Università degli Studi di Genova

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Informatica

SEMPLIFICAZIONE DI MODELLI GEOMETRICI IN MEMORIA SECONDARIA

Candidato:
David Canino

Relatori:
Prof. P. Magillo
Dott. D. Sobrero

Correlatore:
Prof. P. Boccacci

Anno Accademico 2006-2007

Indice

1 Introduzione **1**
- 1.1 Stato dell'arte . . . 2
- 1.2 Il lavoro realizzato . . . 5
- 1.3 L'organizzazione della tesi . . . 8

2 Nozioni di base **11**
- 2.1 Elementi di topologia euclidea . . . 11
  - 2.1.1 Lo spazio metrico euclideo . . . 11
  - 2.1.2 Gli insiemi regolari e le varietá . . . 14
- 2.2 I complessi simpliciali euclidei . . . 16
- 2.3 La modellazione di superfici . . . 21
  - 2.3.1 Le approssimazioni simpliciali . . . 21
  - 2.3.2 Elementi di modellazione . . . 23
- 2.4 Esempi applicativi . . . 26
  - 2.4.1 La rappresentazione dei terreni . . . 26
  - 2.4.2 Il metodo degli elementi finiti . . . 28

3 La memoria secondaria **29**
- 3.1 La macchina di von Neumann . . . 29
- 3.2 L'organizzazione della memoria . . . 31
- 3.3 L'organizzazione dei dischi fissi . . . 32
  - 3.3.1 L'organizzazione logica . . . 34
  - 3.3.2 La memorizzazione fisica dei dati . . . 35
- 3.4 L'organizzazione dei dati . . . 37
  - 3.4.1 Il concetto di file . . . 37
  - 3.4.2 La gestione dello spazio su disco . . . 38
  - 3.4.3 Il modello PDM . . . 40

4 Le basi di dati **43**
- 4.1 Gli aspetti introduttivi . . . 43
  - 4.1.1 Basi di dati e DBMS . . . 43
  - 4.1.2 I modelli dei dati . . . 45
  - 4.1.3 I livelli nella rappresentazione dei dati . . . 47

4.1.4 I linguaggi di accesso e manipolazione 48
4.2 L'accesso ad una base di dati 48
4.2.1 Le transazioni 48
4.2.2 Il controllo della concorrenza 50
4.2.3 I livelli di isolamento 52
4.3 Le tecniche di ripristino 53
4.3.1 Il log incrementale con modifiche differite 54
4.3.2 Il log incrementale con modifiche immediate 54
4.3.3 La tecnica del checkpoint 54
4.4 Il sistema Oracle Berkeley DB 55
4.4.1 Introduzione 55
4.4.2 Le caratteristiche principali 56

5 Le strutture di indicizzazione 65
5.1 Introduzione 66
5.1.1 Una definizione di indice 66
5.1.2 Gli indici spaziali 68
5.2 La struttura dati *Albero Rosso–Nero* 70
5.2.1 Definizione della struttura dati 70
5.2.2 L'operazione di rotazione 72
5.2.3 L'operazione di ricerca 73
5.2.4 L'operazione di inserimento 73
5.2.5 L'operazione di cancellazione 77
5.3 La struttura dati *B–Albero* 81
5.3.1 Definizione della struttura dati 82
5.3.2 L'operazione di ricerca 83
5.3.3 L'operazione di inserimento 84
5.3.4 L'operazione di cancellazione 85
5.4 La struttura dati *Octree* 87
5.4.1 Definizione della struttura dati 87
5.4.2 L'operazione di inserimento 92
5.4.3 L'operazione di cancellazione 94
5.4.4 Le interrogazioni supportate 95
5.5 La struttura dati *K–d tree* 96
5.5.1 Definizione della struttura dati 97
5.5.2 L'operazione di inserimento 99
5.5.3 L'operazione di cancellazione 102
5.5.4 Le interrogazioni supportate 104
5.6 Le strutture dati *ibride* 105
5.6.1 Aspetti introduttivi 106
5.6.2 Le varianti ibride degli indici *Octree* e *K–d tree* 108

6 La libreria OMSM **113**
6.1 Le caratteristiche generali 114
6.1.1 Principi di programmazione ad oggetti 114
6.1.2 Aspetti introduttivi 116
6.1.3 La gestione degli errori 118
6.2 Il componente *OperationTimer* 119
6.3 La struttura dati *ERB–Albero* 123
6.3.1 Le proprietá della struttura dati *ERB–Albero* 124
6.3.2 Il componente *Cache* 129
6.3.3 La tabella delle corrispondenze 132
6.4 La gestione delle triangolazioni 134
6.4.1 L'architettura di I/O su file 135
6.4.2 Il caricamento delle mappe poligonali 138
6.4.3 La verifica delle triangolazioni 144
6.4.4 La conversione delle triangolazioni 151
6.4.5 La decomposizione delle triangolazioni 152
6.5 La memorizzazione dei dati spaziali 154
6.5.1 La persistenza degli oggetti 156
6.5.2 La rappresentazione degli oggetti geometrici 159
6.5.3 Il livello *SPDataIndex* 163
6.5.4 Il livello *NodeHandler* 169
6.5.5 Il livello *NClusterStorager* 174
6.5.6 La configurazione del sistema 176

7 I risultati ottenuti **185**
7.1 I test preliminari 185
7.1.1 Il programma *Types* 186
7.1.2 Il programma *Timers* 187
7.1.3 I programmi *Cubel* e *Cubeb* 189
7.2 La gestione dei modelli geometrici 192
7.2.1 Il programma *Check* 194
7.2.2 Il programma *ToSoup* 196
7.3 La decomposizione dei modelli geometrici 198
7.3.1 Il programma *Omsmconf* 199
7.3.2 Il programma *Decompose* 202

8 Conclusioni e sviluppi futuri **207**
8.1 Un framework utile a molti scopi 209
8.2 Una tecnica utile a molti scopi 210

Bibliografia **215**

Capitolo 1

Introduzione

> Quello che vuoi dire non é che non lo capisci, ma che non l'hai inventato tu. Quello che devi fare, é immaginare che tu sia di nuovo uno studente, portarti l'articolo in camera e leggerlo riga per riga, riverificando tutte le equazioni. Allora ti sará molto facile capire.
>
> *Richard Feynman*

La grafica tridimensionale gioca un ruolo di primo piano in molti domini applicativi e si propone come elemento essenziale nell'interazione uomo–macchina, ma la richiesta di modelli geometrici tridimensionali sempre piú accurati e la disponibilitá di tecnologie in grado di produrli sta facendo crescere rapidamente la dimensione dei modelli geometrici che é necessario codificare. Inoltre, gli ambienti distribuiti forniscono un canale di trasmissione estremamente capillare ed utile per la loro condivisione: la dimensione di tali modelli e la ridotta ampiezza di banda comunemente disponibile implicano peró tempi di attesa molto elevati. Un esempio é dato dai modelli usati nel progetto ABA, (acronimo di *Allen Brain Atlas*, per approfondimenti rifarsi a [ABA06]) il quale si occupa della visualizzazione tridimensionale delle aree di influenza di un singolo gene nel cervello del topo, rendendole disponibili via Web ai ricercatori. Basti pensare che vi sono piú di 20000 aree di interesse e quindi l'intera mappa eccede la quantitá di memoria primaria comunemente disponibile in un elaboratore, rendendola cosí difficilmente gestibile.

Una soluzione possibile sarebbe quella di aumentare la quantitá disponibile di RAM in un elaboratore visto che il suo costo é nettamente in discesa rispetto al passato, tuttavia questa soluzione solleva alcune problematiche:

- non é una tecnica facilmente *scalabile* in quanto i modelli geometrici tendono ad avere dimensioni sempre maggiori e quindi questo aumento di memoria primaria non risolverebbe *veramente* il problema;

- non é una tecnica *fattibile* in quanto é piuttosto complicato stabilire una dimensione fissata di memoria da utilizzare, anche perché una certa quantitá di RAM é necessaria per il funzionamento dell'elaboratore.

Per questi motivi é importante codificare tali modelli nel modo piú efficiente possibile, mantenendo tuttavia la possibilitá di analizzarli e visualizzarli, a seconda delle richieste dell'utente.

1.1 Stato dell'arte

Il primo problema che si incontra nella nostra analisi é quello di modellare in un calcolatore superfici e campi scalari immersi nello spazio euclideo $\mathbb{E}^3$: bisogna utilizzare una rappresentazione discreta, in modo tale da poterla elaborare in maniera efficiente. I due approcci piú comuni in letteratura per modellare un oggetto $\mathcal{O}$ sono:

- fornire una rappresentazione analitica di $\mathcal{O}$;
- scomporre $\mathcal{O}$ in celle, rappresentabili in maniera indipendente.

Le rappresentazioni analitiche sono influenzate da molti parametri e quindi il loro utilizzo é poco flessibile, specialmente con grandi quantitá di dati. In realtá possono essere utilizzate in ambiti ben precisi, come la modellazione di terreni attraverso i frattali, una tecnica molto nota in letteratura: per approfondimenti rifarsi a [Jul22], [Man77], [Hut81], [Lew87], [Vos88], [MM91], [Mus93] e [DK03].

Pertanto é preferibile utilizzare il secondo approccio: questa scelta richiede la definizione di *complesso simpliciale*. Quest'ultimo é un'entitá geometrica composta da un insieme finito di celle, dette *simplessi*, immerse in uno spazio metrico d–dimensionale: le celle devono essere disposte in modo tale da soddisfare delle relazioni fissate. Un complesso simpliciale puó approssimare un certo oggetto geometrico $\mathcal{O}$: ad esempio una triangolazione puó essere utilizzata per rappresentare una superficie mentre una griglia di tetraedri puó descrivere un certo volume. Per approfondimenti sul processo di approssimazione di un oggetto con un complesso simpliciale rifarsi a [PS85], [Ede87], [Man88], [Hof89] e [Req96].

L'accuratezza con cui un complesso simpliciale approssima un oggetto dipende dalla granularitá piú o meno fine dei suoi simplessi, solitamente detta *risoluzione* o *livello di dettaglio* (nota anche come *LOD*, dall'inglese *Level–of–Detail*). In letteratura non vi é una definizione univoca di risoluzione: in questa tesi useremo quella introdotta in [Mag99]. Molte applicazioni richiedono un'alta risoluzione solamente in alcune zone del modello: sarebbe utile adattare localmente la risoluzione, facendola variare in funzione delle richieste dell'utente. Le zone meno importanti dovrebbero essere rappresentate da pochi simplessi a bassa risoluzione mentre quelle piú interessanti per l'utente dovrebbero essere rappresentate con un'accuratezza maggiore. Questo risultato si ottiene applicando degli operatori di modifica alla mappa poligonale: nel nostro caso siamo interessati a degli operatori di semplificazione, i quali ne riducono il livello di dettaglio. Una descrizione piú accurata richiede un

numero maggiore di simplessi e comporta costi di elaborazione piú elevati: l'operazione di semplificazione ne riduce la risoluzione e quindi l'occupazione di memoria. La dimensione degli oggetti geometrici che vogliamo rappresentare in questa tesi eccede la quantitá di memoria *RAM* disponibile in un calcolatore quindi questa tecnica ci consente di ridurre l'occupazione spaziale dei modelli in input. In [PS97] vengono delineate le relazioni esistenti fra l'operazione di semplificazione ed il livello di dettaglio di un'approssimazione simpliciale.

In realtá siamo interessati a delle tecniche che operino in maniera automatica su un complesso simpliciale: in [AS94] viene dimostrato che individuarne la semplificazione ottimale é un problema *NP–completo* e quindi non é facilmente risolvibile. Per gestire questo problema, in letteratura sono state sviluppate moltissime tecniche euristiche per ottenere delle versioni approssimate della mappa poligonale di partenza, come quelle descritte in [RB92], [SZL92], [GH97], [CMS98], [GH98], [Gar99a] e [LRC+02]. Tuttavia tali metodi richiedono la memorizzazione dell'intera mesh in memoria primaria e quindi questa soluzione non é applicabile con modelli geometrici di elevata dimensione come quelli prodotti nel progetto *Digital Michelangelo*, promosso dalla Stanford University: per approfondimenti rifarsi a [LPC+00].

Pertanto sono stati sviluppati dei metodi che permettono la semplificazione di un modello geometrico parzialmente mantenuto in memoria secondaria. In estrema sintesi, il funzionamento di queste tecniche é analogo a quello della *memoria virtuale*, presente ormai su tutti i piú recenti processori: l'idea é quella di decomporre un modello geometrico in porzioni secondo un qualche criterio (funzionale oppure spaziale) per poi mantenere in memoria primaria solamente una certa porzione. Il resto del modello é mantenuto nella memoria secondaria e caricato dinamicamente, se necessario: questo approccio é giustificato dalle possibili esigenze dell'utente. Rifacendosi al progetto *ABA*, l'utente puó essere interessato all'area di influenza di un singolo gene e voler visualizzare solo quell'area, che supponiamo limitata. Con questo approccio possiamo caricare solamente la porzione richiesta, sulla quale é possibile operare a seconda delle varie esigenze e quindi anche semplificarne il livello di dettaglio. In questo modo la dimensione dei modelli gestibili non é piú limitata dalla quantitá di *RAM*, ma da quella della memoria secondaria: quest'ultima é normalmente presente in un calcolatore in grosse quantitá rispetto a quella volatile in quanto il suo costo per megabyte é irrisorio.

Vediamo ora alcune fra le tecniche di semplificazione piú interessanti: in [DdFPS06] vengono passate in rassegna quelle relative alla gestione dei modelli di terreni in memoria secondaria, molte delle quali possono essere estese alle mappe poligonali generiche.

Un gran numero di tecniche si basa sull'operatore di semplificazione *vertex–clustering*, introdotto in [RB92]: si basano sul fatto che l'oggetto da semplificare puó essere suddiviso attraverso una griglia piú o meno regolare, la quale decompone i punti della mappa poligonale in insiemi disgiunti,

contenuti all'interno delle celle della suddivisione. I punti all'interno di ogni cella vengono cancellati e sostituiti da un punto rappresentativo, scelto secondo un qualche criterio. In [Lin00] il punto rappresentativo viene scelto attraverso la misura dell'errore di tipo *QEM* (dall'inglese *Quadric–Error Metric*), introdotta in [GH97], migliorarando in questo modo la qualitá della semplificazione. In [LS01] viene proposta un'estensione di questa tecnica sfruttando una quantitá ridotta di memoria primaria in maniera pressoché indipendente dal modello in esame: questa caratteristica é molto importante con dispositivi aventi ridotte quantitá di RAM come i palmari, i cellulari ed altri sistemi mobili. Un'ulteriore miglioramento di questa tecnica consiste nell'applicare alla mesh iniziale l'algoritmo di compressione introdotto in [IG03]: in questo modo viene costruita una versione della mesh attraverso una codifica che puó essere letta e scritta in maniera efficiente sul supporto di memorizzazione. Su questa codifica é definito un algoritmo di semplificazione, descritto in [ILGS03], il quale si basa sull'analisi di porzioni ridotte della mesh, dette *sequenze*. Una sequenza puó essere facilmente gestita in memoria primaria: una volta semplificata, ogni sequenza viene scritta nuovamente sul supporto di memorizzazione. Questo tipo di tecniche prende il nome di *streaming mesh*: per una completa rassegna di queste tecniche rifarsi a [IL05]. Questi algoritmi possono essere utilizzati anche per la semplificazione di una griglia di tetraedri, come descritto in [VCL$^+$07]. L'implementazione in memoria secondaria dell'operatore *vertex–clustering* é molto semplice e veloce, ma la qualitá del risultato puó essere molto scadente, a causa delle distorsioni indotte dall'utilizzo di una griglia discreta per la semplificazione dell'oggetto geometrico. Per ovviare a questo problema, si puó utilizzare la tecnica della semplificazione iterativa, utilizzando ad esempio l'operatore di collassamento degli spigoli, noto in letteratura come *edge–collapse*. Solitamente questa tecnica viene combinata con una suddivisione gerarchica del dominio del complesso simpliciale realizzata attraverso un indice spaziale, in modo da facilitare le operazioni. Alcuni esempi di indici spaziali sono descritti in [Sam90a], [Sam90b] e [Sam06].

