funzioni dedicate alla grafica 3D, ma è in grado di trattare anche il 2D, considerato come un suo sottoinsieme. Per mantenere la sua indipendenza dalla piattaforma usata, OpenGL non fornisce alcuna funzione di gestione della finestra di visualizzazione. Per questo scopo l'applicazione deve fare ricorso alle funzioni dello specifico sistema operativo oppure usare librerie *platform independent* di terze parti, come ad esempio le *GLUT*[16].

Le funzioni messe a disposizione da OpenGL permettono di tracciare semplici primitive geometriche come *punti*, *linee* e *triangoli*, oppure poligoni convessi composti da un numero arbitrario di lati o curve espresse analiticamente e, mediante un'estensione della libreria, anche superfici *complesse*[17]. Nel trattare queste entità più complesse è la libreria che si fa carico di comporle in primitive semplici quali linee e poligoni. Dal momento che tali superfici curve richiedono un grosso carico computazionale, nella grafica tridimensionale interattiva, le entità che vengono effettivamente usate restano i poligoni, soprattutto i *triangoli*.

Le trasformazioni geometriche necessarie per la visualizzazione sono specificate dall'applicazione ad OpenGL mediante tre matrici: la *model matrix*,

[15] la cui homepage è http://www.opengl.org/

[16] *OpenGL Utility Toolkit*, una cui implementazione OpenSource è disponibile su http://freeglut.sourceforge.net/

[17] NURBS: Non Uniform Rational B-Spline

la *view matrix* e la *perspective matrix*. Le prime due permettono di realizzare la trasformazione di *modelviewing*, la terza consente di effettuale la *proiezione prospettica* delle primitive. Una volta eliminate le primitive che non rientrano nel volume di vista, alle rimanenti viene applicata la trasformazione di *viewport*, con la quale si ottengono le coordinate in pixel per la visualizzazione sullo schermo.

Per specificare l'illuminazione della scena OpenGL prevede tre tipi di sorgenti luminose: *puntiformi*, *direzionali* e di tipo *spot*. La luce puntiforme proviene da una sorgente posta in un determinato punto dello spazio ed emette raggi in ogni direzione. La luce direzionale proviene da una sorgente che si suppone posta all'infinito e con direzione assegnata: i raggi emessi sono così tutti paralleli tra loro. La luce di tipo spot è analoga a quella puntiforme, ma il campo di emissione è ristretto ad un cono del quale è specificata l'apertura angolare.

OpenGL consente infine di tracciare le primitive con materiali uniformi od usando un'immagine come texture.

Le specifiche di OpenGL cono state decise e periodicamente riesaminate da un comitato indipendente di produttori software e hardware (*ARB*[18]), i cui membri permanenti sono: Digital Equipment, IBM, Intel, Microsoft e Silicon Graphics. Tale comitato ratifica le variazioni alle specifiche della libreria, l'uscita di nuove versioni e di test di conformità che le varie implementazioni devono soddisfare.

## 1.2 Rappresentazione delle superfici

Una *superficie poligonale* consiste in un insieme di poligoni locati in uno spazio euclideo tridimensionale $\mathbb{R}^3$. Nel caso più generale è possibile assumere che tali poligoni siano dei *triangoli*, dal momento che qualsiasi faccia avente almeno quattro lati può essere triangolata (Narkhede e Manocha, 1995), ma non viceversa. Tale assunzione comporta numerosi vantaggi in termini di modellazione ed elaborazione per la *computer grafica*[19] ed inoltre ha l'im-

[18] *Architecture Review Board*

[19] si veda la sezione 1.3 nella pagina 19

Figura 1.7: Esempi di simplessi di ordine rispettivamente 0, 1, 2 e 3

portante conseguenza che una superficie poligonale può essere formalizzata utilizzando la teoria dei *complessi simpliciali* (Alexandroff, 1961).

In questa sezione saranno presentate le definizioni relative ad i concetti di tale teoria che saranno utilizzate nei capitoli successivi.

### 1.2.1 Simplessi e complessi simpliciali

#### Definizione di k-simplesso

Si definisce un *k-simplesso* $\sigma^k$ come la combinazione convessa di $k+1$ punti linearmente indipendenti di uno spazio euclideo $\mathbb{R}^n$, con $n \geq k$. $k$ rappresenta l'*ordine* del simplesso, mentre i punti sono denominati *vertici*.

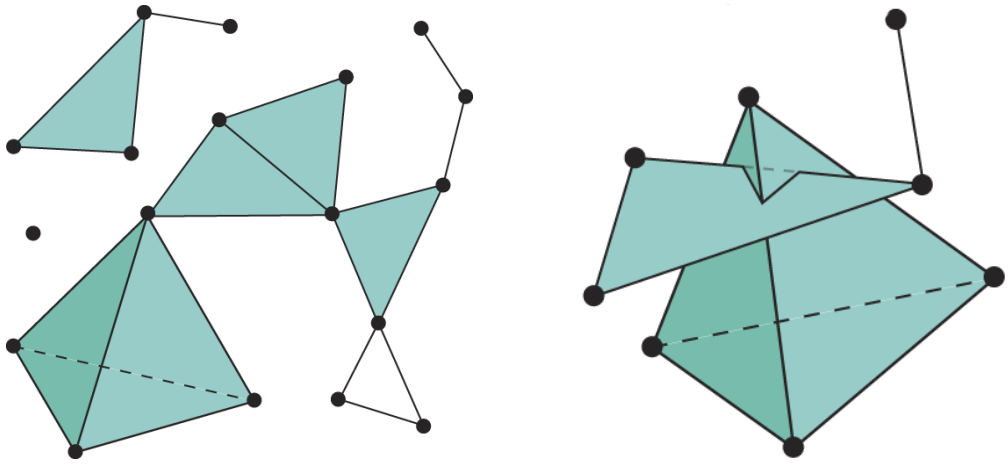
Ad esempio, uno 0-simplesso è un *punto*, un 1-simplesso è un *segmento*, un 2-simplesso è un *triangolo*, un 3-simplesso è un *tetraedro*, e così via (figura 1.7).

#### Facce e co-facce

Ogni simplesso $\sigma$ definito da un sottoinsieme di vertici di $\sigma^k$ è detto *faccia* di $\sigma^k$ e si indica con $\sigma \leq \sigma^k$. Se $\sigma \neq \sigma^k$ tale faccia è detta *propria*. La relazione reciproca è denominata *co-faccia*: in questo caso $\sigma^k$ è co-faccia di $\sigma$ ($\sigma^k \geq \sigma$).

#### Definizione di k-complesso simpliciale

Un *k-complesso simpliciale* $\Sigma$ è definito come una collezione di simplessi che soddisfano le seguenti condizioni:



(a) Complesso simpliciale *valido* (b) Complesso simpliciale *non valido*

Figura 1.8: Esempi di complessi simpliciali

- Per ogni coppia $\sigma_i, \sigma_j \in \Sigma$ deve risultare che $\sigma_i \cap \sigma_j = \emptyset$ oppure $\sigma_i \cap \sigma_j \in \Sigma$. In altre parole, due spigoli possono solo intersecarsi in un vertice comune, oppure due triangoli possono intersecarsi solo lungo uno spigolo od un vertice condiviso.
- Per ogni simplesso $\sigma_i \in \Sigma$, tutte le facce $\sigma_j$ di $\sigma_i$ appartengono a $\Sigma$.

Il grado $k$ del complesso simpliciale $\Sigma$ è uguale al grado massimo dei simplessi $\sigma_i$ che lo compongono.

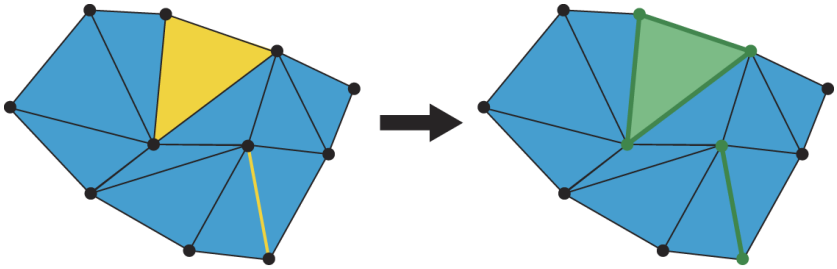
### Chiusura, star e link di un complesso simpliciale

La *chiusura* $C$ del complesso simpliciale $\Sigma$, indicata con $Cl(\Sigma)$, è definita come il più piccolo complesso simpliciale che contenga tutte le facce di $\Sigma$. Siano $\sigma$ i simplessi che compongono $\Sigma$:

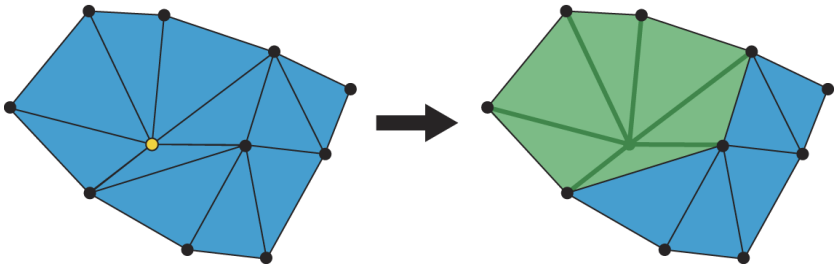
$$Cl(\Sigma) = \{\tau \in C | \tau \leq \sigma \in \Sigma\} \tag{1.1}$$

La *star* $S$ di $\Sigma$ è definita invece come l'insieme delle co-facce di $\Sigma$:

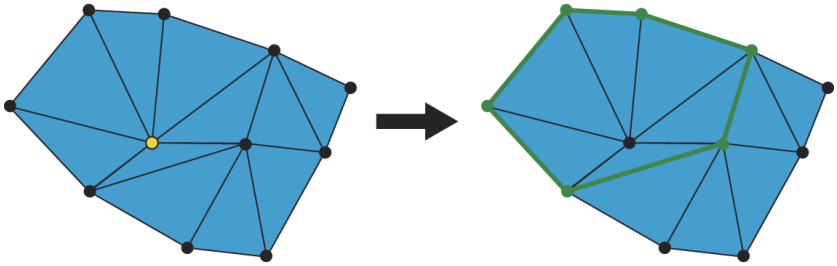
$$St(\Sigma) = \{\tau \in S | \tau \geq \sigma \in \Sigma\} \tag{1.2}$$



(a) *Chiusura* delle parti colorate in giallo



(b) *Star* del punto colorato in giallo



(c) *Link* del punto colorato in giallo

Figura 1.9: Chiusura, star e link di complessi simpliciali

Si definisce inoltre *link* $L$ di $\Sigma$ come il contorno di $St(\Sigma)$:

$$Lk(\Sigma) = Cl(St(\Sigma)) - St(Cl(\Sigma) - \emptyset) \tag{1.3}$$

Un esempio esplicativo di queste ultime definizioni è illustrato nella figura 1.9.

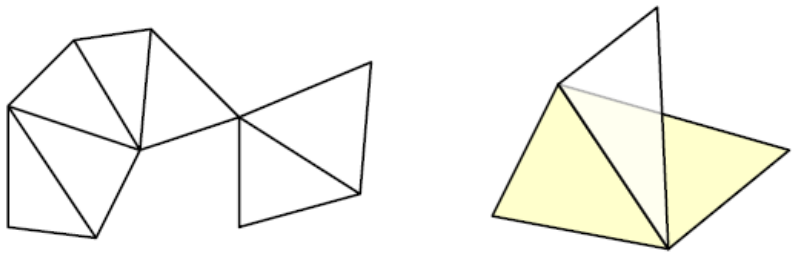


Figura 1.10: Esempi di superfici *non-manifold*

### 1.2.2 Mesh come complessi simpliciali

Grazie alle definizioni fornite nella precedente sezione è facile cogliere l'analogia tra una superficie di triangoli in $\mathbb{R}^3$ ed un 2-complesso simpliciale. Di seguito sono illustrate ulteriori definizioni atte a caratterizzare la topologia delle superfici.

#### Superfici manifold e non manifold

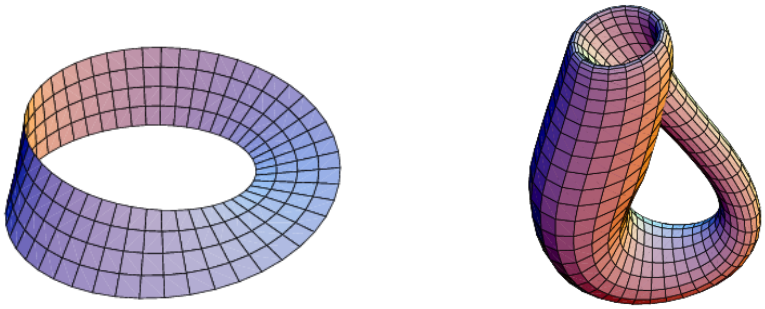
Un *manifold* è una superficie $\Sigma \in \mathbb{R}^3$ tale che ogni punto su $\Sigma$ ha un intorno omeomorfo ad un disco aperto od ad un semi-disco aperto in $\mathbb{R}^2$. In altre parole, ciascuno spigolo delle superficie poligonale deve avere esattamente due facce incidenti; ovviamente, se la superficie presenta dei bordi, gli spigoli che ne fanno parte devono avere una sola faccia incidente. Eventuali spigoli e vertici che non sono conformi a tali proprietà vengono detti *degeneri* (figura 1.10).

Si può generalizzare il concetto di *manifold* per superfici appartenenti a spazi euclidei con dimensione maggiore di tre: un complesso simpliciale è un *n-manifold* se tutti i suoi punti sono omeomorfi ad una n-sfera aperta.

I *manifold* sono molto importanti nella *computer grafica* poiché molte strutture dati ed algoritmi possono gestire soltanto superfici aventi tale topologia.

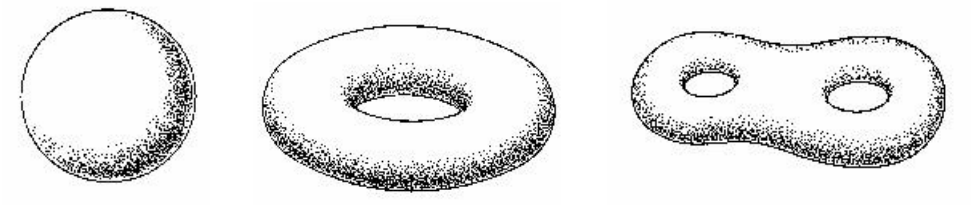
#### Superfici orientabili

Una superficie si dice *orientabile* se è possibile determinare una normale in ciascun punto in modo coerente. Esempi di superfici orientabili sono sfere,



(a) Nastro di Möbius (b) Bottiglia di Klein

Figura 1.11: Esempi di superfici non orientabili



(a) Sfera: *genus* 0 (b) Torus: *genus* 1 (c) Otto: *genus* 2

Figura 1.12: Superfici con differente *genus*

piani o tori; esempi di superfici non orientabili sono invece il nastro di Möbius, la bottiglia di Klein o dei non-manifold (figura 1.11).

### Genus di una superficie

Si definisce *genus* di una superficie connessa, chiusa, orientata e 2-manifold quel numero intero che rappresenta il massimo numero di tagli lungo curve chiuse semplici che è possibile effettuare senza rendere il manifold sconnesso (figura 1.12). Una definizione meno formale indica il *genus* come il numero di maniglie della superficie.

### Caratteristica di Eulero

Sia $\Sigma$ un complesso simpliciale composto da $s_k$ $k$-simplessi. La *caratteristica di Eulero* è definita come:

$$\chi(\Sigma) = \sum_{i=0}^{dim(\Sigma)} (-1)^i s_i = \sum_{\sigma \in \Sigma - \emptyset} (-1)^{dim(\sigma)} \tag{1.4}$$

Restringendosi al caso di un 2-complesso simpliciale si ottiene che[20]:

$$\chi(\Sigma) = v - e + f \tag{1.5}$$

Dove $v$ è il numero di vertici, $e$ il numero di spigoli ed $f$ il numero di facce del 2-complesso simpliciale. La caratteristica di Eulero rappresenta un'invariante di $\Sigma$, vale a dire che tale quantità non dipende dalla particolare triangolazione utilizzata, ma soltanto dalla topologia della superficie. Vale infatti la seguente equivalenza:

$$\chi(\Sigma) = 2(c - g) - b \tag{1.6}$$

Dove $c$ è il numero di componenti non connesse della mesh, $g$ è il genus della superficie e $b$ il suo numero di bordi. Poiché in una mesh di triangoli manifold e chiusa ciascuno spigolo ha esattamente due triangoli incidenti e ciascun triangolo ha tre spigoli incidenti, si ha che $2e = 3f$. Sostituendo tale espressione nell'equazione (1.5) si ottiene:

$$\chi = v - \frac{3}{2}f + f = v - \frac{1}{2}f \Rightarrow 2\chi = 2v - f \tag{1.7}$$

Dal momento che la caratteristica $\chi$ è tipicamente un valore molto piccolo, per mesh con un numero elevato di vertici si ha la relazione notevole (Gotsman *e altri*, 2001):

$$f \approx 2v \tag{1.8}$$

[20] si può trovare una dimostrazione su
http://en.wikipedia.org/wiki/Euler_characteristic

## 1.3 Mesh di triangoli

Nell'ambito delle applicazioni grafiche interattive, le *mesh poligonali* costituiscono il modo più comune di rappresentare i modelli tridimensionali. Una mesh poligonale è costituita da una *collezione di vertici e di poligoni* che definiscono la forma di un *poliedro* nello spazio. Le mesh poligonali riescono ad approssimare qualsiasi tipo di superficie e permettono di visitare e processare gli elementi del modello molto efficientemente qualora si vogliano eseguire elaborazioni su di esso.

Una classe particolare di mesh poligonali spesso usata nella computer grafica è la *mesh di triangoli*, dove cioè i poligoni che la costituiscono devono essere esclusivamente dei triangoli. Tale tipologia di mesh è largamente diffusa poiché l'hardware dedicato adotta il triangolo come una primitiva grafica. Di conseguenza una grande porzione del lavoro di ricerca in computer grafica riguarda elaborazioni applicabili a superfici così rappresentate.

### 1.3.1 Strutture dati

Le strutture dati atte ad organizzare gli elementi di una mesh poligonale si pongono generalmente due obiettivi: ottenere la massima *flessibilità* e la massima *efficienza*.

Per *flessibilità* si intende la capacità di fornire delle interfacce per poter accedere in qualunque momento ad un arbitrario elemento della mesh. Un altro aspetto che rende una struttura dati *flessibile* è quello di offrire la possibilità di rappresentare ogni genere di mesh, indipendentemente dal tipo di poligoni che la compongono.

Inoltre, una struttura dati deve essere *efficiente*, sia in termini di *spazio* di memoria necessario che di *tempo* di accesso agli elementi durante la sua navigazione.

Una mesh poligonale consiste di un insieme di *vertici*, *spigoli*, *facce* e delle *relazioni topologiche* che legano tali elementi. Le strutture dati che li possono contenere sono raggruppate in due categorie: quelle *face-based* od *edge-based*.

Nelle strutture *face-based* ogni poligono è memorizzato con una struttura contenente i puntatori ad i vertici che lo compongono ed eventualmente i

puntatori alle facce adiacenti. Questa semplice rappresentazione, sebbene possa essere efficiente in talune situazioni (ad esempio nella visualizzazione), comporta un'elevata complessità di calcolo nella gestione di mesh composte da poligoni con un numero variabile di lati (Botsch *e altri*, 2002).

### 1.3.2 Winged-edge data structure

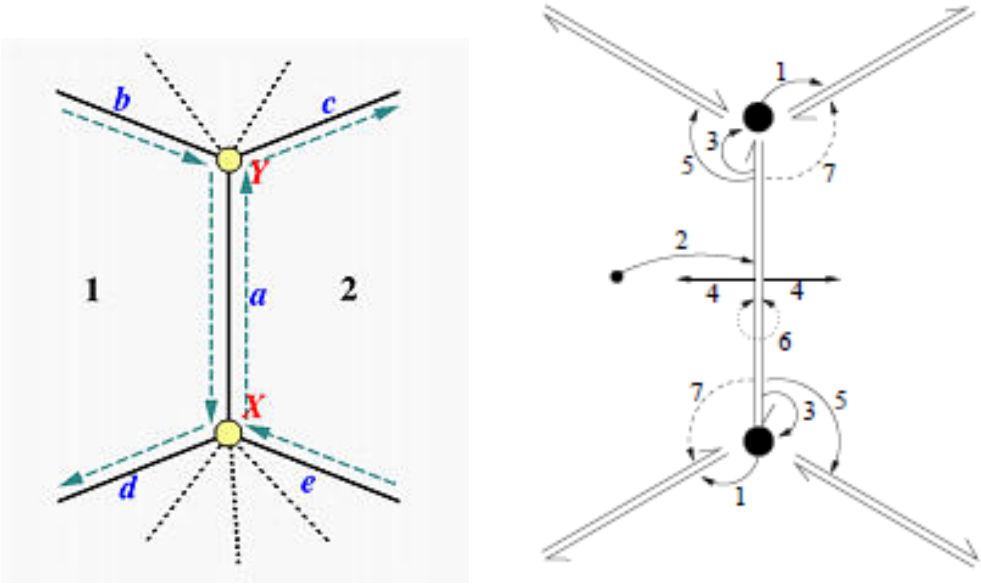
Per superare questo problema sono stati sviluppati altri tipi di *b-rep*[21] più complessi di tipo *edge-based*. Un metodo comune di tale rappresentazione è la *winged-edge data structure* (Baumgart, 1975), (Weiler e Electric, 1985). Con questa tecnica le informazioni relative alla connettività sono memorizzate in ciascuno degli oggetti che rappresentano uno spigolo. In particolare ogni spigolo tiene un puntatore ad i seguenti elementi:

1. i due vertici ai suoi estremi
2. le due facce adiacenti, chiamate rispettivamente *faccia destra* e *faccia sinistra*
3. lo spigolo che lo precede e quello che lo succede nella faccia destra
4. lo spigolo che lo precede e quello che lo succede nella faccia sinistra

Nella figura 1.13(a) nella pagina seguente sono evidenziati gli elementi che devono essere puntati dallo spigolo $a$. Affinché tutti gli elementi siano raggiungibili tra loro, ogni vertice ed ogni faccia deve inoltre possedere un puntatore ad uno degli spigoli ad esso incidente.

Questa struttura permette di riferire con un tempo costante[22] solo ad i vertici ed alle facce adiacenti ad uno spigolo. Per altri tipi di interrogazioni[23] sono necessari calcoli più dispendiosi. Un altro problema della winged-edge data structure è rappresentato dal fatto che gli spigoli non sono orientati esplicitamente, quindi ad ogni accesso è necessario riconoscere quale sia l'estremità dello spigolo interessata.

[21] *boundary representation*
[22] o meglio, un tempo costante per ciascuna informazione ottenuta
[23] ad esempio le facce adiacenti ad una faccia data




(a) Winged-edge data structure (b) Half-edge data structure

Figura 1.13: Strutture dati di tipo *edge-based*

### 1.3.3 Half-edge data structure

L'*half-edge data structure*[24] (Kettner, 1999) è un'altra *b-rep* per mesh poligonali in cui ogni spigolo è considerato come l'insieme di due entità, chiamate appunto *half-edges*, ciascuna delle quali punta verso una sua estremità. Ogni spigolo risulta così *orientato* ed il suo verso è determinato dal fatto che vi si riferisca con uno o l'altro half-edge che lo compongono. Oltre a superare questo limite esistente nella winged-edge data structure, con la HDS si riescono ad ottenere risultati eccellenti con tempi di accesso costanti per la maggior parte delle comuni interrogazioni.

Utilizzando la HDS una mesh poligonale viene rappresentata con un insieme di *vertici*, *half-edges* e *facce*. Gli elementi centrali della struttura sono ovviemente gli half-edges, che devono essere collegati tra di loro e con tutti gli altri elementi della mesh come mostrati nella figura 1.13(b):

1. *vertice* $\mapsto$ uno degli half-edges uscenti da esso

[24] può essere abbreviata con la sigla *HDS*

2. *faccia* $\mapsto$ uno dei suoi half-edge

3. *half-edge* $\mapsto$ il vertice al quale esso punta (*target vertex*)

4. *half-edge* $\mapsto$ la *faccia* alla quale appartiene[25]

5. *half-edge* $\mapsto$ l'half-edge che lo segue nella sua faccia

6. *half-edge* $\mapsto$ l'half-edge *opposto*, cioè la sua rispettiva metà

7. *half-edge* $\mapsto$ l'half-edge che lo precede nella sua faccia

Il collegamento numero (7) è disegnato con una linea tratteggiata poiché ridondante. L'half-edge precedente può essere infatti ricavato percorrendo la catena degli half-edges successivi. In particolare, se la mesh è composta da soli triangoli, dato un qualunque half-edge $h$ sussiste la relazione: $previous(h) = next(next(h))$.

Un'altra relazione importante che deve essere verificata è la seguente:

$$\forall h : opposite(opposite(h)) = h \tag{1.9}$$

La condizione (1.9) ha il significato che uno spigolo può essere rappresentato da al più due *half-edges* che siano rispettivamente opposti. Questo fatto limita la topologia delle superfici rappresentabili: la HDS non è adatta per alcuni tipi di superfici *non-manifold*[26]. Un superficie *non-manifold* può presentare due forme di elementi degeneri:

- *vertici complessi*: sono i vertici il cui intorno non è omomorfo ad un disco[27]

- *spigoli complessi*: sono quelli spigoli che appartengono a più di due facce

[25] mentre un *edge* è condiviso dalle facce ad esso adiacenti, un *half-edge* appartiene esclusivamente ad un'unica faccia

[26] si veda la sezione 1.2.2 nella pagina 16

[27] od ad un semi-disco se il vertice considerato sta sul bordo

Da queste definizioni si intuisce facilmente che non è possibile rappresentare mesh aventi *spigoli complessi* con una HDS. Strutture dati più complesse capaci di gestire superfici non-manifold sono la *radial edge structure* (Weiler, 1988) o la *partial entity structure* (Lee e Lee, 2001).

## 1.4 Gestione di modelli di grandi dimensioni

Nel campo della computer grafica si ha spesso a che fare con modelli o scenari composti da molti milioni di poligoni. Nonostante il grande incremento di prestazioni verificatosi negli ultimi anni relativamente ad i processori grafici, la visualizzazione di modelli di tali dimensioni, quando l'applicazione richiede un certo grado di interattività e di rapidità dei tempi di risposta, rende necessario il ricorso a particolari tecniche di ottimizzazione. E' riportata di seguito una panoramica di tali tecniche.

### 1.4.1 Tecniche di riduzione della complessità

Di questa categoria fanno parte tutti i metodi e gli algoritmi attraverso i quali i modelli subiscono delle trasformazioni finalizzate ad ottenere una versione dell'originale costituita da un numero minore di poligoni; in questo modo si riesce ad alleggerire sia il carico della GPU che la memoria necessaria a memorizzare i modelli, tipicamente a scapito della loro qualità visiva.

Tali strategie sono generalmente conosciute sotto il nome di *tecniche di semplificazione*, a cui è dedicato l'intero capitolo successivo.

Fa parte delle tecniche di riduzione della complessità anche l'uso di *texture*[28], con le quali è possibile incrementare il dettaglio di un modello tridimensionale senza accrescerne il numero di primitive grafiche, e quindi la complessità.

[28]si pensi al *texture mapping*, e soprattutto al *bump mapping* (sezione 1.1.3)

### 1.4.2 Tecniche di ottimizzazione eseguite a run-time

Queste tecniche consistono in una serie di strategie atte a minimizzare il numero di primitive grafiche da inviare alla pipeline della scheda video a seconda del valore di alcuni parametri variabili nel tempo e che sono calcolabili solo a *run-time* per ciascun frame. Esistono ad esempio criteri di ottimizzazione che operano sulla base di una valutazione della porzione di scena che rientra nel campo visivo o della distanza che sussiste tra il soggetto e l'osservatore.

#### Gestione dei livelli di dettaglio

Un ottimo metodo finalizzato a diminuire il carico del processore grafico è quello di utilizzare modelli forniti in diverse copie aventi differente livello di dettaglio[29]: è così possibile, ad esempio, inviare alla pipeline grafica versioni del soggetto composte da un numero inferiore di primitive se esso si trova in una posizione molto lontana dall'osservatore. Per ulteriori dettagli e riferimenti relativi a questa tecnica e su come possono essere creati i modelli multi-risoluzione si veda la sezione 2.1.1 nella pagina 28.

#### Visibility e Occlusion culling

La determinazione della porzione del campo visivo è da sempre un problema fondamentale della computer grafica. La sua importanza è stata evidenziata soprattutto negli ultimi anni a causa del crescere della complessità dei modelli utilizzati nelle applicazioni interattive, il cui *rendering* in tempo reale effettuato con gli approcci classici è divenuto praticamente impossibile.

Per *visibility culling* si intende l'insieme delle tecniche con cui è possibile scartare rapidamente il maggior numero di primitive che non sono visibili, cioè che non contribuiscono al colore di nessun pixel dello schermo. Le due tecniche classiche di *visibility culling* sono il *view-frustum culling* (Foley, 1995) ed il *back-face culling*.

[29] *modelli multi-risoluzione*

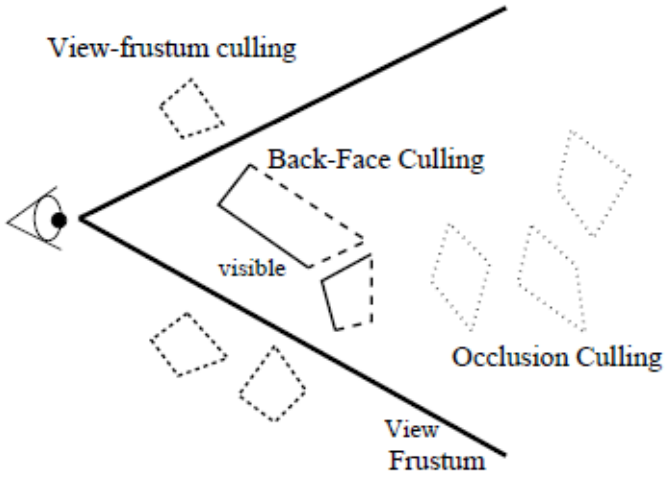



Figura 1.14: Tipologie di *culling*

- *View-frustum culling*: consiste nell'individuare e scartare tutte le primitive che risultano esterne al *frustum*, cioè al tronco di cono che costituisce il campo visivo in una proiezione prospettica (sezione 1.1.3 nella pagina 4). Una implementazione *conservativa* di tale tecnica prevede di raggruppare le primitive di uno stesso oggetto in un'unica entità e di eseguire il test sulla sua *bounding box* o *bounding sphere*. Nel caso in cui lo scenario presenti un numero troppo elevato di oggetti, effettuare il test su ciascuno di essi può rappresentare un overhead non sostenibile. Il problema può essere evitato organizzando gli oggetti in una struttura *gerarchica* (James, 1976), (Slater e Chrysanthou, 1997): l'overhead è così compensato quando grossi rami dell'albero degli oggetti sono scartati.

- *Back-face culling*: gli algoritmi che implementano questa tecnica scartano le facce che sono rivolte nel verso opposto alla direzione di vista. Questa tecnica è realizzata in hardware nei moderni dispositivi grafici ed è attivabile tramite il comando OpenGL `glEnable(GL_CULL_FACE)`. Per ciascun poligono è calcolata l'equazione del piano su cui esso giace: se la distanza tra tale piano ed il punto di vista è negativa, il poligono in questione può essere scartato.

Le tecniche di *occlusion culling* hanno invece lo scopo di scartare le primitive la cui visibilità è ostruita da altre parti della scena. Si tratta di strategie di tipo *globale*, cioè che coinvolgono l'intera scena ed il cui risultato dipende dalle posizioni relative tra i vari oggetti che la compongono. Eseguire il test di occlusione per ciascun poligono può essere quindi un'operazione molto costosa, pertanto, come nel caso del *view-frustum culling*, gli algoritmi di *occlusion culling* operano su oggetti organizzati in strutture gerarchiche (Garlick *e altri*, 1990). Uno degli algoritmi di occlusion culling più usati è lo *hierarchical Z-buffering* (Greene *e altri*, 1993).

Le tecniche presentate in questa sezione sono di solito usate congiuntamente in modo da ottimizzare il più possibile il rendering della scena: vengono prima selezionati gli oggetti contenuti nel volume di vista con un test di *view-frustum culling*; ad essi è poi applicato un algoritmo di *occlusion culling* fino ad ottenere un insieme di oggetti potenzialmente visibili. Da ultimo, tali oggetti sono inviati alla pipeline grafica che ne effettuerà un *back-face culling*.

# Capitolo 2

# Semplificazione di mesh poligonali

## 2.1 Introduzione

Il processo si *semplificazione* consiste nel *trasformare* una mesh poligonale in un'altra con un numero *inferiore* di facce, spigoli e vertici. La mesh risultante è una versione meno complessa dell'originale. Il requisito principale che deve avere un buon algoritmo di semplificazione è quello di produrre un'approssimazione più possibilmente simile al modello iniziale. Il processo di semplificazione può essere però un'operazione molto onerosa, per cui spesso la qualità della mesh semplificata è limitata da requisiti di efficienza.

Vi sono molti campi applicativi in cui è utile approssimare una mesh con un'altra meno dettagliata. Uno di questi è la gestione di modelli ottenuti tramite dispositivi di scansione tridimensionale oppure di isosuperfici costruite con algoritmi come ad esempio il *marching cubes* (Lorensen e Cline, 1987). Spesso tali modelli risultano essere sovra-campionati e la loro complessità troppo elevata. Solitamente le mesh ottenute tramite le suddette procedure sono tessellate uniformemente su tutta la loro superficie; per ragioni di economia si può pensare invece di risparmiare alcuni poligoni laddove la curvatura della superficie sia meno pronunciata. A tale scopo esistono algoritmi si semplificazione capaci di *adattare* la densità locale dei poligoni alla curva-

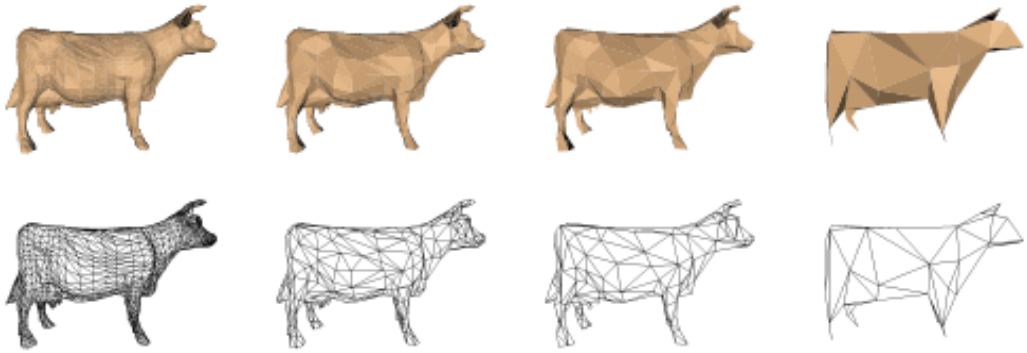


Figura 2.1: Livelli di dettaglio discreti di un modello (Garland)

tura locale della superficie. Queste tecniche permettono di risparmiare fino alla metà dei poligoni senza provocare degradazioni rilevanti della qualità del modello.