Una delle prime tecniche sviluppate attraverso l'operatore di collassamento degli spigoli é quella descritta in [Hop98], adatta alla semplificazione di terreni. Si basa sulla decomposizione gerarchica del dominio in porzioni di una certa grandezza: ogni blocco della suddivisione puó essere semplificato attraverso l'applicazione iterativa dell'operatore di collassamento degli spigoli. Una volta terminata l'analisi di una singola porzione, é possibile fonderla con quelle adiacenti e ripetere ricorsivamente il processo di semplificazione. Lo scopo di questa tecnica é quello di ottenere una rappresentazione semplificata del modello di partenza, memorizzato su un numero ridotto di blocchi e direttamente gestibile in memoria primaria. In [Pri00] questa tecnica viene estesa per poter supportare la gestione di una generica griglia di triangoli. Con queste due tecniche non é possibile applicare l'operatore di semplificazione sugli spigoli incidenti nel contorno dei blocchi e quindi si pos-

sono ottenere degli sgradevoli artefatti, ben visibili sul modello semplificato, in quanto sono a risoluzione maggiore rispetto al resto. Questo problema viene risolto dalla tecnica descritta in [CMRS03], la quale scompone il complesso simpliciale in input attraverso l'indice spaziale *Octree*, introdotto in [FB74]. Anche in questo caso l'algoritmo di semplificazione parte delle foglie: la differenza rispetto alle tecniche precedenti consiste nel fatto che durante l'operazione di modifica vengono caricati i nodi adiacenti in modo da poter operare sugli spigoli appartenenti al contorno della porzione di mesh contenuta nell'ottante descritto dalla foglia sotto analisi. Una volta semplificate due o piú foglie adiacenti, é possibile fonderle ed applicare ricorsivamente l'algoritmo di modifica. Anche in questo caso l'obiettivo di questa tecnica é quello di ottenere una rappresentazione semplificata di quella di partenza che sia memorizzata in un unico nodo foglia e quindi direttamente gestibile in memoria primaria. Le tecniche di semplificazione descritte in [CMRS03] costituiscono il punto di partenza per la nostra analisi: in questa tesi vedremo come poterle generalizzare rispetto all'indice spaziale e all'algoritmo di semplificazione utilizzati.

1.2 Il lavoro realizzato

Gran parte delle tecniche descritte nel paragrafo 1.1 si basano sulla *decomposizione* di una certa mappa poligonale in moduli ed ogni suddivisione viene eseguita secondo un qualche criterio, fissato a priori: i moduli cosí ottenuti verranno modificati con un particolare operatore di semplificazione.

La presenza di molte tecniche di decomposizione disponibili non facilita la modularitá delle applicazioni visto che un cambiamento dell'algoritmo utilizzato potrebbe condurre ad uno stravolgimento pressoché completo dell'applicazione stessa. In realtá questo processo potrebbe essere facilitato dall'esistenza di una piattaforma comune di sviluppo: solitamente si usa il termine inglese *framework* a tale proposito. In questa maniera é possibile implementare vari metodi di decomposizione senza stravolgere il funzionamento di una applicazione: a conferma di quanto detto é possibile osservare, ad esempio, che il processo di decomposizione eseguito da uno specifico indice spaziale si caratterizza per la politica di suddivisione utilizzata e non influisce su altri aspetti, come la memorizzazione fisica dei dati, i quali rimangono pressoché inalterati. Nella nostra ricerca identificheremo la memoria secondaria con il disco fisso, anche se questo termine ha un significato piú generale (pensiamo ad esempio all'approccio distribuito), cercando di creare un sistema il piú modulare possibile e funzionante con modelli parzialmente memorizzati in memoria secondaria.

Il primo problema da risolvere per poter raggiungere il nostro obiettivo é quello di creare un framework per poter memorizzare in memoria secondaria una mesh simpliciale. In letteratura sono state sviluppate una serie di tec-

niche, le quali gestiscono la memorizzazione dei dati geometrici in memoria secondaria: questo problema é stato ampiamente trattato in letteratura e le soluzioni proposte sono divenute ormai di uso comune. Inoltre sono stati sviluppati esempi di sistemi per la gestione di dati spaziali come quelli descritti in [IBM01], [BGK04] e [OSP05]. Il funzionamento di tali architetture puó essere descritto studiando questi tre aspetti:

- l'utilizzo di una struttura ausiliaria di accesso ad albero, detta *indice spaziale*, per facilitare le operazioni di aggiornamento e di interrogazione sui dati: alcuni esempi di indici sono descritti in [Sam06];
- la suddivisione dei nodi dell'indice spaziale in cluster secondo una certa politica in modo da minimizzare il numero di accessi alla memoria secondaria, la quale é piú lenta di quella primaria: per *cluster* si intende un gruppo di nodi, scelti in base ad un certo criterio, i quali possono essere considerati un'unica entitá;
- la gestione dinamica dei cluster in memoria secondaria: questo aspetto puó essere gestito attraverso varie tecniche di memorizzazione, le quali dipendono anche dalla distribuzione fisica dei dati.

Come si puó notare, questi tre aspetti possono essere considerati indipendenti fra loro e le tecniche utilizzate per la loro gestione possono essere combinate in maniera ortogonale. Molte architetture di memorizzazione prevedono una serie di scelte fissate a priori, soprattutto per quanto riguarda gli ultimi due aspetti, mentre é oramai consuetudine fornire la possibilitá di scegliere fra piú indici spaziali. Ad esempio, l'architettura *GiST*, introdotta in [HNP95], permette di variare solamente il tipo di indice spaziale, mentre la sua politica di clustering dei nodi é quella descritta in [DRSS96]: inoltre questo framework é capace di gestire solamente un database locale. Quindi le soluzioni proposte in letteratura riguardano solamente determinati aspetti del problema e non si adattano in maniera dinamica alle varie esigenze di memorizzazione.

In questa tesi viene definita la struttura del framework *OMSM* (dall'espressione inglese *Objects Management in Secondary Memory*), adatto alla gestione di grosse quantitá di dati geometrici in memoria secondaria: questa architettura puó integrare fra loro le diverse tecniche sviluppate in letteratura e garantisce la massima flessibilitá possibile nella risoluzione di questo problema. La struttura di questo framework é multi–livello e ricalca quella del modello *PDM* (dall'inglese *Parallel Disk Model*), introdotto in [VS94a] e [VS94b]: ognuno dei tre aspetti delineati precedentemente viene gestito da uno specifico componente. Per garantire un ampio grado di modularitá non viene assunto l'utilizzo di una particolare tecnica per ogni livello, ma solamente che un certo livello sia in grado di offrire dei servizi in relazione al suo ruolo. Quindi il funzionamento di un livello puó essere considerato

indipendente da quello degli altri: l'utente é in grado di modificare l'effettivo funzionamento del framework, dunque é possibile ottenere diversi comportamenti dell'architettura *OMSM*, a seconda delle varie esigenze. Una caratteristica importante di questo framework é la possibilitá di memorizzare un certo insieme di entitá geometriche di dimensione arbitraria: l'utilizzo di una tecnica di persistenza indipendente dalla dimensione permette di rappresentare oggetti geometrici di dimensione topologica diversa, ma immersi nello stesso spazio metrico euclideo. Ad esempio questo framework é in grado di memorizzare sia griglie di triangoli immersi in $\mathbb{E}^2$ o in $\mathbb{E}^3$ e sia griglie di tetraedri, le quali stanno assumendo un ruolo sempre piú importante nella visualizzazione scientifica. L'indicizzazione spaziale dei dati avviene tramite il concetto di *punto rappresentativo*: ad ogni oggetto γ viene associato, secondo un qualche criterio, un punto multi–dimensionale P_γ, il quale *"descrive"* le proprietá piú importanti di quella entitá. In questo modo si potrá utilizzare P_γ come chiave all'interno di un indice spaziale, mentre l'oggetto geometrico γ sará il dato associato a P_γ.

É evidente che l'obiettivo dell'architettura *OMSM* non é quello di *simulare* il comportamento di un certo indice spaziale, come avviene nel sistema *GiST*, il quale garantisce il bilanciamento della struttura a prescindere dal tipo di indice utilizzato: questa proprietá puó essere fuorviante nell'analisi del comportamento di una struttura di accesso in memoria secondaria. Invece il framework *OMSM* fornisce una piattaforma comune con la quale implementare le operazioni sui dati, utilizzando le funzionalitá fornite dai tre livelli appena descritti. A priori non viene garantita l'efficienza delle operazioni sui dati in quanto essa é la conseguenza delle scelte effettuate in fase di configurazione: ad esempio non viene garantito il bilanciamento dell'indice spaziale utilizzato in quanto questa proprietá dipende dalla struttura ausiliaria di accesso che si vuole utilizzare. Questa proprietá permette una migliore comprensione del comportamento di un indice in memoria secondaria e dunque il framework *OMSM* si prefigura come un ottimo strumento per la verifica sperimentale del funzionamento di una certa tecnica, in maniera indipendente dalle altre. In letteratura sono state sviluppate architetture simili come quelle discusse in [Bow01], [VOU04], [AWC05] e [SGV+05].

Per dimostrare la fattibilitá della soluzione proposta, é stata sviluppata una versione preliminare dell'architettura *OMSM* in grado di decomporre una griglia di triangoli immersa nello spazio euclideo a tre dimensioni, supportando diverse scelte implementative per i vari livelli che compongono il framework. In questo prototipo vengono supportati gli indici spaziali *K–d tree*, *Octree*, *PR K–d trie* e *PR Quadtrie* di tipo ibrido: per approfondimenti su tali strutture rifarsi a [Sam06]. La politica di suddivisione dei nodi in cluster utilizzata é quella secondo la quale un cluster puó contenere un solo nodo dell'indice, mentre i cluster possono essere memorizzati in un database locale, realizzato attraverso il sistema software *Oracle Berkeley DB*, un esempio molto noto di *DBMS* di tipo embedded, introdotto in [Bdb06d]. Bi-

sogna ricordare che la versione realizzata é un prototipo iniziale quindi potrá essere ulteriormente estesa in futuro: questo framework é contenuto all'interno della libreria *OMSM*, la quale contiene l'ampia parte implementativa del lavoro di ricerca sviluppato in questa tesi.

Anche se le tecniche di semplificazione in memoria secondaria sono le uniche con le quali gestire modelli di elevate dimensioni, il loro utilizzo solleva alcune problematiche riguardanti l'utilizzo dei supporti di memorizzazione. Un primo problema é legato alla dimensione del disco fisso ed alle sue caratteristiche fisiche in quanto la densitá di memorizzazione é ormai prossima alla saturazione, a meno di modificare completamente il paradigma di sviluppo di un calcolatore. La risoluzione di questo problema esula decisamente dagli scopi di questa tesi, ma ne verranno delineati i caratteri principali. Un'altra problematica é relativa all'organizzazione dei dati memorizzati in modo da minimizzarne il tempo di accesso, visto che la memoria secondaria é nettamente piú lenta della RAM: si puó ovviare a questo fatto con un'adeguata politica di *caching* e con l'utilizzo di strutture ausiliarie di accesso ai dati memorizzati.

1.3 L'organizzazione della tesi

Di seguito vedremo l'organizzazione di questo documento, che idealmente puó essere suddiviso in tre parti. Nella prima parte vengono proposte alcune nozioni di base riguardanti la topologia euclidea e le caratteristiche della memoria secondaria: limiteremo la nostra indagine solamente alla topologia euclidea visto che é quella in cui sono definiti i modelli geometrici che costituiscono l'oggetto di studio di questa tesi. Nella seconda parte vengono descritte le proprietá strutturali tipiche dei database e vengono affrontate le tematiche della decomposizione spaziale e della costruzione di un indice in memoria secondaria. Infine, nella terza parte della tesi verranno descritte le proprietá del framework *OMSM*, introdotto in questa tesi per la memorizzazione di oggetti geometrici: inoltre ne verranno presentati i risultati sperimentali ed i possibili sviluppi di questa architettura.

La prima parte della tesi comprende i capitoli 2 e 3, riguardanti rispettivamente alcuni principi di topologia euclidea e le proprietá piú importanti della memoria secondaria. Nel capitolo 2 studieremo le proprietá delle superfici immerse nello spazio metrico euclideo $\mathbb{E}^3$ e presenteremo alcuni concetti chiave della topologia euclidea. Le superfici sono oggetti continui e non sono facilmente gestibili su calcolatore: sono peró approssimabili attraverso i complessi simpliciali euclidei che hanno una natura discreta. É una tecnica che ha trovato grande diffusione nelle applicazioni: la rappresentazione dei terreni ed il metodo degli elementi finiti costituiscono due esempi importanti che verranno trattati in questo capitolo, visto che hanno fornito spunti interessanti per la realizzazione del framework *OMSM*. Nel capitolo 3 verranno

analizzate le proprietá fondamentali della memoria secondaria all'interno di un calcolatore. Nelle applicazioni vi é l'esigenza di memorizzare in maniera permanente le informazioni quindi bisogna integrare nell'architettura di un calcolatore dei dispositivi adeguati allo scopo: la scelta progettuale piú frequente é quella di considerarli generici componenti di I/O. Il componente di memorizzazione piú noto é il *disco fisso*, le cui proprietá verranno analizzate in questo capitolo. Infine verrá proposto un modello efficiente per la gestione delle operazioni di I/O.

La seconda parte della tesi comprende i capitoli 4 e 5, riguardanti rispettivamente le proprietá delle basi di dati e quelle delle strutture di indicizzazione. Nel capitolo 4 verranno analizzate le proprietá fondamentali delle basi di dati, il cui accesso é mediato da un particolare componente, detto *DBMS*. Inoltre verrá introdotto il concetto di *transazione* in una base di dati in modo da permettere a piú utenti di accedere e modificarne il contenuto in maniera concorrente. In questo capitolo ci occuperemo anche dello sviluppo di una tecnica per garantire il ripristino della base di dati in caso di fallimento. Inoltre analizzeremo le proprietá dell'*Oracle Berkeley DB*, il quale é un esempio di *embedded DBMS*. Nel capitolo 5 verranno analizzate le proprietá delle *strutture di indicizzazione*, necessarie per ottimizzare l'esecuzione delle operazioni su grosse quantitá di dati: inoltre verrá introdotto il concetto di *indice spaziale* ed alcune sue proprietá. Verranno inoltre proposti alcuni esempi di indici sia per dati unidimensionali e sia per dati di tipo spaziale con particolare enfasi sulle strutture ausiliarie implementate nel framework *OMSM*, il quale si occupa della gestione di un insieme di oggetti geometrici immmersi nello spazio metrico euclideo $\mathbb{E}^d$, con d generico.

La terza parte della tesi comprende i capitoli 6,7 e 8, riguardanti rispettivamente l'analisi delle caratteristiche tecniche dell'architettura di memorizzazione dei dati geometrici, i risultati sperimentali ottenuti e l'analisi dei possibili sviluppi del framework e della libreria *OMSM*. Nel capitolo 6 verranno descritte le proprietá piú importanti della libreria *OMSM*, la quale comprende tutta la parte implementativa realizzata durante la nostra ricerca. In generale, questa libreria contiene alcuni componenti necessari alla gestione di triangolazioni la cui dimensione eccede la quantitá di memoria primaria normalmente disponibile in un calcolatore. Fra i molti componenti presenti, verrá posta particolare enfasi sulla descrizione della versione preliminare del framework *OMSM*. Nel capitolo 7 verranno presentate le caratteristiche tecniche piú interessanti della libreria *OMSM* e metteremo in luce il suo comportamento in base alle piattaforme di sviluppo supportate attraverso la realizzazione di alcuni programmi dedicati. Inoltre verranno presentati i risultati sperimentali ottenuti utilizzando il framework *OMSM* per l'analisi completa di una serie di triangolazioni immerse nello spazio metrico euclideo $\mathbb{E}^3$, in maniera tale da chiarirne le proprietá piú importanti. Infine, nel capitolo 8 verranno delineate alcune possibili estensioni del framework *OMSM*: data la sua natura modulare, questo sistema puó esteso introducendo nuovi

componenti e variando l'implementazione dei vari livelli che lo compongono. Il framework *OMSM* é in grado di gestire oggetti geometrici di dimensione arbitraria e quindi possiamo memorizzare sia griglie di triangoli immersi nello spazio euclideo $\mathbb{E}^3$ (ma anche in quello $\mathbb{E}^2$) e sia griglie di tetraedri. Pertanto é possibile utilizzare il framework *OMSM* con un qualsiasi algoritmo di semplificazione iterativa in memoria secondaria sia per triangolazioni e sia per tetraedralizzazioni, sfruttandone le funzionalitá: il punto di partenza della nostra analisi sará l'approccio descritto in [CMRS03], generalizzandolo rispetto alle varie tecniche di indicizzazione e rispetto alla tecnica di semplificazione utilizzate. Quindi l'architettura *OMSM* potrá essere utilizzato per costruire un modello multirisoluzione in memoria secondaria, un argomento di grande interesse nella ricerca attuale: un modello multi–risoluzione fornisce una gamma di rappresentazioni di un dato oggetto a differenti livelli di dettaglio, come descritto in [DdFM+06].

Capitolo 2

Nozioni di base

> La matematica pura mi sembra lo scoglio contro cui ogni forma di idealismo inevitabilmente si infrange: 317 é un numero primo, non perché lo pensiamo noi o perché la nostra mente é conformata in un modo piuttosto che in un altro, ma perché é cosí, perché la realtá matematica é fatta cosí.
>
> *Godfrey H. Hardy*

In questo capitolo verranno introdotti i fondamenti teorici di questa tesi. Non si ha alcuna pretesa di completezza: nel seguito della trattazione verranno proposti alcuni approfondimenti. Nel paragrafo 2.1 vengono introdotti alcuni concetti elementari di topologia euclidea con cui poter affrontare lo studio dei complessi simpliciali euclidei nel paragrafo 2.2. Quindi presenteremo una tecnica per la rappresentazione di superfici nel paragrafo 2.3 ed alcuni esempi applicativi nel paragrafo 2.4.

2.1 Elementi di topologia euclidea

L'oggetto di studio di questa tesi é costituito dalle superfici immerse nello spazio metrico euclideo $\mathbb{E}^3$ e quindi possiamo limitarci allo studio della topologia euclidea. Per una trattazione piú generale sulla topologia si rimanda a [Ada90], [BW02] e [Hat02]. Nella sezione 2.1.1 affronteremo la definizione di *spazio metrico euclideo* e studieremo alcune sue proprietá. Nella sezione 2.1.2 vedremo gli *insiemi regolari* e le *varietá*.

2.1.1 Lo spazio metrico euclideo

Come primo passo dobbiamo fissare il concetto di *spazio metrico*.

Definizione 2.1. Una *distanza* su un insieme X é una funzione $d : X^2 \to \mathbb{R}$ che soddisfa i seguenti assiomi per ogni $x, y, z \in X$:

- $d(x, y) \geq 0$

- $d(x, y) = 0 \Leftrightarrow x \equiv y$
- $d(x, y) = d(y, x)$
- $d(x, y) \leq d(x, z) + d(z, y)$

Definizione 2.2. Si dice *spazio metrico* una coppia (X, d), dove X é un insieme e d una qualsiasi funzione *distanza* sull'insieme X.

Come si osserva dalla definizione 2.2, non vi é una preferenza per una particolare distanza: si accetta una qualsiasi funzione che ne soddisfa gli assiomi. Questo fatto porta ad avere un numero *infinito* di spazi metrici: l'uso di una specifica funzione distanza puó essere motivato da particolari esigenze come, ad esempio, la complessitá computazionale del calcolo.

Nel seguito useremo la nozione di *spazio metrico euclideo* che puó essere definito come lo spazio $\mathbb{R}^n$ dotato di una funzione distanza, detta *distanza euclidea*. Il valore di n viene spesso indicato come *dimensione* dello spazio: ovviamente vale $n \geq 1$. É necessario dare la definizione di *punto* in $\mathbb{R}^n$.

Definizione 2.3. Un generico elemento $x \in \mathbb{R}^n$ si dice anche *punto* ed é nella forma $x = (x_1, \ldots, x_n)$ in cui vale $x_i \in \mathbb{R}$ per ogni $i = 1, \ldots, n$.

Come detto, vi sono sono infinite distanze euclidee: un esempio é dato dalla *distanza di Manhattan*.

Definizione 2.4. Siano due punti $x, y \in \mathbb{R}^n$ tali che $x = (x_1, \ldots, x_n)$ e $y = (y_1, \ldots, y_n)$. Allora possiamo definire la *distanza di Manhattan* come la funzione $d_1 : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ tale che:

$$d_1(x, y) = \|x - y\|_1 = \sum_{i=1}^{n} |x_i - y_i|$$

Ma quella piú nota in letteratura é la *distanza pitagorica*.

Definizione 2.5. Siano due punti $x, y \in \mathbb{R}^n$ tali che $x = (x_1, \ldots, x_n)$ e $y = (y_1, \ldots, y_n)$. Allora possiamo definire la *distanza pitagorica* come la funzione $d_2 : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ tale che:

$$d_2(x, y) = \|x - y\|_2 = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

La scelta di una qualsiasi distanza euclidea permette la definizione formale di un generico *spazio metrico euclideo*.

Definizione 2.6. Si definisce *spazio metrico euclideo* la coppia $\mathbb{E}^n = (\mathbb{R}^n, d)$ dove d é una qualsiasi *distanza euclidea*.

Gli esempi piú noti sono quello bidimensionale $\mathbb{E}^2$ e quello tridimensionale $\mathbb{E}^3$, il quale descrive la realtá fisica in cui viviamo.

Per poter studiare le proprietá degli insiemi trattati, dobbiamo introdurre il concetto di *intorno* di un punto in un certo spazio metrico euclideo.

Definizione 2.7. Fissati un insieme $\Gamma \subseteq \mathbb{R}^n$, un punto $x_0 \in \Gamma$ ed un valore $r \in \mathbb{R}^+$ si definisce *intorno* del punto x_0 di raggio r come l'insieme:

$$\delta(x_0, r) = \{x \in \Gamma . d(x, x_0) < r\}$$

Questa definizione serve a fissare il concetto di *punto di accumulazione* di un certo insieme Γ.

Definizione 2.8. Fissati un insieme $\Gamma \subseteq \mathbb{R}^n$ ed un punto $x_0 \in \mathbb{R}^n$ si dice che x_0 é un *punto di accumulazione* per Γ se vale

$$\forall r \in \mathbb{R}^+ . \delta(x_0, r) \setminus \{x_0\} \cap \Gamma \neq \emptyset$$

Questa definizione ci dice che ogni intorno del punto x_0 in questione deve contenere un punto dell'insieme Γ diverso da x_0 e ció descrive intuitivamente la *distribuzione* dei punti di Γ e quindi la sua struttura. In seguito saranno di interesse gli insiemi aperti e quelli chiusi: vediamone le definizioni.

Definizione 2.9. Un insieme $\Gamma \subseteq \mathbb{R}^n$ si dice *aperto* se non contiene i suoi punti di accumulazione.

Definizione 2.10. Un insieme $\Gamma \subseteq \mathbb{R}^n$ si dice *chiuso* se contiene tutti i suoi punti di accumulazione.

Secondo queste definizioni, un insieme é aperto se é possibile spostarsi di poco in ogni direzione a partire da un punto dell'insieme senza uscire dall'insieme stesso: in un insieme chiuso questo non puó succedere. Da queste considerazioni possiamo dedurre il concetto intuitivo di *estensione* di un insieme che si formalizza attraverso le definizioni di *interno*, *chiusura* e *contorno* che hanno un ovvio significato nell'esperienza quotidiana.

Definizione 2.11. Dato un insieme $\Gamma \subseteq \mathbb{R}^n$ si definisce *interno* di Γ il piú grande insieme aperto contenuto in Γ: lo denoteremo con $i(\Gamma)$.

Definizione 2.12. Dato un insieme $\Gamma \subseteq \mathbb{R}^n$ si definisce *chiusura* di Γ il piú piccolo insieme chiuso che contenga Γ: lo denoteremo con $c(\Gamma)$.

Definizione 2.13. Dato un insieme $\Gamma \subseteq \mathbb{R}^n$ si definisce *contorno* di Γ la differenza fra Γ stesso ed il suo interno: lo denoteremo con $b(\Gamma)$. Formalmente vale $b(\Gamma) = \Gamma \setminus i(\Gamma)$.

Alcune proprietá degli insiemi, come la loro dimensione, sono descrivibili attraverso il *disco unitario* ed il *semidisco unitario*.

Definizione 2.14. Si definisce *disco unitario di dimensione* n l'insieme B^n:

$$B^n = \{x \in \mathbb{R}^n . \|x\| \leq 1\}$$

Definizione 2.15. Si definisce *semidisco unitario di dimensione* n l'insieme indicato da B^{n+}:

$$B^{n+} = \{(x_1, \dots, x_n) \in \mathbb{R}^n . \|x\| \leq 1 \wedge x_1 \geq 0\}$$

Queste definizioni permettono di studiare la *dimensione* degli insiemi trattati: intuitivamente essa é il numero di parametri indipendenti necessari alla completa descrizione dell'insieme, quindi é un numero naturale. Per esempio un punto sul piano é descritto dalle sue coordinate cartesiane quindi diremo che la sua dimensione é 2. Ma questa definizione fornisce risultati scorretti con insiemi molto *irregolari* come i frattali. Per trattare correttamente questi oggetti bisogna definire la *dimensione di Hausdorff* che é un valore frazionario: ció esula dagli scopi di questa tesi. Per approfondimenti rifarsi a [Jul22], [Man77] e [DK03]. Nella nostra analisi ci limiteremo allo studio di insiemi che hanno dimensione intera $k \in \mathbb{N}$, noti come *insiemi uniformemente k–dimensionali*. Per prima cosa introduciamo il concetto di *insieme uniformemente k–dimensionale* in un punto.

Definizione 2.16. Dato un insieme chiuso $\Gamma \subseteq \mathbb{R}^n$ ed un punto $x \in \Gamma$ si dice che Γ é *uniformemente k–dimensionale nel punto x*, con $k \leq n$ se qualsiasi intorno del punto x é omeomorfo a $i(B^k)$ oppure a $i(B^{k+})$.

La definizione precedente sfrutta il concetto di *omeomorfismo*: intuitivamente diremo che due oggetti sono *omeomorfi* se esiste una deformazione continua che permette di trasformarli l'uno nell'altro. É un concetto fondamentale in topologia: per una trattazione piú formale di questo concetto rifarsi a [Hat02]. Possiamo quindi estendere la k–dimensionalitá uniforme a tutti i punti di un insieme.

Definizione 2.17. Un insieme Γ si dice *uniformente k–dimensionale* se, per ogni suo punto x, é *uniformemente k–dimensionale* ma non *uniformente* $(k+1)$*–dimensionale* in x.

Nel resto della trattazione parleremo di *insiemi di dimensione k* riferendoci agli insiemi uniformemente k–dimensionali.

2.1.2 Gli insiemi regolari e le varietá

Nel seguito vedremo due esempi di spazi metrici euclidei molto importanti quali gli *insiemi regolari* e le *varietá*.

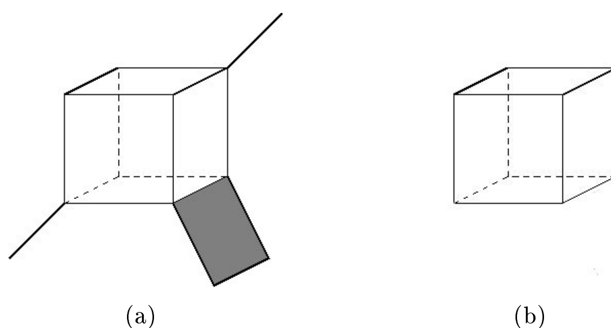
Gli *insiemi regolari* sono stati fra i primi ad essere rappresentati nei sistemi di Computer–Graphics e sono noti anche con il termine inglese *r–set*: per la loro definizione dobbiamo introdurre quella di *regolarizzazione*.

Definizione 2.18. Si definisce *regolarizzazione* di un insieme $\Gamma \subseteq \mathbb{R}^n$ la chiusura dell'interno di Γ: formalmente si indica con $r(\Gamma) = c(i(\Gamma))$.

Questo concetto permette la definizione di *insieme regolare*.

Definizione 2.19. Un insieme $\Gamma \subseteq \mathbb{R}^n$ si dice *insieme regolare* se e soltanto se coincide con la sua regolarizzazione cioé vale $r(\Gamma) = \Gamma$.

Questo significa che l'insieme Γ non deve essere composto da parti di dimensionalitá diverse: ad esempio l'insieme immerso in $\mathbb{E}^3$ della figura 2.1(a) non é un insieme regolare, infatti la sua regolarizzazione non coincide con l'insieme stesso come dimostra la figura 2.1(b).



(a) (b)

Figura 2.1: esempio di insieme non regolare (a) e sua regolarizzazione (b).

La definizione di insieme regolare é piuttosto *forte*: nella realtá quotidiana é difficile che venga completamente soddisfatta. In [QS05] e [QS06] ne viene proposta una versione indebolita.

Un'altra classe interessante di spazi metrici euclidei é quella delle *varietá*, note anche con il termine inglese *manifold*. Si differenziano in *varietá* con o senza contorno: vediamone la definizione formale.

Definizione 2.20. Un insieme $\Gamma \subseteq \mathbb{E}^n$ si dice *k–varietá senza contorno*, con $k \leq n$, se e soltanto se ogni punto $x \in \Gamma$ possiede un intorno omeomorfo al disco aperto $i(B^k)$.

Definizione 2.21. Un insieme $\Gamma \subseteq \mathbb{E}^n$ si dice *k–varietá con contorno*, con $k \leq n$, se e soltanto se ogni punto $x \in \Gamma$ possiede un intorno omeomorfo al disco aperto $i(B^k)$ oppure al semidisco aperto $i(B^{k+})$.

Vediamo in figura 2.2(a) una 2–varietá immersa in $\mathbb{E}^3$ e costituita dal cubo in giallo. In questa figura sono mostrate le intersezioni con tre differenti sfere che rappresentano gli intorni di altrettanti punti: come si puó notare soddisfano la definizione di varietá. Se si ripete mentalmente questo procedimento per ogni suo punto si capisce che il cubo é una 2–varietá. Invece la figura 2.2(b) non rappresenta una varietá: viene mostrato l'intorno di un

punto che viola la definizione di varietá. Per un'analisi dettagliata della geometria e della topologia manifold rifarsi a [Thu97]. É possibile classificare qualsiasi 3–varietá, a meno di omemorfismi, attraverso la congettura sulla *geometrizzazione* di Thurston: come caso particolare si ottiene la *congettura di Poincaré*. Per approfondimenti rifarsi a [Per02], [Per03a] ed a [Per03b].

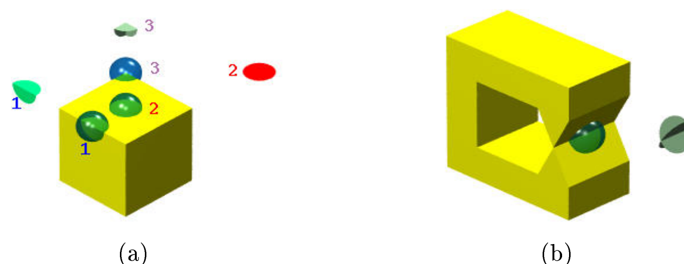


Figura 2.2: sottoinsiemi bidimensionali di $\mathbb{E}^3$ manifold e non manifold: una 2–varietá (a), una non–varietá (b)

2.2 I complessi simpliciali euclidei

In questo paragrafo cercheremo di esporre le caratteristiche piú importanti dei *complessi simpliciali euclidei*, che costituiscono l'oggetto di studio privilegiato di questa tesi. Partiremo dalla nozione di complesso simpliciale astratto e studieremo come immergere tali strutture nello spazio metrico euclideo $\mathbb{E}^3$: queste immersioni sono note come *complessi simpliciali euclidei.*

Nello spazio euclideo un *complesso simpliciale* é un'aggregazione ordinata di un certo numero di poliedri con certe caratteristiche, detti *simplessi*, che si intersecano fra loro solo su facce comuni. Piú in generale, in topologia un complesso simpliciale é un'aggregazione ordinata di insiemi che approssima uno spazio topologico in modo che alcune proprietá siano di facile calcolo. La nozione principale per studiarne le caratteristiche é quella di *complesso simpliciale astratto*: vediamone la definizione.

Definizione 2.22. Un *complesso simpliciale astratto* é una coppia del tipo $K = (X, \Theta)$ data da un insieme X e da un insieme Θ di sottoinsiemi finiti e non vuoti di X che verifica i seguenti assiomi:

- $\forall x \in X.\{x\} \in \Theta$
- $\forall \gamma \in \Theta.\forall \gamma' \subset \gamma, \gamma' \neq \emptyset.\gamma' \in \Theta$

Possiamo definire *simplesso* un generico insieme $\gamma \subset \Theta$. Se $n \geq 0$ definiamo *simplesso di dimensione* n un simplesso γ con $n+1$ elementi, usando la notazione $dim(\gamma) = n$.

Nel resto della trattazione studieremo delle superfici finite quindi sará utile imporre che i complessi simpliciali abbiano un numero finito di elementi.

Definizione 2.23. Il complesso simpliciale K é *finito* se l'insieme X é finito. Se le dimensioni di tutti i simplessi di K hanno un massimo finito diremo che il complesso simpliciale K ha dimensione finita.

L'idea é di immergere un complesso simpliciale astratto nello spazio metrico euclideo $\mathbb{E}^n$: per studiare le immersioni euclidee bisogna partire dal concetto di *simplesso euclideo.*

Definizione 2.24. La combinazione convessa di $k+1$ punti linearmente indipendenti $p_0, \ldots, p_k$ con $p_i \in \mathbb{E}^n$ per ogni $i = 0, \ldots, k$ puó essere espressa nel seguente modo:

$$< p_0, \ldots, p_k >= \{\sum_{i=0}^{k} \lambda_i p_i | \lambda_i \geq 0, \sum_{i=0}^{k} \lambda_i = 1\}$$

Tale combinazione convessa viene detta *k–simplesso euclideo* o *simplesso euclideo di dimensione k.*

Un concetto molto importante é quello di *faccia* di un simplesso.

Definizione 2.25. Sappiamo che un k–simplesso é una combinazione lineare convessa dell'insieme di punti $P_k = \{p_0, \ldots, p_k\}$. Sia $P_j \subset P_k$ un insieme composto da $j+1$ punti di P_k con $j < k$: questo genera un j–simplesso σ'. Si definisce σ' come *faccia propria* di dimensione j del k–simplesso γ.

Il concetto di *faccia* é necessario per la definizione di *adiacenza* e di *incidenza* fra simplessi: vediamo come.