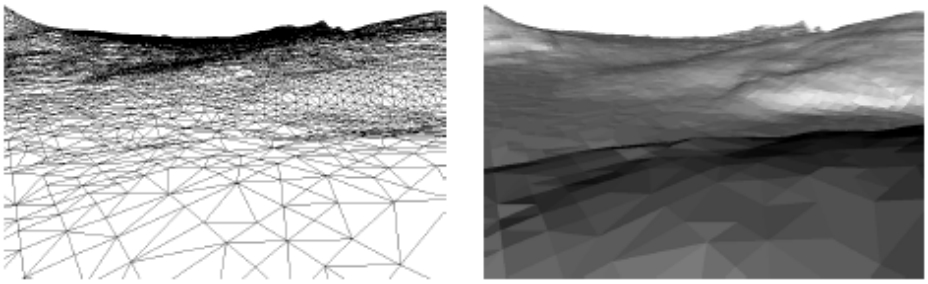
Un altro settore in cui sono spesso usate mesh semplificate è quello del *rendering* di modelli tridimensionali. Nell'ambito della visualizzazione di modelli 3D nasce l'idea secondo cui, a seconda della posizione dell'osservatore, non tutte le porzioni visibili del modello necessitano dello stesso grado di fedeltà della rappresentazione, ma possono essere semplificate in base alla loro angolazione o alla loro distanza dell'osservatore.

### 2.1.1 Level of detail

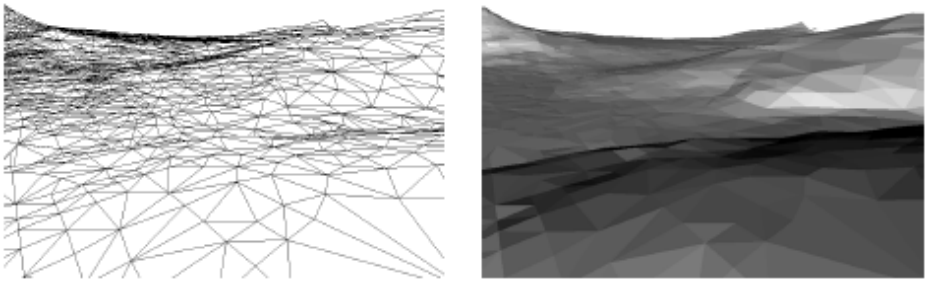
Il concetto di *Level of Detail*[1] (*LOD*) si fonda sul principio che oggetti lontani dall'osservatore non necessitano di una rappresentazione estremamente particolareggiata, dal momento che i piccoli dettagli risultano indistinguibili oltre una certa distanza. Di ciascun modello della scena sono fornite più rappresentazioni progressivamente meno dettagliate man mano che aumenta la distanza tra esso e l'osservatore.

Viene fatta una distinzione in base al tempo di creazione di queste rappresentazioni: si parla di modelli a *multi-risoluzione discreta* qualora le sue versioni semplificate siano create *off-line*, e di modelli a *multi-risoluzione con-*

[1] le cui origini sono attribuite a Clark (1976)



(a) LOD *discreto*



(b) LOD *continuo* e *adattativo*

Figura 2.2: Livelli di dettaglio di un terreno (Garland)

*tinua* quando la semplificazione è eseguita in modo *dinamico* parallelamente alla visualizzazione della scena.

La tecnica del LOD che fa uso di modelli a *multi risoluzione discreta* è chiamato *discrete LOD*. Secondo questo approccio (Funkhouser e Séquin, 1993), per ogni *frame* ciascun elemento della scena è selezionato tra un insieme finito di LOD precostruiti. La scelta dei *gradi di approssimazione* e delle rispettive *soglie di selezione* è di critica importanza per evitare il fenomeno del *popping*, cioè lo sgradevole effetto di notare bruschi cambiamenti dell'apparenza del modello in seguito ad un suo avvicinamento od allontanamento. Oltre a questi accorgimenti esistono anche tecniche più complesse per evitare gli artefatti causati dal *popping*. Una di queste (Funkhouser *e altri*, 1992) fa uso dell'*alpha blending*[2] per distribuire il cambiamento del LOD su più *frames* creando un effetto di dissolvenza.

Le versioni semplificate possono essere create *manualmente* oppure *auto-*

[2]proprietà di un materiale che indica il suo grado di *opacità*

*maticamente* a partire dal modello avente risoluzione massima tramite tecniche di *semplificazione*. La precostruzione dei LOD non ha vincoli di interattività, per cui gli algoritmi di semplificazione utilizzati hanno come obiettivo principale la ricerca della qualità.

Il *continuous LOD* è invece un esempio cui è molto importante che l'algoritmo di semplificazione sia il più efficiente possibile. Livelli di dettaglio continui sono utilizzati nella visualizzazione di grandi superfici, ad esempio il terreno. Se il tassellamento della superficie fosse *uniforme*, le zone di terreno lontane dall'osservatore apparirebbero dettagliate tanto quanto quelle vicine. Per eliminare questo spreco sono state sviluppate tecniche di multi-risoluzione *continua e adattativa* (Duchaineau *e altri*, 1997) (Lindstrom *e altri*, 1996), in cui il dettaglio della superficie non è uniforme, ma si adatta continuamente nello spazio e nel tempo (figure 2.2(a) e 2.2(b) nella pagina precedente).

### 2.1.2 Progressive Mesh

Un'altra applicazione che fa uso di algoritmi di semplificazione è quella delle *progressime mesh* (Hoppe, 1996) (Hoppe, 1998). Per *progressive mesh* si intende il processo di costruzione di un modello a partire da un livello a bassa risoluzione detto *base mesh* che viene progressivamente dettagliato attraverso l'applicazione di *operazioni topologiche inverse di semplificazione*[3].

Tale tecnica è usata in applicazioni interattive dove i modelli sono trasmessi sulla rete, come ad esempio Internet. In questo modo già dopo una prima trasmissione della *base mesh* l'utente può farsi un'idea del modello in questione. Mano a mano che i dati giungono a destinazione il modello risulterà sempre più dettagliato fino ad ottenere il risultato finale. Inoltre, se per qualche motivo il flusso dei dati è interrotto, l'utente è in grado di utilizzare ugualmente il modello, anche se incompleto.

Nel suo lavoro, Hoppe (1996) ha dimostrato che la tecnica delle progressive mesh può anche costituire un metodo efficiente per la compressione di mesh poligonali[4]. In particolare, in un suo successivo articolo (Hoppe, 1998),

[3]per una panoramica delle operazione di semplificazione si veda la sezione 2.3.1 nella pagina 34

[4]si veda il capitolo 3 nella pagina 46

egli mostra come sia possibile ordinare le operazioni[5] in modo che la loro codifica occupi in media 6 bits di memoria.

## 2.2 Tecniche di semplificazione

Esistono moltissime tecniche con le quali è possibile realizzare in modo automatico la semplificazione di mesh poligonali. Esse possono differire nella qualità del risultato, nell'efficienza computazionale e nel controllo dell'errore. In questa sezione è mostrata una panoramica delle tecniche più famose ed importanti illustrandone sinteticamente il funzionamento e mettendone in evidenza le caratteristiche principali.

Nelle sezioni restanti, invece, il metodo scelto per l'implementazione sarà analizzato più in dettaglio.

### 2.2.1 Vertex Clustering

Gli algoritmi appartenenti a questa categoria sono caratterizzati da un'elevata efficienza di calcolo e robustezza. Sono applicabili a qualsiasi tipo di mesh (funzionano anche nel caso di mesh non-manifold) e non hanno bisogno di informazioni relative alla connettività. Nonostante questi vantaggi, purtroppo gli algoritmi vertex clustering non producono spesso approssimazioni di buona qualità.

Un algoritmo sviluppato da Rossignac e Borrel (1993) prende il nome di *uniform vertex clustering* (figura 2.3 nella pagina seguente). Questo metodo prevede la suddivisione dello spazio della bounding box in celle (*cluster*) di lato $l$. Per ciascuna cella $C$ viene calcolato un punto $p$ del volume occupato[6] e tutti i punti $p_0, \ldots, p_n \in C$ sono riposizionati in $p$. Sia $p$ il vertice rappresentativo della cella $P$ e $q$ il vertice rappresentativo della cella $Q$. Nella mesh semplificata i punti $p$ e $q$ vengono connessi se e solo se nella mesh originale esiste almeno una coppia $(p_i, q_i)$ di vertici connessi.

Questo algoritmo può causare notevoli mutamenti alla topologia della

[5] ad ognuna delle quali corrisponde l'inserimento di un vertice

[6] questo particolare punto è detto *vertice rappresentativo* della cella

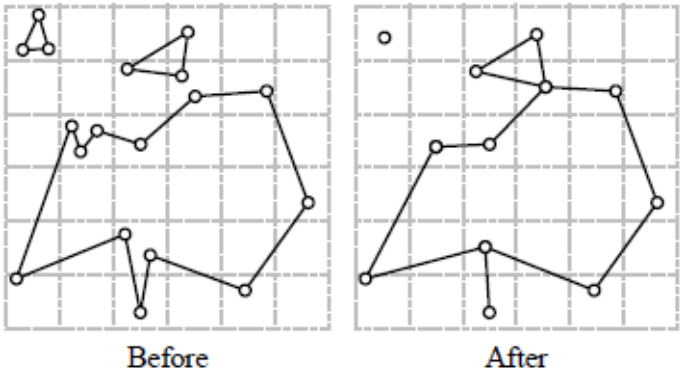



Figura 2.3: Uniform vertex clustering

mesh, in particolare non si può escludere di ottenere un modello semplificato non-manifold a partire da uno che non lo era. Inoltre, il vertex clustering funziona efficacemente solo se la superficie dei triangoli è molto minore della dimensione dei cluster. In altre parole, maggiore è il numero medio di vertici per cluster, tanto più è aggressiva la semplificazione, a discapito di modificazioni topologiche.

### 2.2.2 Region merging

L'algoritmo *superfaces* descritto in Kalvin e Taylor (1994) e in Kalvin e Taylor (1996) partiziona la superficie da semplificare in *regioni disgiunte* secondo dei criteri di complanarità. In seguito, per ciascuna regione sono cancellati i poligoni che ne fanno parte e ne viene eseguita una nuova triangolazione.

Con questo metodo è garantito un limite massimo all'errore introdotto, ma tale errore è difficilmente controllabile ed inoltre l'approssimazione ottenuta non è la migliore in rapporto al numero di triangoli della mesh finale. Per i suddetti motivi ed a causa dell'elevata complessità di implementazione rispetto ad altre tecniche, il metodo *superfaces* è raramente utilizzato.

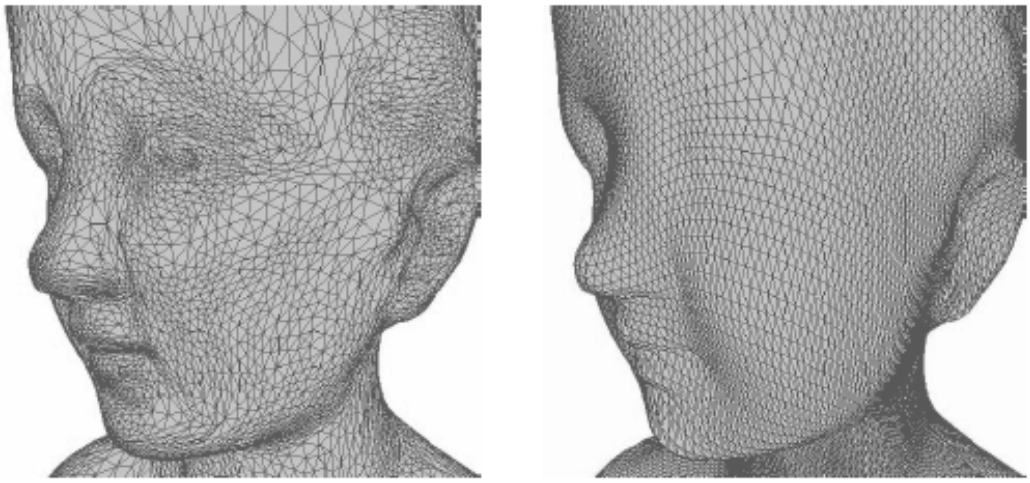


Figura 2.4: Esempio di resampling di una superficie (Gotsman *e altri*, 2001)

### 2.2.3 Resampling

Le tecniche presentate finora hanno la particolarità di semplificare una superficie eliminandone e/o riposizionandone i vertici. Differente è l'approccio adottato nei metodi di *resampling*: la superficie originale viene parametrizzata, solitamente su di un dominio bidimensionale, e tale parametrizzazione viene sfruttata per costruire un nuovo campionamento. I campioni così ottenuti vengono infine triangolati, consentendo di ottenere una mesh di diversa complessità. Nella figura 2.4 è raffigurata una superficie prima e dopo l'applicazione di un algoritmo di *resampling*.

Per ulteriori dettagli sull'argomento si veda l'articolo di Turk (1992) in cui sono presentate varie tecniche di approssimazione di una superficie utilizzando questo metodo.

### 2.2.4 Metodi iterativi

Gli algoritmi iterativi semplificano la superficie ripetendo una serie di passaggi dove in ciascuno di essi viene rimosso un vertice secondo un determinato criterio. La rimozione di un vertice comporta modifiche alla geometria e alla connettività della mesh nel suo intorno. L'algoritmo viene iterato fintanto che la mesh non ha raggiunto il desiderato grado di semplificazione. Ad esempio

è possibile proseguire nel processo di semplificazione fino al raggiungimento di un numero desiderato di poligoni.

La scelta del vertice da rimuovere è determinata da criteri basati sull'errore di approssimazione che tale operazione introduce. A ciascuna possibile operazione viene associato un costo proporzionale a tale errore ed utilizzando una struttura dati tipo *heap* esse sono mantenute in ordine, in modo da rendere possibile la selezione di quella con minor costo.

Esistono molti modi di calcolare il costo di un'operazione di semplificazione. I più usati sono il *quadric error metric* (Garland e Heckbert, 1997) e la *distanza di Hausdorff* (Klein *e altri*, 1996).

Ulteriori dettagli ed esempi relativi a queste tecniche sono presentati nella sezione 2.3.

## 2.3 Semplificazione incrementale

Come già accennato nella precedente sezione, con le tecniche di *semplificazione incrementale* è possibile ottenere un'approssimazione di una superficie eliminando ripetutamente dalla mesh originale un singolo vertice, fino al raggiungimento di un obiettivo prefissato (ad esempio numero di vertici desiderato o entità dell'approssimazione). Tipicamente ogni iterazione consiste nella *contrazione* oppure nella *rimozione* di un vertice.

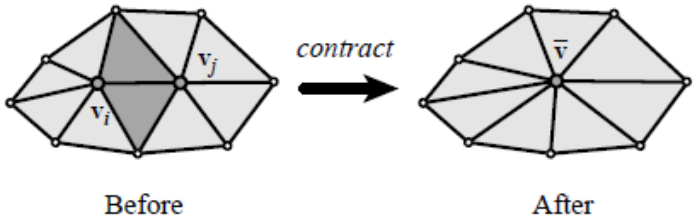
### 2.3.1 Operazioni topologiche

#### Contrazione di un vertice

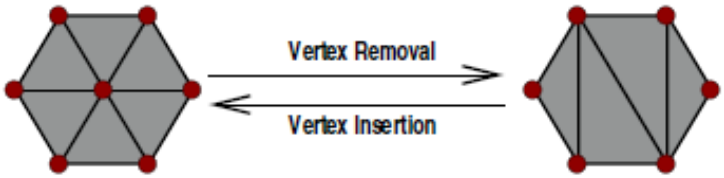
Questa operazione viene indicata con in simbolo $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \bar{\mathbf{v}}$ ed è definita come l'insieme dei tre passi seguenti:

1. Riposizionamento dei vertici $\mathbf{v}_i$ e $\mathbf{v}_j$ in $\bar{\mathbf{v}}$
2. Sostituzione di tutti i riferimenti a $\mathbf{v}_j$ con $\mathbf{v}_i$
3. Rimozione del vertice $\mathbf{v}_j$ e di tutte le facce divenute *degeneri*[7]

[7] un tirangolo è *degenere* quando non ha tre vertici distinti




(a) *Contrazione* di un vertice




(b) *Rimozione* di un vertice

Figura 2.5: Operazioni topologiche

Il risultato dell'operazione è mostrato in figura 2.5(a). Si noti che il primo passo modifica la *geometria* della mesh, mentre il secondo la *connettività*. L'ultimo passo toglie gli elementi che non sono più necessari.

La contrazione del vertice $\mathbf{v}_j$ può anche essere vista come la contrazione dello spigolo $(\mathbf{v}_i, \mathbf{v}_j)$. Per questo motivo in letteratura tale operazione è anche spesso riferita come *edge collapese*. Il caso particolare di *edge collapse* in cui $\bar{\mathbf{v}}$ coincide con $\mathbf{v}_j$ è chiamato invece *half-edge collapse*.

### Rimozione di un vertice

L'operazione di rimozione del vertice $\mathbf{v}$ (figura 2.5(b)) può essere descritta con i seguenti passi:

1. Cancellazione del vertice $\mathbf{v}$
2. Cancellazione di tutte le facce adiacenti a $\mathbf{v}$
3. Triangolazione della lacuna formatasi

Questa operazione sta alla base dell'algoritmo *vertex decimation* proposto da Schroeder *e altri* (1992).

Si noti che l'*half-edge collapse* può anche essere visto come un caso particolare di rimozione del vertice in cui la triangolazione risultante consiste nel collegare ciascuno spigolo del bordo della lacuna con $\mathbf{v}_j$.

### 2.3.2 Il processo di semplificazione

L'algoritmo implementato per la semplificazione di mesh ricalca il lavoro svolto da Garland e Heckbert (1997). Si tratta di un metodo di *semplificazione incrementale* relativamente efficiente con il quale si possono ottenere approssimazioni di alta qualità. Sono inoltre ben controllabili sia l'errore che il livello di semplificazione. L'algoritmo si basa su due concetti fondamentali:

- Iterazione dell'operazione *edge collapse*
- Stima dell'errore usando la funzione di costo denominata *Quadric Error Metric*[8]

Questi due elementi sono strettamente correlati per il fatto che, usando tale metrica, il *costo* è associato a ciascuno spigolo della mesh. L'algoritmo può essere descritto come segue:

1. Scelta delle coppie di vertici $(\mathbf{v}_i, \mathbf{v}_j)$ secondo determinati criteri di ammissibilità[9]
2. Assegnamento del *costo* a ciascuna coppia selezionata
3. Inserimento delle coppie in una struttura dati di tipo *heap* ordinato per costo crescente
4. Ripetizione dei seguenti passi fino all'ottenimento dell'approssimazione desiderata:
   - Estrazione della testa dello *heap*, cioè della coppia $(\mathbf{v}_i, \mathbf{v}_j)$ avente minor costo
   - Esecuzione della contrazione $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \bar{\mathbf{v}}$

[8] vedi sezione 2.4.1 nella pagina 39
[9] vedi sezione 2.3.3 nella pagina seguente

- Aggiornamento dei costi per tutti gli elementi dello *heap* relativi a $\mathbf{v}_i$

In questa sede è importante citare anche un'altra categoria di algoritmi, vale a dire gli algoritmi di *decimazione*. Ne è un esempio il *vertex decimation* (Schroeder *e altri*, 1992), il cui funzionamento è riassunto nei seguenti punti:

1. Selezione di un insieme di vertici *meno importanti*
2. Rimozione dei vertici selezionati
3. Triangolazione delle lacune formatesi

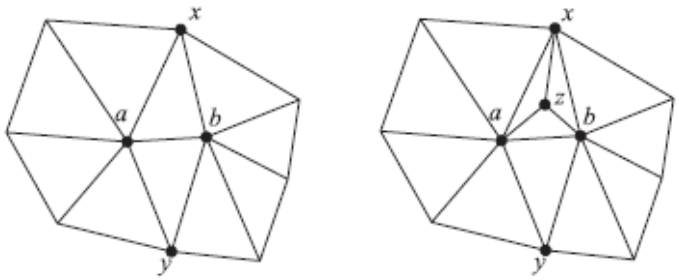
Questi passi possono essere ripetuti per diminuire ulteriormente i dettagli della mesh. La differenza sostanziale tra le tecniche di *decimazione* e quelle di *semplificazione* è la seguente: mentre nelle prima gli elementi da eliminare vengono tutti selezionati inizialmente, nella seconda le informazioni relative all'errore sono tenute costantemente aggiornate dopo ogni operazione e la scelta di ciascun elemento dipende da quelli rimossi precedentemente. Se ne deduce che la *decimazione* è molto più efficiente in termini computazionali, ma l'approssimazione ottenuta è meno precisa e di qualità inferiore. Spesso le due tecniche sono usate in sequenza: dopo una prima *decimazione* più o meno aggressiva viene effettuata una *semplificazione* in modo da ottenere l'approssimazione cercata con estrema accuratezza.

### 2.3.3 Selezione delle coppie di vertici candidate

L'operazione topologica scelta per l'algoritmo è quella della contrazione del vertice. Pertanto, una condizione necessaria affinché una coppia di vertici $(\mathbf{v}_i, \mathbf{v}_j)$ possa essere selezionata è che deve esistere lo spigolo $(\mathbf{v}_i, \mathbf{v}_j)$.

Tuttavia, al fine di ottenere i risultati desiderati, spesso tale condizione non è sufficiente. In particolare, solitamente si richiede che le operazioni di semplificazione non modifichino la topologia della superficie rappresentata dalla mesh[10]. L'operazione di *edge-collapse* non garantisce tale condizione,

[10] è invece ovvio come tali operazioni modifichino la topologia della mesh




(a) Contrazione *valida* (b) Contrazione *non valida*

Figura 2.6: Condizione di omeomorfismo

ovvero è possibile che l'esecuzione di una tale operazione produca una mesh non-manifold a partire da una manifold.

Una condizione necessaria e sufficiente a preservare la topologia della superficie della mesh in seguito ad un *edge-collapse* è stata presentata da Dey *e altri* (1999). Sia $St(\sigma)$ l'insieme delle co-facce[11] del simplesso $\sigma$ e $Lk(\sigma)$ l'insieme delle facce appartenenti a $St(\sigma)$ ma non incidenti su $\sigma$. Sia inoltre $M_i$ un 3-complesso simpliciale *privo di bordi*, ed $e = (\mathbf{v}_i, \mathbf{v}_j)$ un suo spigolo, ed $M_{i+1}$ il complesso dopo la contrazione di $e$. $M_i$ e $M_{i+1}$ sono *omeomorfi* se e solo se:

$$Lk(v_i) \cap Lk(v_j) = Lk(e) \tag{2.1}$$

Nel caso in cui $M_i$ *abbia bordi* è possibile ricondursi al caso generale aggiungendo un *dummy vertex* $w$ per ciascuna lacuna della mesh e collegandolo con gli spigoli che stanno sul bordo.

Nella figura 2.6 sono mostrati due esempi di *edge-collapse* dello spigolo $ab$. In 2.6(a) la condizione $Lk(a) \cap Lk(b) = \{x, y\} = Lk(ab)$ indica che la contrazione è valida. Per contro, in 2.6(b) è facile verificare che $Lk(a) \cap Lk(b) = \{x, y, z, zx\} \neq Lk(ab)$.

[11] si veda la sezione 1.2.1 nella pagina 13

## 2.4 Costo della semplificazione

Come è stato analizzato nella sezione 2.3.2 nella pagina 36, ad ogni spigolo è assegnato un *costo*. Tale *costo* determina la selezione del vertice da contrarre all'iterazione considerata.

Sia $M_i$ il modello all'iterazione $i$. Se il modello a quel passo ha $n$ coppie di vertici valide, il modello ottenuto dopo la successiva iterazione è uno soltanto degli elementi dell'insieme $\{M_{i+1}^1, \ldots, M_{i+1}^n\}$, dove con $M_{i+1}^s$ è indicato il modello ricavato dal passo $i$ contraendo lo spigolo $s$.

Dal momento che l'approssimazione ha lo scopo di rappresentare la mesh originale più fedelmente possibile, lo spigolo $s$ da contrarre è selezionato in modo tale che $M_{i+1}$ si discosti minimamente da $M_i$. La differenza tra i modelli di due passi successivi rappresenta infatti l'*errore di approssimazione*, una cui stima è data proprio dal *costo* assegnato all'elemento che è stato contratto. In altre parole più piccolo è il *costo* assegnato ad uno spigolo, minore è il cambiamento che la sua contrazione introduce.

### 2.4.1 Metrica dell'errore

Il costo basato sulla *quadric error metric* si prefigge lo scopo di fornire una buona caratterizzazione dell'errore di approssimazione, senza richiedere un eccessivo costo computazionale. Questa tecnica rappresenta un ottimo compromesso tra *efficienza* ed *accuratezza*.

Seguendo il lavoro svolto da Ronfard e Rossignac (1996), ad ogni vertice del modello è associato un insieme di *piani*. Viene poi definito l'*errore relativo al vertice* come la somma dei quadrati delle distanze tra il vertice considerato ed i piani ad esso associati.

I piani associati a ciascun vertice non sono altro che i piani individuati dalla facce ad esso adiacenti. Ciascuna faccia determina un piano con equazione $\mathbf{n}^T\mathbf{v} + d = 0$, dove $\mathbf{n} = [n_x, n_y, n_z]^T$ è la normale alla faccia e $d$ una costante. Il quadrato della distanza tra un vertice $\mathbf{v} = [x, y, z]^T$ e tale piano è dato da:

$$D^2 = (\mathbf{n}^T\mathbf{v} + d)^2 = (\mathbf{n}^T\mathbf{v} + d)(\mathbf{n}^T\mathbf{v} + d) = \mathbf{v}^T(\mathbf{n}\mathbf{n}^T)\mathbf{v} + 2d\mathbf{n}^T\mathbf{v} + d^2 \quad (2.2)$$

Si definisca la *quadrica* $Q$ come la tripletta $Q = (\mathbf{A}, \mathbf{b}, c) = (\mathbf{n}\mathbf{n}^T, d\mathbf{n}, d^2)$, la cui *metrica* è data da:

$$Q(\mathbf{v}) = \mathbf{v}^T\mathbf{A}\mathbf{v} + 2\mathbf{b}^T\mathbf{v} + c \quad (2.3)$$

Sia $\{f_1, \ldots, f_k\}$ l'insieme delle facce adiacenti al vertice $\mathbf{v}$. Ciascuna faccia $f_i$ possiede una normale $\mathbf{n}_i$ che, insieme ad un suo punto $\mathbf{p}_i$, individua la *quadrica fondamentale* del piano su cui essa giace, calcolabile come:

$$Q_i = (\mathbf{A}_i, \mathbf{b}_i, c_i) = (\mathbf{n}_i\mathbf{n}_i^T, -\mathbf{A}_i\mathbf{p}_i, \mathbf{p}_i^T\mathbf{A}_i\mathbf{p}_i) \quad (2.4)$$

L'errore $E_Q$ associato al vertice $\bar{v}$ non è altro che la somma delle quadriche $Q_i$ relative alla facce $f_i$ incidenti.

$$E_Q(\mathbf{v}) = \sum_{i=1}^{k} D_i^2(\mathbf{v}) = \sum_{i=1}^{k} Q_i(\mathbf{v}) = Q(\mathbf{v}) \quad (2.5)$$

dove $Q = \sum_{i=1}^{k} Q_i$. La somma tra quadriche è definita come la quadrica le cui componenti sono la somma delle componenti degli addenti: $(Q_i + Q_j) = (\mathbf{A}_i + \mathbf{A}_j, \mathbf{b}_i + \mathbf{b}_j, c_i + c_j)$.

Quando lo spigolo $(\mathbf{v}_i, \mathbf{v}_j)$ viene contratto, la quadrica risultate $Q$ è data semplicemente dalla somma delle quadriche dei vertici coinvolti $Q = Q_i + Q_j$. Si definisce così il *costo della contrazione* $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \bar{\mathbf{v}}$ come:

$$Q(\bar{\mathbf{v}}) = Q_i(\bar{\mathbf{v}}) + Q_j(\bar{\mathbf{v}}) \quad (2.6)$$

Il calcolo della quadrica per ciascun vertice può essere eseguito con complessità *costante*. Questa caratteristica rende molto efficienti gli algoritmi basati su questa metrica.

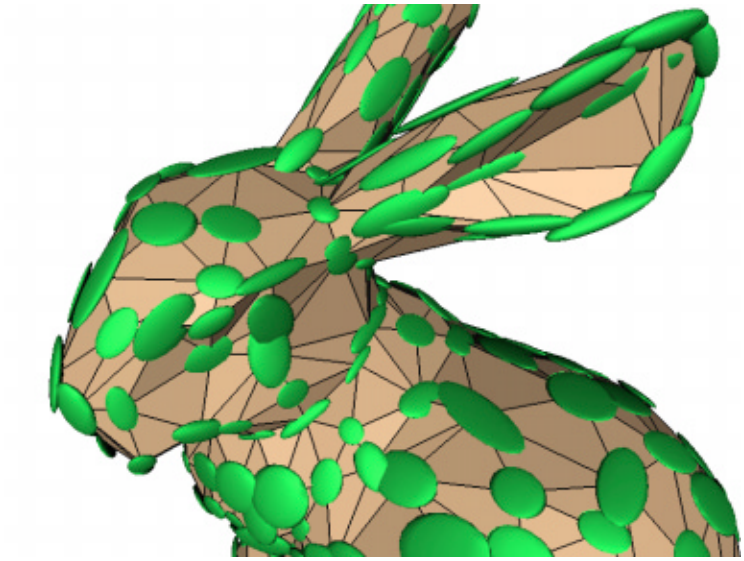


Figura 2.7: Rappresentazione grafica delle quadriche (Garland e Heckbert, 1998)

### 2.4.2 Interpretazione geometrica della quadrica

Le quadriche descritte in questa sezione hanno un'interessante interpretazione geometrica. Per una data quadrica $Q$ si definisce l'*iso-superficie* $Q(\mathbf{v}) = \varepsilon$ come l'insieme dei punti il cui errore vale $\varepsilon$ (relativamente a $Q$). In altra parole si tratta del luogo dei punti in cui un vertice può essere riposizionato senza alterare l'errore associato. Tali superfici sono *ellissoidi* i cui assi principali sono definiti dagli autovalori ed autovettori della matrice $\mathbf{A}$. Quando la matrice $\mathbf{A}$ è *singolare*[12] gli ellissoidi sono *degeneri*: se il numero di autovalori nulli è uno, allora le iso-superfici sono cilindri di lunghezza infinita; se invece gli autovalori nulli sono due, allora le iso-superfici degenerano in coppie di piani paralleli.

La figura 2.7 illustra il risultato dell'approssimazione del modello di un coniglio. Sono evidenziate le iso-superfici delle quadriche utilizzate per tale semplificazione. La figura mostra come le quadriche caratterizzano localmente la superficie della mesh: le quadriche associate ad i vertici situati a cavallo di forti curvature si presentano più allungate nella direzione della curvatu-

[12] almeno un suo autovalore vale 0

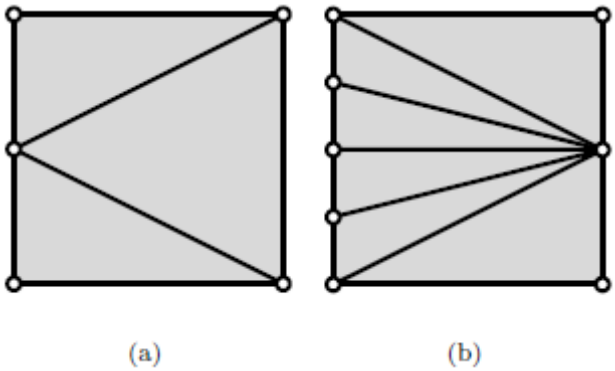



Figura 2.8: Due differenti triangolazioni di una regione piana

ra stessa; dove invece la superficie è più piatta, la quadriche associate ad i vertici che ne fanno parte sono più rotonde.

### 2.4.3 Normalizzazione della quadrica

Le quadriche associate ad i vertici calcolate come in (2.5) producono però alcuni effetti indesiderati. Si consideri la figura 2.8. Dal momento che tutti i triangoli risiedono sullo stesso piano, la quadrica fondamentale associata a ciascuno di essi è la stessa e vale $Q$. Si calcoli ora la quadrica dell'intera regione come la somma delle quadriche dei vertici che ne fanno parte. Nella triangolazione (a) la quadrica della regione vale $9Q$, mentre nel caso (b) vale $18Q$. Questo effetto va contro il principio intuitivo secondo il quale la quadrica associata ad una regione deve essere la stessa indipendentemente dalla triangolazione utilizzata.

Allo scopo di evitare questo inconveniente, si introduce il concetto di *quadrica pesata* $Q$, definita come:

$$Q = (w\mathbf{A}, w\mathbf{b}, wc) \tag{2.7}$$

L'errore al vertice (2.5) viene generalizzato nel seguente modo:

$$E_Q(\mathbf{v}) = \sum_{i=1}^{k} w_i Q_i(\mathbf{v}) = \left( \sum_{i=1}^{k} w_i Q_i \right)(\mathbf{v}) \tag{2.8}$$

Il peso $w$ può essere scelto secondo diversi criteri. In particolare, per risolvere il problema di avere una quadrica associata ad una regione che sia indipendente dal suo tassellamento, si può scegliere il peso $w_i$ come l'*area* del triangolo $f_i$ relativo al piano la cui quadrica fondamentale è $Q_i$.

Inoltre si desidera che l'errore sia anche *indipendente dalla scala*: di due regioni simili che differiscono solo per la scala si vuole che l'errore associato alla regione più piccola sia minore. Questa proprietà è ottenuta evitando di dividere la (2.8) per $\sum_{i=1}^{k} w_i$.

### 2.4.4 Politiche di posizionamento dei vertici

Nell'esecuzione delle operazioni di *edge collapse* $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \bar{\mathbf{v}}$ è di fondamentale importanza la scelta del vertice $\bar{\mathbf{v}}$. Questa scelta determina infatti l'errore introdotto $Q(\bar{\mathbf{v}})$.

Se ad esempio non si vuol andare a modificare la geometria della mesh è possibile usare una forma particolare di *halfedge collapse* in cui il target è $\mathbf{v}_i$ se $Q(\mathbf{v}_i) < Q(\mathbf{v}_j)$ e $\mathbf{v}_j$ altrimenti. Questa politica di selezione del target è chiamata *subset placement*.

Il posizionamento *ottimale*[13] si può ottenere invece minimizzando l'errore[14] $Q(\bar{\mathbf{v}})$. Dal momento che la quadrica è una funziona quadratica, trovarne il minimo si riduce a risolvere un'equazione lineare.

$$\nabla Q(\bar{\mathbf{v}}) = 2\mathbf{A}\bar{\mathbf{v}} + 2\mathbf{b} = 0 \tag{2.9}$$

Da cui si ricava:

$$\bar{\mathbf{v}} = -\mathbf{A}^{-1}\mathbf{b} \tag{2.10}$$

L'errore introdotto scegliendo il target in questo modo vale:

$$Q(\bar{\mathbf{v}}) = \mathbf{b}^T\bar{\mathbf{v}} + c = -\mathbf{b}^T\mathbf{A}^{-1}\mathbf{b} + c \tag{2.11}$$

Calcolare il minimo del gradiente della quadrica ha il significato geometri-

[13] *optimal placement*

[14] si noti che questo non implica l'approssimazione ottima

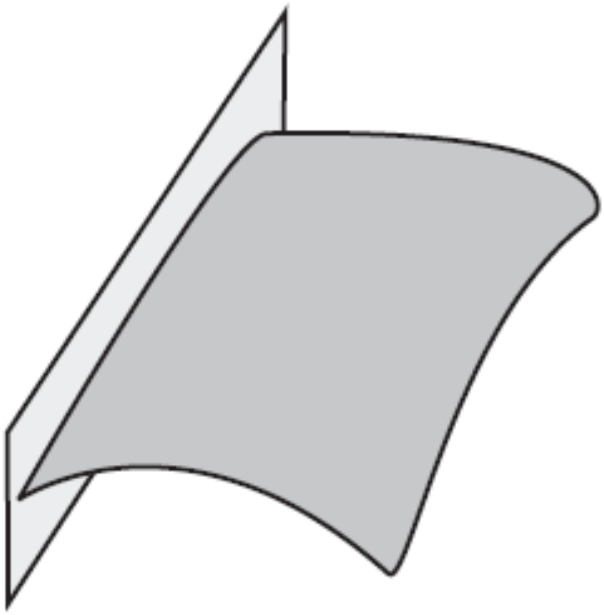


Figura 2.9: Esempio di *constraint plane*

co di trovare il centro delle *iso-superfici* ellissoidali descritte nella sezione 2.4.2 nella pagina 41.

Rispetto alla tecnica subset placement, il posizionamento ottimale produce approssimazioni dall'aspetto più simile in confronto all'originale ed in cui i triangoli hanno aree più uniformi.

### 2.4.5 Come preservare bordi e discontinuità

L'algoritmo visto fino a questo punto non utilizza alcun accorgimento particolare nel caso in cui la mesh da approssimare abbia dei bordi. In tal caso è possibile che i bordi si *spostino* durante la semplificazione incrementale, modificando eccessivamente il modello originale.

Questo problema può essere risolto calcolando la quadrica dei vertici che stanno sui bordi in modo diverso. Sia $E = \{e_1, \ldots, e_n\}$ l'insieme degli spigoli appartenenti ai bordi ed $F = \{f_1, \ldots, f_n\}$ l'insieme delle rispettive facce. Per ogni spigolo $e_i \in E$ si calcoli il *constraint plane*, cioè il piano passante per $e_i$ e perpendicolare al piano individuato da $f_i$ (vedi figura 2.9). Per ciascun

*constraint plane* si calcoli inoltre la quadrica fondamentale pesandola per un coefficiente $w$ molto grande. Le quadriche fondamentali dei *constraint planes* vanno così a contribuire alla quadriche dei vertici ad essi adiacenti come di consueto.

Questa tecnica funziona anche nel caso in cui la superficie presenti altre forme di discontinuità, per esempio di colore o di altri attributi generici.

# Capitolo 3

# Compressione di mesh poligonali

## 3.1 Introduzione

Le informazioni necessarie a descrivere una mesh sono di due tipi: quelle concernenti la *connettività* e quelle relative alla *geometria*. La *connettività* consiste nelle informazioni sulle interconnessioni degli elementi della mesh, mentre la *geometria* specifica la posizione dei vertici nello spazio.

La tecnica standard di rappresentazione di una mesh descrive la connettività attraverso una *tabella di incidenza faccia/vertici*: per ogni faccia è data una lista orientata di indici che rappresentano i vertici della faccia stessa. Si può dimostrare[1] che alcune delle informazioni contenute nella tabella di incidenza sono ridondanti e sono ricavabili implicitamente analizzando la connettività della mesh. Per questo motivo e con il fine di rendere più efficiente l'uso della memoria necessaria all'archiviazione di modelli tridimensionali, sono state presentate numerose tecniche di compressione specifiche.

Prima di esaminare in dettaglio il problema della codifica di mesh poligonali, di seguito vengono ricordati alcuni concetti fondamentali relativi alla codifica e compressione dell'informazione in generale.

[1] sezione 3.3 nella pagina 53

## 3.2 Tecniche di codifica dell'informazione

Le tecniche di codifica dei dati si dividono in due categorie: le tecniche *lossless* e le tecniche *lossy*. La carateristica fondamentale di una compressione lossless è che il prodotto di una codifica e decodifica è la copia identica dell'originale, senza nessun tipo di cambiamento o perdita. Le tecniche lossy sono invece applicate nei casi in cui è possibile fare una scelta controllata delle informazioni che possono essere perse, senza che se ne modifichi sensibilmente la percezione.

### 3.2.1 Tecniche lossless

In generale un flusso di dati può essere visto come una sequenza di *simboli*, il cui insieme costituisce l'*alfabeto* $\mathcal{A} = \{\sigma_1 ... \sigma_a\}$ di cardinalità $a$. Sia ad esempio $S_n = \sigma_{i1}\sigma_{i2}...\sigma_{in}$ una sequenza di $n$ simboli, ognuno dei quali può essere rappresentato con $\lceil log_2 a \rceil$ bits. L'intera sequenza $S_n$ occupa così $n\lceil log_2 a \rceil$ bits di memoria. Per aumentare l'efficienza della memorizzazione si può pensare di rappresentare i simboli con un numero *variabile* di bits, per esempio dando maggiore importanza ai simboli più ricorrenti. Sia $\#_i(S_n)$ il numero di simboli $\sigma_i$ all'interno della sequenza $S_n$ e $p_i(S_n) = \#_i(S_n)/n$ la probabilità del simbolo $\sigma_i$. Si definisca inoltre l'*entropia* del simbolo $\sigma_i$ come $e_i = \log_2(1/p_i)$. L'ottimale numero medio di bits per simbolo è dato da

$$\sum_{i=0}^{a} p_i \cdot \log_2 \frac{1}{p_i} \tag{3.1}$$

### 3.2.2 Codifica di Huffman

L'algoritmo di codifica di Huffman[2] (Nelson e Gailly, 1995) associa a ciascun simbolo un codice di un numero variabile di bits in modo che a probabilità più alte corrispondano codici più corti. I codici di Huffman hanno il vincolo del *prefisso unico* per poter essere riconosciuti nonostante la lunghezza variabile. Questo può essere garantito costruendo i codici a partire

[2]per le origini di questa tecnica si veda Huffman (1952)

da un albero binario le cui foglie sono associate ai simboli. Il percorso dalla radice al simbolo $\sigma_i$ costituisce un codice $c_i$ tale che ad ogni livello dell'albero corrisponde un bit che specifica il percorso seguito: ad esempio 0 se si procede con il figlio sinistro, 1 con quello destro.

L'algoritmo di Huffman specifica come costruire l'albero dei simboli e come ricavarne i codici, sia in fase di codifica, che di decodifica. Inizialmente ad ogni simbolo viene associato un nodo ed ad ogni nodo un peso pari alla probabilità di apparizione del simbolo che rappresenta. I passi per costruire l'albero sono i seguenti:

1. Selezione dei due nodi liberi con il minor peso.

2. Creazione di un nuovo nodo da porre come padre per i nodi liberi individuati. Il peso di questo nuovo nodo è la somma dei pesi dei figli.

3. Associazione di un valore binario ad i rami dell'albero che collegano il nodo appena creato con quelli selezionati. Per convenzione si attribuisca il valore 0 al ramo sinistro ed 1 a quello destro. Questo valore servirà per costruire i codici.

4. Inserimento del nuovo nodo nella lista dei nodi liberi e rimozione dei due nodi selezionati dalla stessa.

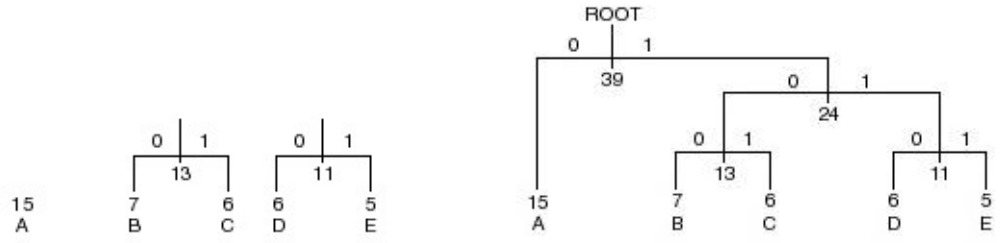
5. Ripetere i precedenti passi finché la lista dei nodi liberi non contenga un unico elemento. Tale nodo costituirà la radice dell'albero.

Una volta costruito l'albero, encoder e decoder possono compiere la *traduzione* dei simboli nel modo più efficiente possibile. Ad esempio, l'encoder può costruire i codici in una fase di preprocessing visitando l'albero in ordine anticipato (*preorder*) in modo da poterli così accedere con accesso diretto.

Durante la fase di decodifica invece il decoder riconosce i simboli percorrendo l'albero a partire dalla radice. Per ogni bit letto il decoder visita il figlio destro se il il valore di tale bit è uno, o il figlio sinistro altrimenti. Quando il nodo visitato è una foglia viene emesso il simbolo ad esso associato. Con la lettura del prossimo bit la visita dell'albero è ripresa a partire dalla radice, e così via.

| Simbolo | Frequenza |
|---|---|
| A | 15 |
| B | 7 |
| C | 6 |
| D | 6 |
| E | 5 |

Tabella 3.1: Tabella delle frequenze




(a) Dopo due passi

(b) Al termine dei passi

Figura 3.1: Fasi della costruzione dell'albero di Huffman

Di seguito è mostrato un esempio sulla costruzione dei codici di Huffman. Nella tabella 3.1 è riportato l'alfabeto e la frequenza di ciascun simbolo.

L'algoritmo inizia scegliendo i due simboli con frequenza più bassa: D ed E. Viene creato un nuovo nodo, padre dei nodi D ed E, la cui frequenza è impostata ad 11. Alla foglia D è assegnato il ramo 0, mentre alla foglia E è assegnato il ramo 1. Al passo successivo i due nodi liberi con minor frequenza sono il B ed il C. Viene creato quindi un nuovo nodo padre di questi ultimi. Il risultato è riportato in figura 3.1(a).

Iterando le regole dell'algoritmo di Huffman riportate alla pagina 47, si ottiene l'albero come in figura 3.1(b).

I codici di ciascun simbolo sono ricavati percorrendo l'albero dalla radice alla foglia corrispondente. Il risultato ottenuto è mostrato in tabella 3.2 nella pagina successiva.

| Simbolo | Codice |
|---|---|
| A | 0 |
| B | 100 |
| C | 101 |
| D | 110 |
| E | 111 |

Tabella 3.2: Codici di Huffman

### 3.2.3 Codifica aritmetica

L'algoritmo di Huffman è ottimale soltanto se le probabilità dei simboli sono potenze negative di due, poiché in tal caso i codici hanno una lunghezza intera di bits. Se ad esempio la probabilità fosse $\frac{1}{5}$, l'ottimo sarebbe avere una codifica di $\log_2 5 = 2.3219$ bits. Nella codifica aritmetica (Witten *e altri*, 1987) non vengono assegnati codici ai simboli, ma l'intero stream viene rappresentato con un singolo numero in virgola mobile compreso tra 0 ed 1.

Si supponga di comprimere una sequenza $S_n$ di $n$ simboli il cui alfabeto $\mathcal{A}$ ha cardinalità $a$ . Ciascun simbolo $\sigma_i$ ha associata una probabilità di apparizione $p_i$. Sia definisca l'*intervallo di riferimento* $w = [w_L, w_H]$ e si indichi con $w_m$ l'intervallo di riferimento relativo all'iterazione $m$. L'algoritmo della compressione aritmetica procede nel seguente modo:

1. Inizializzazione dell'intervallo di riferimento $w = w_0$ con $w_L = 0$ e $w_H = 1$
2. Suddivisione di $w$ in $a$ sotto intervalli $v^i = [v^i_L, v^i_H]$ in modo tale che $(v^i_H - v^i_L) \propto p_i$
3. Lettura del prossimo simbolo $\sigma_s$ in ingresso
4. Restrizione dell'intervallo di riferimento $w$ a $v^s$, vale a dire $w_L = v^s_L$ e $w_H = v^s_H$
5. Ripetizione dei passi 2 - 4 per ciascuno degli $n$ simboli in ingresso
6. La codifica corrisponde ad un valore $W$ scelto in modo tale che $W \in [w_L, w_H]$

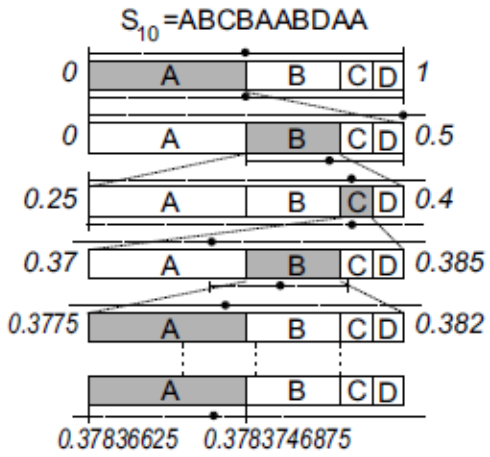



Figura 3.2: Compressione Aritmetica

Un esempio della codifica aritmetica è illustrato nella figura 3.2.

Si facciano alcune considerazioni sul numero di bits necessari a rappresentare $W$. La prima suddivisione dell'intervallo di riferimento produce dei sotto intervalli $v^i$ di ampiezza $\bar{v}^i = p_i$, dal momento che $\bar{w_0} = \sum_{i=1}^{k} p_i = 1$. Dopo $n$ suddivisioni l'ampiezza dell'intervallo di riferimento è determinata dai simboli $\sigma_i \in S_n$ incontrati e sarà pari a:

$$\bar{w_n} = \prod_{i=1}^{n} p_i \tag{3.2}$$

Si supponga adesso di suddividere l'intervallo di riferimento iniziale $[0, 1]$ in sotto intervalli di ampiezza $\bar{w_n}$, ciascuno dei quali è individuato da un indice. Uno soltanto di questi intervalli corrisponde a $w_n$. Il valore della codifica $W$ può anche essere visto come il suo indice. Il numero $N$ di tali intervalli è dato da:

$$N = \frac{\bar{w_0}}{\bar{w_n}} = \frac{1}{\prod_{i=1}^{n} p_i} \tag{3.3}$$

Il numero $B$ di bits necessari per rappresentare $N$ è quindi:

$$B = \log_2 N = \log_2 \frac{1}{\prod_{i=1}^{n} p_i} = \sum_{i=1}^{n} \log_2 \frac{1}{p_i} \tag{3.4}$$

Dal momento che l'*entropia* $e_i$ del simbolo $\sigma_i$ è data da $\log_2 \frac{1}{p_i}$, si ottiene il risultato notevole:

$$B = \sum_{i=1}^{n} e_i \tag{3.5}$$

Si riesce così a dimostrare che con la codifica aritmetica si ottiene l'ottimo[3], cioè la minima entropia.

### 3.2.4 Tecniche lossy

Nelle tecniche di compressione *lossy* viene tollerata la perdita di parte dell'informazione associata ai dati originali al fine di consentire un maggiore livello di compressione. Questa perdita di informazione deve essere tale da non degradare eccessivamente la percezione dell'oggetto codificato. Un esempio di tale tecnica è la codifica delle immagini in formato JPEG (Skodras *e altri*, 2001) o dei video MPEG (Le Gall, 1991).

Queste tecniche possono essere applicate alla *geometria* di una mesh, vale a dire all'elenco delle coordinate dei vertici. La rappresentazione della geometria nei formati più comuni (VRML, OBJ, ASC) consiste infatti in un vettore di numeri a virgola mobile, ciascuno dei quali è rappresentato generalmente con 32 o 64 bits di memoria. Nel caso di molte applicazioni di visualizzazione di scene tridimensionali, tale livello di accuratezza non è necessario ed è quindi conveniente *quantizzare* tali valori su un numero minore di bits.

### 3.2.5 Quantizzazione

Nel processo di *quantizzazione* i valori di un *dominio continuo* (ad esempio la retta reale) sono approssimati con un insieme relativamente piccolo di valori discreti.

L'esempio più semplice di quantizzazione è la *quantizzazione scalare*, il cui dominio è la retta reale. Tale operazione può essere espressa nella seguente forma:

[3] vedi la condizione di ottimo (3.1)

$$Q(x) = g(\lfloor f(x) \rfloor) \tag{3.6}$$

Dove:

- $x$ è il numero reale da approssimare
- $\lfloor \cdot \rfloor$ è la funzione che approssima all'intero inferiore. Il risultato di tale operazione rappresenta l'approssimazione cercata $i = \lfloor f(x) \rfloor$ ed è spesso riferito come *indice di quantizzazione*
- $f(x)$ e $g(i)$ sono due arbitrarie funzioni a valori reali

In questo contesto $i$ è l'elemento memorizzato o trasmesso, mentre il valore originale viene ricavato calcolando $g(i)$. La più comune formula di quantizzazione è la *quantizzazione uniforme*, che approssima valori reali dell'intervallo $[0, 1)$ con un interi di $M$ bits.

$$Q(x) = \frac{\lfloor 2^M x \rfloor + 0.5}{2^M} \tag{3.7}$$

Dove gli operatori $f(x)$ e $g(i)$ sono semplici funzioni di scalatura. $g(i)$ inoltre introduce un offset per porre il valore ricavato nel punto medio del relativo *intervallo di quantizzazione* $2^{-M}$.

## 3.3 Compressione della connettività

Comprimere la connettività significa riuscire a rappresentare le informazioni contenute nella tabella delle incidenze faccia/vertice nel modo più efficiente possibile. Secondo il metodo di rappresentazione standard (sezione 3.1 nella pagina 46), la connettività di una mesh composta da $f$ triangoli e $v$ vertici occupa $3f\lceil \log_2 v \rceil \approx 6v\lceil \log_2 v \rceil$ bits, cioè circa $6\lceil \log_2 v \rceil$ bits per vertice (bpv). Ne è un esempio di questa rappresentazione il nodo *IndexedFaceSet* del linguaggio *VRML*[4].

[4]per approfondimenti sul linguaggio *VRML* è possibile consultare ad esempio Hartman e Wernecke (1996)

```
Shape {
        appearance Appearance {
                material Material {
                        emissiveColor .8 0.2 0
                }
        }
        geometry IndexedFaceSet {
                coord Coordinate {
                        point [
                                -1 0 1,
                                1 0 1,
                                1 0 -1,
                                -1 0 -1,
                                0 2 0
                        ]
                }
                coordIndex [
                        0, 3, 2, 1, -1
                        0, 1, 4, -1
                        1, 2, 4, -1
                        2, 3, 4, -1
                        3, 0, 4, -1
                ]
        }
}
```

Questo codice descrive una piramide con il linguaggio *VRML*. In esso si nota il nodo *point* in cui è definita la lista delle coordinate dei vertici (la *geometria*) ed il nodo *coordIndex* che specifica la tabella delle incidenze faccia/vertice (la *connettività*). È facile osservare come l'efficienza di tale rappresentazione decresca al crescere delle dimensioni della mesh, rendendola poco indicata per mesh di grandi dimensioni.

Al fine di evitare tale limitazione sono stati sviluppati numerosi algoritmi di compressione, nei quali il numero di bits richiesti per rappresentare un vertice non dipende dalla dimensione della mesh. Una caratteristica comune

a tali metodi è la visita progressiva delle facce che compongono la superficie (*conquista*): gli elementi della mesh vengono codificati nel momento in cui entrano a far parte della *regione conquistata*. I metodi di codifica della connettività sono raggruppati in tre categorie a seconda dell'elemento che gioca il ruolo dominante nella compressione.

- *Tecniche basate sulle facce*: Cut-Border Machine (Gumhold e Straßer, 1998) e Edge Breaker (Rossignac, 1999)
- *Tecniche basate sugli spigoli*: Face Fixer (Isenburg e Snoeyink, 2000)
- *Tecniche basate sui Vertici*: Valence Based Encoding (Touma e Gotsman, 2000)

### 3.3.1 Generalità sugli algoritmi di codifica

Come già accennato in precedenza, questi algoritmi definiscono una *regione conquistata*[5] che contiene tutti gli elementi della mesh *processati* fino all'istante considerato. Un vertice od uno spigolo si dice *processato* quando siano processate tutte le facce che vi incidono. Un vertice od uno spigolo non ancora processato, ma che ha almeno una faccia incidente processata è detto *attivo*. L'insieme dei vertici e degli spigoli attivi costituisce il *bordo di taglio*, cioè quella linea che delimita la regione di accrescimento e che separa gli elementi processati da quelli non processati. La regione di accrescimento si espande verso le facce non processate incidenti sul bordo di taglio. Un particolare spigolo attivo del bordo di taglio viene denominato *gate* ed il vertice al quale punta viene detto *pivot*. La regione di accrescimento si espande verso le facce non processate incidenti al gate ed al pivot. Le operazioni di espansione sono dette *comandi* ed ad ognuna di esse corrisponde un *simbolo*, che può essere accompagnato da informazioni relative alle relazioni di incidenza tra gli elementi considerati.

Il processo di codifica inizia scegliendo una faccia, i cui lati vanno a costituire il primo *bordo di taglio*, ed uno dei suoi spigoli è scelto come gate.

[5] detta anche *regione di espansione*

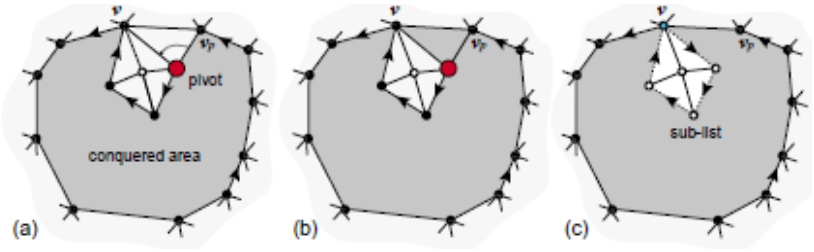
La regione di accrescimento viene espansa progressivamente secondo regole stabilite dal particolare algoritmo, fintanto che la mesh non è conquistata interamente. Il procedimento può essere riassunto nei seguenti passi.

1. Costruzione di una struttura che consenta l'accesso alle incidenze ed alle adiacenze tra gli elementi della mesh
2. Scelta della faccia iniziale. Vengono così inizializzati bordo di taglio, gate e pivot
3. Finché tutta la mesh non è conquistata sono ripetute le seguenti operazioni:
   - Determinazione dell'operazione da eseguire analizzando le incidenze del gate
   - Esecuzione dell'operazione determinata con l'eventuale accrescimento della regione conquistata
   - Aggiornamento delle informazioni di incidenza tra gli elementi e scelta un nuovo gate e pivot, se necessario.

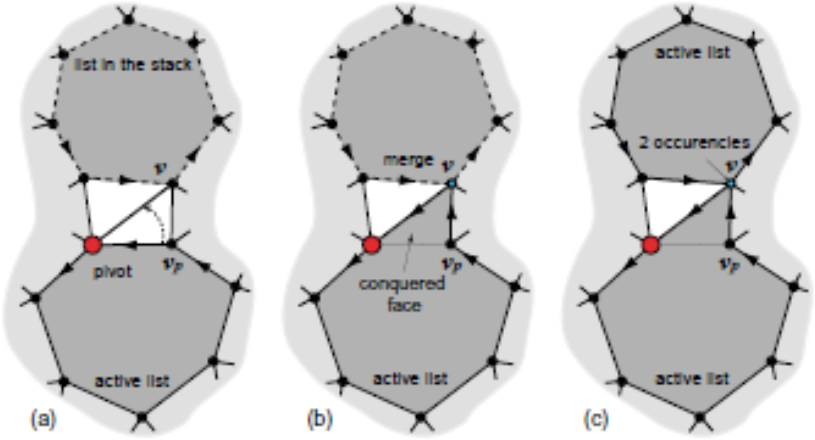
### 3.3.2 Casi particolari: split, merge e mesh con bordi

#### L'operazione Split

Durante l'espansione della regione conquistata può verificarsi il caso in cui il bordo di taglio non sia più un poligono semplice. Ciò accade quando l'algoritmo tenta di conquistare un vertice già processato che appartenga al bordo di taglio corrente. Il verificarsi di questa situazione viene gestito da una procedura denominata *split*, che consiste fondamentalmente nel dividere il bordo di taglio in due parti. Al fine di gestire correttamente la presenza simultanea di più bordi di taglio, questi vengono introdotti in uno *stack*. Tale struttura dati consente di tenere ordinati i bordi di taglio che devono essere ancora processati. Quando si verifica lo *split* di un bordo di taglio, uno dei due bordi generati viene processato immediatamente, mentre l'altro viene inserito nello stack per essere trattato successivamente (si veda la figura 3.3(a) nella pagina seguente).




(a) *Split*: al termine dei passi si ottengono due bordi di taglio




(b) *Merge*: I due bordi di taglio si uniscono

Figura 3.3: Casi particolari: *split* e *merge* (Alliez e Desbrun, 2001)

### L'operazione Merge

Un altro caso particolare che può verificarsi è l'intersezione di due diversi bordi di taglio. Tale situazione viene gestita tramite un'operazione di *merge*. Come avviene nel caso di split, può darsi che si tenti di processare un vertice già conquistato. Se tale vertice appartiene al bordo di taglio corrente si ha uno *split*, altrimenti si ha un *merge* tra il bordo di taglio corrente e quello a cui appartiene il vertice in questione. Quest'ultimo viene unito con il bordo di taglio corrente ed è rimosso dallo *stack* (figura 3.3(b) nella pagina precedente).

### Mesh con bordi

Nel caso in cui la mesh da codificare presenti dei bordi vi sono diversi metodi che possono essere adottati per gestire questa particolarità. L'idea più semplice è quella di considerare il bordo della mesh come il perimetro di una faccia virtuale ad essa complementare. Il problema è risolto aggiungendo alla mesh tante facce virtuali quanti sono i bordi che essa presenta.

Questa tecnica non può essere però adottata se la mesh ha il vincolo di essere composta esclusivamente da triangoli; generalmente il bordo ha più di tre lati e quindi non può essere chiuso con un'unica faccia triangolare. In questo caso è possibile chiudere il bordo della mesh inserendo un vertice *dummy* e tanti triangoli quanti sono i lati del bordo in modo da collegarne ogni spigolo con il vertice dummy.

Si può anche evitare di aggiungere elementi virtuali alla mesh inizializzando lo stack dei bordi di taglio in modo particolare: si creano inizialmente tanti bordi di taglio quanti sono i bordi della mesh. Tutti questi bordi di taglio sono inseriti nello stack. In altre parole, i buchi della mesh sono visti come regioni già conquistate a priori. Per ogni algoritmo incontrato sarà specificato con quale metodo vengono gestiti i bordi della mesh.

### 3.3.3 Metodi basati sulle facce

Nei metodi di codifica della connettività basati sulle facce, le operazioni di accrescimento della regione conquistata sono trattate specificando il tipo di faccia[6] da aggiungere ed altre informazioni riguardanti le relazioni di incidenza con gli elementi adiacenti. In particolare, i metodi descritti in questa sezione varranno solo per mesh di triangoli: il tipo di faccia è ridondante ed i simboli emessi conterranno pertanto solo le informazioni sulle relazioni di incidenza. Ad ogni simbolo corrisponde una particolare operazione. Inoltre, i bordi della mesh non vengono chiusi, ma sono gestiti inizializzando lo stack dei bordi di taglio come spiegato nel paragrafo 3.3.2 nella pagina precedente. Se la mesh non ha bordi, lo stack è inizialmente vuoto e l'algoritmo inizia con un bordo di taglio ottenuto dal perimetro di una faccia scelta arbitrariamente. Le operazioni di conquista sono spiegate di seguito e mostrate in figura 3.4 nella pagina successiva.

1. L'operazione *centro* $C$ aggiunge alla regione conquistata il triangolo adiacente al gate. Il numero dei vertici attici aumenta di uno ed il nuovo gate è collocato in modo tale che punti al pivot.

2. L'operazione *destra* $R$ si verifica quando il triangolo da aggiungere è incidente sia al gate, che allo spigolo successivo del bordo di taglio. In questo caso il triangolo è aggiunto alla regione conquistata e viene scelto un nuovo pivot ed un nuovo gate.

3. L'operazione *sinistra* $L$ è analoga alla $R$, con la differenza che il triangolo da aggiungere è adiacente al gate ed allo spigolo precedente del bordo di taglio. In questo caso non c'è bisogno di scegliere un nuovo pivot, ma soltanto il gate, in modo tale che punti ad esso.

4. L'operazione di *fine* $E$ si verifica quando il triangolo da aggiungere ha tutti e tre gli spigoli incidenti al bordo di taglio. È l'operazione con la quale il bordo di taglio si chiude. Se l'intera mesh è stata codificata il processo di compressione termina.

[6] il numero dei lati

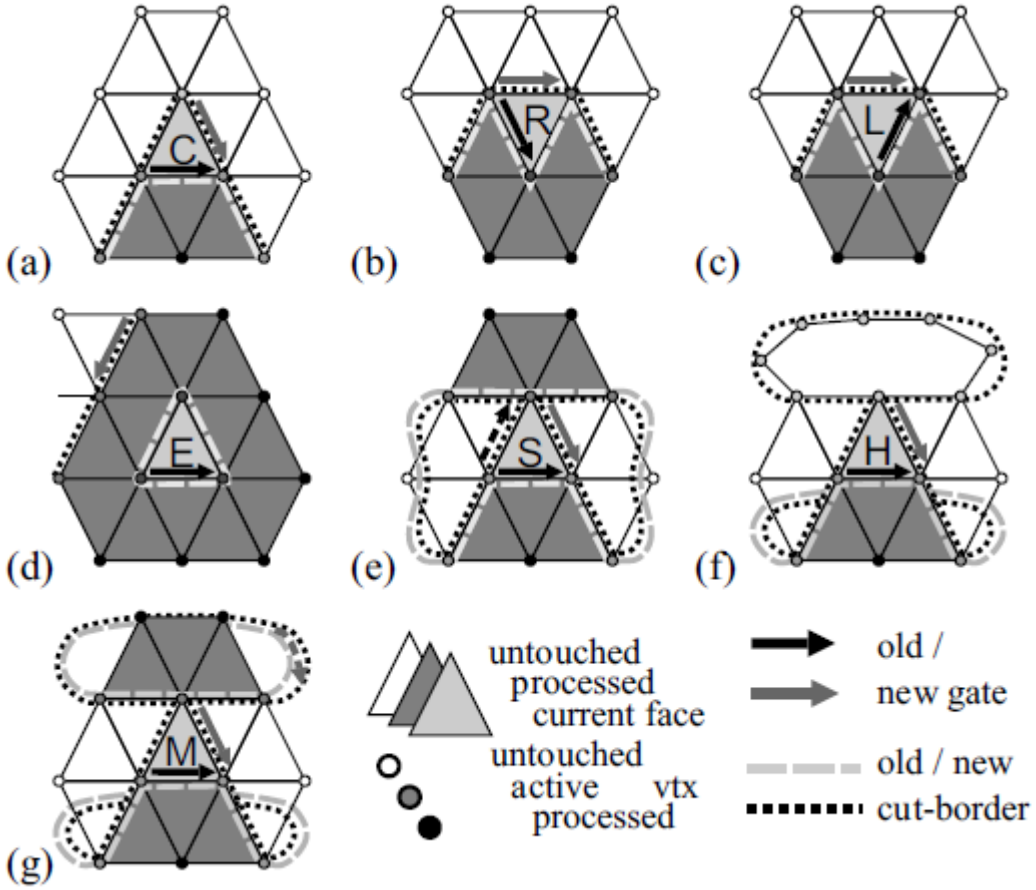



Figura 3.4: Operazioni di conquista dei metodi di codifica basati sulle facce validi per mesh di triangoli (Gotsman *e altri*, 2001)

5. L'operazione *split* $S_i$ si presenta quando il bordo di taglio cresce su sé stesso (figura 3.3(a) nella pagina 57). Il vertice interessato è chiamato *split vertex* e ne viene specificato l'indice $i$, cioè la posizione che ha tale vertice all'interno del bordo di taglio relativamente al *pivot*. Il bordo di taglio viene quindi diviso in due parti, una delle quali è inserita nello stack, mentre la codifica prosegue con l'altra.

6. L'operazione *merge* $M_{s,i}$ corrisponde alla situazione mostrata in figura 3.3(b) nella pagina 57. Il bordo di taglio corrente è unito con quello alla posizione $s$ nello *stack*, a partire dall'$i$esimo vertice[7].

7. L'operazione *hole* $H_l$ si presenta quando il bordo di taglio corrente cresce verso un bordo della mesh. Con l'indice $l$ è specificata la lunghezza del bordo in termini di numero degli spigoli che lo compongono. Dopo l'operazione *hole* il bordo diventa parte del bordo di taglio corrente.

### Cut-Border Machine

Il metodo Cut-Border Machine (Gumhold e Straßer, 1998) è un esempio reale di metodo basato sulle facce. In questo algoritmo l'alfabeto è composto dai simboli $\mathcal{A} = \{C, R, L, H, M, S_i\}$ con $i = \pm 2, \pm 3, \ldots$ indici delle operazioni di *split*. All'alfabeto fanno inoltre parte altri simboli che vanno a rappresentare gli indici per le operazioni di *merge* e *hole*. Più precisamente, è necessario un numero di indici per le operazioni di merge pari al *genus* della mesh, ed un numero di indici per le operazioni di hole pari al numero di bordi.