Definizione 2.26. Due k–simplessi γ e γ' sono detti *adiacenti* se:

- esiste un $(k-1)$–simplesso che sia una faccia propria di entrambi i k–simplessi quando $k > 0$;
- γ e γ' sono le facce di un 1–simplesso quando $k = 0$.

Definizione 2.27. Due simplessi γ e γ' sono detti *incidenti* se γ é una faccia propria di γ' oppure γ' é una faccia propria di γ.

Nel seguito della trattazione risulteranno fondamentali i concetti di *link* e di *star* di un simplesso: per poterli introdurre bisogna fornire una relazione d'ordine parziale fra i simplessi.

Definizione 2.28. Dato un complesso simpliciale Γ e due suoi simplessi γ_1 e γ_2 possiamo definire una relazione d'ordine parziale $<_b$ fra γ_1 e γ_2 tale che:

$$\gamma_1 <_b \gamma_2 \Leftrightarrow dim(\gamma_1) < dim(\gamma_2) \wedge \gamma_1 \neq \gamma_2 \wedge \gamma_1 \in b(\gamma_2)$$

In letteratura questa relazione d'ordine $<_b$ viene chiamata *relazione di contorno* e viene usata per denotare tutti i simplessi σ in Γ per i quali vale $\sigma <_b \gamma$ cioé gli elementi dell'insieme $B(\gamma) = \{ \sigma \in \Gamma . \sigma <_b \gamma \}$. Questo insieme é detto *contorno combinatorico*, in contrapposizione a quello insiemistico della definizione 2.13. Grazie a questa relazione possiamo fissare i concetti di *star* e di *link* di un simplesso γ in un complesso simpliciale Γ.

Definizione 2.29. Sia un complesso simpliciale Γ e sia γ un suo simplesso. Allora si definisce la *star* del simplesso γ come $\gamma^* = \{ \gamma' \in \Gamma . \gamma <_b \gamma' \}$.

La figura 2.3 ne rappresenta un esempio: il punto in giallo é il simplesso di riferimento γ, mentre la sua *star* é costituita dagli spigoli e dai triangoli evidenziati in verde che contengono γ nel loro contorno.

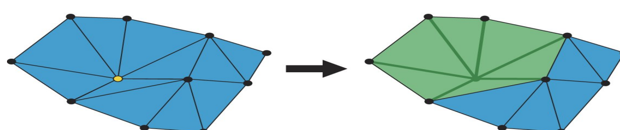


Figura 2.3: esempio di *star* di un simplesso

Definizione 2.30. Sia un complesso simpliciale Γ e sia γ un suo simplesso. Allora si definisce il *link* del simplesso γ come la differenza fra l'unione dei contorni combinatorici dei simplessi appartenenti a γ^*, il simplesso γ ed i simplessi appartenenti a γ^*. In letteratura questo insieme viene solitamente indicato con $Lnk(\gamma)$.

La figura 2.4 ne rappresenta un esempio: il punto in giallo é il simplesso di riferimento γ, mentre i simplessi in verde ne costituiscono il *link*.

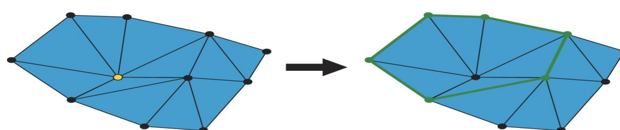


Figura 2.4: esempio di *link* di un simplesso

Possiamo fissare la definizione di *complesso simpliciale euclideo*.

Definizione 2.31. Un *complesso simpliciale euclideo* Γ é un insieme finito di simplessi euclidei che soddisfa le seguenti condizioni:

- se γ é un simplesso di Γ e γ' una delle sue facce, allora anche $\gamma' \in \Gamma$
- siano γ_1 e γ_2 due simplessi euclidei distinti appartenenti a Γ, la loro intersezione o é vuota o é una faccia propria di entrambi.

L'insieme di tutti i k–simplessi del complesso simpliciale Γ si indica con $\Gamma^{(k)}$.

Bisogna osservare che non tutte le immersioni di complessi simpliciali verificano queste proprietá: nella figura 2.5 vediamo un'immersione di un 3–complesso simpliciale nello spazio $\mathbb{E}^3$ che non soddisfa le proprietá di un 3–complesso simpliciale euclideo.

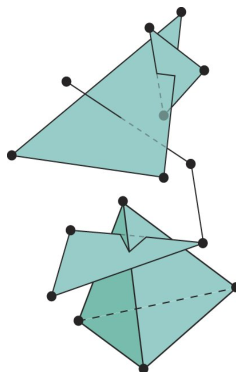


Figura 2.5: esempio di immersione in $\mathbb{E}^3$ di un 3–complesso simpliciale che non soddisfa le proprietá di 3–complesso simpliciale euclideo

I *complessi simpliciali regolari* forniscono un sottoinsieme dei complessi simpliciali euclidei particolarmente interessante: per poterli definire occore la definizione di *simplesso top*.

Definizione 2.32. Dato un complesso simpliciale euclideo Γ si definisce *simplesso top* un simplesso $\gamma \in \Gamma$ se non esiste un simplesso $\gamma' \in \Gamma$ tale che γ sia una faccia propria di γ'.

Questo ci permette di dare la definizione di *complesso simpliciale regolare.*

Definizione 2.33. Si dice che Γ é un *k–complesso simpliciale regolare* se tutti i suoi simplessi top sono di dimensione k.

I complessi simpliciali regolari piú usati nelle applicazioni sono quelli di ordine 2 e 3, immersi nello spazio euclideo $\mathbb{E}^3$: si indicano rispettivamente con i termini *triangolazione* e *tetraedralizzazione.*

Definizione 2.34. Un complesso simpliciale regolare di ordine 2 immerso in $\mathbb{E}^3$ viene detto *triangolazione* o *griglia di triangoli.*

La figura 2.6 mostra un esempio di triangolazione: si tratta della visualizzazione di una griglia di triangoli usando il programma *MeshLab*, che viene sviluppato presso il *CNR Visual Computing Lab* di Pisa e distribuito con licenza *GPL*: é stato usato in questa tesi per la visualizzazione dei vari complessi simpliciali euclidei. Per approfondimenti rifarsi a [Msh05].

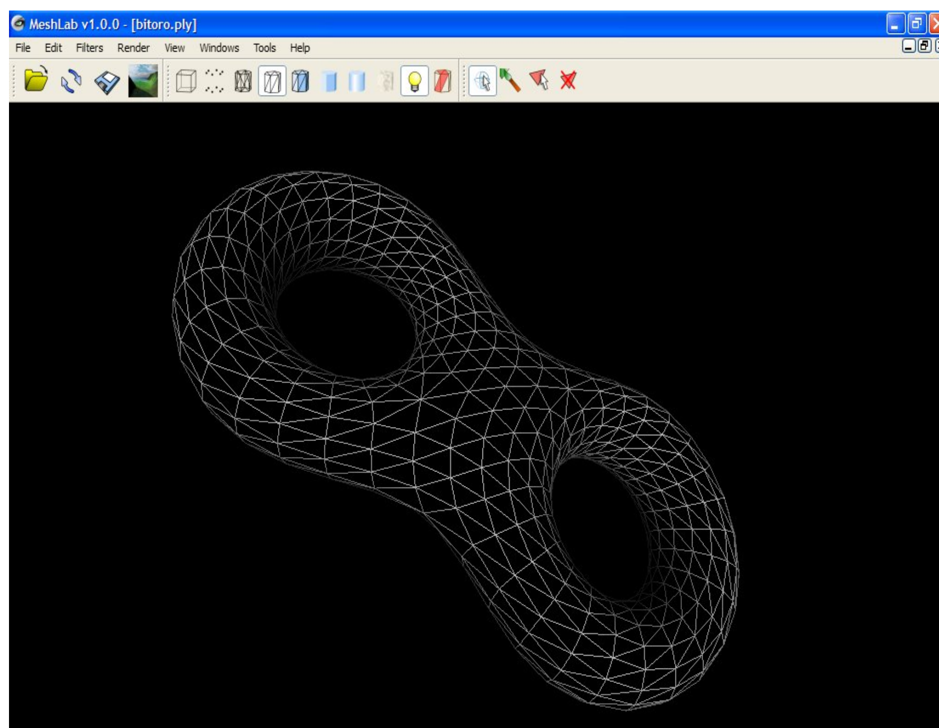


Figura 2.6: visualizzazione di una triangolazione usando *MeshLab*

Definizione 2.35. Un complesso simpliciale regolare di ordine 3 immerso in $\mathbb{E}^3$ viene detto *tetraedralizzazione* o *griglia di tetraedri*.

Il tetraedro in figura 2.7 é un esempio banale di tetraedralizzazione.

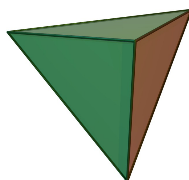


Figura 2.7: un esempio banale di tetraedralizzazione

Un aspetto importante é l'*orientamento* di un complesso simpliciale euclideo: nel resto della trattazione vedremo solamente il caso bidimensionale, gli altri sono analoghi. Come primo passo, bisogna definire gli *orientamenti* di un 1–simplesso e di un 2–simplesso euclidei.

Definizione 2.36. Dato un 1–simplesso euclideo γ generato dall'insieme di punti $P_1 = \{p_0, p_1\}$ si definisce *orientamento* di γ uno dei due possibili ordinamenti dei punti dati da $< p_0, p_1 >$ e da $< p_1, p_0 >$.

Definizione 2.37. Dato un 2–simplesso euclideo γ generato dall'insieme di punti $P_2 = \{p_0, p_1, p_2\}$, si definisce *orientamento* di γ uno dei due possibili ordinamenti dei punti dati da $< p_0, p_1, p_2 >$ e da $< p_2, p_1, p_0 >$, a meno di permutazioni cicliche. L'orientamento scelto induce banalmente anche quello dei tre 1–simplessi di γ.

Queste definizioni ci permettono di formalizzare l'*orientamento coerente* dei 2–simplessi della triangolazione.

Definizione 2.38. Dato un 2–complesso simpliciale Γ che contiene i 2–simplessi γ_1 e γ_2 la cui intersezione é un 1–simplesso γ', si dice che γ_1 e γ_2 sono *orientati coerentemente* se l'orientamento indotto da γ_1 su γ' é opposto a quello indotto da γ_2.

Questo ci permette di dire che cosa si intende per *triangolazione orientabile*.

Definizione 2.39. Una triangolazione si dice *orientabile* se é possibile orientare coerentemente tutti i suoi 2–simplessi.

Nello spazio euclideo $\mathbb{E}^2$ gli orientamenti dei simplessi prendono il nome di *orientamento antiorario* o *orientamento orario*, a seconda del punto di vista dell'osservatore: sono quindi legati al concetto di *visibilitá*. Negli spazi a dimensione maggiore, questa caratteristica viene a mancare visto che non esiste una direzione di vista privilegiata: é possibile, peró, fissarne una in qualche modo e ricondursi alle nozioni precedenti.

2.3 La modellazione di superfici

In questa tesi studieremo come poter rappresentare delle superfici su un calcolatore: nella sezione 2.3.1 verranno introdotti il concetto di *superficie* e quello di *approssimazione simpliciale di una superficie*. Presenteremo poi le caratteristiche fondamentali di un sistema per la loro rappresentazione nella sezione 2.3.2.

2.3.1 Le approssimazioni simpliciali

Il concetto di superficie é noto dall'esperienza quotidiana considerando ad esempio il contorno degli oggetti concreti: in queste ipotesi l'oggetto concreto viene detto *solido* mentre il suo contorno *superficie*.

Definizione 2.40. Si dice *superficie* una k–varietá immersa in $\mathbb{E}^d$ con $k < d$.

Nel resto della tesi chiameremo *superficie* una 2–varietá immersa in $\mathbb{E}^3$.

Definizione 2.41. Si dice *solido* una d–varietá immersa in $\mathbb{E}^d$.

Nel resto della tesi, chiameremo *solido* una 3–varietá immersa in $\mathbb{E}^3$. In letteratura vengono chiamati anche con il termine *politopi*: per una loro classificazione rifarsi a [Joh66], [Zal69] e [Har96]. Nel resto della tesi parleremo principalmente di superfici per poi generalizzare i risultati ottenuti ai solidi, dove possibile.

Molte caratteristiche delle k–varietá possono essere studiate attraverso le loro triangolazioni in maniera piú *semplice*, quindi puó essere utile approssimare una varietá con un complesso simpliciale euclideo. Una k–varietá $\mathcal{O}$ puó essere rappresentata con un complesso simpliciale Γ la cui immersione euclidea $|\Gamma|$ sia a sua volta una k–varietá: ció fornisce una descrizione approssimata della *geometria* di $\mathcal{O}$. Di solito si richiede anche che $|\Gamma|$ sia omeomorfo a $\mathcal{O}$ in modo da preservarne la *topologia*: questo procedimento sfrutta la teoria delle categorie e conduce alla definizione dell'*approssimazione simpliciale* come descritto in [Ada90] e [Hat02]. Questo tipo di trattazione esula dagli scopi di questa tesi: possiamo peró fornirne una descrizione piú *intuitiva*.

Per quanto ci riguarda, un complesso simpliciale euclideo Γ puó essere usato per approssimare una k–varietá $\mathcal{O}$: l'unione dei k–simplessi di Γ fornisce un'approssimazione di $\mathcal{O}$. Ogni k–simplesso γ approssima una porzione della varietá $\mathcal{O}$ che indicheremo con $\mathcal{O}_\gamma$: per misurare la *distanza* di γ da $\mathcal{O}_\gamma$ introduciamo un *errore di approssimazione*. Questo errore puó essere definito da un qualche criterio, nel seguito useremo quello indotto dalla *distanza di Hausdorff*.

La distanza di Hausdorff estende la distanza euclidea introdotta nella sezione 2.1.1 in quanto é definita fra due insiemi $P, Q \in \mathbb{E}^n$. Il primo passo é la definizione della *distanza unilaterale di Hausdorff*.

Definizione 2.42. Si definisce *distanza unilaterale di Hausdorff* di un insieme $P \subseteq \mathbb{E}^n$ da un insieme $Q \subseteq \mathbb{E}^n$ il valore

$$d_{\frac{H}{2}} = sup \ \{ \ inf \ \{ \ d(p,q).q \in Q \ \} \ \}$$

dove $p \in P$ e d é una qualsiasi distanza euclidea.

In generale non é simmetrica quindi non dovrebbe essere chiamata *distanza*: costituisce peró la base per la definizione della *distanza di Hausdorff*.

Definizione 2.43. Si definisce *distanza di Hausdorff* tra l'insieme $P \subseteq \mathbb{E}^d$ e l'insieme $Q \subseteq \mathbb{E}^d$ il valore dato da:

$$d_H(P,Q) = max \ \{d_{\frac{H}{2}}(P,Q) \ , \ d_{\frac{H}{2}}(Q,P)\}$$

La distanza di Hausdorff ci permette la definizione dell'*errore di approssimazione* di un singolo simplesso: vediamo come.

Definizione 2.44. Siano $\mathcal{O}$ una k–varietá, Γ un complesso simpliciale che approssima $\mathcal{O}$ e γ un k–simplesso di Γ allora definiamo $d_H(\gamma, \mathcal{O}_\gamma)$ come l'*errore di approssimazione* del simplesso γ.

Possiamo generalizzare questo risultato definendo l'*errore di approssimazione* del complesso Γ rispetto alla varietá $\mathcal{O}$: vediamo come.

Definizione 2.45. Possiamo definire l'*errore di approssimazione* di un complesso Γ rispetto alla varietá $\mathcal{O}$ come la funzione $\varepsilon : \Gamma^{(k)} \to \mathbb{R}^+$ dove $\varepsilon(\gamma)$ é l'errore di approssimazione del simplesso γ.

Questa definizione é legata all'*accuratezza* della rappresentazione, che intuitivamente decresce all'aumentare dell'errore e viceversa. In letteratura non é definita in modo univoco, per noi sará definita come in [Mag99], cioé come l'inverso dell'errore di approssimazione: vediamone la definizione formale.

Definizione 2.46. Possiamo definire l'*accuratezza* di un k–complesso simpliciale Γ rispetto alla varietá $\mathcal{O}$ come la funzione $\alpha : \Gamma^{(k)} \to \mathbb{R}^+$ dove

$$\alpha(\gamma) = \frac{1}{\varepsilon(\gamma)}$$

per ogni k–simplesso γ in Γ.

Alla luce dell'approssimazione simpliciale possiamo analizzare il concetto di orientabilitá di una superficie: é possibile ridursi all'orientamento della triangolazione attraverso il seguente lemma, enunciato in [Hof89].

Lemma 2.1. *Se una* 2*–varietá* Γ *é orientabile, allora ogni triangolazione di* Γ *lo é. Inoltre, se una triangolazione di una* 2*–varietá* Γ *é orientabile, allora lo é anche* Γ*.*

Il *nastro di Möbius* e la *bottiglia di Klein* sono esempi di superfici non orientabili: per approfondimenti rifarsi a [Gar84], [Lop93], [Hil94] e [Pol03].