All'alfabeto non fa parte il simbolo corrispondente all'operazione *fine* in quanto tale situazione si presenta solo quando la lunghezza del bordo di taglio valga esattamente tre. In questo caso un'operazione *sinistra*, *destra* o *fine* produrrebbe il medesimo risultato. L'operazione di *fine* viene quindi codificata riutilizzando il simbolo $L$ od $R$.

La fase di decodifica di una mesh compressa con Cut-Border Machine viene eseguita nello stesso ordine e seguendo gli stessi passi della codifica.

[7] La posizione è relativa al v*pivot*

**Edge Breaker**

Il metodo Edge Breaker (Rossignac, 1999), pur facendo parte delle tecniche basate sulle facce, presenta delle differenze sostanziali rispetto al precedente algoritmo. Una prima differenza consiste nel fatto che le operazioni di *split* sono codificate senza indicare l'indice $i$. Questa cosa è possibile soltanto se l'operazione di *fine* viene indicata esplicitamente. L'indice del vertice di split infatti può essere ricavato analizzando la tipologia delle operazioni da eseguire a partire dal momento di split fino all'operazione di fine corrispondente. È sufficiente notare che le operazioni $L$ ed $R$ decrementano la lunghezza del bordo di taglio di uno, le operazioni $C$ e $S$ la incrementano di uno, mentre le $E$ la decrementano di tre. L'indice di split $i$ può pertanto essere ricavato con la semplice formula $i = \#_R + \#_L - \#_C - \#_S + 3\#_E - 1$.

A causa di questa particolarità, la decodifica di una mesh viene eseguita in ordine *inverso*, partendo cioè dall'ultimo dei simboli $E$ della sequenza compressa. Per ulteriori informazioni si rimanda a Isenburg e Snoeyink (2001). Un esempio di questo tipo di codifica è illustrato in figura 3.5 nella pagina seguente.

### 3.3.4 Metodi basati sugli spigoli

L'algoritmo di codifica basata sugli spigoli analizzato in questa sezione è quello conosciuto sotto il nome di *Face Fixer* (Isenburg e Snoeyink, 2000). L'alfabeto è composto dall'insieme dei simboli $\mathcal{A} = \{F_l, L, R, S, E, H_l, M_{s,i,l}\}$. La codifica produce un numero di simboli pari esattamente al numero degli spigoli della mesh. Per ogni faccia di $l$ lati si ha in output un simbolo $F_l$; analogamente, per ogni bordo lungo $l$ si ha un simbolo $H_l$. Inoltre, per ogni grado del genus della mash si ha un simbolo $M_{s,i,l}$, dove i tre indici stanno ad indicare:

- $s$: la posizione del secondo bordo di taglio all'interno dello *stack*
- $i$: la posizione del gate all'interno del secondo bordo di taglio
- $l$: la lunghezza del secondo bordo di taglio

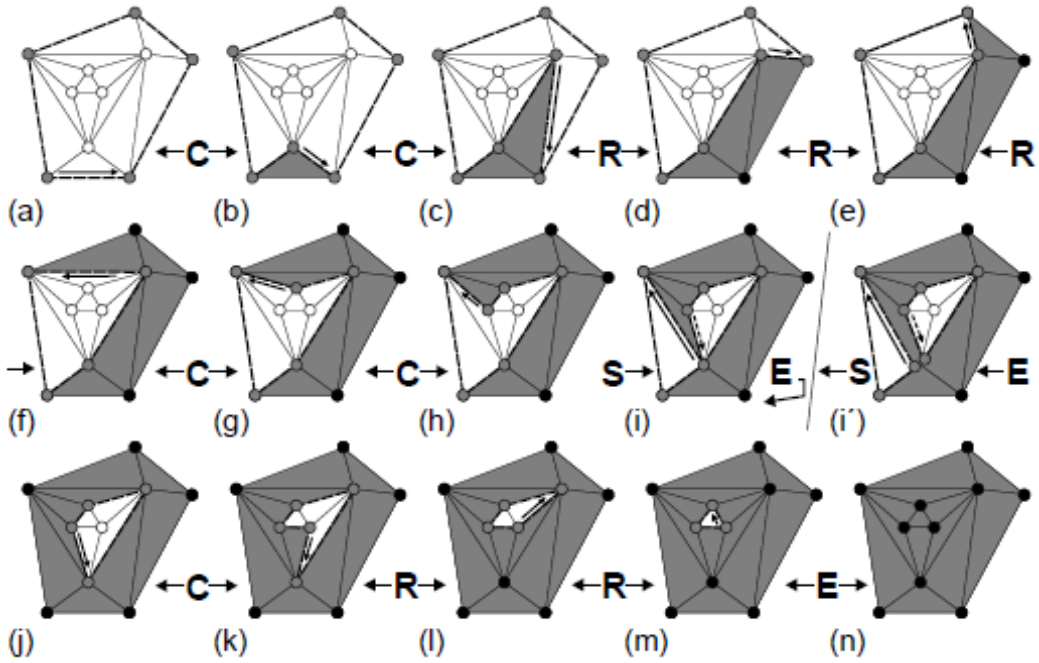



Figura 3.5: Esempio di codifica basata sulle facce (Gotsman *e altri*, 2001)

Le operazioni corrispondenti ai simboli appena presentati sono descritte di seguito.

- $F_l$: Il gate è adiacente ad una faccia con $l$ lati non ancora processata. Il bordo di taglio viene esteso su questa faccia ed il gate viene posizionato in avanti sull'ultimo lato della faccia appena introdotta.

- $R$ ($L$): Il gate è adiacente rispettivamente al prossimo (precedente) spigolo del bordo di taglio. Il gate è quindi ricollocato sul precedente (prossimo) spigolo.

- $S$: Il gate risulta essere adiacente ad uno spigolo del corrente bordo di taglio, ma tale spigolo non è né il suo precedente, né il suo successivo. In questo caso si ha uno *split* del bordo di taglio, di cui una parte è inserita nello *stack* per essere processata successivamente.

- $E$: Il gate è adiacente ad uno spigolo del bordo di taglio che risulta essere sia il precedente, che il successivo. In altre parole il bordo di

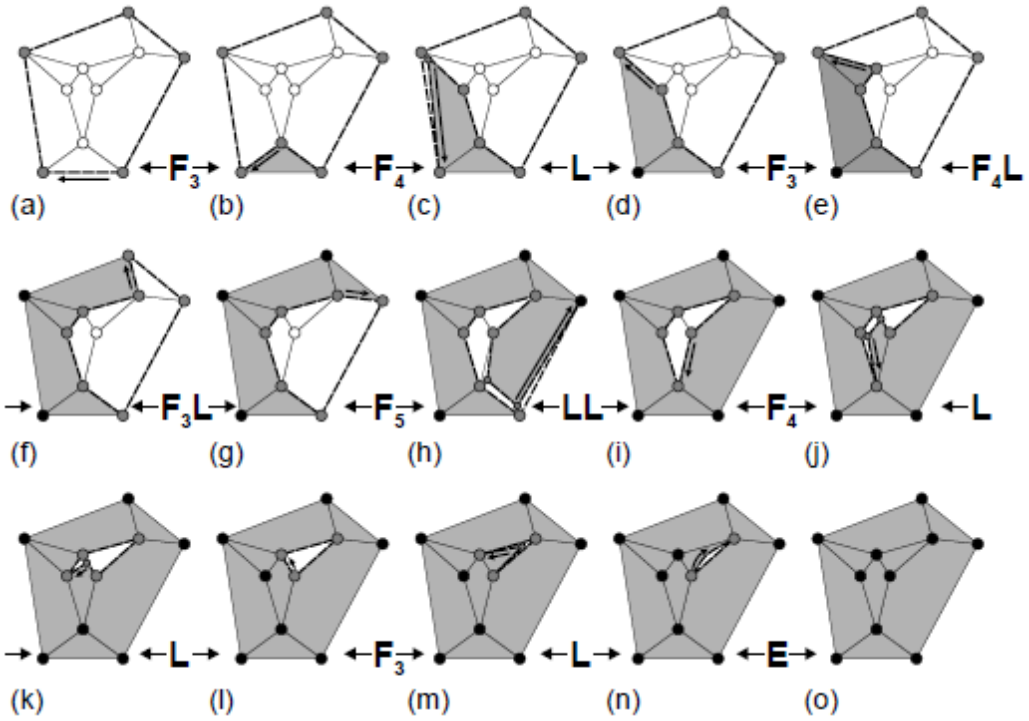



Figura 3.6: Esempio di codifica basata sugli spigoli (Gotsman *e altri*, 2001)

taglio è composto da due soli spigoli: la sua fase di espansione può dirsi completata ed è rimosso dallo stack.

- $H_l$: Il gate è adiacente ad un bordo di $l$ lati non ancora processato. Si aggiungono gli spigoli della lacuna al bordo di taglio.

- $M_{s,i,l}$: Il gate è adiacente ad uno spigolo già processato che fa parte di un bordo di taglio diverso da quello corrente. Come ormai è noto, l'operazione da eseguire è un *merge* tra questi due bordi di taglio.

Un esempio di codifica ottenuta applicando l'algoritmo Face Fixer è illustrata in figura 3.6.

### 3.3.5 Metodi basati sui vertici

La tecnica di codifica basata sui vertici è stata sviluppata da Touma e Gotsman (2000) e successivamente revisionata ed ottimizzata da Alliez e

Desbrun (2001). Il successo di questo algoritmo è dovuto al fatto che l'idea di base è molto semplice ed il suo utilizzo porta ad ottimi risultati. Prima di parlare in dettaglio dell'algoritmo vero e proprio è utile presentare alcune definizioni importanti.

### Definizioni

- *Lista Attiva*: corrisponde con il bordo di taglio visto per i precedenti algoritmi. Si tratta di una lista ciclica di vertici adiacenti a due a due che dividono la mesh in due parti: la parte *conquistata* e quella *non conquistata*.
- *Focus*: un vertice della lista attiva viene denominato *focus*. Non è altro che l'analogo del *pivot*. Il *focus* è l'elemento centrale dell'algoritmo e tutte le operazioni sono eseguite attorno ad esso.
- *Valenza*[8] *di un Vertice*: indica il numero di spigoli che incidono sul vertice considerato.
- *Vertice Libero*: è un vertice non ancora processato.
- *Spigolo Libero*: è uno spigolo non ancora processato.
- *Vertice Pieno*: è un vertice di cui nessuno spigolo incidente è libero.
- *Offset*: dati due vertici appartenenti ad una stessa lista attiva, si chiama *offset* il numero di vertici che li separa contandoli in senso antiorario.

### Funzionamento dell'algoritmo

L'algoritmo è stato studiato per mesh composte da soli triangoli. In una fase di preprocessing, i bordi ed i buchi della mesh sono chiusi con la tecnica del *dummy vertex* (vedi la sezione 3.3.2 nella pagina 58) ed a ciascun vertice è associata la rispettiva *valenza*.

[8] spesso vi si riferisce anche con il termine *grado*

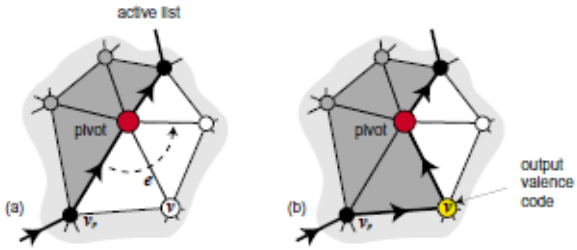



Figura 3.7: Principio di funzionamento della conquista con il *valence based encoding* (Alliez e Desbrun, 2001)

I passi che caratterizzano l'algoritmo vero e proprio seguono lo schema introdotto nella sezione 3.3.1 nella pagina 55, ma in questa sezione ne saranno mostrati maggiori dettagli. Inizialmente è scelto un triangolo della mesh in modo arbitrario ed è inizializzata una lista attiva con i suoi tre vertici. Uno di essi è eletto *focus*. L'algoritmo va a codificare i vertici adiacenti al focus procedendo in senso antiorario. Sia $v_{focus}$ il focus ed $e$ il primo spigolo libero incontrato percorrendo tutti gli spigoli che insistono su di esso in senso antiorario. Si chiami $v$ l'altro estremo di $e$ e $v_p$ il vertice che precede $v_{focus}$ nella lista attiva. Se $v$ è *libero* la conquista coinvolge il vertice $v$ stesso, gli spigoli $e$ e $\{v, v_p\}$ ed il triangolo $\{v_{focus}, v, v_p\}$. Il vertice $v$ è inserito nella lista attiva nella posizione immediatamente precedente al focus. Il passaggio risulta più chiaro analizzando la figura 3.7. Ad ogni vertice conquistato corrisponde un simbolo $A_i$ in uscita che ne codifica la corrispondente *valenza* $i$. Se $v$ è un *dummy vertex* la sua conquista è codificata ugualmente con il simbolo $A_i$, ma in questo caso l'indice $i$ rappresenta la valenza cambiata di segno[9] per contraddistinguerlo dagli altri.

Nel caso in cui il vertice $v$ non fosse libero significa che stiamo considerando un caso particolare di *split* o *merge*, come spiegato nella sezione 3.3.2 nella pagina 56. Ricordiamo che si ha uno *split* della lista attiva se $v$ appartiene alla lista attiva corrente; mentre se $v$ appartiene ad un'altra lista attiva si ha il *merge* tra quest'ultima e la lista attiva corrente. Le operazioni di *split* e *merge* sono codificate in uscita rispettivamente con i simboli $S_i$ e $M_{i,s}$, dove $i$ rappresenta l'*offset* del vertice di split/merge ed $s$ la posizione

[9] per i *dummy vertex* si ha che $i < 0$

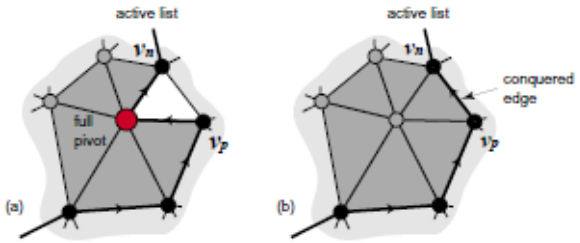



Figura 3.8: Rimozione del focus pieno (Alliez e Desbrun, 2001)

che lista attiva coinvolta nel *merge* ha all'interno dello *stack*.

La tecnica di codifica basata sui vertici non ha ulteriori tipi di operazioni. L'alfabeto $\mathcal{A}$ risulta essere formato dai soli simboli illustrati fino a questo punto $\mathcal{A} = \{A_i, S_i, M_{i,s}\}$.

### Rimozione dei vertici pieni

Fino ad adesso il procedimento di conquista è stato presentato sotto l'ipotesi che il *focus* abbia almeno uno spigolo libero ad esso incidente; se però un tale spigolo non può essere trovato, significa che il focus è *pieno*. Più dettagliatamente, se $v_n$ rappresenta il vertice successivo al focus nella lista attiva, il fatto che il focus sia *pieno* comporta la conquista del triangolo $\{v_n, v_{focus}, v_p\}$ e dello spigolo $\{v_n, v_p\}$ (si veda la figura 3.8). A questo punto non è più possibile conquistare ulteriori elementi attorno al focus, che viene rimosso dalla lista e va a far parte della *regione conquistata*. Il vertice della lista attiva immediatamente successivo a quello appena rimosso viene eletto nuovo *focus*.

Si noti che la conquista legata alla presenza di un vertice pieno non provoca l'emissione di alcun simbolo in uscita. Il decoder possiede infatti tutti gli elementi necessari a dedurre questa situazione: esso deve tenere il conteggio degli spigoli aggiunti attorno al focus durante la fase di ricostruzione e confrontarne il numero con la valenza associata. Quando il numero di spigoli incidenti è uguale alla valenza, vuol dire che il vertice è pieno.

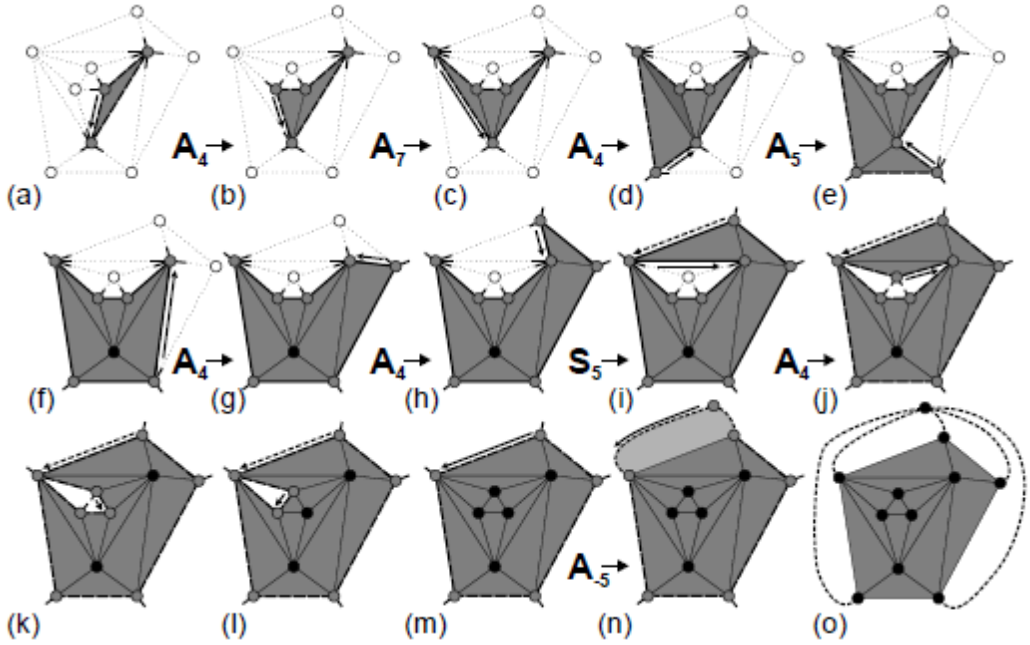



Figura 3.9: Esempio di codifica basata sui vertici (Gotsman *e altri*, 2001)

### Termine del processo di codifica

La conquista di nuovi vertici comporta un aumento della lunghezza della lista attiva, mentre la rimozione di vertici pieni ne provoca un accorciamento. La codifica della mesh prosegue fintanto che tutti i vertici non sono diventati *pieni*. Come è possibile immaginare, al termine del processo di codifica la lista attiva tende a chiudersi su sé stessa mano a mano che tutti i vertici pieni ne vengono rimossi, fino a svuotarsi completamente. Quando la lista attiva corrente è vuota non è ancora certo che l'algoritmo sia terminato: è necessario controllare se ci sono altre liste attive in attesa di essere processate all'interno dello *stack*, derivate ad esempio da una o più operazioni di *split*. In questo caso la conquista prosegue con la lista attiva che sta alla testa dello *stack*. Il processo ha termine quando non vi sono più liste attive nello *stack*.

Un esempio di codifica basata sui vertici è illustrata nella figura 3.9. La decodifica ricostruisce la mesh ripetendo esattamente gli stessi passi compiuti nella fase di codifica.

**Possibili miglioramenti**

Alliez e Desbrun (2001) hanno mostrato una serie di miglioramenti all'algoritmo di cui vale la pena citare una strategia con la quale si riduce il numero di operazioni di split. Per mesh tassellate in modo irregolare si possono infatti avere numerose e frequenti operazioni di split. Lo svantaggio di avere molti simboli $S_i$ è che l'alfabeto cresce notevolmente in quanto gli indici $i$ non sono statisticamente predicibili e presentano una grossa variabilità[10]. Come noto, un codice il cui alfabeto è molto grande viene compresso meno efficientemente usando metodi di compressione ad entropia minima tipo Huffman.

La soluzione proposta da Alliez e Desbrun (2001) è un diverso modo di scegliere il *focus* nella lista attiva. Invece di sceglierlo arbitrariamente, è denominato focus quel vertice della lista attiva che ha il minor numero di *spigoli liberi*. In caso di parità viene considerato come fattore di decisione il numero medio di spigoli liberi dei vertici della lista attiva circostanti a quelli contendenti. In questo modo si riesce a ridurre il numero di *split* anche del 70%.

## 3.4 Tecniche di compressione della geometria

Mentre nella precedente sezione sono stati presentate delle tecniche specifiche per comprimere la connettività di una mesh, di seguito saranno mostrati alcuni metodi per rappresentare le informazioni relative alla geometria in un modo più compatto.

### 3.4.1 Quantizzazione delle coordinate

Come già accennato nella sezione relativa alle tecniche di codifica *lossy* (sezione 3.2.4 nella pagina 52), il primo approccio che si può usare per migliorare l'efficienza della rappresentazione della geometria di una mesh è quello di *quantizzare* le coordinate dei vertici. Spesso succede infatti che le mesh

[10] a differenze dei simboli $A_i$, in cui l'indice $i$ di solito vale 6 o valori ad esso prossimi (tra 4 ed 8 generalmente)

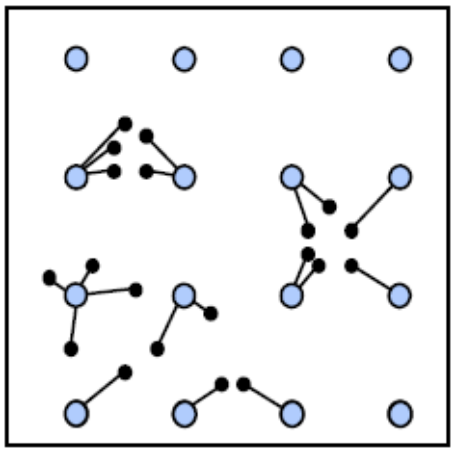


Figura 3.10: Esempio di quantizzazione uniforme

sono ottenute con dispositivi a scansione tridimensionale. Questi dispositivi hardware sono dotati di un'alta precisione (24-32 bits) che sovente non è necessaria e può rappresentare una forma di rumore. Anche usare numeri in virgola mobile per rappresentare le coordinate dei vertici in molti casi può essere un inutile spreco: i 32 bits del tipo *float*, o addirittura i 64 bits del tipo *double*, permettono di posizionare i vertici praticamente in qualsiasi punto dello spazio; ma potrebbero essere sufficienti soltanto un ristretto insieme di livelli. Per questi motivi il primo passo nella compressione della geometria è la *quantizzazione*: vale a dire che ciascun vertice è spostato verso il punto di una griglia tridimensionale ad esso più vicino (*quantizzazione uniforme*, vedi figura 3.10).

Considerando che una mesh ha $n$ vertici $V = \{v_1, \ldots, v_n\}$, che la griglia è composta da $k$ valori quantizzati $Q = \{q_1, \ldots, q_k\}$, e che la funzione che mappa i vertici ai punti della griglia è $m : V \mapsto Q$, dopo tale processo si ottiene un *errore* pari a:

$$E(v, m) = \sum_{i=1}^{n} \|v_i - m(v_i)\|^2 \tag{3.8}$$

Per minimizzare l'errore è necessario che la regione di spazio occupata dalla griglia sia più piccola possibile, altrimenti alcuni livelli rimangono inutilizzati. Per questo motivo la griglia viene scelta in modo che le sue

dimensioni coincidono con la *bounding box* della mesh, cioè il più piccolo parallelepipedo che la contiene interamente. Sovente si fa coincidere la regione di spazio della griglia con la cosiddetta *axis aligned bounding box*[11], cioè una bounding box i cui spigoli sono paralleli agli assi principali dello spazio.

Si consideri ad esempio il solo asse delle $x$. Si supponga inoltre che la bounding box abbia come estremo sinistro $x_0$ e che sia lunga $l$: l'insieme $Q_x$ dei $k$ livelli quantizzati è dato da $Q_x = \{x_0, x_0 + \frac{l}{k-1}, x_0 + \frac{2l}{k-1}, \ldots, x_0 + l\}$. In modo analogo sono definiti gli insiemi $Q_y$ e $Q_z$, per un totale di $k^3$ valori. Ciascun vertice può quindi essere rappresentato da una tripletta di interi ciascuno appartenente all'insieme $\{0, \ldots, k-1\}$. Nell'encoder e nel decoder assume quindi vitale importanza la posizione e la dimensione della bounding box della mesh.

Possono essere usate altre forme di quantizzazione mirate a minimizzare l'errore (3.8), ma che sono solitamente molto più costose in termini computazionali.

### 3.4.2 Tecniche di predizione dei vertici

Le tecniche di predizione dei vertici sono utilizzate al fine di ottenere dei simboli che si apprestano meglio ad essere compressi con una codifica ad entropia minima del tipo mostrato nella sezione 3.2.1 nella pagina 47. Il vertice predetto sarà vicino al vertice reale quanto più è buono l'algoritmo di predizione usato. Basta quindi codificare l'*errore di predizione*, cioè la differenza tra il vertice predetto e quello reale, che avrà una distribuzione tipicamente centrata nello 0 ed un dominio di valori molto ristretto.

La formula generica di un *algoritmo di predizione lineare*[12] è data dalla seguente:

$$v_i^p = \sum_{j=1}^{k} a_j v_{i-j} \tag{3.9}$$

Vale a dire che il vertice predetto è la *combinazione lineare* dei vertici già

[11] spesso abbreviata con AABB
[12] LPC, *linear prediction coding*

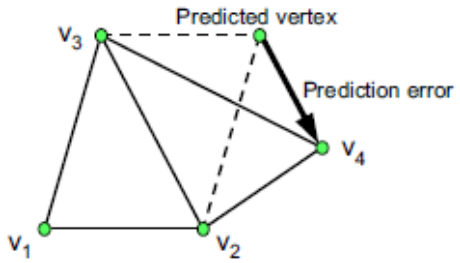



Figura 3.11: Regola del parallelogrammo

processati, solitamente un numero molto ristretto. L'esempio più semplice di predizione lineare è quello in cui i coefficienti della combinazione lineare valgono $a_1 = 1, a_2 = a_3 = \ldots = a_k = 0$. Più in generale per minimizzare l'errore di predizione si scelgono i coefficienti in modo tale che:

$$(a_1, \ldots, a_k) = \arg\min \sum_{i=k+1}^{n} \left\| v_i - \sum_{j=1}^{k} a_j v_{i-j} \right\|^2 \tag{3.10}$$

Risolvere questo sistema richiede una quantità di calcoli molto elevata. Ma andando ad analizzare la struttura triangolare della mesh si nota che è possibile usare un metodo di predizione lineare molto efficiente e che consente di ottenere ottimi risultati.

Il metodo in questione è la cosiddetta *regola del parallelogrammo*, che prende il nome dalla sua interpretazione geometrica. Il principio empirico su cui si basa è infatti quello secondo cui due triangoli adiacenti di una mesh tendono a formare un parallelogrammo.

Il funzionamento di questo metodo è illustrato nella figura 3.11. Quando l'algoritmo va a codificare il vertice $v_4$ è già a conoscenza del triangolo $(v_1, v_2, v_3)$. Il quarto vertice può essere predetto come $v_4^p = v_3 + v_2 - v_1$. Nella figura 3.12 nella pagina successiva è mostrata una tipica distribuzione dell'errore di predizione ottenuta applicando la regola del parallelogrammo.

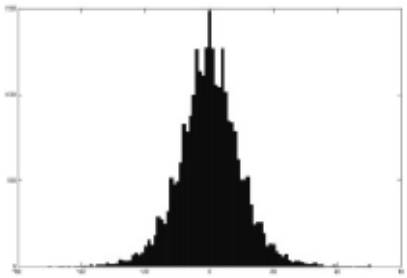


Figura 3.12: Distribuzione dell'errore di predizione con la regola del parallelogrammo (Gotsman *e altri*, 2001)

# Capitolo 4

# Integrazione delle tecniche presentate

## 4.1 XVR: Extreme Virtual Reality

XVR è un framework integrato per lo sviluppo di complesse applicazioni di realtà virtuale sviluppato a partire dal 2001 presso il laboratorio PERCRO, attualmente XVR sta alla base di numerosi progetti di *rendering* in tempo reale, di *interazione* e di *simulazione* di sistemi fisici. Il framework è tenuto costantemente aggiornato secondo le necessità ed il continuo evolversi delle tecnologie. Alcuni esempi di applicazioni XVR sono illustrati nella figura 4.1 nella pagina seguente.

Attualmente XVR offre una vasta serie di funzionalità utili a controllare molti degli aspetti relativi alla programmazione di realtà virtuali, quali *real-time graphics*, *suono olofonico*, *interazione*, controllo di *interfacce aptiche* e *custer rendering* (CAVE) (figura 4.2 nella pagina successiva).

Le applicazioni XVR sono sviluppate attraverso un dedicato linguaggio di scripting (S3D) con il quale i programmatori hanno a disposizione una serie di comandi e costrutti unificati che permettono di controllare ogni aspetto del sistema, dalle animazioni di modelli 3D all'interazione con le interfacce aptiche.

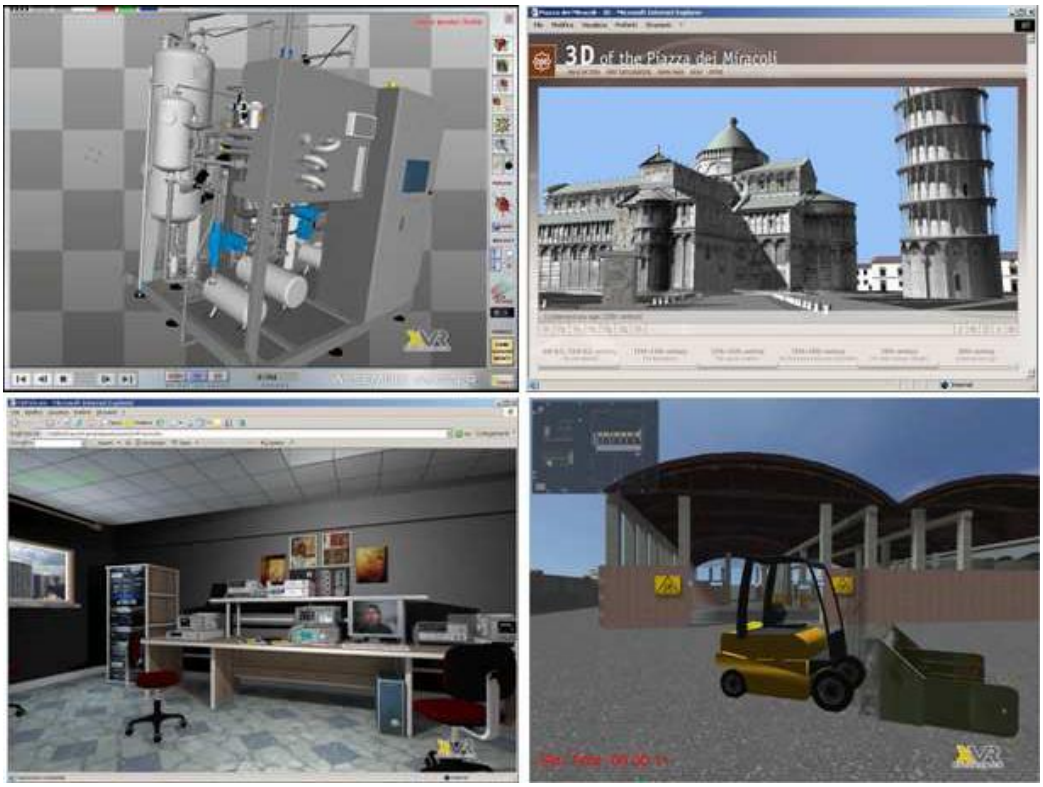



Figura 4.1: Esempi di applicazioni XVR

Figura 4.2: Esempi di sistemi controllati da applicazioni XVR

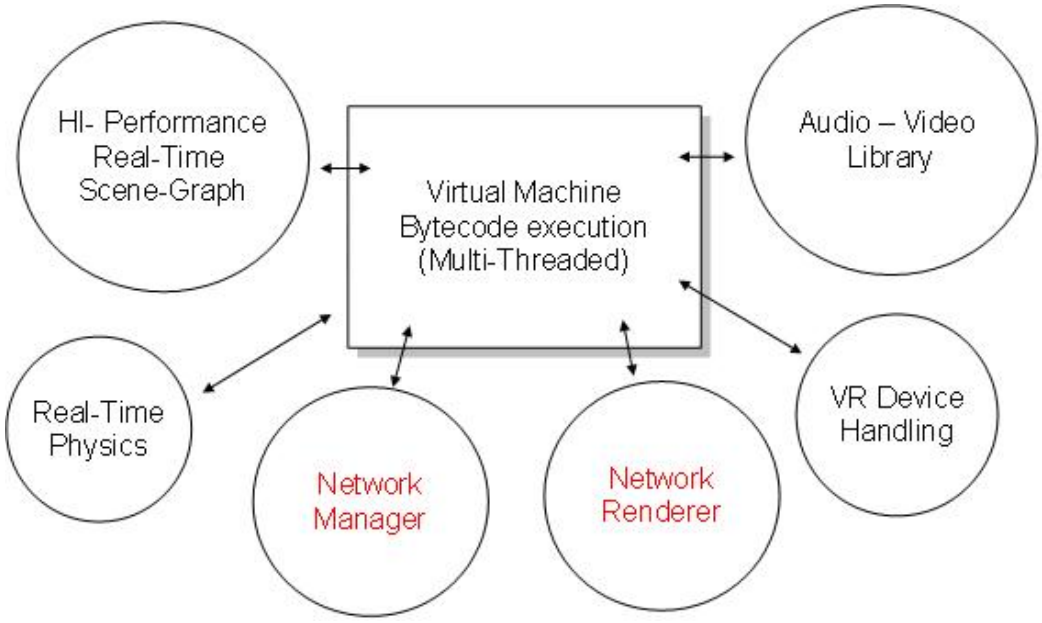



Figura 4.3: L'architettura del framework XVR

### 4.1.1 L'architettura

L'elemento che sta alla base di XVR è una *Virtual Machine*[1] capace di interpretare ed eseguire i programmi scritti con il linguaggio S3D. Tale *Virtual Machine*, come molte altre VMs, possiede un proprio set di istruzioni *byte code*, un set di *registri*, uno *stack* ed un'area di memoria dove memorizzare i metodi.

Il linguaggio di scripting S3D permette di specificare il comportamento dell'applicazione mettendo a disposizione i più comuni costrutti di base ed una serie di *classi* e *funzioni* specifiche per la programmazione di VR. Gli script S3D sono compilati in *byte code* in modo da poter essere eseguiti dalla XVR-VM. Quest'ultima esegue il codice richiamando le funzioni di una libreria software ad alte prestazioni che copre in modo esaustivo tutte le funzionalità tecniche richieste per lo sviluppo di applicazioni complesse, come la visualizzazione di grafica in tempo reale, lo scambio di dati sulla rete locale o il controllo di interfacce esterne dotate ad esempio di dispositivi con ritorno di

[1]per una definizione di *macchina virtuale* si veda ad esempio http://it.wikipedia.org/wiki/Macchina_virtuale. Il termine può essere abbreviato con la sigla VM

forza. Il programmatore può estendere inoltre le funzionalità del linguaggio S3D già esistenti attraverso librerie esterne (DLLs[2]) personalizzate[3].