2.3.2 Elementi di modellazione

In questa sezione descriveremo una tecnica per rappresentare le superfici immerse nello spazio $\mathbb{E}^3$ in un calcolatore. Questa tecnica puó essere definita attraverso un processo di modellizzazione operante su tre livelli:

- lo *spazio reale* contiene gli oggetti fisici;
- lo *spazio di modellazione* contiene gli oggetti matematici che idealizzano gli oggetti fisici: di solito si indica con il simbolo $\mathcal{M}$;
- lo *spazio delle rappresentazioni* definisce quali informazioni memorizzare per descrivere un modello all'interno del calcolatore: di solito si indica con il simbolo $\mathcal{R}$.

Il compito della modellizzazione consiste nell'assegnare ad un oggetto matematico una rappresentazione adatta per essere visualizzata e manipolata su un elaboratore: vedremo alcune definizioni che descrivono questo processo.

Definizione 2.47. Una rappresentazione $r \in \mathcal{R}$ é una collezione finita di simboli appartenenti ad un alfabeto finito che codifica un modello $m \in \mathcal{M}$.

Definizione 2.48. Uno schema di rappresentazione é un'applicazione del tipo $\sigma : \mathcal{M} \rightarrow \mathcal{R}$ che associa un modello $m \in \mathcal{M}$ con una rappresentazione $r \in \mathcal{R}$

Definizione 2.49. Indichiamo con $\mathcal{D}$ e con $\mathcal{V}$ rispettivamente il dominio ed il codominio di uno schema di rappresentazione σ. Allora possiamo dire che una rappresentazione r é *valida* se e soltanto se $r \in \mathcal{V}$. Pertanto, ogni rappresentazione valida é l'immagine di almeno un modello.

Per fissare un sistema di rappresentazione dobbiamo dire cosa contengono lo spazio reale, lo spazio $\mathcal{M}$ e lo spazio $\mathcal{R}$. In questa tesi abbiamo scelto di modellare gli oggetti del mondo reale con delle 2–varietá immerse in $\mathbb{E}^3$ che, come si puó immaginare, non sono facilmente trattabili su un calcolatore in quanto oggetti continui. Nella sezione 2.3.1 abbiamo visto come sia possibile approssimare un k–manifold attraverso i complessi simpliciali. Quest'ultimi hanno una natura discreta e sono piú facilmente gestibili in un calcolatore, quindi li sceglieremo come i nostri modelli e di conseguenza come elementi dello spazio $\mathcal{M}$. Pertanto lo spazio $\mathcal{R}$ contiene le rappresentazioni dei complessi simpliciali euclidei, cioé le strutture dati che verranno usate per memorizzarne le caratteristiche. Ora il problema diventa fissare il contenuto delle strutture dati. Limitiamoci al solo caso bidimensionale: una triangolazione é caratterizzata dalla sua immersione nello spazio metrico euclideo (la sua *geometria*) e dalla struttura di connessione dei simplessi che lo compongono (la sua *topologia*). Nel nostro caso esistono tre tipi di simplessi, quelli di ordine 0,1 e 2. In letteratura gli 0–simplessi prendono il nome di *vertici*, gli 1–simplessi quello di *spigoli* e i 2–simplessi quello di *triangoli*: si utilizza anche il termine *entitá* per indicare l'insieme di tutti i simplessi di una triangolazione. Nel resto della trattazione indicheremo con n il numero totale dei vertici, con e quello degli spigoli e con t quello dei triangoli: in [Ede87] si dimostra che $e \sim 3n$ e $t \sim 2n$. La topologia di una triangolazione viene espressa tramite due tipi di relazioni topologiche fra le entitá: quelle di *adiacenza* e quelle di *incidenza*. A loro volta le relazioni di incidenza si dividono in relazioni di *incidenza boundary* e quelle di *incidenza coboundary*. Nel paragrafo 2.2 abbiamo giá visto le definizioni di *simplessi adiacenti* e quella generale di *simplessi incidenti*: ora vediamo quelle relative ai tipi di incidenza.

Definizione 2.50. Un k–simplesso γ_1 é in relazione di *incidenza boundary* con un j–simplesso γ_2 se $j < k$ e γ_2 é una faccia propria di γ_1.

Definizione 2.51. Un k–simplesso γ_1 é in relazione di *incidenza coboundary* con un j–simplesso γ_2 se $k < j$ e γ_1 é una faccia propria di γ_2.

Vedremo nel seguito le relazioni di adiacenza in una triangolazione:

- la relazione *Vertice–Vertice* associa ad un vertice tutti i vertici ad esso adiacenti attraverso uno spigolo: convenzionalmente i vertici sono ordinati in un qualche ordine, ad esempio quello antiorario;
- la relazione *Spigolo–Spigolo* mette in relazione due spigoli che condividono un vertice: il numero di spigoli adiacenti ad un dato spigolo é generalmente arbitrario, ma comunque limitato da e. Nelle strutture dati se ne memorizza un sottoinsieme sufficiente per ricostruire la relazione completa.
- la relazione *Triangolo–Triangolo* intercorre fra due triangoli adiacenti: ogni triangolo é adiacente con al piú tre triangoli, ordinati in un qualche ordine, ad esempio quello antiorario.

Vedremo nel seguito le relazioni di incidenza di tipo *boundary*:

- la relazione *Spigolo–Vertice* associa ad uno spigolo i suoi due vertici estremi in un qualche ordine, che ne determina l'orientamento;
- la relazione *Triangolo–Vertice* associa ad un triangolo i suoi tre vertici estremi: convenzionalmente i vertici sono ordinati in maniera coerente con i triangoli in relazione *Triangolo–Triangolo*;
- la relazione *Triangolo–Spigolo* associa ad un triangolo i suoi tre spigoli: convenzionalmente gli spigoli sono ordinati in modo coerente con i vertici in relazione *Triangolo–Vertice* e quindi con i triangoli in relazione *Triangolo–Triangolo*.

Vedremo nel seguito le relazioni di incidenza di tipo *coboundary*:

- la relazione *Vertice–Spigolo* associa ad un vertice tutti gli spigoli in esso incidenti: convenzionalmente gli spigoli sono ordinati in modo coerente con i vertici in relazione *Vertice–Vertice*;
- la relazione *Vertice–Triangolo* associa ad ogni vertice i triangoli in esso incidenti: convenzionalmente i triangoli sono ordinati in modo coerente con i vertici in relazione *Vertice–Vertice* e quindi con gli spigoli in relazione *Vertice–Spigolo*;
- la relazione *Spigolo–Triangolo* associa ad uno spigolo i triangoli che lo contengono nel loro contorno. Ogni spigolo é in relazione con al piú due triangoli elencati secondo l'orientamento dello spigolo, definito dalla relazione *Spigolo–Vertice*.

Le relazioni presentate sono valide se il dominio da approssimare é una varietá. Nel caso non–manifold vanno estese: per approfondimenti rifarsi a [dFMPS04] e [dFH05b]. In generale, un complesso simpliciale é completamente definito dai suoi simplessi top: tutte le strutture dati devono memorizzare le informazioni geometriche relative a tali simplessi e le arricchiscono

con quelle relative alle relazioni di adiacenza. In letteratura ne sono state sviluppate molte: per una panoramica generale rifarsi a [PS85], [Man88], [Req96] e [dFH05a]. Si puó estendere questa tecnica alle griglie di tetraedri: per approfondimenti rifarsi a [dFH03], [dFGH04], [LLLV05] e [HVdF06].

2.4 Esempi applicativi

Nel paragrafo 2.3 abbiamo visto come definire ed usare una triangolazione per manipolare superfici. Nella sezione 2.4.1 vedremo l'uso della decomposizione simpliciale per la gestione di terreni, mentre nella sezione 2.4.2 ne vedremo l'uso nel campo ingegneristico.

2.4.1 La rappresentazione dei terreni

In questa sezione vedremo come utilizzare l'approssimazione simpliciale nella modellazione di un terreno. Un modello digitale tridimensionale di un terreno é piú comprensibile di una cartina topografica e diventa fruibile per la visualizzazione e per le operazioni di interrogazione in quanto viene integrato nelle banche dati geografiche, dette *GIS* (dall'inglese *Geographic Information Systems*). Nelle applicazioni vogliamo modellare una porzione di terreno usando un *Modello Digitale di un Terreno* (spesso abbreviato in *DTM*, dall'inglese *Digital Terrain Model*), il quale é descritto da un campionamento finito di punti nello spazio. Puó essere modellato con una delle seguenti tecniche:

- la *rete a maglie triangolari irregolari*, la quale é solitamente abbreviata in *TIN* (dall'inglese *Triangular Irregular Networks*);

- la *gliglia a maglie regolari quadrate*, la quale é solitamente abbreviata in *RSG* (dall'inglese *Regular Square Grid*).

Entrambe si basano sulla decomposizione simpliciale e sfruttano il concetto di *campo scalare*.

Definizione 2.52. Una funzione $\varphi : \Omega \subseteq \mathbb{E}^d \to \mathbb{E}$ é detta *campo scalare*.

I terreni vengono approssimati da un campo scalare facendo corrispondere una quota h_P ad ogni punto P del dominio Ω bidimensionale e questo viene fatto in maniera automatica fornendo il *DTM* di partenza. Entrambe le tecniche si basano sulla creazione (in qualche modo) di una decomposizione simpliciale del dominio Ω e sull'associazione di una certa quota ad ogni 2–simplesso del dominio, ottenendo di conseguenza un 2–complesso simpliciale immerso nello spazio $\mathbb{E}^3$.

Nella tecnica *RSG* l'insieme dei punti originali viene trasformato in una griglia regolare a maglie quadrate, impiegando tecniche di interpolazione.

Nel DTM risultante ogni nuovo punto interpolato appartiene ad un quadrato della griglia: per ottenere una decomposizione simpliciale del dominio si triangola ogni maglia quadrata attraverso la sua diagonale. Questa tecnica non é adattiva: con terreni piuttosto *irregolari*, come quello in figura 2.8, tende a semplificare le superfici montuose ed a rappresentare quelle pianeggianti con *troppi* punti. Fornisce risultati migliori per terreni abbastanza *regolari* senza grosse discontinuitá o brusche variazioni.

Con la tecnica *TIN* é possibile produrre modelli costituiti da un insieme di punti quotati collegati da segmenti che formano una rete continua di triangoli. La scelta dei punti significativi puó basarsi su vari metodi ed il loro collegamento deve assicurare la continuitá del terreno: la superficie di ogni triangolo é definita attraverso l'elevazione dei suoi tre vertici. In figura 2.8 ne vediamo un esempio. L'uso del modello *TIN* consente di rappresentare il terreno con meno punti rispetto a quello RSG: la densitá dei punti puó essere adattata al livello di complessitá locale della superficie. Le *TIN* si prestano meglio a rappresentare aree ove le pendenze variano bruscamente in quanto gli spigoli possono allinearsi esattamente con le linee che segnano la discontinuitá della pendenza: dunque é vantaggioso utilizzarle in quanto la loro accuratezza si adatta alle caratteristiche del terreno.

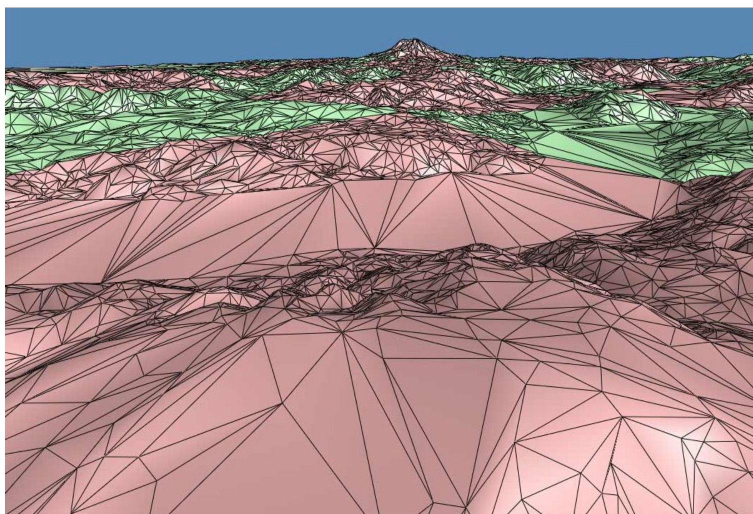


Figura 2.8: modello di terreno attraverso una *TIN*

Sono stati sviluppati molti sistemi per la gestione di terreni: per approfondimenti rifarsi a [DWS+97], [Dis97], [dFMP00] e [LP02]. Per approfondimenti sui sistemi *GIS* rifarsi a [dFPM99]. Le tecniche fin qui presentate non sono le uniche: ad esempio é possibile usare un frattale per modellare un terreno. La difficoltá maggiore di questo metodo sta nel trovare il frattale adatto allo scopo: inoltre soffre di una serie di problematiche descritte in [Lew88] e

non é adatto per l'esecuzione di interrogazioni. Per approfondimenti rifarsi a [Vos88], [MM91] e [Mus93].

2.4.2 Il metodo degli elementi finiti

Un altro importante esempio applicativo é il *metodo degli elementi finiti*, che indicheremo con *FEM* nel seguito, dall'inglese *Finite Elements Method*. Costituisce un metodo numerico assai versatile per la soluzione approssimata di equazioni differenziali che compaiono in molti problemi di ingegneria come ad esempio nella risposta strutturale di un certo oggetto meccanico. La caratteristica principale di questo metodo consiste nella discretizzazione del dominio continuo di partenza ottenendone uno discreto, detto *mesh*, mediante l'uso di componenti, detti *elementi finiti*, di dimensione finita ma non infinitesima. Su ciascun elemento la soluzione del problema puó essere espressa come combinazione lineare di funzioni dette *funzioni di base*. Rimanendo nell'ambito dell'ingegneria strutturale la suddivisione dei solidi continui in un numero finito di elementi consente di ricondurre il calcolo della risposta strutturale alla soluzione numerica di un problema avente un numero finito di gradi di libertá: ció permette l'analisi di problematiche alquanto generali ed eventualmente caratterizzate da geometrie molto complesse, per le quali la determinazione analitica della risposta tenso–deformativa risulterebbe molto complicata. Per approfondimenti sul *FEM* rifarsi a [Naf83] e [TC01]. Per gli scopi di questa tesi, non ci interessa entrare nel merito di questo tipo di analisi, ma ci preme mettere in luce l'analogia con la decomposizione simpliciale di superfici, peraltro molto evidente. Inoltre, molte tecniche sviluppate per il *FEM* possono essere modificate ed adattate alla rappresentazione di superfici: tra queste ricordiamo quelle sviluppate nelle librerie *ETREE* e *Zoltan* le quali memorizzano le mesh come se fossero complessi simpliciali.

La libreria *ETREE*, sviluppata alla *Carnegie Mellon University* nell'ambito del *Progetto Euclid*, permette la memorizzazione degli elementi finiti in memoria secondaria attraverso la struttura dati *Octree* della quale parleremo nel paragrafo 5.4. Per approfondimenti rifarsi a [TOL02] e [TOL03].

La libreria *Zoltan*, sviluppata nei *Sanda National Laboratories*, permette l'elaborazione degli elementi finiti in un ambiente distribuito o multiprocessore. Per approfondimenti rifarsi a [BDF$^+$99], [DBH$^+$02], [BDF$^+$06a], [BDF$^+$06b], [DBH$^+$06] e [CBD$^+$07].

Alcune delle idee sviluppate in queste librerie hanno fornito importanti spunti per la realizzazione dell'architettura di memorizzazione di dati spaziali all'interno della libreria *OMSM*: ne parleremo nel capitolo 6.

Capitolo 3

La memoria secondaria

La velocitá é importante, ma la precisione é tutto.
Wyatt Earp

A partire dalla *macchina analitica*, progettata da Charles Babbage, vi é stata l'esigenza di archiviare i dati in maniera permanente per poterli poi riutilizzare e consultare in successive computazioni: per approfondimenti rifarsi a [Men42] e [Bab88]. L'esigenza di memorizzare i dati ha portato allo sviluppo dei supporti di memorizzazione: in questo capitolo ne vedremo una panoramica generale. Non si ha alcuna pretesa di completezza al riguardo: nel seguito della trattazione verranno proposti alcuni approfondimenti. Nel paragrafo 3.1 richiameremo il modello della macchina di von Neumann per capire la funzione dei supporti di memorizzazione all'interno di un moderno calcolatore. In questa tesi tratteremo modelli geometrici che eccedono la quantitá di memoria primaria disponibile in un calcolatore, la quale verrá studiata nel paragrafo 3.2. Il supporto piú economico e adatto alla memorizzazione dei dati é il *disco fisso*, il quale verrá analizzato nel paragrafo 3.3. Infine vedremo nel paragrafo 3.4 alcuni principi per organizzare in maniera efficiente la memorizzazione dei dati sul disco fisso.

3.1 La macchina di von Neumann

Con l'espressione *macchina di von Neumann* (o *architettura di von Neumann*) ci si riferisce ad uno schema di progettazione di calcolatori elettronici definito dal matematico ungherese John von Neumann: per approfondimenti rifarsi a [vN45]. Questo schema si basa su quattro componenti fondamentali:

- l'*unitá di memoria*, la quale memorizza i dati ed i programmi da eseguire: si indica con *RAM*, dall'inglese *Random Access Memory*;
- l'*unitá di controllo*, la quale esegue i programmi contenuti nell'unitá di memoria: si indica con *CPU*, dall'inglese *Central Processing Unit*;

- l'*unitá di input*, tramite la quale i dati vengono inseriti nel calcolatore per essere elaborati;
- l'*unitá di output*, la quale restituisce all'esterno i dati elaborati.

La *CPU* é composta da molte sottocomponenti: una delle piú rilevanti é l'*unitá aritmetico–logica* (spesso indicata con *ALU*, dall'inglese *Arithmetic–Logic Unit*) che puó eseguire alcune istruzioni di tipo logico/matematico. All'interno dell'*ALU* é presente un registro detto *accumulatore*, che fa da tramite fra l'unitá di input e quella di output grazie a speciali istruzioni che caricano un dato dalla memoria all'accumulatore e viceversa. La figura 3.1 mostra lo schema di questa architettura.

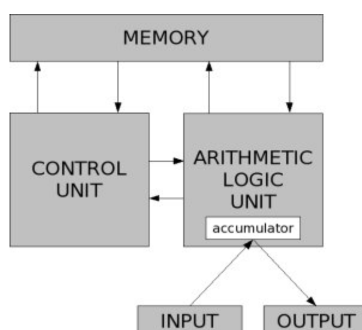


Figura 3.1: lo schema della macchina di von Neumann

Dunque, il funzionamento di un elaboratore é (teoricamente) piuttosto *semplice*: il dato in input viene caricato nella *ALU*, la quale invia i risultati all'unitá di output alla fine del calcolo richiesto. Non ci interessa formalizzarne il funzionamento in maniera precisa: si rimanda alla vastissima letteratura sull'argomento, ad esempio [Tan91]. Sebbene sia un modello molto semplice, la macchina di von Neumann descrive il funzionamento della maggior parte dei moderni elaboratori. Come si puó notare, questo modello non prevede i supporti di memorizzazione come entitá distinte dalle altre. Possiamo osservare che i dati devono essere caricati nella *RAM* per poter essere elaborati dalla *CPU* a prescindere dalla loro provenienza quindi possiamo considerare i *supporti di memorizzazione* come dispositivi di I/O. Questi ultimi si dividono in due categorie: quelli a *blocchi* e quelli a *carattere.*

Definizione 3.1. Un dispositivo si dice *a blocchi* se scompone i dati in parti di lunghezza fissa, ognuna delle quali é direttamente indirizzabile.

Definizione 3.2. Un dispositivo si dice *a caratteri* se gestisce un flusso di caratteri senza supportare alcun tipo di indirizzamento diretto degli stessi.

Usare questi componenti per modellare i supporti di memorizzazione é una scelta che rende la gestione delle operazioni indipendente dal dispositivo: si

assume che quest'ultimo sia a blocchi e si lascia al software di basso livello la parte dipendente dall'hardware. Ogni unitá di I/O é costituita da una componente meccanica e da una elettronica in modo da rendere la sua gestione il piú modulare possibile. La componente elettronica é chiamata *controllore del dispositivo* (in inglese *controller*) mentre quella meccanica é il dispositivo stesso. In questo modo il sistema operativo comunica solamente con il controllore operando a livello di blocco (in lettura o in scrittura). Sará il controllore stesso ad operare correttamente sul dispositivo, in quanto é l'unico che ne conosce la vera natura. Per approfondimenti sulle varie architetture di costruzione del sistema di I/O rifarsi a [Tan91] ed a [Tan92].

3.2 L'organizzazione della memoria

Come abbiamo visto nel paragrafo 3.1, la *memoria* é quella parte del calcolatore in cui sono immagazzinati i dati. É costituita da un certo numero di *celle* (dette anche *locazioni*), ognuna delle quali puó immagazzinare una quantitá fissata di informazione. Ogni locazione é identificata dal suo *indirizzo*: se ve ne sono n, esse avranno gli indirizzi compresi fra 0 e $n-1$. Si possono indicare gli indirizzi delle celle di memoria con numeri binari: se un indirizzo ha m bit, il numero massimo di locazioni direttamente indirizzabili é dato da 2^m ed é indipendente dalla dimensione della cella. Tutte le celle contengono lo stesso numero di bit: se una locazione é costituita da k bit allora potrá contenere una delle 2^k diverse combinazioni di bit. Si utilizza il termine *parola* come sinonimo di cella di memoria e di solito molte istruzioni operano su parole intere. Con queste considerazioni un calcolatore con parole da 64 bit contiene 8 byte per parola, registri da 64 bit ed istruzioni per manipolare parole a 64 bit. I byte di una parola possono essere numerati da sinistra verso destra o viceversa: questa scelta potrebbe sembrare irrilevante, ma ha delle implicazioni importanti. Storicamente nei calcolatori della famiglia *Motorola* i byte sono numerati da sinistra verso destra, nella famiglia *Intel* é avvenuto il contrario. Nel primo caso la numerazione comincia dalla cifra di maggior peso e si parla di calcolatore a *finale grande* (in inglese *big–endian*) in contrapposizione al calcolatore a *finale piccolo* (in inglese *little–endian*). L'uso di questi termini, in informatica, é stato introdotto in [Coh81]. Entrambe le rappresentazioni sono efficaci: il problema nasce quando due macchine a finale diverso cercano di condividere delle parole di memoria, ad esempio inviandole byte a byte. Supponiamo che il calcolatore a finale grande spedisca una parola a quello a finale piccolo un byte alla volta, a partire dal byte 0. Questo finisce nel byte 0 della numerazione a finale piccolo e cosí via. A questo punto la macchina *little–endian* interpreta la parola come se fosse invertita rispetto a quella di partenza.

Passiamo ora alla realizzazione fisica della memoria: la trattazione verterá sull'analisi logica dei circuiti. Il nostro obiettivo é la costruzione di un

generico modulo di memoria organizzato in 2^k locazioni di m bit ciascuna come mostrato in figura 3.2: il bus A é quello dedicato agli indirizzi mentre quello D é dedicato ai dati. Come noto, ogni *filo* puó contenere un valore 0 oppure 1. Gli ingressi *CS* e R/nW sono variabili di controllo per le operazioni sulla memoria. L'ingresso *CS* disabilita il modulo indirizzato dal bus A quando ha valore 0 ed in questo caso il valore di R/nW é irrilevante. Se *CS* ha valore 1, il modulo indirizzato dal bus A viene abilitato ed il valore di R/nW indica il tipo di operazione richiesta (con R/nW=*1* indichiamo l'operazione di lettura, con R/nW=*0* indichiamo quella di scrittura). Sul bus D vedremo i valori richiesti in base alla combinazione di questi ingressi.

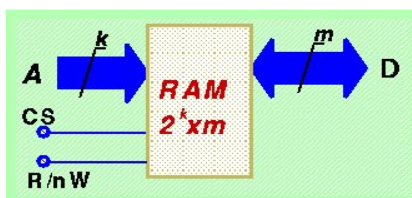


Figura 3.2: il diagramma logico di un generico modulo di memoria

Per realizzare questo componente si sfrutta la tecnologia dei circuiti sequenziali la cui implementazione fisica prevede l'uso dei condensatori. Recentemente si sono imposti i transistor ad effetto di campo, noti come *MOSFET*. Il loro nome deriva dall'inglese *Metal Oxide Semiconductor Field Effect Transistor*: sono un tipo particolare di transistor, usato nei dispositivi digitali grazie al basso consumo di energia ed alla conseguente dispersione di calore molto ridotta. Per approfondimenti rifarsi a [LP81] e [GKS98].

3.3 L'organizzazione dei dischi fissi

Nel paragrafo 3.2 abbiamo visto alcune caratteristiche della memoria: é un costoso componente dalle prestazioni eccellenti e perció non é conveniente averne grossi quantitativi. Di conseguenza si usano delle memorie secondarie, piú economiche e capaci della *RAM*: la tecnologia usata le rende peró piú *lente*. Questo fatto ha reso necessario lo sviluppo di tecniche di tipo hardware e/o software per migliorarne le prestazioni. Come abbiamo accennato nell'introduzione di questo capitolo, studieremo solamente le caratteristiche dei *dischi fissi*.

La tecnologia delle unitá a disco fisso presenta una serie di vantaggi nel campo dell'archiviazione dei dati quali una densitá di memorizzazione elevatissima ed un costo relativamente contenuto rispetto ad altre soluzioni. Il sistema di memorizzazione é costituito da *dischi* ricoperti di materiale ferromagnetico su cui vengono scritti i dati. Le operazioni di registrazione

e rilettura dei dati vengono effettuate dalle *testine*, una per ogni faccia dei piatti. Le testine operano lungo *tracce* concentriche del disco. I dischi sono calettati sullo stesso albero (l'asse di rotazione centrale): in questo modo é possibile introdurre un componente, detto *attuatore*, che controlla in maniera centralizzata le testine. La figura 3.3 mostra la struttura di un disco fisso.

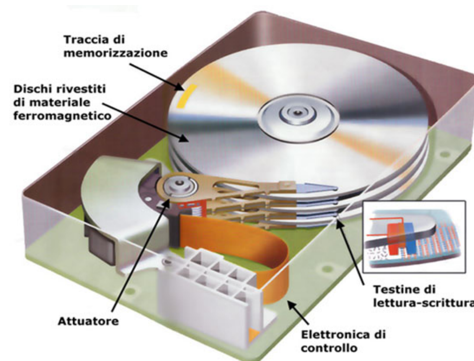


Figura 3.3: Schema di una moderna unitá a disco rigido

La *capacitá*, il *tempo di accesso* e la *velocitá di trasferimento* sono importanti indicatori delle prestazioni di un disco fisso.

La *capacitá* di un disco fisso misura la quantitá di dati memorizzabili: puó essere aumentata incrementando la densitá con cui i dati vengono memorizzati oppure la dimensione od il numero dei dischi. Il *tempo di accesso* é il tempo medio necessario perché un dato posto in una parte a caso del disco fisso possa essere reperito: la testina deve spostarsi e contemporaneamente il disco deve ruotare fino a quando non si é nella posizione richiesta. Ció introduce un ritardo nelle operazioni, detto *latenza rotazionale*. La *velocitá di trasferimento* é definita come la quantitá di dati che il disco fisso é in grado di gestire in un determinato periodo di tempo. Bisogna dire che, a parte casi particolari, la velocitá di trasferimento teorica viene raramente raggiunta ed il tempo di accesso é quello che maggiormente influenza le prestazioni.

Vi sono altre due grandezze che influenzano in misura minore le prestazioni di un disco fisso: il *buffer di memoria* e la *velocitá dell'interfaccia*. Il *buffer di memoria* é una piccola memoria cache posta a bordo del disco fisso, la quale ha il compito di memorizzare gli ultimi dati letti o scritti nel disco: in caso di ripetuti accessi le informazioni possono essere reperite nel buffer invece che dal disco. Visto che il buffer é un componente elettronico, le sue prestazioni sono molto elevate. L'*interfaccia* di collegamento tra l'hard disk ed il controller specifica la velocitá massima alla quale le informazioni possono essere trasferite da e per il disco fisso.

Gli algoritmi e le tecniche in memoria secondaria si occupano della gestione esplicita dei blocchi di dati da e verso i dispositivi di memorizzazione

(i dischi fissi nel nostro caso): in [dM07b] si dimostra come l'organizzazione interna dei supporti di memorizzazione possa favorire le operazioni di lettura e di scrittura. Per questo motivo é importante conoscere il funzionamento dei supporti usati per la memorizzazione dei dati: nella sezione 3.3.1 descriveremo l'organizzazione logica di un disco fisso mentre nella sezione 3.3.2 vedremo alcuni principi che stanno alla base della memorizzazione fisica dei dati sui piatti di un hard–disk.

3.3.1 L'organizzazione logica

I dati sono generalmente memorizzati su disco seguendo uno schema logico ben definito. Uno dei piú diffusi é quello che suddivide un disco in cilindri, settori e tracce. In letteratura é noto come schema *CHS*, acronimo dei termini inglesi *Cylinder*, *Head* e *Sector*. Ogni piatto si compone di numerosi anelli concentrici e numerati, detti *tracce*. Il *cilindro* corrisponde all'insieme delle tracce aventi lo stesso codice identificativo su piatti diversi. Il *settore*, detto anche *blocco disco*, é l'unitá di memorizzazione minima a cui si possa accedere e quindi corrisponde ad una porzione di traccia. La figura 3.4 mostra l'organizzazione logica di un disco fisso indotta dal modello *CHS*.

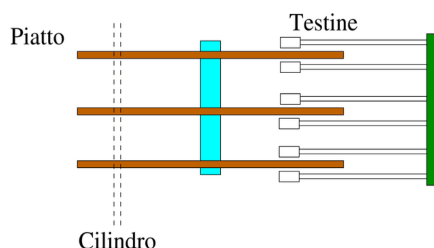


Figura 3.4: la suddivisione CHS in un disco fisso

Per definire le caratteristiche di un disco si parla di *geometria*: questa si esprime attraverso l'indicazione del numero di cilindri, di testine e di settori a disposizione. Ad esempio, se un hard disk si compone di 2 dischi (o in alternativa 4 piatti), 16384 cilindri (ció significa 16384 tracce per piatto) e 16 settori di 4096 bytes per traccia, allora la capacitá del disco sará il prodotto di queste quantitá:

$$(4\ piatti)(16384\ \frac{tracce}{piatti})(16\ \frac{settori}{tracce})(4096\ \frac{bytes}{settori}) \sim 4Gb$$

Per individuare i dati in un disco fisso, si puó usare una tripla del tipo (p, t, s) dove p é il numero del piatto, t é il codice identificativo della traccia mentre s indica il settore: in questo modo la testina deve spostarsi e contemporaneamente il disco deve ruotare fino a raggiungere il settore s. Questo processo é nettamente piú lento dell'identificazione di un blocco della memoria in

quanto si sfruttano dei componenti meccanici e non elettronici. Il tempo necessario ad operare su un blocco del disco é determinato da tre fattori: il *tempo di posizionamento* della testina sul cilindro richiesto, il *ritardo di rotazione* necessario al settore per passare sotto alla testina ed il *tempo di trasferimento* vero e proprio. Per la maggior parte dei dischi, il tempo di riposizionamento é quello predominante quindi ridurlo significa incrementare le prestazioni del sistema: per far questo dobbiamo sviluppare un algoritmo per la schedulazione del braccio del disco. Possiamo osservare che mentre il braccio delle testine sta operando possono arrivare altre richieste che verranno memorizzate dal controller. Una volta soddisfatta una richiesta, bisogna scegliere la prossima da gestire: gli algoritmi di schedulazione si differenziano su questo punto. Per approfondimenti sugli algoritmi di schedulazione del braccio di un disco rifarsi a [Smi81] ed a [GD87].

Un primo algoritmo consiste nel gestire le richieste una alla volta e nell'ordine di arrivo: questa schedulazione é nota come *FCFS*, dall'inglese *First–Come First–Served* e puó essere molto inefficiente, a seconda dell'ordine di arrivo delle richieste, in quanto costringe il braccio del disco a muoversi in maniera pressoché casuale.

In alternativa si potrebbe scegliere l'operazione relativa al cilindro piú vicino in modo da minimizzare il tempo di posizionamento: questo algoritmo é noto come *SSF* (dall'inglese *Shortest Seek First*) e riduce di circa la metá gli spostamenti del braccio del disco rispetto a quello *FCFS*. Purtroppo non garantisce il soddisfacimento delle richieste relative ai cilindri periferici, piú lontani dal centro del disco: é sufficiente l'arrivo in sequenza di richieste con spostamenti minimi o comunque relative ai cilindri centrali.

Un'ulteriore alternativa é ereditata dai sistemi di controllo degli ascensori i quali iniziano a muoversi in una direzione e la cambiano solo quando non vi sono piú richieste pendenti nel verso in cui stanno procedendo. Questo algoritmo é noto in letteratura come *algoritmo dell'ascensore* e richiede la memorizzazione della *direzione corrente* attraverso un flag che puó assumere i valori *UP* e *DOWN*. Il valore *UP* impone l'accettazione di richieste per numeri crescenti di cilindro, *DOWN* per numeri decrescenti di cilindro. Quando una richiesta é stata completata si controlla il flag: se é *UP* il braccio si muove verso il prossimo blocco con numero crescente di cilindro, se é *DOWN* si muove verso il prossimo blocco con numero decrescente di cilindro. In entrambi i casi si inverte la direzione di ricerca, modificando il flag, se non ci sono altre richieste in quella direzione.

3.3.2 La memorizzazione fisica dei dati

In questa sezione vedremo alcune nozioni di base sulla memorizzazione fisica dei dati all'interno di un disco rigido: non si hanno pretese di completezza, per una esposizione piú completa rifarsi ad un qualsiasi testo che tratti l'elettromagnetismo, come ad esempio [GKS98].