L'utilizzo di un linguaggio di scripting dedicato per lo sviluppo di applicazioni di VR ha due importanti conseguenze: una di queste consiste nel fatto che il programmatore ha a disposizione un linguaggio unico ed integrato con tutte le funzionalità necessarie per programmare un'applicazione completa. Questo rappresenta il punto di forza di questa tecnologia, permettendo di ridurre notevolmente il tempo necessario allo sviluppo di un'applicazione di VR rispetto ad altri framework basati sul linguaggio C++. L'altra conseguenza sta nel fatto che, in quanto linguaggio di scripting, la sua esecuzione non è così veloce come quella del codice assembly generato direttamente da un compilatore C++. In altre parole l'uso di un linguaggio di scripting introduce alcune penalizzazioni sulle prestazioni dell'applicazione sviluppata. Tuttavia tali penalizzazioni sono appena percettibili nella maggioranza dei casi: il tempo di esecuzione viene infatti impiegato soprattutto nell'esecuzione delle funzioni delle librerie ad alte prestazioni realizzate a sua volta in C++. Il linguaggio di scripting può essere visto come l'elemento *glue*, che tiene collegate tra loro tutte le varie tecnologie ed il cui codice solitamente è molto più semplice, chiaro e conciso rispetto a listati C o C++.

### 4.1.2 L'ambiente di sviluppo XVR

La piattaforma XVR è distribuita con alcuni programmi ed utilità per la progettazione e l'implementazione di applicazioni di VR. Nella figura 4.4 è possibile osservare l'IDE[4] ufficiale del framework, denominato XVR Studio. Tra le sue funzionalità vi sono:

- un editor per il linguaggio di scripting S3D
- un project manager
- un compilatore

[2] *dynamic linked library*
[3] si veda la sezione 4.2 nella pagina 81
[4] *Integrated Development Environment*

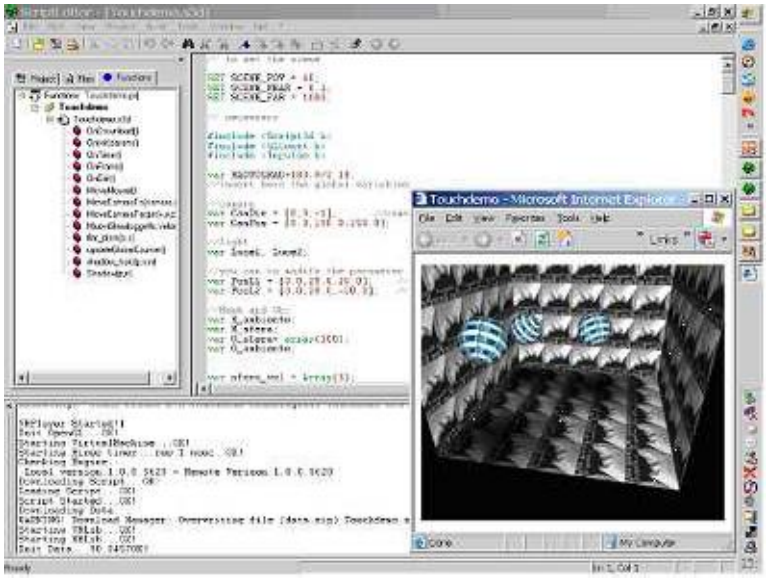



Figura 4.4: XVR Studio

- un modulo per pubblicare il progetto sul web
- una finestra di output dove vengono mostrate informazioni utili alla fase di *debug* e di *profiling*[5]

Al fine di velocizzare ulteriormente la preparazione di un'applicazione interattiva, XVR Studio offre un pratico *wizard* che guida il programmatore durante il setup dell'applicazione con una serie di passi. Con tale *wizard* è possibile scegliere i parametri di base relativi alla scena virtuale, tra cui ad esempio il numero e la posizione delle luci e del punto di vista. Il *wizard* produce inoltre il codice che implementa un sistema di navigazione dell'ambiente virtuale tramite l'uso del mouse del calcolatore.

Il risultato di questa fase si inizializzazione è uno script S3D completo, in cui sono presenti alcune sezioni contrassegnate dalla sigla *to-do* su il programmatore può intervenire per personalizzare l'applicazione inserendo il proprio codice.

[5]cioè l'analisi delle prestazioni in termini di *frequenza* e *durata* delle chiamate a funzione

### 4.1.3 Il linguaggio di scripting S3D

Indipendentemente dalla complessità dall'applicazione che si desidera sviluppare, uno script S3D deve fornire l'implementazione di almeno sei funzioni fondamentali. Tali funzioni fanno parte del linguaggio e costituiscono la base del funzionamento di tutte le applicazioni XVR:

- onDownload()
- OnInit()
- OnFrame()
- OnUpdate()
- OnEvent()
- OnExit()

La funzione OnDownload() è la prima ad essere eseguita dal framework e serve a reperire tutte le risorse[6] necessarie al corretto funzionamento dell'applicazione.

La corpo della funzione OnInit() serve invece ad eseguire tutti i comandi di inizializzazione dell'applicazione. Tali comandi sono eseguiti sequenzialmente. Nessuna delle altre funzioni viene richiamata fintanto che la OnInit() non è terminata.

La funzione OnFrame() è invece quella responsabile alla produzione di output grafico. Dal momento che il contesto grafico è accessibile soltanto durante l'esecuzione di tale funzione, essa è l'unica che può e deve contenere tutte le chiamate a funzioni e metodi atti a tale scopo. Se i comandi di grafica vengono richiamati dal corpo di altre funzioni, essi non produrranno alcun effetto.

Un'altra funzione predefinita non elencata sopra, ma che può essere utile implementare in molte situazioni è la OnTimer(). Essa è eseguita a ad intervalli regolari indipendentemente dalla OnFrame(). Si tratta del luogo giusto

[6] modelli 3D, librerie personalizzate, ecc.

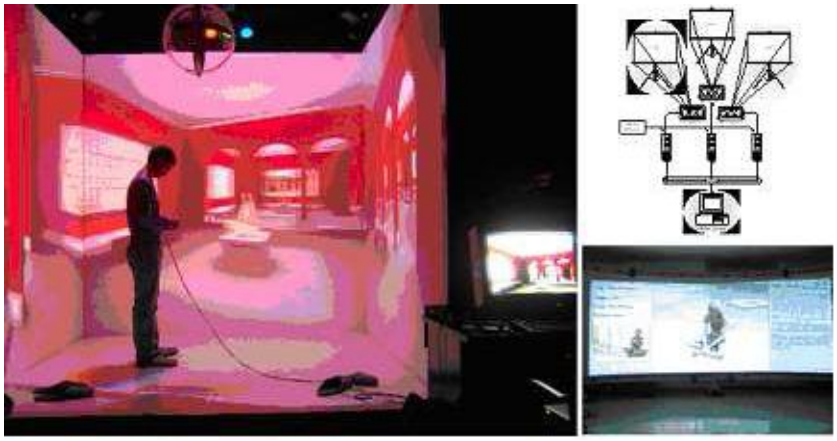


Figura 4.5: XVR cluster rendering

dove inserire quei comandi che non sono correlati con il *rendering* grafico. E' possibile impostare alcuni parametri per selezionare la frequenza di esecuzione di questa funzione che, grazie all'alta risoluzione del *timer*, può essere richiamata fino a mille volte al secondo.

La funzione OnEvent() è invece indipendente sia dalla OnFrame() che dalla OnTimer(). Essa è richiamata ogni qual volta l'applicazione riceve dei *messaggi*. Questi *messaggi* possono essere generati *esternamente* dal sistema operativo, oppure *internamente* dal programma XVR stesso. Lo scambio di *messaggi* è utile alle applicazioni XVR per avere la flessibilità di poter eseguire taluni compiti che non è necessario svolgere ad intervalli regolari di tempo.

Infine, la funzione OnExit() è richiamata nel momento in cui l'applicazione termina la propria esecuzione o quando l'utente chiude la finestra che la contiene.

### 4.1.4 Cluster rendering

Per offrire all'utente un elevato livello di immersione, molte delle installazioni di Realtà Virtuale fanno uso delle tecnologie a proiezione. Al fine di ottenere un miglioramento globale della risoluzione od un più esteso campo visivo (come in un CAVE), i più complessi sistemi a proiezione esistenti

attualmente sono composti da molteplici proiettori. Qualora il numero di tali proiettori sia così elevato da non consentirne il pilotaggio con un singolo calcolatore, si rende necessario adottare un sistema di *Cluster Rendering*.

A tale scopo XVR è dotato di speciali *network drivers* capaci di distribuire il compito di rendering di una singola scena su di un cluster di workstations. L'interoperabilità tra gli elementi del cluster è implementata con flussi di dati che viaggiano sulla rete sotto forma di *stream UDP broadcast*. I *network drivers* di XVR correggono automaticamente molti degli errori prospettici tipici di queste installazioni e permettono inoltre di adattare una qualsiasi applicazione XVR ad ambienti di tipo CAVE senza dover apportare alcuna modifica al progetto.

### 4.1.5 Un'applicazione di esempio

Il codice riportato nella figura 4.6 costituisce un esempio di un'applicazione XVR il cui scopo è quello di illustrare la facilità di uso del linguaggio di scripting e di integrazione di più tecnologie interattive. L'applicazione visualizza una mesh roteante su di uno schermo stereoscopico.

## 4.2 Modularità di XVR

### 4.2.1 Importazione di funzioni da librerie esterne

Le funzioni offerte dal framework XVR possono essere aggiunte e personalizzate caricando dinamicamente una o più *librerie DLL esterne*. A tale scopo XVR dispone della classe `CVmExternDLL` con la quale è possibile *caricare* una DLL specificandone il percorso all'interno del filesystem:

```
var library = CVmExternDLL("myLib.dll");
```

Una volta creato, l'oggetto `CVmExternDLL` riferisce la DLL, ma è sostanzialmente vuoto. E' necessario infatti *importare* una ad una le funzioni della libreria che si intendono utilizzare nell'applicazione con il metodo `__AddFunction()`, specificandone la *signature*:

```
#include <Script3d.h>

//Global variables
var M_hello
var O_hello;

function OnDownload()
{
    FileDownload("http://www.cs.ucl.ac.uk/F.Tecchia/helloworld.zip");
}

functon OnInit()
{
            M_hello = LoadMesh("helloworld.wrl")
            O_hello = NewObject(M_hello);

            SetCameraPosition(0,0,10);
            SetFrameRate(30); SetTimer(10);

            SceneSetParam(VR_HEADTRACKER,1);
            SceneSetParam(VR_TRACKER_POSITION,0,0,1.5);
            SceneSetParam(VR_SCREEN_SIZE,2,1.5);
            SceneSetParam(VR_EYE_SEPARATION,0.04);
}

function OnFrame()
{
            glDrawBuffer( GL_BACK_LEFT );
            SceneBegin(VR_STEREO_LEFT);
              hello.Draw();
            SceneEnd();

            glDrawBuffer( GL_BACK_RIGHT );
            SceneBegin(VR_STEREO_RIGHT);
              hello.Draw();
            SceneEnd();
}

function OnTimer()
{
            O_hello.Rotate(0.5,0,1,0);
}

function OnEvent()
{
            //NOT USED
}
```

Figura 4.6: Una semplice applicazione di esempio

```
library.__AddFunction(C_FLOAT, "myFunction", C_PFLOAT, C_INT);
```

In seguito sarà possibile richiamare la funzione importata come metodo dell'oggetto `CVmExternDLL` nel seguente modo:

```
var vec = [1.0, 2.0, 3.0];
var num = library.myFunction(vec, 4);
```

### 4.2.2 Esportazione di funzioni personalizzate

La programmazione di un modulo per XVR deve prevedere l'*esportazione* di un insieme di funzioni (*interfaccia*) affinché esse possano essere importate come mostrato nella precedente sezione.

Supponendo di scrivere un modulo con il linguaggio C++, le funzioni da esportare devono essere dichiarate specificando le parole chiave:

```
extern "C" __declspec(dllexport)
```

Questa notazione indica al *linker* che tali funzioni devono essere rese *pubbliche* ed accessibili con un identificatore che coincide con il nome dato alla funzione stessa.

La funzione `myFunction` usata nell'esempio precedente può essere esportata nel seguente modo:

```
extern "C" __declspec(dllexport)
float myFunction(float *f, int n){
   float a = 0;
   for (int i = 0; i < n; i++)
      a += f[i];
   return a;
}
```

## 4.3 Architettura dei moduli sviluppati

Gli algoritmi di elaborazione dei modelli 3D realizzati sono stati implementati utilizzando il linguaggio C++ e facendo uso della libreria OpenGL

per la comunicazione con il dispositivo grafico dell'elaboratore. Le funzioni e le classi sono state incluse in una libreria dinamica in modo da poter essere importate nell'ambiente di sviluppo per applicazioni interattive XVR.

Il modulo realizzato si compone fondamentalmente di tre componenti:

- Classi che implementano l'algoritmo di semplificazione incrementale di mesh basato sulla *quadric error metric*
- Classi per l'archiviazione e la lettura di mesh utilizzando il metodo di codifica *valence based encoding*
- Integrazione delle due componenti precedenti tramite classi per la gestione della visualizzazione di scene facendo uso dei *livelli di dettaglio* e *view frustum culling*

### 4.3.1 Formato e tipo dei modelli utilizzati

Alcuni dei modelli utilizzati per i test del modulo sono stati forniti dal laboratorio PERCRO nel formato proprietario AAM specifico per l'uso con XVR. Si tratta di un formato ASCII molto complesso, che permette di definire interi scenari, sia statici, che animati. Ciascun oggetto della scena è denominato *subset* ed è definito come un insieme di poligoni che condividono lo stesso *materiale*, dove per materiale si intende l'insieme degli attributi *glMaterial* di OpenGL (componente ambiente, diffusa, speculare, ecc) relativi al calcolo dell'illuminazione. Ogni file AAM specifica il numero $n$ di subsets che compongono la scena, ed il numero $f$ di *frames* dell'animazione[7]. Ciascun frame $f$ deve descrivere la geometria e la connettività degli $n$ subsets, oltre ad eventuali attributi e coordinate utili al *mapping* delle textures, quando presenti. La geometria è descritta con una lista di triplette $(x, y, z)$, mentre la connettività con una tabella di incidenza faccia-vertici, dove a ciascun triangolo corrispondono una tre indici $(v_1, v_2, v_3)$.

Il modulo realizzato è stato pensato per lavorare con modelli statici e non-textured, incentrando l'interesse sulla sola *geometria* e *connettività* delle

[7] nelle scene statiche il numero di frames è ovviamente uno

mesh. Per tale motivo è stato pensato di utilizzare una versione semplificata del formato AAM, che contenesse soltanto le informazioni necessarie agli scopi prefissati. Inoltre si è pensato di utilizzare un formato *binario* piuttosto che ascii, in modo da ridurre al minimo le ridondanze che non sono relative alla topologia del modello. Questo accorgimento consente di ottenere risultati più coerenti nella valutazione del rapporto di compressione raggiunto durante i test relativi alla compressione.

La conversione dei modelli AAM nel formato appena descritto è stata eseguita in una fase di preprocessing tramite un *parser* generato con Bison[8], facendo uso dell'analizzatore lessicale Flex[9].

Un altro formato con cui è compatibile il modulo è l'*Object File Format OFF*, sviluppato da Rost (1989) presso la Digital Equipment Corporation's Workstations Systems Engineering ed in seguito reso disponibile per la pubblica distribuzione. Si tratta di un formato ASCII molto semplice ed intuitivo che si è reso utile in due situazioni: grazie ad esso è stato possibile usare molti dei modelli reperibili in rete[10] ed inoltre è uno dei formati supportati dal programma *Metro*[11], un tool usato in fase di test dell'algoritmo di semplificazione per il calcolo dell'errore.

### 4.3.2 Rappresentazione di modelli 3D

I modelli tridimensionali utilizzati sono gestiti con una *Half-Edge Data Structure* che ricalca gli schemi illustrati nella sezione 1.3.3 nella pagina 21. Si ricorda inoltre che la HDS non è in grado di gestire correttamente mesh *non-manifold*, per cui i modelli selezionati per i test devono rispettare tale vincolo.

L'implementazione della struttura segue il diagramma *UML*[12] riportato nella figura 4.7 nella pagina seguente: si nota che ciascun elemento possiede i

[8] *GNU Parser Generator*, http://www.gnu.org/software/bison/

[9] *Fast Lexical Analizer*, http://flex.sourceforge.net/

[10] ad esempio dal sito http://shapes.aim-at-shape.net/viewmodels.php?page=1

[11] http://vcg.sourceforge.net/tiki-index.php?page=Metro

[12] *Unified Modeling Language*: http://en.wikipedia.org/wiki/Unified_Modeling_Language. I diagrammi UML sono stati realizzati con *StarUML*, un tool opensource reperibile da http://staruml.sourceforge.net/en/

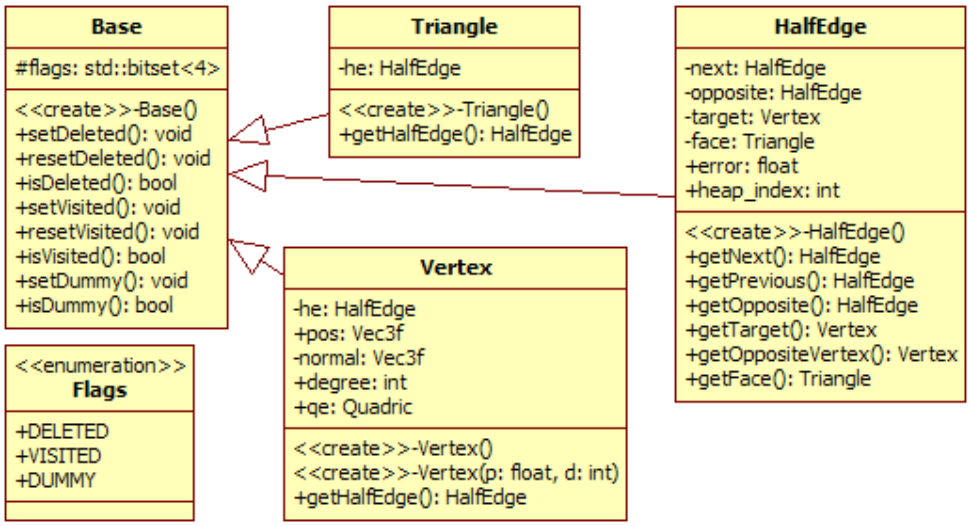



Figura 4.7: Diagramma UML dell'Half-Edge Data Structure

collegamenti agli elementi adiacenti (che costituiscono le informazioni relative alla connettività) ed altri membri dati atti a memorizzare la geometria od ulteriori informazioni importanti per le procedure realizzate. La classe `Base`, da cui tutti gli elementi derivano, fornisce un'interfaccia per poterne marcare lo stato. I possibili *flags* applicabili agli elementi della mesh sono i membri dell'enumerato denominato, appunto, `Flags`.

La classe che rappresenta la mesh vera e proprio è il `Subset`, che, seguendo la stessa nomenclatura adottata dal formato AAM, costituisce un singolo oggetto della scena. Un diagramma semplificato di tale classe è mostrato nella figura 4.8 nella pagina successiva. I dati membro più importanti della classe `Subset` sono i contenitori degli elementi della mesh, implementati con dei `vector` della libreria *STL* del C++.

### Classe Subset - Inizializzazione

Durante la fase di inizializzazione della classe, il costruttore invoca la specifica funzione `createElementsFrom{source}()` in modo da allocare in memoria tutte le componenti della mesh. Inizialmente i puntatori tra gli ele-

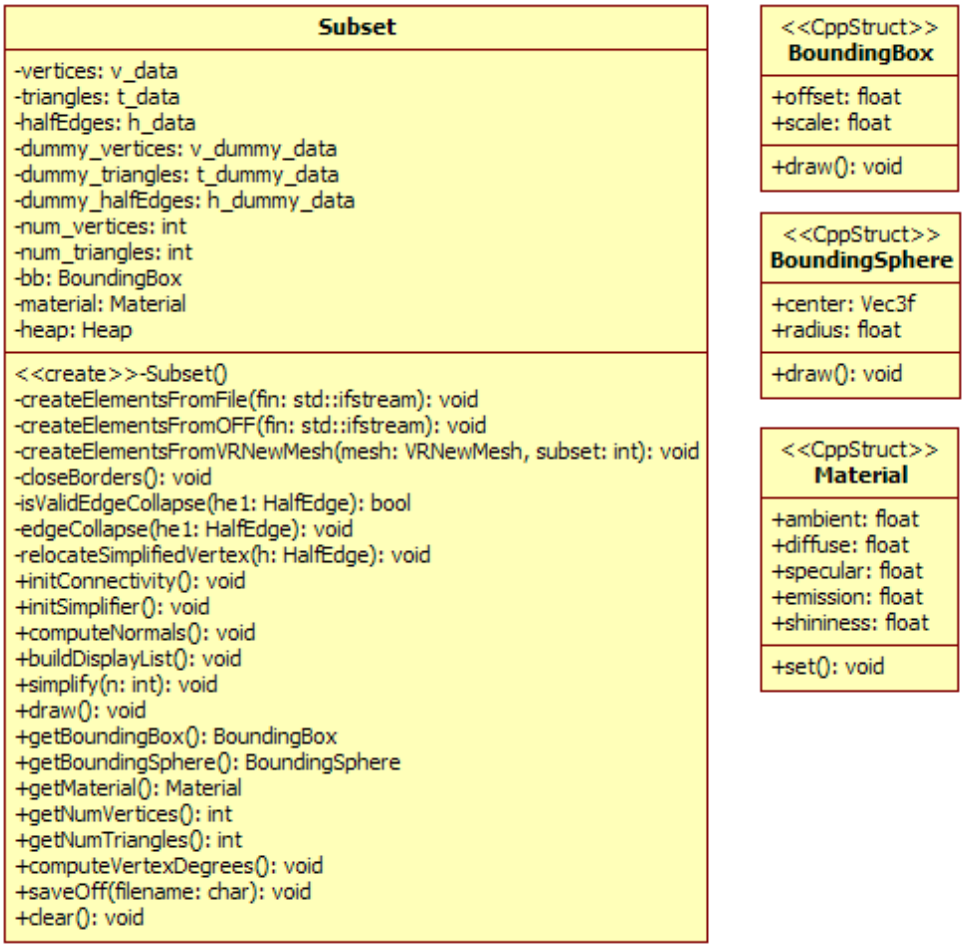



Figura 4.8: Diagramma UML semplificato della classe `Subset`

menti della mesh sono impostati ad un valore nullo; sarà poi compito della funzione `initConnectivity()` valutare correttamente tali campi in modo da ricostruire completamente le informazioni relative alla connettività. Se, dopo questa fase, alcuni elementi della mesh risultano ancora non connessi, significa che il modello possiede uno o più bordi. La funzione `closeBorders()` è stata implementata appositamente per individuare e chiudere tali bordi attraverso l'introduzione di nuovi elementi *dummy*.

### Classe Subset - Navigazione

Al fine di rendere più agevole la navigazione tra le componenti di una mesh, l'interfaccia della classe `Subset` prevede una serie di metodi[13] che restituiscono *iteratori* e *circolatori* ad i suoi elementi.

Gli *iteratori* agli elementi di una mesh sono analoghi agli iteratori dei contenitori STL. Un oggetto di tipo `VertexIterator`, ad esempio, permette di scandire uno ad uno tutti i vertici della mesh sfruttando l'overloading dell'operatore di incremento `++`. Similmente sono state implementate le classi `TriangleIterator` e `HalfEdgeIterator`.

Più interessante è invece l'uso dei *circolatori*. Tali oggetti possiedono un riferimento ad un elemento $e$ della mesh e permettono di iterare tra gli elementi ad esso adiacenti di un certo tipo. Ad esempio, un'istanza della classe `VertexTriangleIter` permette di scorrere le facce adiacenti ad un vertice dato. I circolatori implementati sono:

- `VertexVertexIter`: vertice $\mapsto$ vertici adiacenti
- `VertexIHalfEdgeIter`: vertice $\mapsto$ half-edges entranti
- `VertexOHalfEdgeIter`: vertice $\mapsto$ half-edges uscenti
- `VertexTriangleIter`: vertice $\mapsto$ triangoli adiacenti
- `TriangleVertexIter`: triangolo $\mapsto$ vertici che gli appartengono

[13] questi metodi non sono riportati nel diagramma UML della figura 4.8 nella pagina precedente per brevità

- `TriangleHalfEdgeIter`: triangolo $\mapsto$ half-edges che gli appartengono
- `TriangleTriangleIter`: triangolo $\mapsto$ triangoli adiacenti

#### Visualizzazione grafica

L'oggetto `Subset` così ottenuto può essere visualizzato sullo schermo invocando il metodo `draw()`, che disegna la mesh utilizzando le primitive OpenGL. Per velocizzare il processo di rendering, è possibile configurare l'oggetto in modo tale che la `draw()` faccia uso di una *display list* (Shreiner *e altri*, 2003), che va opportunamente compilata richiamando la funzione membro `buildDisplayList()`. Nel caso di *shaded rendering*, per una corretta computazione dell'illuminazione dell'oggetto, è possibile pre-calcolare le normali ad i vertici con il metodo `computeNormals()`.

### 4.3.3 Semplificazione di mesh

Al fine di effettuare l'operazione di semplificazione incrementale, alle strutture dati finora presentate sono state aggiunte due classi: la classe `Quadric` e la classe `Heap` (figura 4.9 nella pagina successiva).

Come illustrato precedentemente nella sezione 2.4.1, si ricorda che l'algoritmo di semplificazione prevede che ad ogni vertice $\mathbf{v}_i$ sia assegnata una quadrica da utilizzare nel calcolo dell'errore delle contrazioni $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \bar{\mathbf{v}}$, con $\mathbf{v}_j$ vertice adiacente a $\mathbf{v}_i$; inoltre, tutte le coppie ammissibili di vertici[14] vanno ordinate in una struttura dati di tipo *heap* in modo da essere poi estratte a partire da quella avente costo minore.

Le fasi di attribuzione delle quadriche ad i vertici e di inizializzazione dell'heap sono compiute dal metodo `initSimplifier()` della classe `Subset`.

#### Calcolo delle quadriche e degli errori di contrazione

La classe `Quadric` prevede un costruttore che ha come parametri in ingresso la normale $\mathbf{n}$ al piano $T$ ed un punto $\mathbf{p} \in T$: tale costruttore esegue il calcolo dei parametri $A$, $b$ e $c$ della *quadrica fondamentale* associata al piano

[14] in questo contesto coincidono con gli half-edges che li collegano

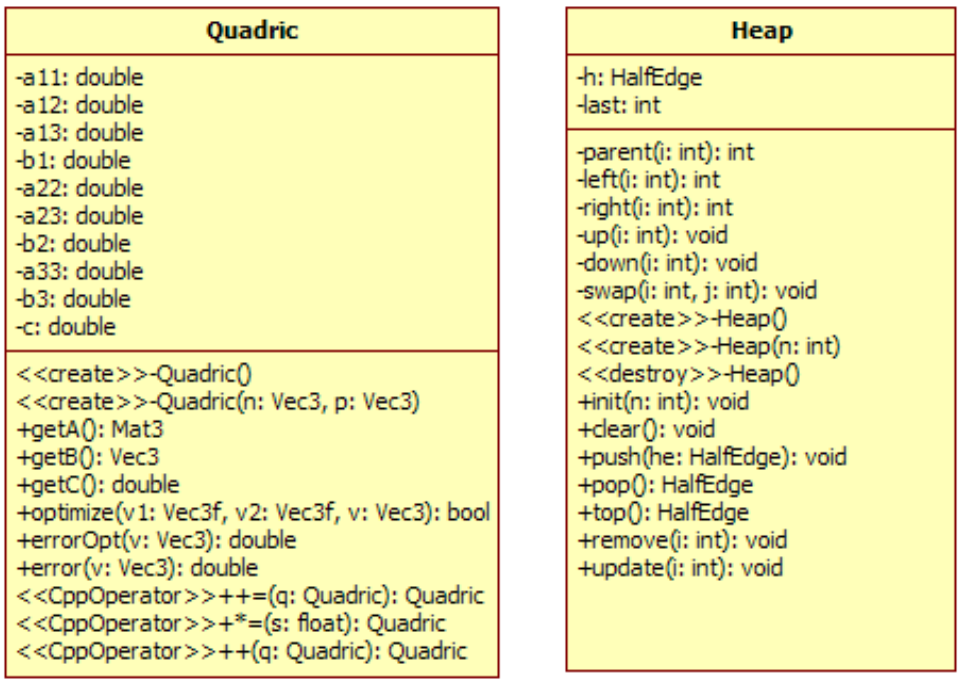



Figura 4.9: Strutture dati per la semplificazione

$T$. Durante la fase di inizializzazione, per ciascun triangolo viene calcolata la quadrica fondamentale associata al piano su cui giace e, una volta pesata (moltiplicata) per l'area del triangolo stesso, essa viene sommata al membro di tipo `Quadric` dei suoi tre vertici.

L'errore accumulato nella contrazione $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \bar{\mathbf{v}}$ è calcolato a partire dall'oggetto `Quadric` ottenuto come somma delle due quadriche associate ad i vertici $\mathbf{v}_i$ e $\mathbf{v}_j$. Dal momento che ad ogni coppia ammissibile di vertici corrisponde un `HalfEdge`, si è pensato di associare un errore a ciascuno di essi, memorizzandolo nel dato membro `error`.

Per il calcolo dell'errore, la classe `Quadric` mette a disposizione il metodo `error(v)`, il cui parametro in ingresso indica la posizione del vertice $\bar{\mathbf{v}}$ dopo la contrazione. In alternativa è possibile usare la funzione `errorOpt(v)`, che calcola l'errore utilizzando una formula ottimizzata, ma valida solo nel caso in cui `v` sia l'*optimal placement*.

La posizione di $\bar{\mathbf{v}}$ può invece essere trovata con il metodo `optimize()`: se la quadrica non è degenere, tale funzione ritorna l'*optimal placement*, altrimenti il punto medio dei due vertici di partenza.

### Implementazione ed utilizzo dell'heap

Come mostrato nel diagramma UML della figura 4.8, la classe `Subset` possiede un dato membro di tipo `Heap` atto a svolgere le funzioni sopra citate: si tratta di un *heap* di puntatori ad `HalfEdge` ordinati secondo il membro `error` degli oggetti da essi riferiti.

La classe `Heap` è implementata secondo il metodo tradizionale in cui i suoi elementi sono organizzati in un albero binario memorizzato in un *array*, in questo caso di puntatori ad `HalfEdge`; oltre alle comuni operazioni `push()` e `pop()`, la classe fornisce altri metodi utili per l'uso specifico cui è destinata.

- `remove(int i)`: rimuove dall'heap l'elemento che occupa la $i$esima posizione all'interno dell'*array*.
- `update(int i)`: ristabilisce la consistenza della struttura dati se l'errore associato all'$i$esimo `HalfEdge` è mutato. Si tratta di una scorciatoia che evita l'estrazione ed il successivo inserimento dello stesso elemento.

Tra i metodi della classe `Heap` non vi è una funzione per la ricerca di un determinato elemento, cioè una procedura che, dato un `HalfEdge`, ne restituisce la posizione all'interno dell'heap. Per ragioni di efficienza, è stato infatti scelto di memorizzare tale posizione direttamente nelle istanze di ogni `HalfEdge` utilizzando il dato membro `heap_index`. In questo modo l'operazione di ricerca assume un costo *costante* anziché *logaritmico*.

Tutte le ottimizzazioni sopra elencate rendono estremamente efficiente e performante l'uso dell'`Heap`, in particolare l'accesso ad i suoi elementi e la loro gestione. Un'ulteriore ottimizzazione, questa volta in termini di memoria, è stata quella di rappresentare la matrice $\mathbf{A}$ di dimensione $3 \times 3$ con sei anziché nove coefficienti, dal momento che si tratta di una matrice simmetrica. Non è stato possibile invece utilizzare variabili di tipo `float` anziché `double` per memorizzare tali coefficienti a causa dell'elevato grado di precisione richiesto per il calcolo degli errori.

### Implementazione dell'algoritmo

La classe `Subset` fornisce alcuni metodi tramite i quali viene effettuata la semplificazione della mesh che rappresenta:

- `simplify(int)`
- `simplyfyPolygon(int)`
- `simplifyUpTo(int)`
- `simplifyPercentage(float)`

Ciascuno di essi calcola prima un numero di triangoli obiettivo, e dopo cicla la funzione `collapseFirstEdge()` fino al suo raggiungimento. Tale metodo compie i seguenti passi:

1. Estrae l'`HalfEdge` che si trova alla testa dello *heap*
2. Verifica che la contrazione dell'`HalfEdge` selezionato non vada a modificare la topologia della superficie della mesh valutando il valore booleano ritornato dalla funzione `isValidEdgeCollapse()`

3. Effettua la contrazione dell'`HalfEdge` selezionato eseguendo prima le operazioni relative alla geometria (`relocateSimplifiedVertex()`) e poi alla connettività (`edgeCollapse()`)

4. Aggiorna l'errore degli `HalfEdge`s incidenti sul vertice target della contrazione e li riordina nell'*heap*

### 4.3.4 Compressione e decompressione di mesh

Le classi che effettuano la compressione e decompressione di una mesh sono rispettivamente la classe `Compressor` e `Uncompressor` (figura 4.10 nella pagina seguente). Esse hanno molte caratteristiche in comune: la più evidente è che entrambe derivano dalla classe `Huffman`, che implementa l'algoritmo per la costruzione dell'albero e dei codici di Huffman e dispone dei metodi per la loro lettura e scrittura su file.

#### Implementazione della codifica di Huffman

Come noto, i codici di Huffman sono rappresentati con un numero variabile di bits; le comuni classi di IO presenti nella Standard C++ Library non sono quindi adatte a questo scopo poiché orientate al byte. Si è reso perciò necessario realizzare una classe di IO specifica, denominata `BitFile`, i cui metodi `read()` e `write()` operano con un numero arbitrario di bits anziché con multipli interi di bytes.

I codici di Huffman non possono essere emessi durante l'esecuzione dell'algoritmo di compressione poiché per poterli calcolare devono essere note le frequenze di apparizione di tutti i simboli. Per questa ragione il compressore accumula i simboli di output in una coda e ne aggiorna le frequenze tramite la funzione `count(Symbol)` della classe `Huffman`[15]. Al termine del processo di compressione viene chiamato il metodo `buildTrees()` con cui sono costruiti due alberi di Huffman distinti: uno per i simboli relativi alla geometria ed uno per quelli concernenti la connettività. I codici sono quindi

[15]per la precisione tale coda contiene soltanto i *riferimenti* ad i simboli, ottenuti come valore di ritorno della funzione `count()`

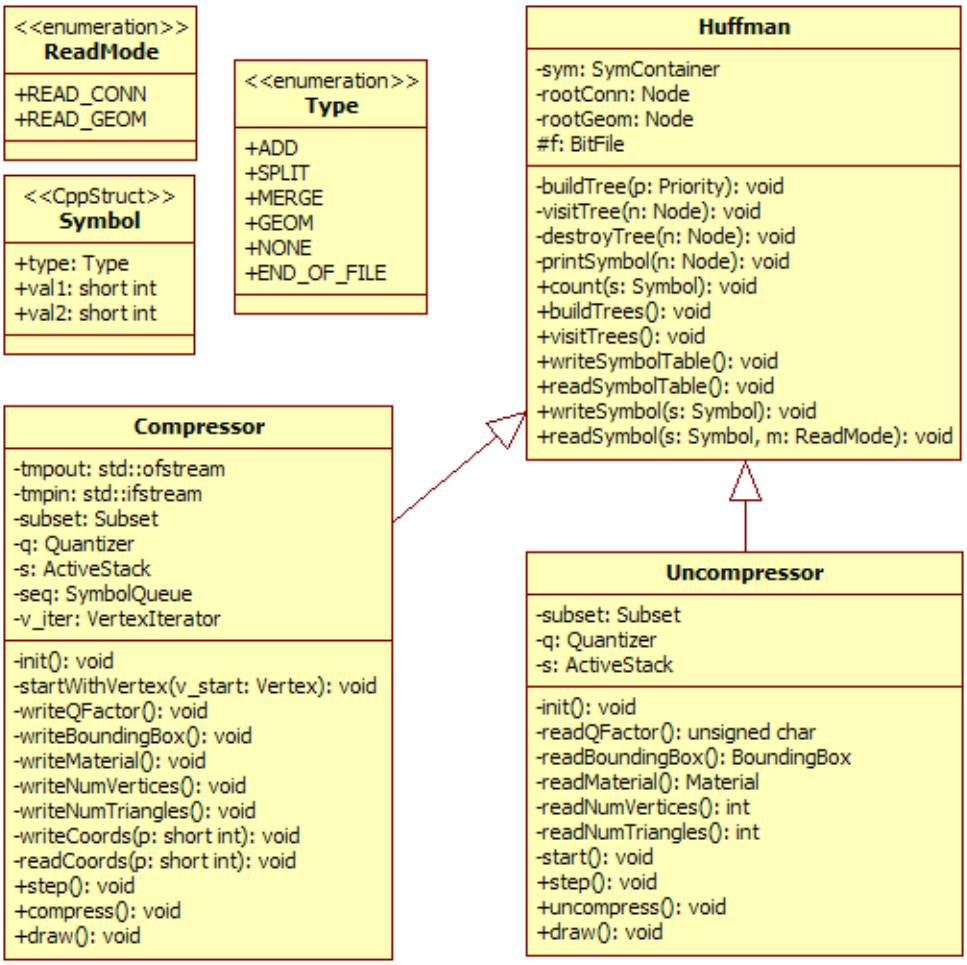



Figura 4.10: Schema UML del compressore e del decompressore

calcolati effettuando una visita ricorsiva dell'albero a partire dalla radice con la funzione `visitTrees()`.

### Formato del file compresso

L'ultima operazione eseguita dal compressore è la scrittura del file di uscita. Tale procedura comprende l'emissione di un *header* al quale seguono le traduzioni dei simboli presenti nella coda di output. L'*header* è così composto:

- *Fattore di quantizzazione*[16] → `writeQFactor()`
- *Bounding Box* → `writeBoundingBox()`
- *Materiale* → `writeMaterial()`
- *Numero di vertici* → `writeNumVertices()`
- *Numero di triangoli* → `writeNumTriangles()`
- *Tabella dei simboli* → `writeSymbolTable()`

Alcune componenti dell'*header* possono apparire ridondanti, ma servono per l'inizializzazione del decompressore. La bounding box, ad esempio, può essere calcolata automaticamente dopo aver letto le coordinate di tutti i vertici; in questo caso però essa deve essere nota sin dall'inizio al decompressore per poter ricostruire le coordinate quantizzate. Anche il numero di vertici e di triangoli è ridondante ai fini della compressione, ma è utile al decompressore per allocare la memoria necessaria a contenere tutti gli elementi della mesh.

### La quantizzazione della coordinate

L'algoritmo di codifica implementato include una parte *lossy*: la compressione della geometria prevede infatti che i simboli relativi alle coordinate dei vertici rappresentino il valore degli *indici di quantizzazione*, piuttosto che

[16] l'uso del fattore di quantizzazione è spiegato successivamente

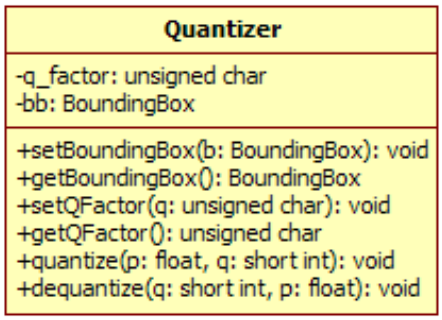



Figura 4.11: La classe `Quantizer`

gli effettivi numeri a virgola mobile. Il *fattore di quantizzazione* è un parametro del costruttore della classe `Compressor`, con il quale esso inizializza il membro `q` di tipo `Quantizer` (figura 4.11).

La classe `Quantizer` è in grado di effettuare una quantizzazione uniforme delle coordinate di un qualsiasi punto interno alla bounding box. Con il metodo `quantize(float *p, short *q)` ciascuna coordinata del punto `p` è trasformata in un punto dell'intervallo $[0; 1]$ relativo alla bounding box; il rispettivo indice di quantizzazione viene restituito con il vettore `q`. Il metodo `dequantize(short *q, float *p)` ricava invece il valore quantizzato delle coordinate a partire dagli indici di quantizzazione.

### Implementazione del valence based encoding

L'algoritmo *valence based encoding* possiede la particolarità di avere una fase di decompressione perfettamente simmetrica a quella di compressione. Per questo motivo le classi `Compressor` ed `Uncompresor` sono molto simili tra loro ed hanno metodi e dati membri analoghi, perciò questa analisi sarà incentrata in modo particolare soltanto sul compressore.

Durante la fase di inizializzazione ogni vertice del `Subset` è marcato come *non visitato* ed il suo membro `degree` è impostato al valore della valenza. Quando un vertice verrà *conquistato*, il suo membro `degree` sarà decrementato mano a mano che gli spigoli ad esso incidenti sono a sua volta conquistati, in modo tale da tenere il conto di quelli ancora *liberi*. Quando la variabile

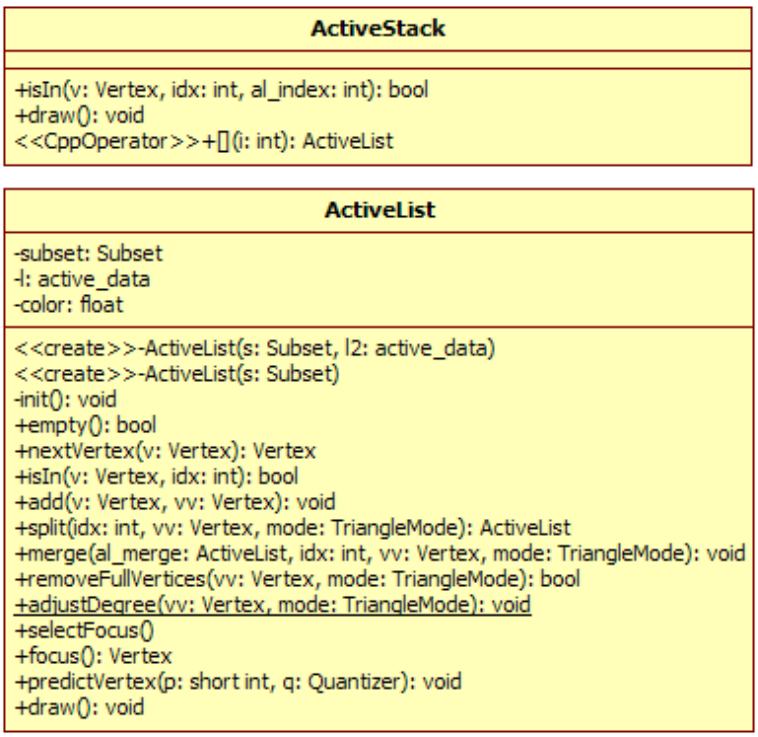



Figura 4.12: Diagramma UML della classe `ActiveList`

`degree` giungerà a zero significa che il rispettivo vertice è divenuto *pieno* e potrà essere rimosso dalle *liste attive* cui appartiene.

Dopo questa fase ha inizio la visita della mesh. Il punto di partenza è un triangolo arbitrario[17]: una nuova *lista attiva* è creata ed inizializzata con i vertici di tale triangolo. In seguito la classe `Compressor` itera la funzione `step()` fino a quando tutta la mesh non sarà conquistata.

L'elemento centrale del processo di codifica è lo *stack* delle *liste attive*. Esso è stato implementato con la classe `ActiveStack`, che deriva il tipo `stack<ActiveList*>` della STL aggiungendo il metodo `isIn(Vertex)` atto a restituire l'indice della lista attiva alla quale appartiene un determinato vertice.

Gli elementi dell'`ActiveStack` sono i puntatori alle istanze della classe `ActiveList` (figura 4.12 nella pagina precedente), che rappresentano le *liste attive* correntemente non vuote. La classe `ActiveList` memorizza i vertici del bordo di taglio ad essa associato in un contenitore di tipo `list<Vertex*>`. Questa classe è la responsabile dell'esecuzione delle *operazioni di conquista*:

- `add(Vertex v)`: aggiunge il vertice `v` al bordo di taglio marcandolo come *visitato*
- `split(int idx)`: esegue lo *split* del bordo di taglio restituendo l'indirizzo di una nuova `ActiveList` creata per ospitare uno dei due bordi di taglio formatisi
- `merge(ActiveList l, int idx)`: unisce il bordo di taglio corrente con quello della lista attiva `l`
- `removeFullVertices()`: rimuove tutti i vertici *pieni* dalla lista

Ciascuna di queste funzioni restituisce inoltre un vettore di tre puntatori a `Vertex` ed un valore del tipo enumerato `TriangleMode`. Tali elementi indicano quali sono i vertici a cui è necessario modificare il `degree` ed in

[17] cioè quello relativo al primo elemento del vettore dei vertici che non è marcato come visitato

che modo. Il metodo statico `adjustDegrees()`, prendendo in input queste informazioni, esegue l'aggiornamento delle variabili `degree` coinvolte.

La politica di selezione del *focus* è stata implementata invece nel metodo `selectFocus()`: esso sceglie come *focus* il vertice del bordo di taglio con minore `degree`, cioè quello a cui resta da aggiungere il minor numero di spigoli incidenti.

Un'altra tecnica implementata dalla classe `ActiveList` è quella della *predizione del vertice* secondo la *regola del parallelogrammo*. Il metodo atto a svolgere tale funzione è `predictVertex()`, che restituisce il vettore delle coordinate del vertice predetto. Esso opera con le coordinate già quantizzate, in modo da annullare l'errore di predizione in fase di ricostruzione.

Per seguire in modo interattivo la fase di compressione/decompressione di una mesh è possibile eseguire la procedura in una modalità *passo-passo* in cui il metodo `step()` viene richiamato una sola volta per iterazione. Tramite il metodo `draw()` della classe `Compressor`/`Uncompressor` è possibile inoltre seguire lo stato di espansione della regione conquistata disegnando sullo schermo i bordi di taglio delle liste attive, ciascuna con un colore differente.

### 4.3.5 Gestione dei livelli di dettaglio

Per mostrare un esempio pratico di utilizzo dei *livelli di dettaglio* ed i vantaggi che essi comportano sono state implementate le classi `Model`, `Object` e `Scene` (figure 4.13 e 4.14 nella pagina successiva). Di seguito, per ciascuna di esse sono presentati i dettagli implementativi e funzionali.

#### La classe Model

Ogni istanza di tale classe rappresenta un modello tridimensionale con *multi-risoluzione discreta* utilizzabile per comporre la scena da visualizzare. Il modello è caricato in modo tale che la sua bounding box e bounding sphere sia centrata nell'origine, memorizzando solo le informazioni relative alle dimensioni della bounding box (vettore `sc`) ed il raggio della bounding sphere (variabile `r`).

Lo scopo della classe `Model` è quello di gestire le *display lists* relative alle

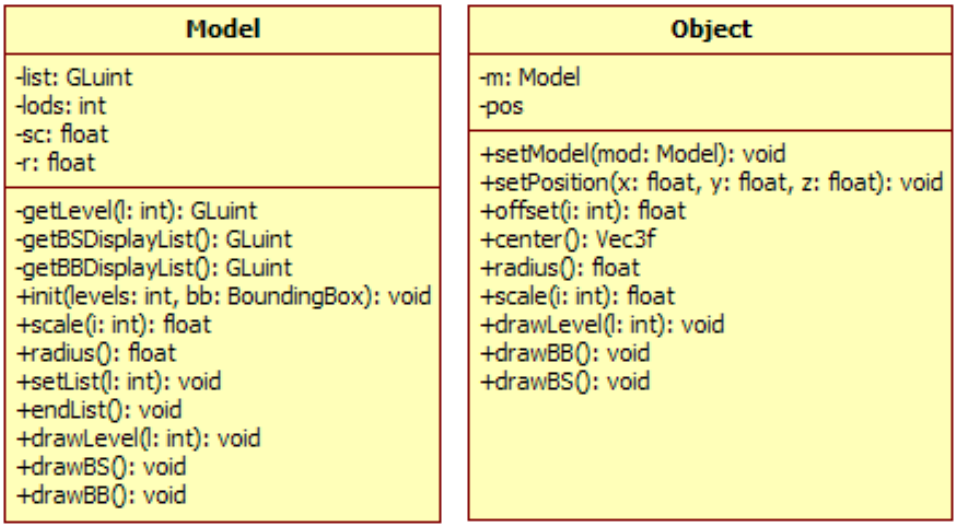



Figura 4.13: Diagramma UML delle classi Model ed Object

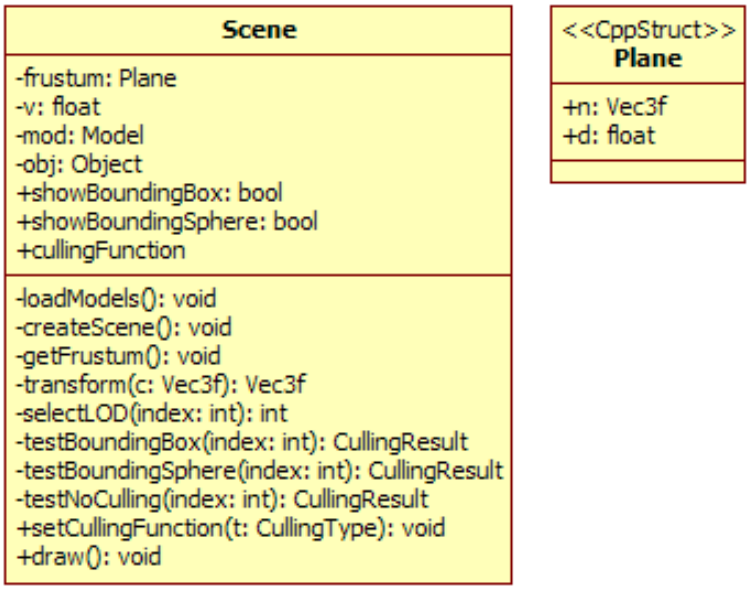



Figura 4.14: Diagramma UML della classe Scene

varie versioni del modello in modo trasparente. L'inizializzazione di un'istanza della classe `Model` richiede come parametro il numero $n$ di versioni del modello che si desiderano utilizzare: sono così allocate $n$ *display lists* per contenere le versioni, più una *diaplay list* per la bounding box e per la bounding sphere. Mentre queste ultime due sono compilate automaticamente durante al fase di inizializzazione, quelle riservate alle versioni multi-risoluzione discrete devono essere compilate una ad una racchiudendo i comandi OpenGL tra le chiamate ad i metodi `setList(int l)` ed `endList()`.

Infine, le display lists possono essere visualizzate sullo schermo tramite i metodi `drawLevel(int l)`, `drawBB()` e `drawBS()`, che disegnano rispettivamente la versione `l` del modello, la sua bounding box e la sua bounding sphere.

### La classe Object

Le istanze della classe `Object` rappresentano i componenti veri e propri della scena. Per ciascuno di essi è specificato il modello tridimensionale da utilizzare e la sua posizione nello spazio. Questa tecnica è stata pensata per avere più repliche di uno stesso modello dislocate in punti diversi della scena senza tenere copie multiple delle sue display lists in memoria.

I metodi atti a disegnare un oggetto sono semplicemente dei *wrapper* dei rispettivi metodi del modello di riferimento, ad i quali è aggiunta l'operazione di *traslazione*.

### La classe Scene

Coerentemente alla sua denominazione, la classe `Scene` (figura 4.14 nella pagina precedente) rappresenta la scena da visualizzare. Essa possiede un vettore di modelli e di oggetti che vengono inizializzati nel costruttore rispettivamente con le funzioni `loadModels()` e `createScene()`.

- `loadModels()`: carica i modelli leggendoli da file in formato binario od OFF ed associa ciascuno di essi ad un'istanza della classe `Model`; per ogni modello crea poi una serie di versioni semplificate e ne compila le display lists utilizzando i metodi del relativo contenitore.

- `createScene()`: alloca un vettore di `Object` ed a ciascuno di essi associa un modello valido ed una posizione.

Il rendering della scena è affidato al metodo `draw()`. Esso ha il compito di disegnare tutti gli oggetti della scena e per ognuno di essi esegue alcune ottimizzazioni come riportato nei seguenti passi:

1. esegue un *test di visibilità*[18]. Se tale test fallisce l'oggetto è scartato.
2. seleziona il *livello di dettaglio* dell'oggetto più adatto tramite la funzione `selectLOD()`. La politica implementata in questo esempio prevede la selezione delle versioni in modo da avere un numero di poligoni inversamente proporzionale alla distanza tra l'oggetto considerato e l'osservatore.
3. disegna la versione dell'oggetto selezionata al passo precedente
4. disegna la bounding box o la bounding sphere dell'oggetto se i rispettivi membri booleani `showBoundingBox` e `showBoundingSphere` sono settati.

### View Frustum Culling

Il test di visibilità accennato precedentemente consiste nel *view frustum culling*: in altre parole, se la bounding box o la bounding sphere dell'oggetto di riferimento rimane completamente fuori dal campo visivo, il test fallisce.

La classe `Scene` è fornita delle strutture dati e delle funzioni per poter svolgere tale compito. Il *view frustum* è rappresentato da un vettore di sei oggetti di tipo `Plane`, ognuno dei quali memorizza la normale ed il termine noto dell'equazione di uno dei piani del view frustum. Tali parametri sono aggiornati ad ogni *frame* tramite la funzione `getFrustum()` che implementa il metodo descritto in Gribb e Hartmann (2001). In questo articolo è esposto come ricavare i parametri del view frustum a partire dalle matrici di *model-view* e di *projection* di OpenGL.

[18] si veda il seguente paragrafo

È possibile selezionare la funzione che effettua il test tra tre possibili implementazioni:

- `testBoungindBox()`: esegue il test utilizzando come riferimento la bounding box dell'oggetto
- `testBoundingSphere()`: esegue il test utilizzando come riferimento la bounding sphere dell'oggetto
- `testNoCulling()`: è un test *dummy*, che non fallisce mai

L'esito del test è un valore del tipo enumerato `CullingResult` che può assumere tre valori a seconda che la bounding box (sphere) intersechi o no il view frustum:

1. *INSIDE*: l'oggetto è completamente contenuto nel campo visivo.
2. *INTERSECTING*: l'oggetto potrebbe essere parzialmente visibile.
3. *OUTSIDE*: l'oggetto non è visibile e può pertanto non essere disegnato.

La funzione per il test di visibilità può essere scelta attraverso il metodo `setCullingFunction(CullingType t)`.

Per avere un riscontro visivo del risultato del test, durante la fase di visualizzazione, le bounding box o le bounding sphere sono disegnate con colori differenti.

# Capitolo 5

# Test ed analisi delle performance

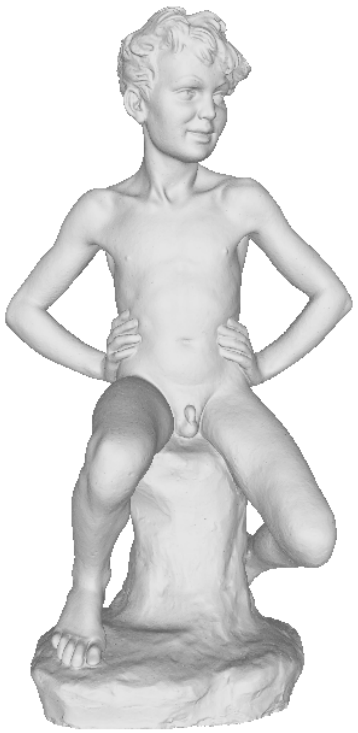
## 5.1 Algoritmo di semplificazione

Per l'analisi delle prestazione degli algoritmi di semplificazione e compressione implementati sono stati utilizzati i tre modelli tridimensionali mostrati nella figura 5.1 nella pagina successiva. I modelli "Ragazzo" e "Madonna" sono stati forniti dal laboratorio PERCRO, mentre il modello "Carter" è stato reperito in rete dal repository pubblico del sito AIM@SHAPE[1].

I test consistono nell'applicare l'algoritmo di semplificazione incrementale alla mesh originale al fine di generare una serie di *livelli di dettaglio* (LOD) aventi un numero di poligoni sempre inferiore. Di tali approssimazioni sono poi stati valutati e confrontati tre parametri:

- L'*errore* di approssimazione rispetto alla mesh originale
- Il *tempo* impiegato per la generazione del LOD a partire dal modello originale
- Il *carico* computazionale richiesto dalla visualizzazione del LOD valutato come il tempo impiegato a generare un *frame* che lo ritrae

[1] http://shapes.aim-at-shape.net



(a) “Ragazzo”

(b) “Madonna”

(c) “Carter”

Figura 5.1: Versioni originali dei modelli utilizzati

| Modello | Numero di Vertici | Numero di Triangoli |
|---|---|---|
| Ragazzo | 999599 | 1999210 |
| Madonna | 600801 | 1199999 |
| Carter | 535199 | 1070426 |

Tabella 5.1: Complessità dei modelli utilizzati

Il sistema utilizzato per l'esecuzione dei test è un notebook dotato di processore Pentium Centrino Duo con doppio core da 1.83GHz, 1GB di memoria RAM e scheda video nVidia GeForce go 7400 con 256MB di memoria video.

Prima di analizzare i risultati dei test, di seguito sono date ulteriori informazioni riguardo al metodo utilizzato per il calcolo dell'errore di approssimazione.

### 5.1.1 Metrica di errore utilizzata

Lo strumento usato per il calcolo dell'errore di approssimazione è *Metro* (Cignoni *e altri*, 1998b), un programma capace di confrontare numericamente due mesh di triangoli che descrivono la stessa superficie a differenti livelli di dettaglio. Le differenze geometriche sono valutate come la *distanza* tra le superfici delle due mesh di seguito definita.

#### Definizione di distanza tra due mesh

Dato un punto $p$ ed una superficie $S$ si definisce la distanza $e(p, S)$ come:

$$e(p, S) = \min_{p' \in S} d(p, p') \tag{5.1}$$

Dove $d()$ rappresenta la distanza euclidea tra due punti in $\mathbb{R}^3$. Si può estendere questa definizione alla distanza tra due superfici nel seguente modo:

$$E(S_1, S_2) = \max_{p \in S_1} e(p, S_2) \tag{5.2}$$

Si noti che la definizione appena fornita non è simmetrica: in generale $E(S_1, S_2) \neq E(S_2, S_1)$. Per tale motivo spesso viene usata la *distanza di Hausdorff*, vale a dire il valore massimo tra $E(S_1, S_2)$ ed $E(S_2, S_1)$.

Si dice infine *distanza media* $E_m$ tra due superfici $S_1$ ed $S_2$ l'integrale di superficie della distanza normalizzato per l'area di $S_1$:

$$E_m(S_1, S_2) = \frac{1}{|S_1|} \int_{S_1} e(p, S_2) ds \tag{5.3}$$

**L'approccio utilizzato da Metro**

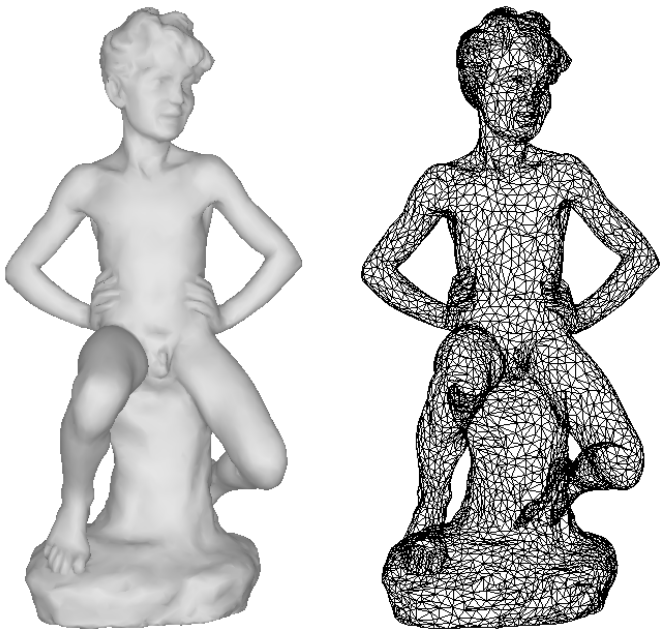
Il tool *Metro* confronta numericamente due mesh $S_1$ ed $S_2$ tramite il calcolo della loro *distanza* secondo le definizioni appena date. Tale forma di *errore* costituisce una valutazione quantitativa della *qualità* del livello di dettaglio.

Date due mesh, *Metro* ricampiona la superficie della prima e calcola la distanza tra ciascun campione e la seconda mesh. Tramite un processo di integrazione vengono calcolate poi la distanza media, massima e di Hausdorff tra le due superfici. La precisione di tale integrazione dipende dalla risoluzione del ricampionamento. Ottimi risultati sono ottenuti con uno *step size* pari all'1% della lunghezza della diagonale della bounding box.

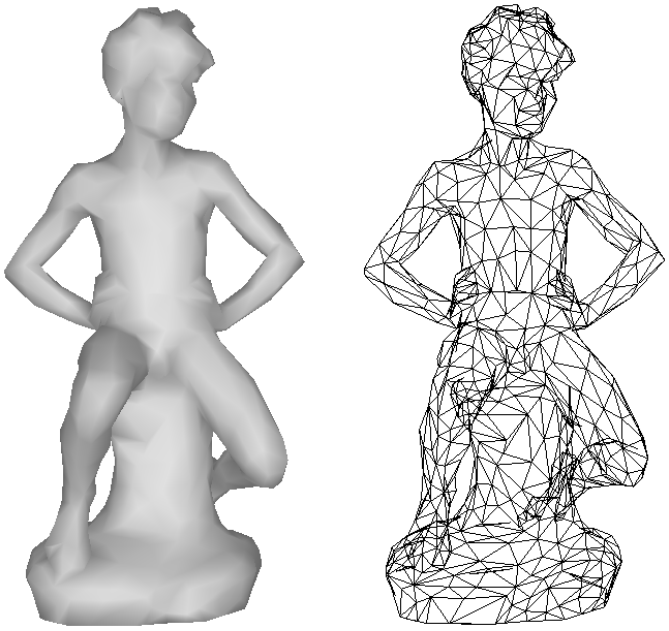
### 5.1.2 Qualità dei modelli semplificati

Per ciascun modello sono state costruite delle versioni semplificate successive in modo da poterne osservare le caratteristiche sia da un punto di vista qualitativo, che quantitativo.

Nelle figure 5.2, 5.3 e 5.4 nelle pagine seguenti sono mostrati alcuni passi del processo di semplificazione dei modelli utilizzati. I livelli illustrati costituiscono alcuni dei risultati finali della semplificazione. Tali livelli sono stati scelti in modo da avere una triangolazione che sia distinguibile visivamente per poterne così analizzare le caratteristiche. Si ricorda infatti che i modelli originali possiedono milioni di poligoni, ed una loro visualizzazione in *wireframe* non permette di distinguere i triangoli che li compongono.



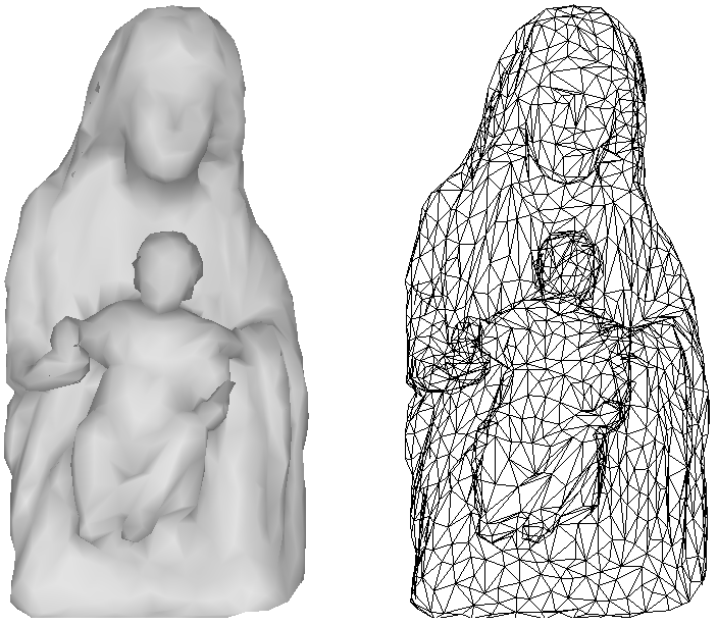
(a) Versione con 16000 poligoni



(b) Versione con 2000 poligoni

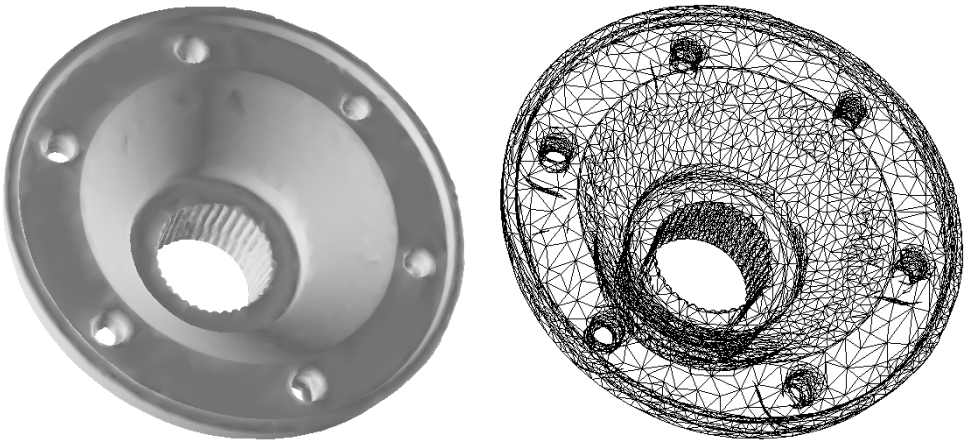
Figura 5.2: Approssimazioni successive del modello “Ragazzo”

(a) Versione con 18000 poligoni

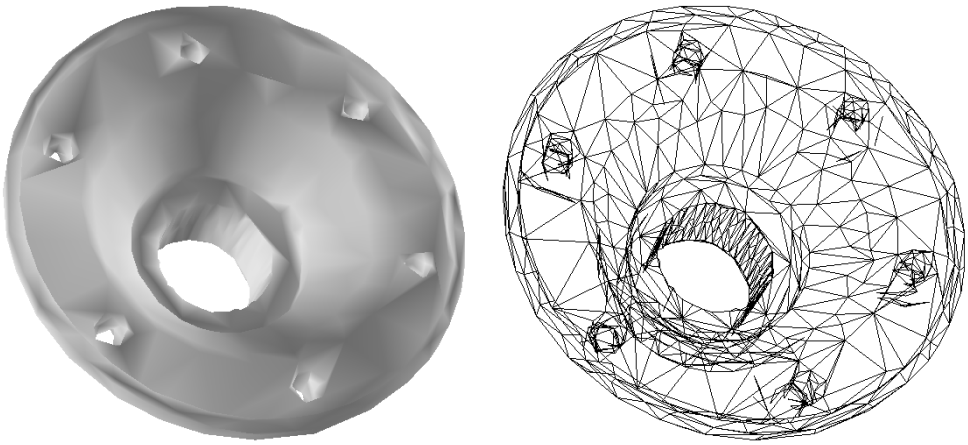


(b) Versione con 1200 poligoni

Figura 5.3: Approssimazioni successive del modello “Madonna”



(a) Versione con 17600 poligoni



(b) Versione con 2200 poligoni

Figura 5.4: Approssimazioni successive del modello “Carter”

### Analisi qualitativa

Dalle figure precedenti si evince come l'algoritmo produca delle versioni semplificate molto simili all'originale, preservandone la morfologia anche quando il livello abbia solo lo 0,1% dei poligoni a discapito, naturalmente, dei dettagli ad alta frequenza della superficie.

Si nota inoltre come l'uso della *quadric error metric* influenzi il tassellamento della mesh durante l'applicazione dell'algoritmo di semplificazione incrementale: nelle figure è possibile constatare che i triangoli si dispongono in un modo che segue la topologia locale della superficie della mesh, allungandosi longitudinalmente lungo la direzione di minima curvatura. Sono esempio di questo aspetto i triangoli che costituiscono le braccia e le gambe di Ragazzo, le pieghe delle vesti di Madonna od il cono centrale del Carter.

### Analisi quantitativa

Una stima dell'errore di approssimazione prodotto dall'applicazione dell'algoritmo di semplificazione è stata calcolata con *Metro* per vari livelli di dettaglio relativi a ciascuno dei tre modelli. I risultati di questo test sono riportati nei grafici delle figure 5.5, 5.6 e 5.7 nelle pagine seguenti.

In tali grafici è disegnato su scala logaritmica l'errore *medio*, *quadratico medio* e *di Hausdorff* dei livelli di dettaglio al variare del numero di poligoni che li compongono. L'errore, o distanza tra le due mesh, è stato normalizzato rispetto alla diagonale della bounding box del modello per poter essere confrontato tra modelli differenti.