Per prima cosa dobbiamo osservare che i piatti dei dischi sono coperti da materiale ferromagnetico e quindi possiamo considerare un bit di informazione come una piccola regione del piatto che produce un campo magnetico. Quindi una testina deve essere in grado di riconoscere questa variazione del campo e di poter operare in maniera corretta su tale regione, la quale é nota in letteratura come *dominio di Weiss*. Inizialmente si sfruttava il fenomeno della magnetoresistenza: cerchiamo di descriverne in maniera intuitiva i principi di base. In un materiale la velocitá di movimento degli elettroni di conduzione varia a seconda che la loro direzione sia o meno parallela all'orientazione magnetica di un certo strato: quando gli elettroni si muovono nella stessa direzione della polarizzazione magnetica la loro velocitá é minima e quindi la resistenza é massima. Se si muovono nella direzione opposta, la resistenza é ovviamente minima. A seconda del verso della magnetizzazione che caratterizza l'area sulla quale staziona l'elemento sensibile, l'informazione verrá interpretata come 0 oppure 1. Il problema principale di questo approccio é il rapporto segnale/rumore, quest'ultimo generato dal magnetismo del materiale. Per comprendere la situazione, possiamo pensare gli atomi come una folla di calamite fluttuanti: questi producono dei *segnali spurii* che rovinano in qualche modo il segnale rilevato. Questo approccio rientra nel paradigma secondo cui il funzionamento di un dispositivo elettronico si deve basare sul trasporto della carica elettrica.

In realtá vi sono una serie di effetti fisici simili alla magnetoresistenza che si basano sulle proprietá di *spin*: ció ha portato allo sviluppo di una nuova branca della fisica della materia chiamata *spintronica* in cui é lo spin che gioca il ruolo principale nei meccanismi di trasporto. I dispositivi spintronici operano con alcune quantitá che dipendono dal grado di allineamento dello spin. L'effetto sicuramente piú noto é la *Magneto–Resistenza Gigante*, abbreviato spesso in *GMR* (dall'inglese *Giant Magneto–Resistance*): nel seguito della trattazione cercheremo di mettere in luce le idee fondamentali di questo effetto. La *GMR* é definibile come il cambiamento drastico della resistenza di una struttura composta da strati magnetici e non (nota come *struttura multilayer*) a seguito dell'applicazione di un campo magnetico di forte intensitá. Variando lo spessore degli strati non magnetici, si rendono antiparallele (in assenza di campo) le magnetizzazioni di due strati magnetici adiacenti: questa configurazione é nota come *configurazione anti–ferromagnetica* o AF. A seguito dell'applicazione di un campo magnetico sufficientemente intenso, le magnetizzazioni degli strati si orientano nella direzione del campo: questa é nota come la *configurazione ferromagnetica* o *FM*. In questa configurazione, la resistenza che il multilayer oppone al passaggio di corrente é molto piú piccola di quella presente nella configurazione AF. Come si puó notare, é un fenomeno del tutto analogo alla magnetoresistenza ordinaria. É possibile realizzare delle testine per disco fisso che sfruttano questo fenomeno: per approfondimenti rifarsi a [BG03]. Una variante della *GMR* é la *magnetoresistenza straordinaria* che si osserva in un reticolo semiconduttore composto

da diversi strati non magnetici e questo rende quasi nullo il rumore magnetico. Attualmente questo fenomeno non é stato pienamente compreso: per una trattazione piú approfondita si rimanda a [Sol04].

3.4 L'organizzazione dei dati

Le applicazioni su calcolatore hanno bisogno di memorizzare e gestire informazioni da e per il disco fisso garantendo tre requisiti fondamentali:

- deve essere possibile memorizzare una grossa quantitá di informazioni in maniera efficiente;
- l'informazione deve essere conservata anche dopo la terminazione del processo che vi accede;
- piú processi devono avere la possibilitá di accedere all'informazione in modo concorrente.

In questo paragrafo vedremo la soluzione offerta dal sistema operativo, introducendo il concetto di *file* nella sezione 3.4.1. Una soluzione piú evoluta é data dalle *basi di dati* che verranno introdotte nel capitolo 4. Nella sezione 3.4.2 vedremo come poter gestire in maniera efficiente lo spazio su disco. Infine, vedremo nella sezione 3.4.3 un modello evoluto di gestione dell'I/O che estende le tecniche precedenti.

3.4.1 Il concetto di file

Il *file* é un concetto introdotto dal sistema operativo per astrarre la memorizzazione di dati su un disco in modo tale che l'utente non debba conoscere il funzionamento del supporto di memorizzazione. Puó essere strutturato in molti modi: quello piú comune é la sequenza di byte, ordinata con un certo finale. Per il sistema operativo non é importante il contenuto di un file: un qualsiasi significato viene attribuito a livello di programma utente. Ogni sistema operativo fornisce agli utenti un metodo per indicizzare ed organizzare file detto *filesystem*. In questa tesi non ci interessa descrivere un filesystem in particolare ma piuttosto la memorizzazione del file.

Il fattore chiave nell'implementazione di un file consiste nel tenere traccia di quali blocchi disco memorizzano il suo contenuto: ogni sistema operativo sfrutta una particolare implementazione interna. Non ci interessa fissarne una in particolare in modo da non legarci ad un particolare sistema operativo. Per approfondimenti rifarsi alla foltissima letteratura sull'argomento come ad esempio [DN65], [RT74], [Tan92] e [Pic02]. Nel seguito della trattazione ci basta assumere che un file venga descritto come una sequenza di blocchi disco che memorizzano il suo contenuto.

3.4.2 La gestione dello spazio su disco

Nella sezione 3.4.1 abbiamo visto come un file sia una sequenza di byte: quasi tutti i filesystem suddividono i file in blocchi di dimensione fissa che non necessitano di essere adiacenti. Una volta decisa questa organizzazione, sorge la questione di quanto grandi dovranno essere i blocchi. Nella sezione 3.3.1 abbiamo visto l'organizzazione logica di un disco fisso quindi il settore, la traccia ed il cilindro sono ovvi canditati come unitá di allocazione. Avere una grande unitá di allocazione, come un cilindro, puó peró provocare un grande spreco di spazio: un file di dimensioni ridotte impegna un cilindro intero. In [MT84] e [THB06] viene mostrato che in ambiente UNIX la grandezza media di un file é di circa 1K e quindi per allocare un cilindro di 32K viene sprecato circa il 97% dello spazio totale del disco. Ma usare una piccola unitá di allocazione porterá ogni file ad essere composto da molti blocchi: la lettura di ogni blocco richiede normalmente una ricerca ed un ritardo di rotazione e questo rallenta la gestione di un singolo file. Quindi la scelta della dimensione del blocco deve garantire un buon utilizzo dello spazio disco ed una buona velocitá di trasferimento: purtroppo queste due proprietá sono inversamente proporzionali. Pertanto la dimensione del blocco sará un compromesso fra queste due proprietá.

Una volta scelta la dimensione del blocco disco, uno dei problemi che possono sorgere é quello di tenere traccia dei blocchi liberi. Vi sono due metodi il cui uso é particolarmente frequente nelle applicazioni: la *lista concatenata* e la *mappa dei bit* dei blocchi disco. Il primo metodo sfrutta una lista concatenata di blocchi disco ed ognuno di questi puó memorizzare un certo numero di indici di blocchi liberi: con questa tecnica si utilizza un numero di blocchi disco proporzionale a quello dei blocchi liberi. Il secondo metodo richiede una mappa di n bit, uno per ognuno degli n blocchi del disco: quelli liberi vengono rappresentati con il bit 1 nella mappa, 0 altrimenti. Vediamo lo spazio occupato da queste due tecniche, supponendo di voler gestire un disco da *20 Mb*, con la dimensione del blocco pari ad *1 Kb* ed un numero di blocco rappresentabile su 16 bit. Nel primo caso ogni blocco nella lista contiene i numeri di 511 blocchi liberi e quindi il disco necessita al massimo di 40 blocchi per contenere tutti i 20000 numeri di blocco disponibili. Dunque dobbiamo usare *40 Kb* per memorizzare la lista dei blocchi liberi. Nel secondo caso usiamo solamente tre blocchi per memorizzare la mappa composta da *20000 bit*. In generale, la tecnica della mappa di bit necessita di meno spazio, se vi é una quantitá sufficiente di memoria: solamente nel caso di disco quasi pieno la lista dei blocchi liberi sará piú economica.

Gli accessi al disco sono molto piú lenti di quelli alla memoria: per questa ragione bisogna ridurne il numero. La tecnica usata piú comunemente é la *cache dei blocchi*: si memorizza un insieme limitato di blocchi disco in modo da velocizzarne l'accesso. Il sistema di funzionamento é piuttosto semplice: quando si richiede un blocco, si controlla se questo é contenuto nella cache. In

caso affermativo l'operazione puó essere soddisfatta senza accedere al disco altrimenti il blocco va letto dal disco e memorizzato nella cache. Il buffer ha una dimensione limitata e quando diventa pieno bisogna sostituire un blocco attualmente memorizzato per far posto ad uno nuovo. Le politiche di sostituzione sono del tutto simili a quelle usate nella paginazione per sostituire una pagina di memoria: per approfondimenti rifarsi a [Smi78]. Tra le varie politiche di sostituzione possiamo ricordarne alcune, particolarmente importanti nelle applicazioni:

- la politica *LRU* (dall'inglese *Least–Recently–Used*) viene usata per sostituire il blocco acceduto meno recentemente fra quelli presenti nella cache: assieme al blocco bisogna tenere traccia del suo ultimo accesso;
- la politica *LFU* (dall'inglese *Least–Frequently–Used* viene usata per sostituire il blocco acceduto meno frequentemente fra quelli presenti nella cache: assieme al blocco bisogna tenere traccia degli accessi.

Dobbiamo peró osservare una differenza importante fra l'uso della cache per le pagine di memoria e per i blocchi disco. Una pagina di memoria viene allocata per un certo processo il quale é l'unico che vi puó accedere mentre i blocchi disco sono per loro natura condivisi fra piú processi. Quindi l'uso della cache nella paginazione avvantaggia banalmente un singolo processo in quanto é l'unico che puó operare su quella determinata pagina di memoria. Al contrario, l'uso della cache nella lettura dei blocchi permette al sistema operativo di risparmiare la gestione di alcuni accessi al disco, senza agevolare in realtá alcun processo che accede concorrentemente a quel blocco. Nel primo caso si parla di *localitá forte* delle pagine, mentre nel secondo di *localitá debole* dei blocchi. Questo concetto é particolarmente importante in fase di scrittura: con la localitá forte si é certi che é stato il processo proprietario a modificare la pagina di memoria, con la localitá debole il blocco disco puó essere stato modificato da uno dei tanti processi che lo condividono. Questa situazione puó essere gestita nell'ambito del controllo della concorrenza: parleremo di questo argomento in maniera piú approfondita nel capitolo 4. Abbiamo visto come la memoria cache possa essere usata per la paginazione della memoria e per la gestione dei blocchi disco: in realtá é possibile usarla anche per dati generici. In questa tesi abbiamo implementato una cache per la memorizzazione di coppie del tipo *chiave/dato* che sfrutta la politica di sostituzione *LRU*: descriveremo le caratteristiche fondamentali di questo componente nel capitolo 6.

L'uso delle tecniche di *caching* non é l'unico metodo per migliorare le prestazioni di un filesystem: possiamo ridurre il movimento delle testine del disco sistemando i blocchi che potrebbero essere acceduti preferibilmente nello stesso cilindro. Quando si scrive un file, si devono allocare i blocchi uno alla volta, man mano che questi vengono richiesti. Se i blocchi liberi sono memorizzati in una mappa di bit interamente contenuta in memoria,

é abbastanza facile scegliere un blocco libero il piú vicino possibile a quello di partenza; con una lista libera, parzialmente contenuta nel disco, é piú complicato allocare assieme i blocchi vicini. In questo caso é piú conveniente mantenere traccia della memoria del disco in gruppi di blocchi consecutivi. Vediamo un esempio pratico: se una traccia é formata da 64 settori di 512 byte, il sistema potrebbe usare blocchi di *1 Kb* (equivalenti a 2 settori) ed allocare la memoria disco in unitá composte da 2 blocchi (e quindi 4 settori). Ció non equivale ad avere un blocco disco di *2 Kb* poiché la memoria cache potrá ancora usare blocchi di *1 Kb*, ma la struttura sequenziale ridurrá di un fattore due il numero di ricerche necessarie. Per approfondimenti su queste tecniche rifarsi a [MJLF84].

3.4.3 Il modello PDM

In letteratura é stato sviluppato un modello che descrive in maniera accurata le problematiche relative all'I/O: si tratta del modello *PDM* (dall'inglese *Parallel Disk Model*), introdotto in [VS94a], [VS94b] e [Vit01]. Questo modello descrive un'architettura a due livelli di memoria: il primo livello é costituito dalla memoria principale ed il secondo da una serie di dischi fissi. Questa struttura si interfaccia con un certo numero di processori: non ne viene specificato il numero preciso e la loro modalitá di interconnessione é indifferente. In [Vit01] viene dimostrato che non si limita il potere espressivo del modello usando un solo processore: nel seguito assumeremo la presenza di una sola *CPU*, come mostra la figura 3.5.

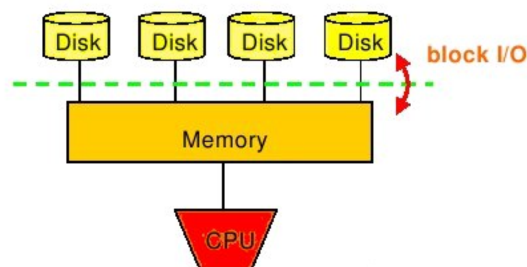


Figura 3.5: l'architettura a due livelli di memoria usata nel modello *PDM*

In questo modello si assume che la complessitá della maggior parte degli algoritmi in memoria secondaria (detti algoritmi *EM*, dall'inglese *External Memory*) possa essere descritta attraverso questi parametri:

- la dimensione dei dati di input, che indicheremo con N;
- la capacitá della memoria disponibile, che indicheremo con M;
- la dimensione del blocco disco, che indicheremo con B;

- il numero dei dischi, che indicheremo con D.

I parametri N, M e B devono essere composti da un numero di byte che possa essere espresso come una potenza di due in modo da poter implementare in maniera efficiente le operazioni. Nel seguito assumeremo che $M < N$: in questo modo vengono rispettate le ipotesi di partenza di questa tesi in cui la dimensione dei modelli geometrici eccede la quantitá di memoria disponibile. Fra la memoria principale ed i dischi vengono scambiati blocchi di dimensione B: si assume che l'accesso ai dischi venga svolto in maniera *sincrona*, ovvero ogni operazione opera su D blocchi contemporaneamente e gestisce una quantitá di dati pari a DB: per ragioni di efficienza si assume che $DB < \frac{M}{2}$. Per ottimizzare queste operazioni, si organizzano i blocchi in *strisce*: si opera contemporaneamente su D blocchi appartenenti a dischi diversi (uno per disco), ma nella stessa posizione. Questa tecnica é nota in letteratura con il termine inglese *striping* ed é stata introdotta nella descrizione dell'architettura *RAID*, (dall'inglese *Redundant Array of Inexpensive Disks*) la quale aumenta l'affidabilitá e la disponibilitá dei dati memorizzati: per approfondimenti rifarsi a [CGK+88], [PGK88] e [Gib92].

Nel seguito, indicheremo con modello *classico* l'organizzazione delle operazioni di I/O introdotta nell'architettura di von Neumann della qualle abbiamo discusso inizialmente in questo capitolo. Nel modello *PDM* si assume che ogni algoritmo possa essere descritto in termini di queste quattro operazioni:

- la *scansione* sequenziale di N record: nel modello *classico* questa operazione ha una complessitá $\Theta(N)$ e quindi $\Theta(\frac{N}{BD})$ nel nostro caso, visto che ogni operazione opera su BD blocchi;
- l'*ordinamento* di N record: come noto, nel modello *classico*, questa operazione ha una complessitá $\Theta(N \log N)$, in [AV88] viene dimostrato che questa vale $\Theta(\frac{N}{BD}\frac{\log N - \log B}{\log M - \log B})$;
- la *ricerca* di un determinato record all'interno dei dati: nel caso peggiore é equivalente alla scansione sequenziale di N record;
- la *risoluzione* di una certa interrogazione sui dati.

Vi sono applicazioni, come le permutazioni, per le quali queste primitive non sono sufficienti: inoltre questo modello fornisce una stima incoerente della complessitá. Nel modello *classico* il problema di permutare N oggetti ha complessitá $\Theta(N)$ quindi, in base alle considerazioni precedenti, si dovrebbe avere una complessitá $\Theta(\frac{N}{BD})$: invece in [CH96] si dimostra che questa vale $\Theta(\frac{N}{BD}\frac{\log N - \log B}{\log M - \log B})$, cioé equivalente al caso del sorting. Questa discrepanza viene dal fatto che in questo modello si misura la complessitá in termini delle operazioni di I/O, tralasciando le fasi di calcolo e di comunicazione fra memoria e disco: in certe situazioni, queste operazioni hanno complessitá non trascurabile e spesso superiore rispetto a quelle di I/O. Per queste ragioni, il

modello *PDM* non ha trovato molta diffusione nelle applicazioni ad eccezione della libreria *TPIE*.