Nello stesso grafico usando l'asse $y$ di destra è riportato inoltre l'andamento di un parametro di prestazioni importante (in rosa): il tempo necessario al motore grafico per generare un frame contenente il soggetto.

Dal grafico si può constatare come le prestazioni crescano in modo inversamente proporzionale al numero di poligoni che costituiscono il livello, a discapito di un errore sempre maggiore; si può notare però che tale errore resta sensibilmente basso (dell'ordine di milionesimi di diagonale della bounding box) fino ad un livello di dettaglio composto da circa il 15-20% dei poligoni dell'originale. Pertanto, per modelli di queste dimensioni tale valore

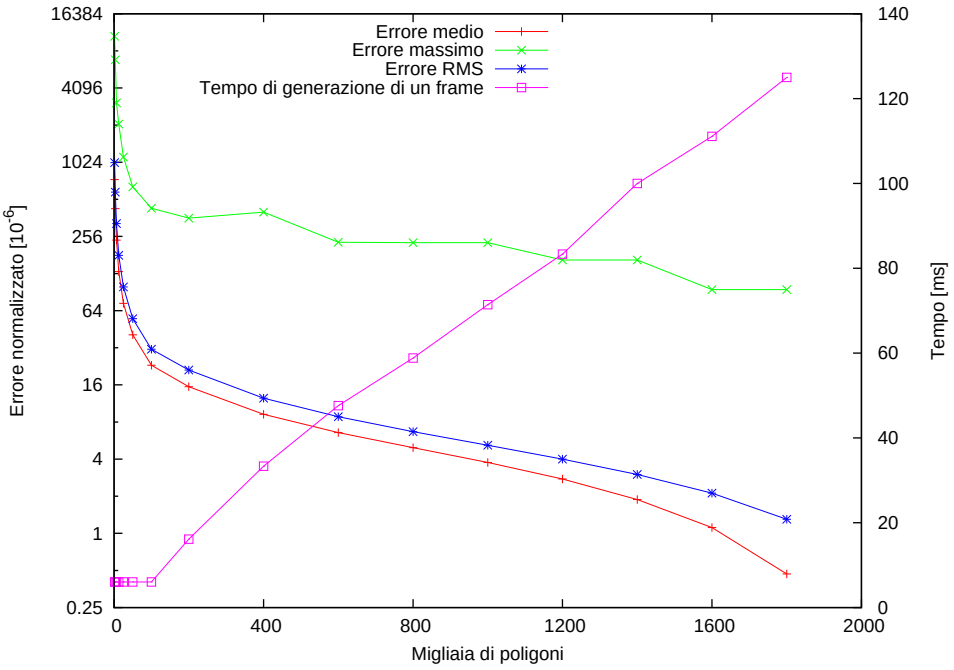



Figura 5.5: Errore di approssimazione e tempo di rendering (“Ragazzo”)

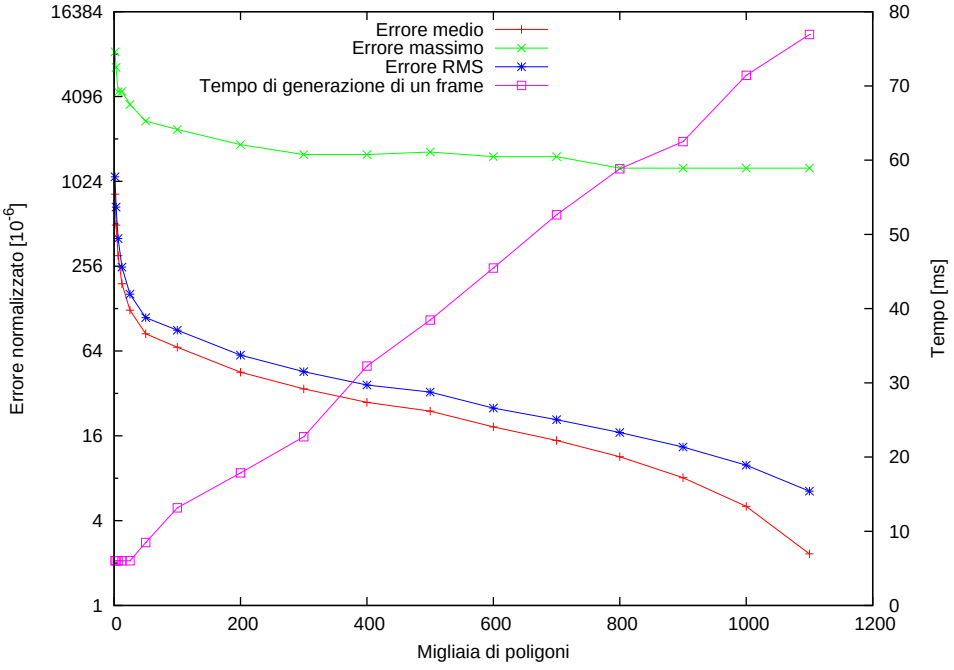



Figura 5.6: Errore di approssimazione e tempo di rendering (“Madonna”)

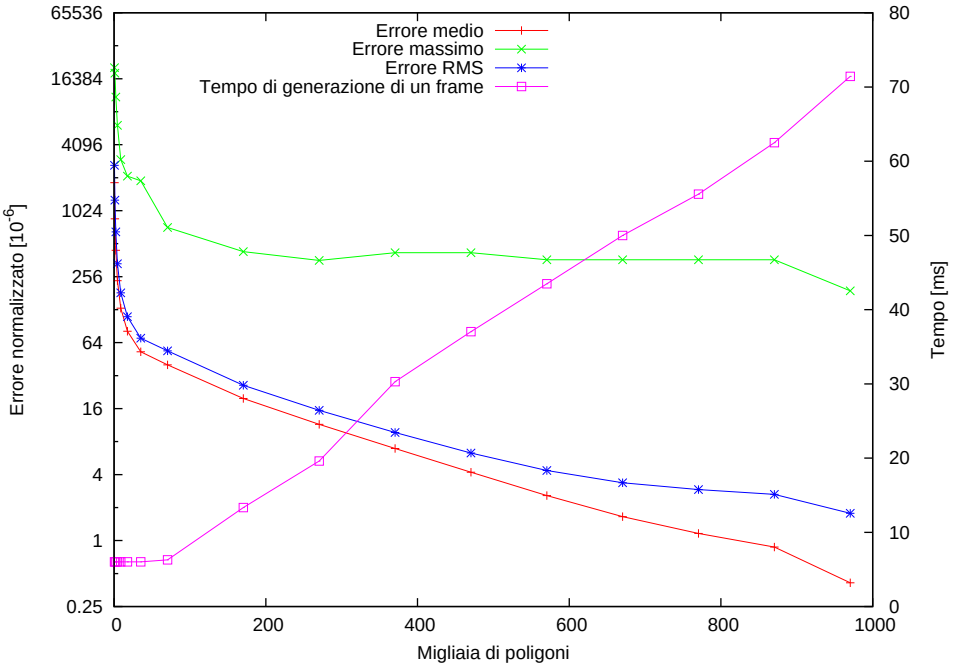



Figura 5.7: Errore di approssimazione e tempo di rendering ("Carter")

rappresenta un buon *trade-off* tra l'incremento delle prestazioni e la perdita della qualità visiva.

### 5.1.3 Performance dell'algoritmo

Le prestazioni dell'algoritmo sono state misurate in termini del tempo necessario ad effettuare la contrazione di tutti i vertici della mesh, ovvero finché essa presenta coppie di vertici ammissibili. Il grafico della figura 5.8 nella pagina seguente riporta tali misurazioni.

I modelli usati per i test, composti da milioni di poligoni, sono stati semplificati interamente in poche decine di secondi. La cosa più interessante è però il fatto che le curve del grafico hanno un andamento *lineare*, vale a dire che il tempo di semplificazione è direttamente proporzionale al numero dei poligoni. Da tale considerazione si può dedurre che il tempo necessario ad effettuare la contrazione di un vertice ed ad aggiornare l'errore relativo alle coppie di vertici coinvolte è pressoché *costante*.

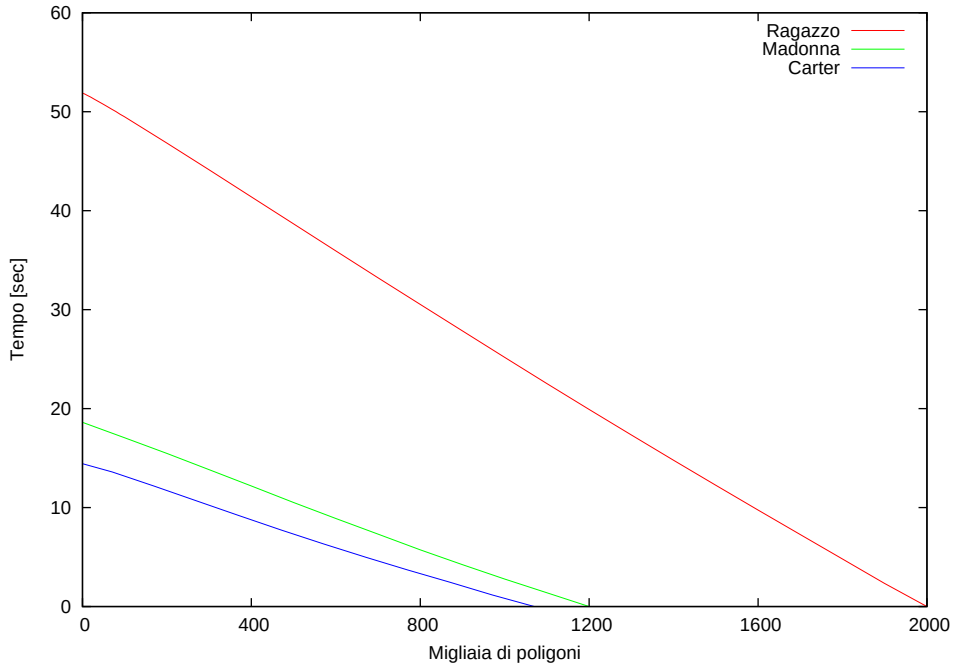


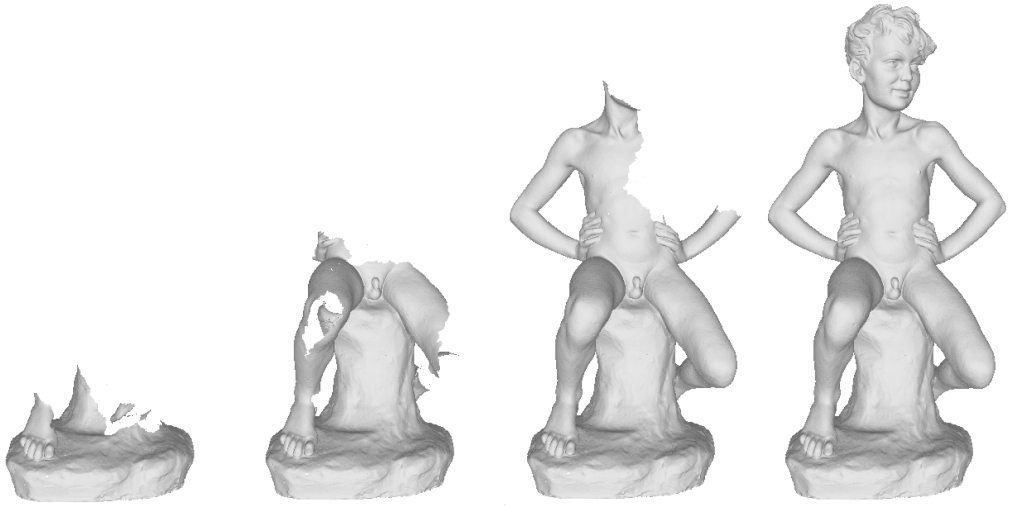

Figura 5.8: Tempo di semplificazione dall'originale fino a 0 poligoni

Tale peculiarità è resa possibile grazie alla particolare implementazione delle funzionalità dell'*heap* che tiene ordinati gli errori delle operazioni di semplificazione (vedi sezione 4.3.3 nella pagina 91).

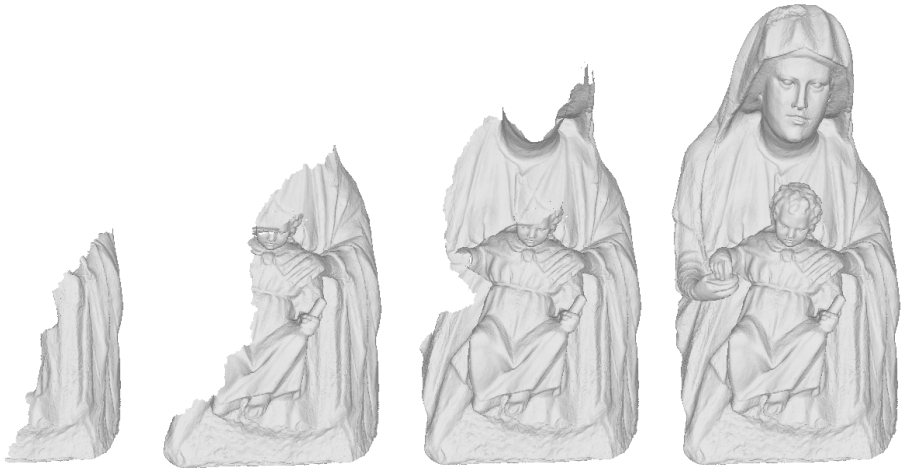
## 5.2 Algoritmo di compressione

I test reletivi all'algoritmo di compressione della connettività e della geometria di mesh triangolari sono stati effettuati con gli stessi tre modelli presentati precedentemente. I fattori analizzati nei i suddetti test sono:

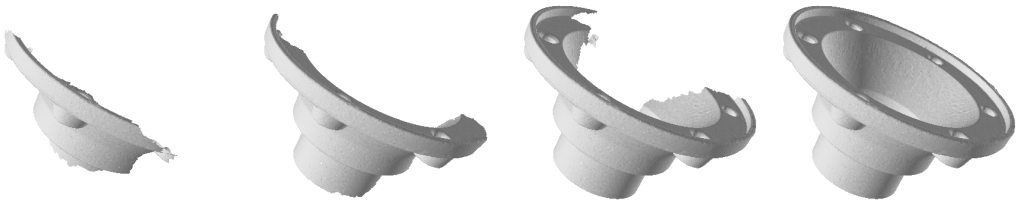
- l'*efficacia* dell'algoritmo, cioè la capacità di rendere le informazioni che descrivono la mesh più compatte
- l'*efficienza*, cioè i tempi necessari alla procedura di codifica e decodofica
- l'*errore* commesso durante la fase *lossy* dell'algoritmo



(a) Modello “Ragazzo”



(b) Modello “Madonna”



(c) Modello “Carter”

Figura 5.9: Fasi della decodifica dei modelli compressi

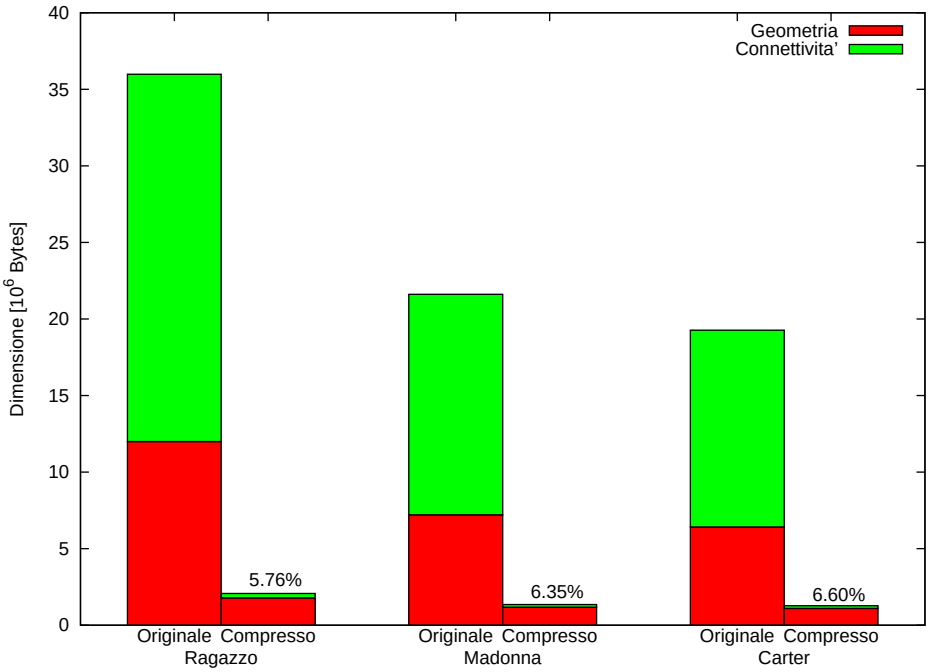



Figura 5.10: Stistiche sulla compressione

### 5.2.1 Analisi dell'efficacia dell'algoritmo

Questi test consistono nel confrontare il file prodotto come risultato della procedura di codifica con quello originale. A tale scopo il file originale è dato in un formato binario in cui le informazioni sulla geometria di ciascun vertice consistono in tre numeri di tipo `float` per un totale di 12 bytes per vertice; la connettività, a sua volta, si compone da una lista di tre indici di tipo `int` a cui corrispondono 12 bytes per triangolo. La codifica è stata svolta applicando l'algoritmo implementato con una configurazione che prevede la *quantizzazione* delle coordinate su 12 bits.

I risultati dei test sono illustrati nel grafico della figura 5.10 e nelle tabelle 5.2 e 5.3.

L'istogramma evidenzia come il file ottenuto dalla compressione sia più compatto di quello originale, con un fattore di compressione superiore al 90%. Inoltre si può constatare come la connettività, predominante nel file originale, sia di modesta entità nel file compresso. Le informazioni relative alla connettività associate a ciascun vertice risultano infatti di circa 2,5 bits

| Modello | Poligoni | Originale | Compresso | Ratio |
|---|---|---|---|---|
| Ragazzo | 2000000 | 35143 kB | 2052 kB | 94,34% |
| Madonna | 1200000 | 21104 kB | 1364 kB | 93,65% |
| Carter | 1000000 | 19267 kB | 1273 kB | 94,40% |

Tabella 5.2: Risultati della compressione

| Modello | Conn. bpv | Conn. Ratio | Geom. bpv | Geom. Ratio |
|---|---|---|---|---|
| Ragazzo | 2,28 | 98,81% | 14,2 | 85,2% |
| Madonna | 2,31 | 98,80% | 15,7 | 83,7% |
| Carter | 2,64 | 98,63% | 16,3 | 83,0% |

Tabella 5.3: Statistiche del file compresso

contro i 192 bits del file originale, con un rapporto di compressione che si avvicina al 99% pur essendo frutto di una una procedura di tipo *lossless*, che fa uso cioè della tecnica di *valence based encodig* e della codifica di *Huffman*.

Anche la geometria raggiunge un buon rapporto di compressione, superiore all'80%, grazie all'integrazione delle tecniche di quantizzazione (*lossy*), di predizione del vertice e di codifica basata su entropia. Per la sola geometria il rapporto di compressione può essere aumentato ulteriormente utilizzando un fattore di quantizzazione meno elevato; questo metodo ha però dei contro, come mostrato nelle successive sezioni.

### 5.2.2 Tempi di codifica e decodifica

I test mostrano come, utilizzando la tecnica di codifica implementata, a fronte di una rappresentazione più compatta delle informazioni, si abbia lo svantaggio di tempi di caricamento più lunghi. Ciò è dovuto alle elaborazioni compiute durante la fase di ricostruzione della mesh (figura 5.11 nella pagina successiva).

In questo contesto il tempo di caricamento considerato è soltanto quello necessario alla creazione degli elementi della mesh (vertici, triangoli ed half-egdes) escludendo quindi i tempi impiegati nella costruzione della connettività e della display list, non rilevanti ai fini del confronto.

Dal grafico è possibile inoltre constatare che, nonostante le precedenti con-

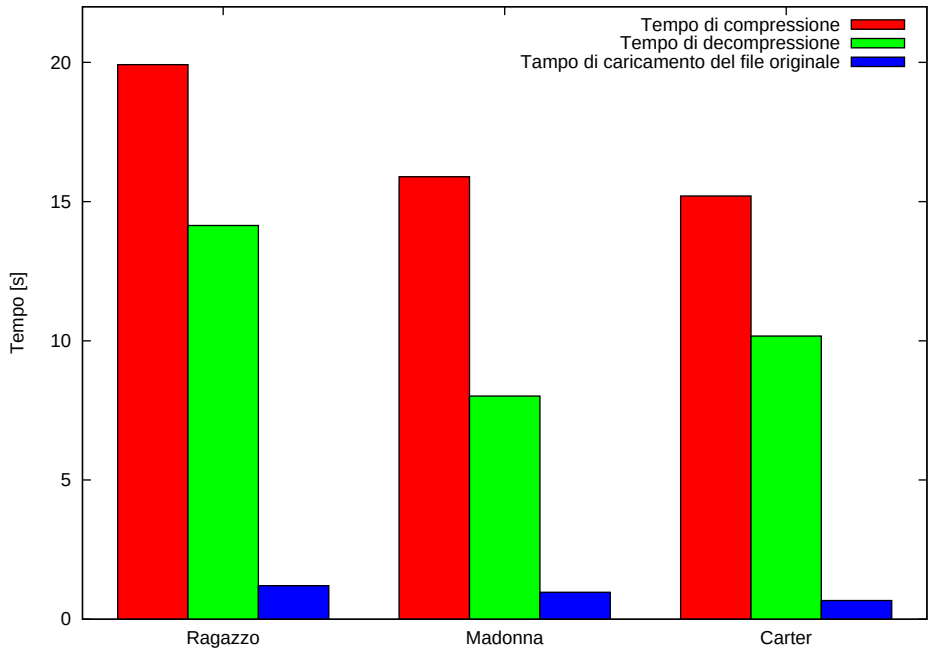



Figura 5.11: Tempi di compressione, di decompressione e di caricamento

siderazioni, i tempi di codifica e decodifica di una mesh complessa costituita da milioni di poligoni sono comunque dell'ordine di pochi secondi. Per tale motivo l'algoritmo di codifica può essere considerato idoneo all'archiviazione di modelli e scenari 3D anche molto complessi.

### 5.2.3 Valutazione dell'errore di quantizzazione

Si ricorda che l'algoritmo di codifica implementato fa un uso congiunto di tecniche sia *lossless* che *lossy*. In particolare la componente *lossy* è costituita dalla *quantizzazione* delle coordinate della geometria. I seguenti test hanno lo scopo di confrontare la qualità della mesh codificata alterando la configurazione del fattore di quantizzazione.

L'errore di quantizzazione è stato misurato con lo strumento *Metro* (sezione 5.1.1 nella pagina 106) e le prove sono state ripetute per ciascuno dei tre modelli quantizzando le coordinate dei vertici con 14, 12 e 10 bits.

I risultati dei test sono riportati nelle figure 5.12, 5.13 5.14. In tali istogrammi sono mostrate le dimensioni della connettività compressa e della geo-

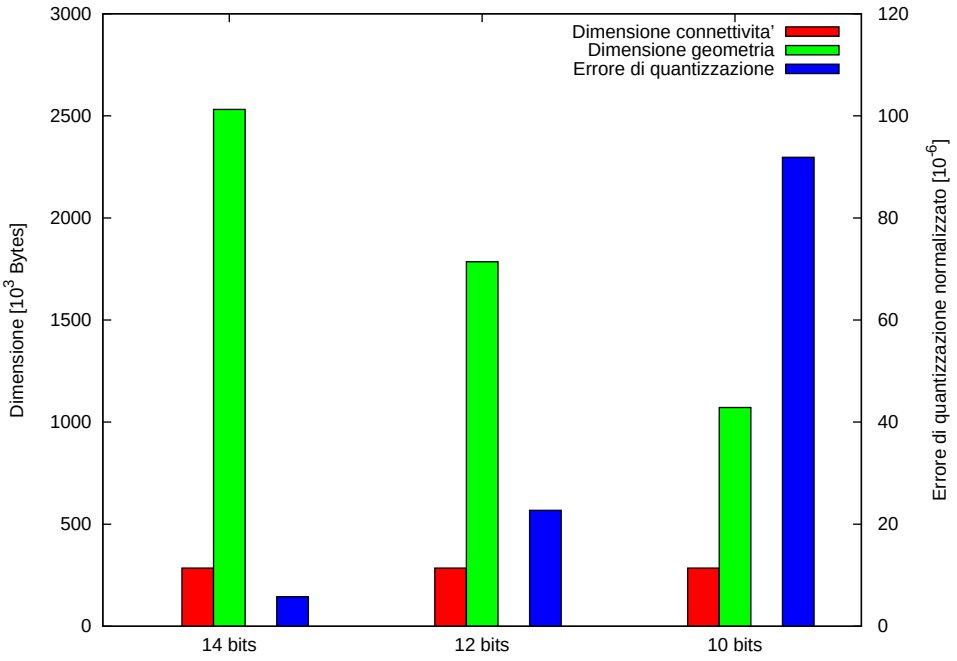



Figura 5.12: Errore di quantizzazione ("Ragazzo")

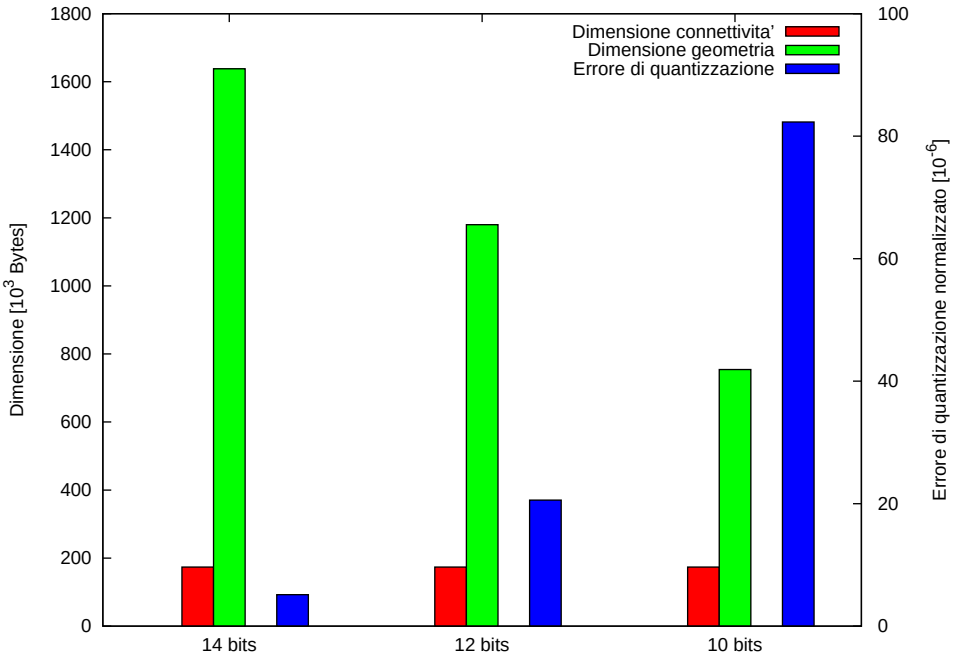



Figura 5.13: Errore di quantizzazione ("Madonna")

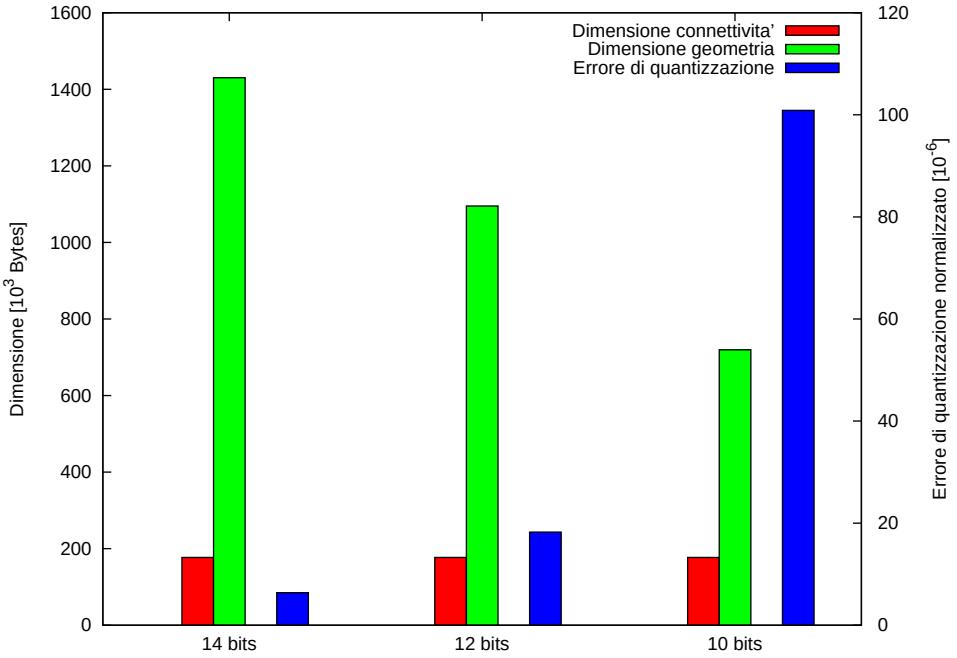



Figura 5.14: Errore di quantizzazione ("Carter")

metria compressa contemporaneamente all'errore di quantizzazione. Come prevedibile, al variare del fattore di quantizzazione la dimensione della connettività resta invariata, poiché la quantizzazione coinvolge soltanto la geometria. Al contrario, la geometria risulta più compatta quando è utilizzata una quantizzazione più aggressiva, a scapito di un errore maggiore.

Per modelli ad elevata risoluzione, come quelli su cui sono stati eseguiti i test, si nota come l'utilizzo di soli 10 bits per coordinata, pur rendendo la geometria molto compatta, porti a risultati di scarsa qualità: la rispettiva mesh ha infatti una qualità confrontabile con quella di una versione del modello composta da sole pochi migliaia di poligoni.

La perdita di una così grande parte dei dettagli è dovuta al fatto che quando i livelli di quantizzazione sono pochi, molti vertici che erano distinti nel modello originale collassano in un unico punto (*clustering*). Tale fenomeno ha l'effetto di diminuire il numero effettivo dei vertici della mesh, di formare triangoli degeneri e di invertire la normale di alcuni poligoni. Tali artefatti sono visibili come macchie più o meno scure sulla superficie della

mesh.

Utilizzando coordinate quantizzate con almeno 12 bits l'errore commesso si aggira intorno a pochi milionesimi della diagonale della bounding box, che corrisponde ad una diminuzione della qualità praticamente trascurabile.

## 5.3 Impiego dei livelli di dettaglio

I seguenti test misurano la performance della fase di rendering di uno scenario complesso che fa uso di classi capaci di gestire più *livelli di dettaglio* dei modelli che lo compongono.

Ad i modelli usati nei precedenti test si aggiungono altri due soggetti reperiti anch'essi dal repository pubblico di AIM@SHAPE. Questi modelli sono chiamati "Horse" e "Raptor" (figura 5.15 nella pagina successiva) e sono composti da circa due milioni di triangoli ciascuno.

### 5.3.1 Descrizione dello scenario

Lo scenario è stato realizzato dislocando varie copie dei cinque modelli su di un piano formando una sorta di scacchiera $5 \times 5$. Una panoramica dell'ambiente virtuale di test è visibile nella figura 5.16 nella pagina 123.

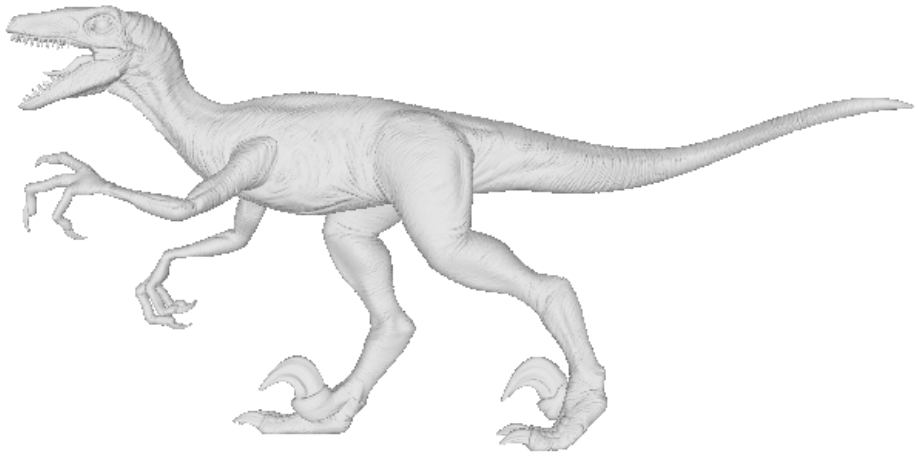
Il gestore grafico si occupa fondamentalmente di due compiti:

1. *creazione delle display lists* relative alle varie versioni di ciascun modello durante una fase di *preprocessing*

2. gestire a *run-time* la selezione del livello per ciascun oggetto della scena in funzione della sua distanza dall'osservatore

Ciascuno dei due compiti può essere configurato, in particolare possono essere specificati il numero di versioni volute per ciascun modello ed il rispettivo numero di poligoni per ciascuna di esse; la funzione di *switching dei LOD* prevede invece la scelta di una distanza $d$ tale che lo *switch* del livello di un oggetto avvenga quando la sua distanza dall'osservatore è un multiplo intero di $d$.

(a) "Horse"



(b) "Raptor"

Figura 5.15: I modelli introdotti per i test sui LOD

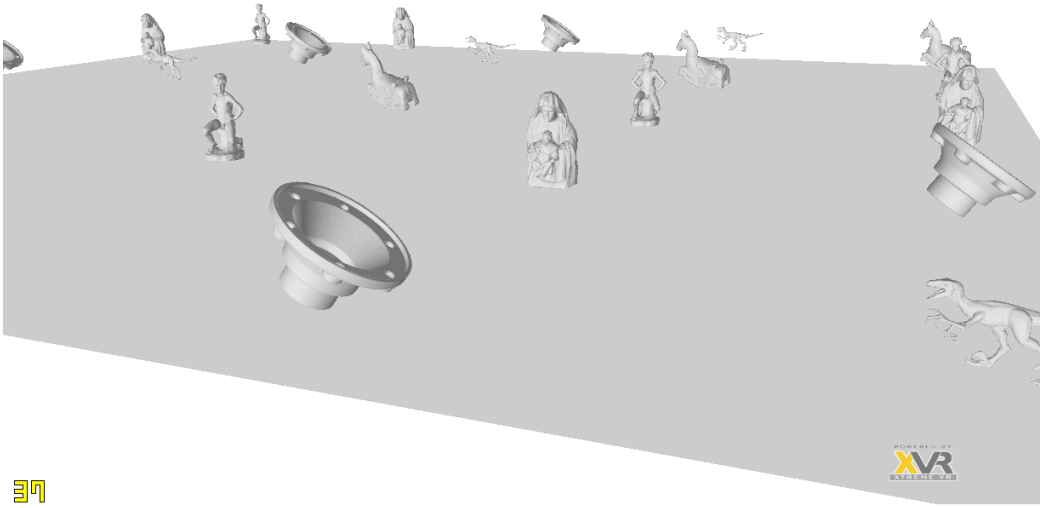


Figura 5.16: Lo scenario utilizzato per i test

Figura 5.17: Lo scenario mostrato in *wireframe*

| | |
|---|---|
| Numero di livelli | 5 |
| Livelli (triangoli) | $200k$, $100k$, $50k$, $20k$, $10k$ |
| Distanza di *switch* | 4.0 |

Tabella 5.4: Configurazione di base usata nei test

### 5.3.2 Configurazione e modalità di test

La configurazione di base utilizzata nei test è riportata nella tabella 5.4. Il livello di dettaglio massimo è stato scelto come trade-off tra la qualità dell'approssimazione e la velocità di rendering. In questo modo lo scenario è composto al più da $25 \times 200k = 5M$ di poligoni quando per ogni oggetto è selezionato il massimo dettaglio. Il numero di poligoni per ciascun livello di dettaglio è scelto in modo tale che il numero di poligoni si dimezza circa al raddoppiare della distanza dall'osservatore. Adottando questo tipo di configurazione si ottiene uno *switch* fluido dei livelli, vale a dire che il passaggio tra due livelli aventi diverso dettaglio non è percepibile visivamente.

Per avere idea in ogni momento dei livelli di dettaglio selezionati è possibile passare ad una modalità di rendering *wireframe* in cui a ciascun LOD corrisponde un colore differente (figura 5.17 nella pagina precedente).

I test sono stati effettuati muovendosi tra gli oggetti dello scenario con una telecamera mobile che segue la traiettoria illustrata nella figura 5.18. L'animazione descritta ha una durata di circa 35 secondi.

Le misurazioni consistono nella registrazione dell'istante relativo alla generazione di ciascun frame, in modo da creare un profilo temporale della fase di rendering. Lo strumento utilizzato per il rilevamento di tali tempi è *Fraps*[2], un tool gratuito che offre funzionalità di *benchmarking* per applicazioni OpenGL.

Per tutti i test è inoltre utilizzata la tecnica del *view-frustum culling* che fa uso della bounding box degli oggetti.

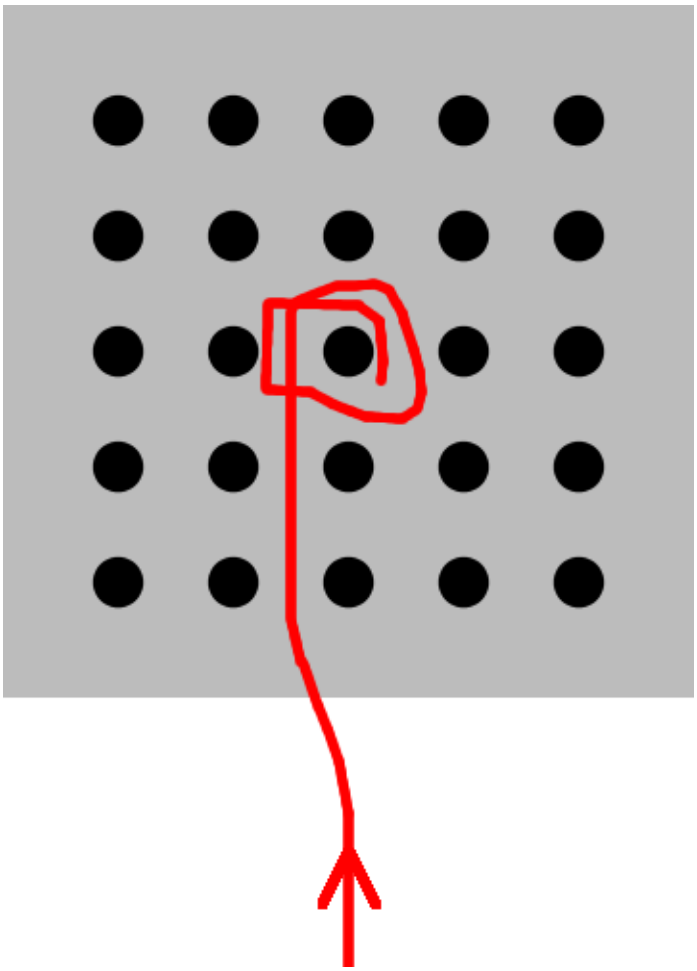


Figura 5.18: Il percorso seguito dalla telecamera durante i test

| | |
|---|---|
| Numero di livelli | 1 |
| Livelli (triangoli) | $200k$ |
| Distanza di *switch* | - |

Tabella 5.5: Configurazione con LOD non attivi

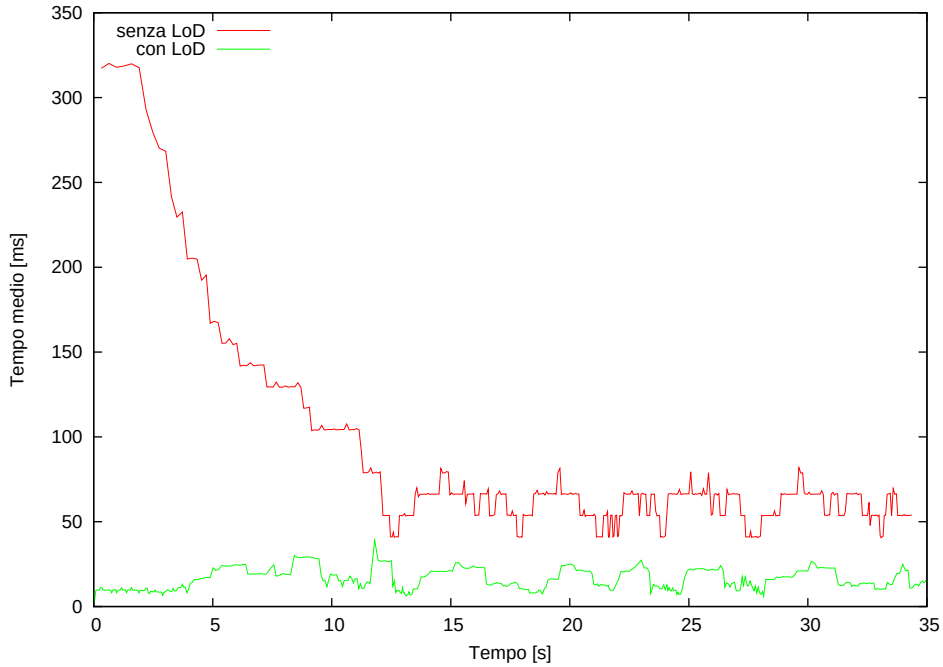



Figura 5.19: Tempo medio di generazione dei *frames*

### 5.3.3 Analisi delle performance

Nel primo test sono messi a confronto i profili temporali della fase di rendering utilizzando prima la configurazione descritta nella precedente sezione, e dopo ripetendo l'esperimento con modelli mono-risoluzione (tabella 5.5).

Il grafico della figura 5.19 mostra come il rendering sia molto più veloce quando i LOD sono attivi. Soprattutto nella fase iniziale in cui la camera, trovandosi molto distante dalla scena, inquadra tutti e 25 gli oggetti, la differenza è netta: l'uso del massimo livello di dettaglio impedisce all'hardware di poter generare più di tre frames al secondo.

Le performance aumentano nella fase centrale del test grazie al culling dei soggetti che non rientrano nell'inquadratura, ma generalmente le prestazioni del test con i LOD attivi sono superiori a causa del minor numero di poligoni da elaborare.

Il grafico della figura 5.20 riporta invece il numero totale di frames ge-

[2] http://www.fraps.com/

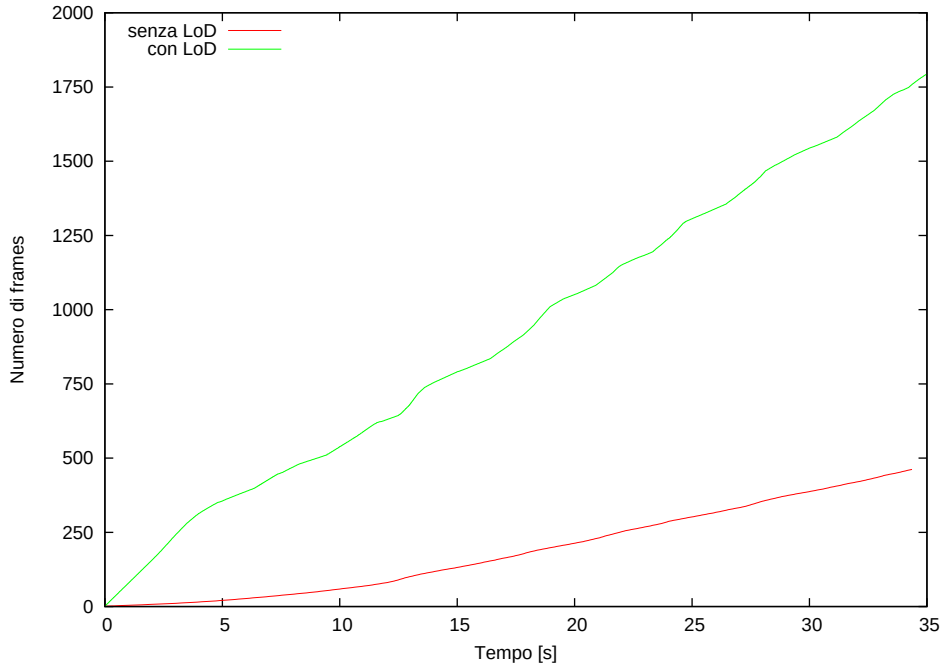



Figura 5.20: Numero di frames generati

nerati fino ad un certo istante: la sua pendenza rappresenta quindi il *frame rate*. Dall'analisi di tale grafico è possibile notare che al termine del test il numero di frames generati quando viene fatto uso di LOD è più del triplo rispetto a quanto tale tecnica non è operativa. In altre parole, impiegare un motore grafico che fa uso di modelli multi-risoluzione comporta un incremento di prestazioni mediamente superiore al 300% per quanto riguarda il rendering di questa specifica scena.

### Confronto tra diverse distanze di switching

In questa prova il test è stato ripetuto tre volte facendo variare ad ogni *run* la distanza di *switching* dei livelli: le configurazioni adottate prevedono l'impostazione della distanza $d$ rispettivamente a 4.0, 2.0 ed 8.0 unità.

Gli altri parametri di configurazione e la traiettoria seguita dalla camera restano invariati rispetto ad i test precedenti.

Dai grafici 5.21 e 5.22 nella pagina successiva si evince che le prestazioni aumentano diminuendo la distanza di *switching*. Tale risultato è motivato dal

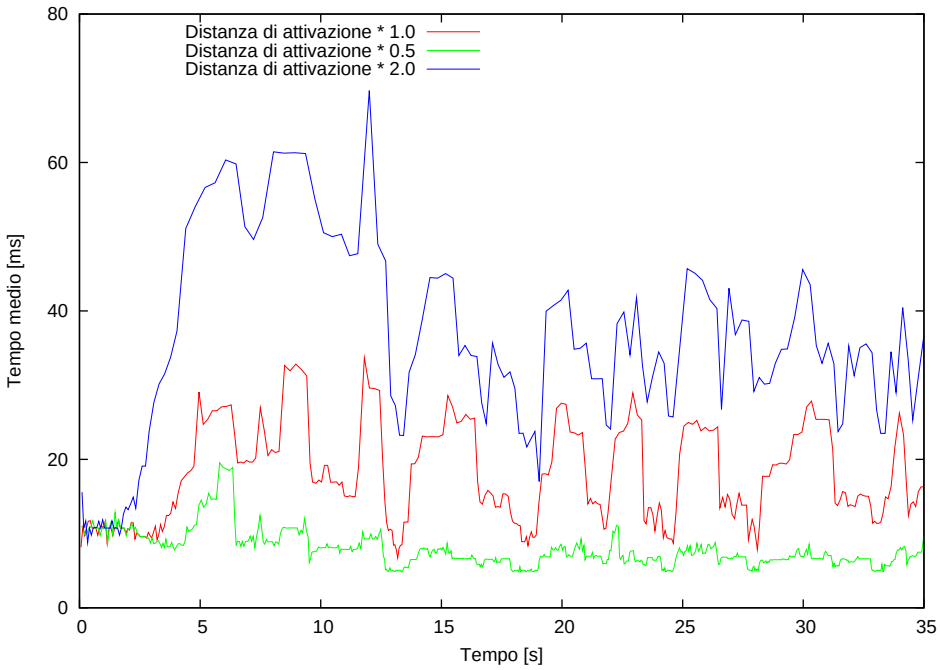



Figura 5.21: Tempo medio di generazione dei *frames* al variare della distanza di *switching* dei LOD

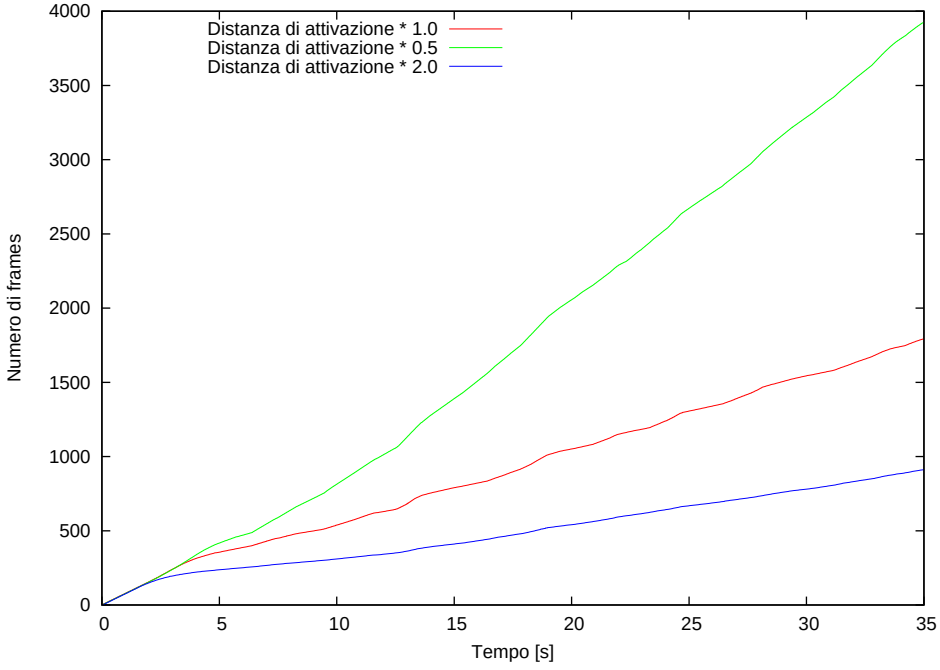



Figura 5.22: Numero di *frames* al variare della distanza di *switching* dei LOD

| | |
|---|---|
| Numero di livelli | 3 |
| Livelli (triangoli) | $200k$, $50k$, $10k$ |
| Distanza di *switch* | 8.0 |

Tabella 5.6: Configurazione con la metà di livelli

| | |
|---|---|
| Numero di livelli | 9 |
| Livelli (triangoli) | $200k$, $150k$, $100k$, $75k$, $50k$, $35k$, $20k$, $15k$, $10k$ |
| Distanza di *switch* | 2.0 |

Tabella 5.7: Configurazione con il doppio di livelli

fatto che per ciascun oggetto, a parità di distanza dell'osservatore, quando la distanza di *switching* è minore viene selezionata una versione del modello con un numero inferiore di poligoni.

Secondo le configurazioni adottate per questi test, l'incremento della velocità di rendering è all'incirca inversamente proporzionale alla distanza scelta.

Nonostante il fatto che la diminuzione della distanza di *switching* comporti un incremento delle prestazioni, è necessario notare che la qualità della scena risulta peggiorata poiché i soggetti sono rappresentati generalmente con un numero inferiore di poligoni.

Nel caso particolare del presente test, valutando empiricamente la qualità dei livelli selezionati quando la distanza di *switching* vale 2.0 unità, è possibile affermare che la perdita dei dettagli continua ad essere trascurabile. Rispetto al *run* privo della gestione dei LOD, tale esecuzione presenta un incremento della velocità di rendering di circa l'800%.

### LOD con diverso numero di livelli

In questi ultimi test, il profilo temporale della fase di rendering ottenuto con la configurazione di base è stato confrontato con quello di due ulteriori *run* le cui configurazioni sono riportate rispettivamente nelle tabelle 5.6 e 5.7.

Nel primo di essi è stato diminuito il numero di versioni, adattando la distanza di *switching* in modo tale che i livelli ad esse corrispondenti siano

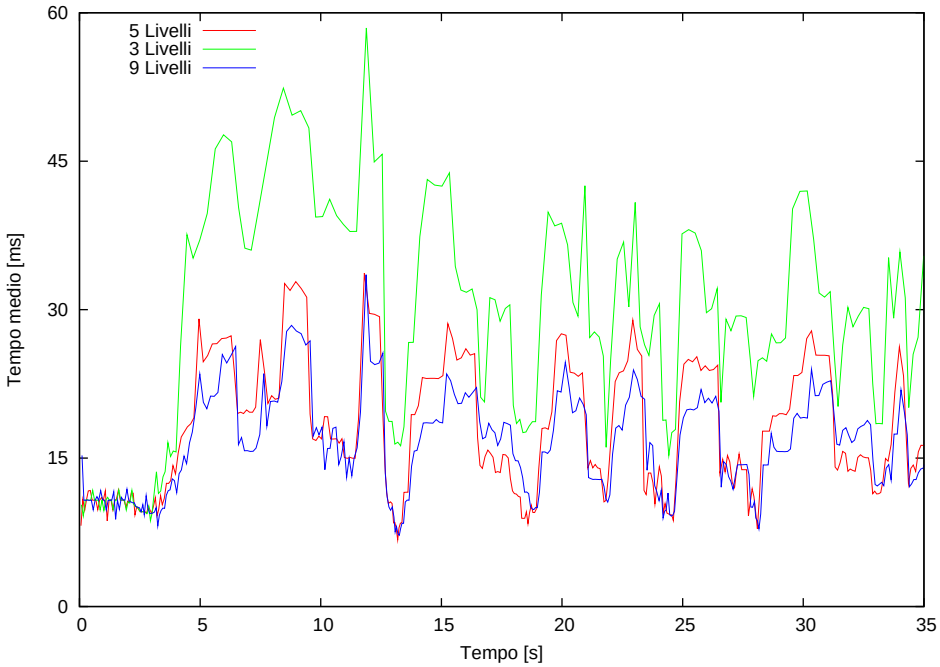



Figura 5.23: Tempo medio di generazione dei *frames* al variare del numero dei LOD

attivati alla stessa distanza con cui versioni aventi lo stesso dettaglio sono attivate nel test di base. Nel secondo *run* il numero di livelli è stato invece aumentato ed anche in questo caso la distanza di *switching* è stata opportunamente proporzionata.

I risultati sono riportati nei grafici delle figure 5.23 e 5.24. Dalla loro analisi si nota che, delle tre soluzioni, quella che comporta una minore velocità di rendering corrisponde a quella con un numero minore di livelli. Questo perché fintanto che la camera si trova ad una distanza inferiore ad 8 unità dal soggetto, di quest'ultimo viene selezionata la versione con il massimo dettaglio. Negli altri casi, invece, a distanze già inferiori possono essere selezionati livelli aventi la metà od addirittura un quarto di poligoni.

In modo analogo, dal grafico si può constatare che aumentando il numero di livelli dei modelli multi-risoluzione le prestazioni migliorano. Tale incremento ha però un limite, infatti quando sono utilizzati modelli dotati di numerose versioni, la differenza del numero di poligoni di due versioni

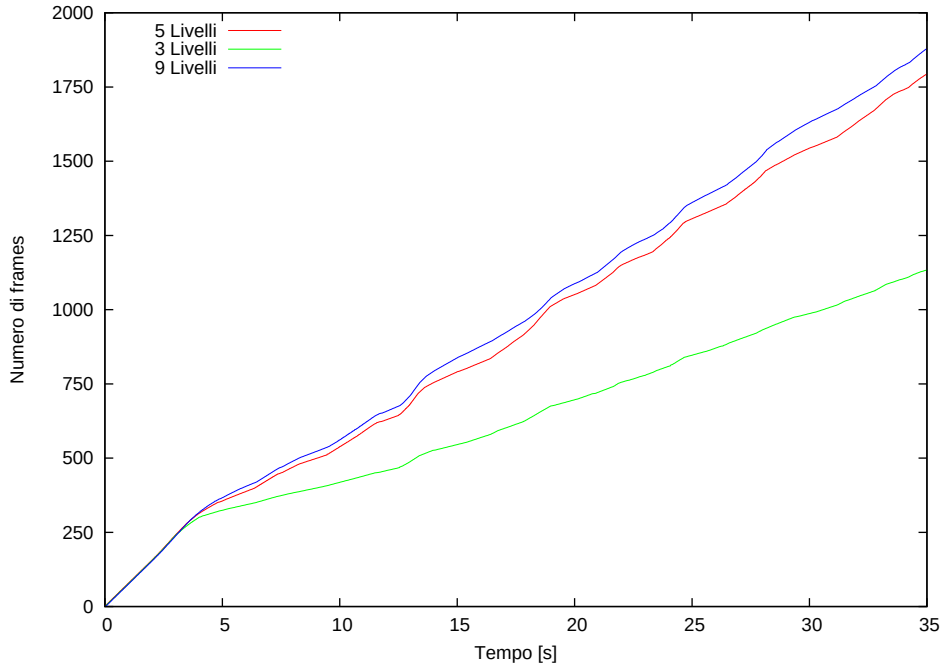



Figura 5.24: Numero di *frames* generati al variare del numero dei LOD

successive si fa sempre più piccola; pertanto la selezione di una o dell'altra delle due può non essere una scelta così determinante ai fini del carico del dispositivo grafico.

Inoltre, se i livelli di dettaglio sono molti, va tenuto in considerazione l'overhead dovuto allo *switching*: quando la memoria necessaria a memorizzare le display lists relative ad i livelli di tutti i modelli è cospicua, la scheda video può subire rallentamenti in corrispondenza delle numerose operazioni di svuotamento e caricamento della *video cache* a causa delle innumerevoli sostituzioni dei livelli da visualizzare.

Sono stati effettuati anche ulteriori test atti a confrontare le varie tecniche di *view-frustum culling* implementate. Tuttavia i miglioramenti introdotti non sono risultati significativi poiché il moderno dispositivo grafico usato per i test dispone di ottimizzazioni hardware che realizzano tali funzionalità.

# Capitolo 6

# Conclusioni e futuri sviluppi

## 6.1 Conclusioni

In questa tesi è stato affrontato lo studio di tecniche di semplificazione di mesh poligonali e di gestione dei livelli di dettaglio con il fine di ottenere una visualizzazione in tempo reale di modelli e di scenari tridimensionali ad elevata complessità. Inoltre, con l'obiettivo di conseguire ad una più efficiente gestione ed archiviazione di tali modelli, è stata svolta un'analisi approfondita degli algoritmi di codifica di mesh poligonali.

Il lavoro ha come risultato finale l'implementazione di un modulo software che integra nel *framework* di sviluppo per applicazioni interattive XVR le suddette funzionalità. In particolare, è stata implementata una *half-edge data structure* con cui è possibile gestire in modo semplice ed efficiente qualsiasi tipo di mesh triangolare *manifold*. Sulla base di ciò è stato realizzato un algoritmo di semplificazione incrementale basato su *quadric error metric* capace di generare livelli di dettaglio aventi un numero arbitrario di poligoni e che possiede le seguenti caratteristiche:

- *efficienza computazionale*: in un tempo dell'ordine di alcuni secondi è possibile ottenere versioni semplificate aventi pochi poligoni a partire da una mesh costituita da milioni di triangoli
- *conservazione della morfologia*: l'algoritmo consente di preservare la topologia, la conformazione locale ed eventuali bordi della mesh, e l'errore

rispetto alla superficie originale aumenta gradualmente al diminuire del numero dei triangoli

Nel modulo software è stato inoltre implementato l'algoritmo di codifica della connettività di mesh triangolari denominato *valence based encoding* e le tecniche di *quantizzazione* e di *predizione dei vertici con la regola del parallelogrammo* per quanto riguarda la compressione della geometria. Tali tecniche sono state poi integrate con l'algoritmo di codifica di *Huffman* per una riduzione dell'entropia dell'informazione.

Il risultato è un compressore che applica congiuntamente le tecniche sopra citate e che è in grado di ridurre di oltre il 90% un file binario contenente una mesh. Dai test è stato inoltre dimostrato che tale compressore è molto efficiente in termini di costo computazionale: modelli costituiti da milioni di triangoli sono stati codificati in poche decine di secondi. La sua efficacia può essere inoltre regolata configurando opportunamente il fattore di quantizzazione in modo da trovare un compromesso tra la dimensione del file compresso e l'errore di quantizzazione commesso.

Nella libreria è stato incluso anche un gestore di livelli di dettaglio con il quale è stato dimostrato sperimentalmente come sia possibile ottenere un incremento della velocità di rendering di oltre il 400%, permettendo la visualizzazione in tempo reale di scenari costituiti da milioni di poligoni; nelle fasi di test sono stati illustrati alcuni dei parametri su cui è possibile agire con lo scopo di trovare un trade-off tra prestazioni e qualità della scena, mostrandone i pro ed i contro.

## 6.2 Futuri sviluppi

A seguito di questi risultati gli sviluppi futuri del presente lavoro possono essere individuati nei seguenti punti:

- Integrazione del supporto delle *texture coordinates* sia nell'algoritmo di semplificazione che di compressione per poter utilizzare così le techiche di *texture mapping* o di *bump mapping*

- Realizzazione delle ottimizzazioni relative all'algoritmo di compressione *valence driven* citate in Alliez e Desbrun (2001) che non sono state ancora implementate, tra cui la sostituzione della codifica di Huffman con il *range encoder* (Schindler e Billrothstrasse, 1998), un tipo di compressore aritmetico adattativo

- Implementazione di tecniche gerarchiche di *culling*, come ad esempio l'*occlusion culling* tramite l'algoritmo *hierarchical Z-buffer* ed eventualmente uno *hierarchical view-frustum culling*

- Estensione dei LOD a tecniche gerarchiche come gli *HLODs*[1] (Erikson *e altri*, 2001) con cui è possibile migliorare notevolmente la velocità e la fedeltà del rendering di ambienti virtuali di elevate dimensioni

[1] *hierarchical levels of detail*

# Bibliografia


Alexandroff P. (1961). *Elementary Concepts of Topology*. Dover.

Alliez P.; Desbrun M. (2001). Valence-Driven Connectivity Encoding for 3 D Meshes. *Computer Graphics Forum*, **20**(3), 480–489.

Baumgart B. (1975). A polyhedron representation for computer vision. *AFIPS National Computer Conference*, **44**, 589–596.

Botsch M.; Steinberg S.; Bischoff S.; Kobbelt L. (2002). Openmesh-a generic and efficient polygon mesh data structure. *OpenSG Symposium*.

Cignoni P.; Montani C.; Rocchini C.; Scopigno R. (1998a). A general method for preserving attribute values on simplified meshes. *Proceedings of the conference on Visualization*, **98**, 59–66.

Cignoni P.; Rocchini C.; Scopigno R. (1998b). Metro: Measuring Error on Simplified Surfaces. *Computer Graphics Forum*, **17**(2), 167–174.

Clark J. H. (1976). Geometric Models for Visible Surface Algorithms. *Communications*.

Dey T.; Edelsbrunner H.; Guha S.; Nekhayev D. (1999). Topology preserving edge contraction. *Publ. Inst. Math.(Beograd)(NS)*, **66**(80), 23–45.

Duchaineau M.; Wolinsky M.; Sigeti D.; Miller M.; Aldrich C.; Mineev-Weinstein M. (1997). ROAMing terrain: Real-time Optimally Adapting Meshes. *Visualization'97., Proceedings*, pp. 81–88.

Erikson C.; Manocha D.; Baxter III W. (2001). HLODs for faster display of large static and dynamic environments. *Proceedings of the 2001 symposium on Interactive 3D graphics*, pp. 111–120.

Foley J. (1995). *Computer Graphics: Principles and Practice in C*. Addison-Wesley Professional.

Foley J.; Phillips R.; Hughes J.; van Dam A.; Feiner S. (1994). *Introduction to Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Funkhouser T.; Séquin C. (1993). Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pp. 247–254.

Funkhouser T.; Séquin C.; Teller S. (1992). Management of large amounts of data in interactive building walkthroughs. *Proceedings of the 1992 symposium on Interactive 3D graphics*, pp. 11–20.

Garland M.; Heckbert P. (1997). Surface simplification using quadric error metrics. *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 209–216.

Garland M.; Heckbert P. (1998). Simplifying surfaces with color and texture using quadric error metrics. *IEEE Visualization*, **98**, 263–270.

Garlick B.; Baum D.; Winget J. (1990). Parallel Algorithms and Architectures for 3D Image Generation, volume SIGGRAPH'90 Course Notes, Vol 28, chapter Interactive Viewing of Large Geometric Data Bases Using Multiprocessor Graphics Workstations. *ACM SIGGRAPH*, pp. 239–245.

Gotsman C.; Gumhold S.; Kobbelt L. (2001). Simplification and compression of 3d meshes. *European Summer School on Principles of Multiresolution in Geometric Modelling (PRIMUS), Munich*.

Greene N.; Kass M.; Miller G. (1993). Hierarchical Z-buffer visibility. *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pp. 231–238.

Gribb G.; Hartmann K. (2001). Fast Extraction of Viewing Frustum Planes from the World-View-Projection Matrix. *Online document.*

Gumhold S.; Straßer W. (1998). Real time compression of triangle mesh connectivity. *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 133–140.

Hartman J.; Wernecke J. (1996). *The VRML 2.0 handbook.* Addison-Wesley Reading, Mass.

Hoppe H. (1996). Progressive meshes. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 99–108.

Hoppe H. (1998). Efficient implementation of progressive meshes. *Computers & Graphics*, **22**(1), 27–36.

Huffman D. (1952). A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, **40**(9), 1098–1101.

Isenburg M.; Snoeyink J. (2000). Face fixer: compressing polygon meshes with properties. *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 263–270.

Isenburg M.; Snoeyink J. (2001). Spirale Reversi: Reverse decoding of the Edgebreaker encoding. *Computational Geometry: Theory and Applications*, **20**(1-2), 39–52.

James C. (1976). ≪ Hierarchical geometric models for visible surface algorithms≫. *Communications of the ACM*, **19**(10), 547–554.

Kalvin A.; Taylor R. (1994). Superfaces: Polyhedral approximation with bounded error. *Medical Imaging: Image Capture, Formatting, and Display*, **2164**, 2–13.

Kalvin A.; Taylor R. (1996). Superfaces: polygonal mesh simplification with bounded error. *Computer Graphics and Applications, IEEE*, **16**(3), 64–77.

Kettner L. (1999). Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry: Theory and Applications*, **13**(1), 65–90.

Kilgard M. (2000). A Practical and Robust Bump-mapping Technique for Today's GPUs. *Game Developers Conference 2000*.

Klein R.; Liebich G.; Straßer W. (1996). Mesh reduction with error control. *IEEE Visualization*, **96**, 311–318.

Le Gall D. (1991). MPEG: a video compression standard for multimedia applications. *Communications of the ACM*, **34**(4), 46–58.

Lee S.; Lee K. (2001). Partial entity structure: a compact non-manifold boundary representation based on partial topological entities. *Proceedings of the sixth ACM symposium on Solid modeling and applications*, pp. 159–170.

Lindstrom P.; Koller D.; Ribarsky W.; Hodges L.; Faust N.; Turner G. (1996). *Real-time, continuous level of detail rendering of height fields*. ACM Press New York, NY, USA.

Lorensen W.; Cline H. (1987). Marching cubes: A high resolution 3D surface construction algorithm. *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pp. 163–169.

Narkhede A.; Manocha D. (1995). Fast polygon triangulation based on Seidel's algorithm. *Graphics Gems V*, pp. 394–397.

Nelson M.; Gailly J. (1995). *The data compression book*. MIS: Press New York, NY, USA.

Phong B. T. (1975). Illumination for computer generated pictures. *Communications of the ACM*, **18**(6), 311–317.

Ronfard R.; Rossignac J. (1996). Full-range Approximation of Triangulated Polyhedra. *Computer Graphics Forum*, **15**(3), 67–76.

Rossignac J. (1999). Edgebreaker: connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, **5**(1), 47–61.

Rossignac J.; Borrel P. (1993). Multi-resolution 3D approximations for rendering complex scenes. *Modeling in Computer Graphics*, pp. 455–465.

Rost R. (1989). OFF-A 3D Object File Format. *Digital Equipment Corporation Technical Report, October*.

Schindler M.; Billrothstrasse V. (1998). A fast renormalisation for arithmetic coding. *Data Compression Conference, 1998. DCC'98. Proceedings*.

Schroeder W.; Zarge J.; Lorensen W. (1992). Decimation of triangle meshes. *ACM SIGGRAPH Computer Graphics*, **26**(2), 65–70.

Sheridan T. (1992). *Telerobotics, Automation, and Human Supervisory Control*. MIT Press.

Shreiner D.; Woo M.; Neider J.; Davis T. (2003). The OpenGL Programming Guide (Red Book).

Skodras A.; Christopoulos C.; Ebrahimi T. (2001). The JPEG 2000 still image compression standard. *Signal Processing Magazine, IEEE*, **18**(5), 36–58.

Slater M.; Chrysanthou Y. (1997). View volume culling using a probabilistic caching scheme. *Proceedings of the ACM symposium on Virtual reality software and technology*, pp. 71–77.

Touma C.; Gotsman C. (2000). Triangle mesh compression. US Patent 6,167,159.

Turk G. (1992). Re-tiling polygonal surfaces. *Computer Graphics(ACM)*, **26**(2), 55–64.

Weiler K. (1988). The radial edge structure: A topological representation for non-manifold geometric boundary modeling. *Geometric Modeling for CAD Applications*, **14**, 2002–3286.

Weiler K.; Electric G. (1985). Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments. *Computer Graphics and Applications, IEEE*, **5**(1), 21–40.

Witten I.; Neal R.; Cleary J. (1987). Arithmetic coding for data compression. *Communications of the ACM*, **30**(6), 520–540.