La libreria *TPIE* (acronimo di *Transparent and Parallel I/O Environment*) é stata sviluppata alla *Duke University* ed implementa l'architettura a due livelli descritta dal modello *PDM* appena introdotto: per approfondimenti rifarsi a [ABH+97] e [ABH+02]. La struttura di tale libreria é modulare e si basa sui seguenti componenti:

- il modulo *AMI* (dall'inglese *Access Method Interface*) fornisce al programmatore dei metodi per l'accesso all'architettura *PDM*;
- il modulo *BTE* (dall'inglese *Block Transfer Engine*) si occupa del trasferimento dei blocchi di dati fra la memoria ed i dischi fissi;
- il modulo *MM* (dall'inglese *Memory Manager*) si occupa dell'allocazione e della gestione della memoria necessaria.

Ogni operazione puó essere descritta attraverso l'interazione fra questi tre moduli: quelli *BTE* ed *MM* sono transparenti per l'utente il quale sfrutta solamente il componente *AMI*. La libreria *TPIE* é stata usata in applicazioni di calcolo scientifico come descritto in [VV96] e [PAAV03]. Essa costituisce uno strumento alternativo con il quale gestire dati in memoria secondaria: la sua implementazione attuale sfrutta le primitive fornite da un kernel *UNIX* non compatibile con lo standard *POSIX* e quindi questa libreria non é portabile e non supportata da altri sistemi. La non aderenza allo standard *POSIX*, il quale definisce un interfaccia dei servizi comunemente forniti da un sistema operativo, ne limita la diffusione: per approfondimenti sulle primitive dello standard *POSIX* rifarsi a [IEE85], [OG04], [Tri05] e [LSB07]. In questa tesi abbiamo tenuto in debita considerazione la compatibilitá di quanto realizzato con la maggior parte dei sistemi attualmente esistenti e quindi l'uso di questa libreria sarebbe stato un ostacolo per il nostro obiettivo. Per questa ragione abbiamo utilizzato il sistema software *Oracle Berkeley DB* al posto di questa libreria il quale ci ha permesso di creare un prototipo di *DBMS* spaziale funzionante su vari sistemi. In realtá la libreria *OMSM* ricalca le scelte architetturali di quella *TPIE* in quanto é anch'essa realizzata tramite tre componenti che interagiscono fra loro con compiti pressoché simili a quelli dei moduli *AMI*, *BTE* ed *MM*. Descriveremo il componente *Oracle Berkeley DB* nella sezione 4.4 e la libreria *OMSM* nel capitolo 6.

Capitolo 4

Le basi di dati

La matematica non possiede solo la veritá, ma anche una bellezza fredda ed austera come quella della scultura.

Bertrand Russell

In questa tesi é stato sviluppato un prototipo di *DBMS* spaziale di tipo *embedded* e quindi ci interessa rapportarne le caratteristiche principali con le problematiche *classiche* della teoria delle basi di dati, argomento di questo capitolo. Non si ha alcuna pretesa di completezza: nel seguito della trattazione verranno proposti alcuni approfondimenti. Nel paragrafo 4.1 verranno descritte le proprietá generali di una *base di dati*. Nel paragrafo 4.2 vedremo come sia possibile strutturarne correttamente l'accesso: inoltre studieremo nel paragrafo 4.3 una tecnica per garantire il ripristino dei dati in caso di fallimento. Infine presenteremo le caratteristiche del sistema software *Oracle Berkeley DB* nel paragrafo 4.4: questo componente é stato usato in questa tesi per la realizzazione del *DBMS* spaziale.

4.1 Gli aspetti introduttivi

In questo paragrafo verranno introdotte le proprietá fondamentali di una base di dati. Non si hanno pretese di completezza: per una trattazione piú completa rifarsi a [EMN89], [Ull89], [KS91] e [AHV95]. Nella sezione 4.1.1 verranno introdotte le proprietá di un *DBMS*, il quale utilizza un modello per la rappresentazione dei dati, come vedremo nella sezione 4.1.2. Nella sezione 4.1.3 vedremo come un *DBMS* possa fornire varie rappresentazioni dei dati, gestibili attraverso dei linguaggi che descriveremo nella sezione 4.1.4.

4.1.1 Basi di dati e DBMS

Due concetti molto importanti sono quelli di *dato* e di *informazione*: questi termini vengono spesso usati impropriamente come sinonimi e quindi dobbiamo ricordarne le definizioni, espresse per la prima volta in [BdPLS90].

Definizione 4.1. Per *informazione* si intende tutto ció che produce variazioni nel patrimonio conoscitivo di un soggetto.

Definizione 4.2. Per *dato* si intende una registrazione della descrizione di una qualsiasi caratteristica della realtá su un supporto che ne garantisca la conservazione, la comprensibilitá e la reperibilitá.

Per capire cosa sia una *base di dati* possiamo ripercorrere brevemente l'evoluzione dei sistemi informativi. I primi sistemi erano basati sull'uso di archivi separati a cui ogni applicazione accedeva in modo autonomo. In seguito si sviluppó un'architettura in cui il sistema operativo costituiva l'interfaccia fra ogni applicazione ed i propri archivi offrendo dei metodi di accesso generalizzato: per approfondimenti rifarsi a [Gra78]. Infine, si é passati ad un approccio in cui i dati vengono organizzati in un unico insieme logicamente integrato, detto *base di dati*. Tale insieme é gestito dal *DBMS*, (dall'inglese *Data Base Management System*) come mostrato in figura 4.1.

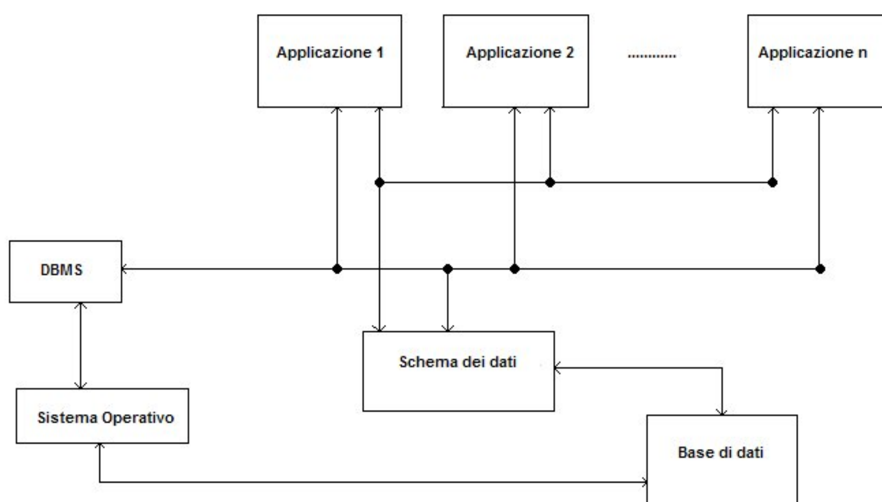


Figura 4.1: l'architettura di una base di dati con *DBMS*

Nel seguito della trattazione cercheremo di fissare i concetti di *base di dati* e di *DBMS*.

Definizione 4.3. Una *base di dati* puó essere definita come un insieme di dati strettamente correlati e memorizzati su un supporto di memoria di massa che possono essere manipolati da vari programmi applicativi.

Definizione 4.4. Un *DBMS* é un sistema hardware o software che si occupa dell'aggiornamento, della gestione e della consultazione di una base di dati registrata in una memoria di massa.

L'introduzione dei *DBMS* assicura tre obiettivi fondamentali: l'*integrazione*, l'*indipendenza* ed il *controllo centralizzato* dei dati. L'*integrazione* garantisce una definizione logica e centralizzata dei dati, detta *schema*, la quale viene descritta da un formalismo ad alto livello: quest'ultimo é detto *modello dei dati* e sará argomento della sezione 4.1.2. Questo meccanismo permette alle applicazioni di accedere in varie modalitá alla base di dati, richiedendo solamente le informazioni di interesse. L'*indipendenza* dei dati opera almeno su due livelli, il *livello fisico* e quello *logico*. In questo modo la rappresentazione fisica dei dati é indipendente da come il *DBMS* organizza i dati per gli utenti: questo meccanismo sará l'argomento della sezione 4.1.3. Ad esempio, in questa tesi le applicazioni si interfacciano al *DBMS* operando su oggetti geometrici, non sapendo che questi vengono inseriti in un particolare indice spaziale e che vengono memorizzati sotto forma di cluster di nodi: vedremo nel dettaglio questo procedimento nel capitolo 6. Il *controllo centralizzato* sui dati ne fornisce un accesso condiviso, garantendo la qualitá, la privatezza e la sicurezza degli stessi in quanto ogni richiesta di accesso alla base di dati viene interpretata, analizzata ed eseguita dal *DBMS* stesso. Un *DBMS* offre dei servizi per garantire gli obiettivi delineati, fra i quali ricordiamo:

- la *descrizione dei dati*, la quale specifica cosa memorizzare nel database: tratteremo questo aspetto nella sezione 4.1.2;
- la *manipolazione dei dati*, la quale permette la gestione degli stessi, introducendo diversi livelli di rappresentazione, ognuno dei quali fornisce dei linguaggi adatti alla gestione dei dati: analizzeremo questi aspetti nelle sezioni 4.1.3 e 4.1.4;
- le *strutture di memorizzazione*, le quali garantiscono la durabilitá dei dati: questo aspetto verrá brevemente delineato nelle sezioni 4.1.3 e 4.1.4. Le strutture di memorizzazione dipendono dal tipo di dati che vogliamo memorizzare: nel capitolo 5 ne vedremo alcuni esempi adatti alla gestione di dati spaziali e multidimensionali;
- il *controllo dell'integritá*, il quale evita la memorizzazione di dati non corretti: si basa sull'introduzione di un accesso centralizzato alla base di dati, aspetto che verrá trattato nel paragrafo 4.2;
- il *ripristino della base di dati*, il quale evita l'inconsistenza e la perdita dei dati memorizzati: tratteremo questo aspetto nel paragrafo 4.3.

4.1.2 I modelli dei dati

Per utilizzare i servizi di un *DBMS* é necessario rappresentare le *entitá* del mondo reale mediante un insieme di concetti che un *DBMS* é in grado di trattare. Tale insieme di concetti é detto *modello dei dati* e rappresenta il punto di partenza della nostra analisi.

Definizione 4.5. Un *modello dei dati* é un formalismo che descrive:

- i *dati*, i quali rappresentano le entitá di un certo ambito applicativo;
- le *interrelazioni fra dati*, le quali rappresentano le associazioni semantiche fra le entitá descritte dai dati;
- i *vincoli semantici*, i quali rappresentano le proprietá significative per le entitá rappresentate dai dati.

Dato che i modelli servono per rappresentare una certa realtá di interesse é utile identificarne i concetti di base, tra i quali ricordiamo:

- *entitá*: oggetto della realtá applicativa di interesse;
- *attributo*: una caratteristica di una data entitá che é significativa ai fini della descrizione della realtá applicativa di interesse e che assume uno o piú valori, detti *valori dell'attributo*, appartenenti ad un insieme di valori possibili, detto *dominio* dell'attributo;
- *insieme di entitá*: raggruppamento di entitá *simili* aventi gli stessi attributi, anche se non necessariamente gli stessi valori degli attributi;
- *associazione*: una corrispondenza fra piú insiemi di entitá.

Questo ci permette di definire lo *schema* di una base di dati.

Definizione 4.6. La descrizione dei dati specificata tramite un modello é detta *schema* della base di dati.

Un esempio molto noto di modello dei dati é il *modello relazionale*, basato sul concetto di *relazione*: essa puó essere vista come una tabella con righe e colonne contenenti dati di un certo tipo. Ogni riga della tabella viene detta *tupla*. Una tabella rappresenta un insieme di entitá, una tupla rappresenta un'entitá appartenente all'insieme modellato dalla relazione a cui la tupla appartiene, mentre una colonna di una tupla rappresenta il valore di un particolare attributo dell'entitá rappresentata dalla tupla. Esiste un'ampia letteratura sull'argomento in quanto é il modello che viene maggiormente usato nelle applicazioni: per approfondimenti rifarsi ad esempio a [Cod85] e [Kan90]. Il *DBMS* realizzato in questa tesi non sfrutta questo modello: nella sezione 4.4.2 vedremo il modello usato dal sistema *Oracle Berkeley DB*.

L'evoluzione dei *DBMS* é stata guidata da quella dei modelli dei dati. I primi *DBMS* sfruttavano il modello gerarchico prima e quello reticolare poi: per approfondimenti rifarsi a [Oll78] e [Bas96]. Negli anni '80 comparve il modello relazionale, la cui semplicitá permise lo sviluppo di un linguaggio di accesso come quello *SQL*: nella sezione 4.1.4 spiegheremo cosa si intende per linguaggio di accesso. Alcuni esempi di *DBMS* relazionali (noti in letteratura anche come *RDBMS*, dall'inglese *Relational DBMS*) sono descritti in

[Sql96] e [Hip00]. Le applicazioni piú recenti richiedono la gestione di dati complessi e multimediali: sono stati sviluppati *DBMS* che integrano la tecnologia ad oggetti, come descritto in [CFB00], [Web00], [CFB+03] e [Ora05]. Sono divenuti molto importanti anche i *DBMS* spaziali, i quali gestiscono entitá spaziali e geografiche utili ad esempio nelle applicazioni GIS: il sistema realizzato in questa tesi appartiene a questa categoria. Per approfondimenti rifarsi a [Par94], [SA94], [GS95], [HNP95], [Sam95], [AGS97], [BBC97], [ES97], [Pro97b], [AI01a] e [EEA05]. Come si puó notare, la tendenza nello sviluppo di nuovi modelli dei dati é sempre stata quella di aumentare il potere espressivo del modello, rendendolo in grado di rappresentare in modo diretto la realtá di interesse per le applicazioni e di rendere la rappresentazione dei dati il piú indipendente possibile da aspetti implementativi.

4.1.3 I livelli nella rappresentazione dei dati

Uno degli scopi del *DBMS* é quello di fornire una rappresentazione logica dei dati, nascondendone i dettagli implementativi: in questo modo l'utente concepisce le informazioni secondo strutture riferite direttamente alle entitá reali di interesse. Per realizzare ció si introducono i tre livelli seguenti:

- il *livello fisico*, il quale riguarda l'effettiva memorizzazione dei dati;
- il *livello concettuale*, il quale riguarda la struttura logica assunta dai dati;
- il *livello esterno*, il quale si riferisce al modo in cui ogni utente vede ed accede ai dati.

Il *livello fisico* della base di dati contiene le strutture usate per memorizzare e gestire i dati in maniera efficiente. Queste strutture sono dette *indici* e servono a facilitare e velocizzare le operazioni sui dati: ne vedremo alcuni esempi nel capitolo 5. Per l'utente, questo livello é del tutto trasparente: egli si deve occupare solo del valore informativo dei dati e non di come vengono registrati sui supporti. Il *livello concettuale* rappresenta la struttura globale della base di dati descritta tramite il modello dei dati, formalismo introdotto nella sezione 4.1.2. La descrizione é realizzata tramite il *linguaggio di descrizione dei dati* che studieremo nella sezione 4.1.4. Una volta definito lo schema concettuale, é possibile realizzare accessi e navigazioni personalizzate dei dati che prendono il nome di *viste* logiche: queste costituiscono il *livello esterno* nella rappresentazione dei dati.

Definizione 4.7. Una *vista* é l'astrazione di una parte della base di dati concettuale che coinvolge i dati dell'istanza della base di dati limitatamente alla porzione interessata.

4.1.4 I linguaggi di accesso e manipolazione

Un *DBMS* fornisce un insieme di linguaggi che permettono agli utenti di specificare i dati di interesse. Tra questi linguaggi possiamo ricordare il *linguaggio di definizione dei dati*, il *linguaggio di manipolazione dei dati* ed il *linguaggio di definizione delle strutture di memorizzazione*.

Il *linguaggio di definizione dei dati* é spesso indicato con *DDL* (dall'inglese *Data Definition Language*) e descrive lo schema concettuale della base di dati, fornendo la notazione che permette di specificarne le strutture. Un esempio classico di tale linguaggio é quello *Entitá–Relazione*, noto anche come modello *ER* (dall'espressione inglese *Entity–Relationship*): per approfondimenti rifarsi a [Che76]. Il *linguaggio di manipolazione dei dati* é spesso indicato con *DML* (dall'inglese *Data Manipulation Language*) e permette agli utenti di accedere ed operare sulla base di dati. Le operazioni supportate sono generalmente l'inserimento, la cancellazione e la ricerca di dati. Un esempio di tale linguaggio é quello *SQL*: per approfondimenti rifarsi a [MS93] e [Sql03]. Il *linguaggio di definizione delle strutture di memorizzazione* é spesso indicato con *SDL* (dall'inglese *Storage Definition Language*) e fissa la corrispondenza fra le strutture logiche dei dati e quelle di memorizzazione, a cui abbiamo accennato nella sezione 4.1.3. L'utente puó influenzare le scelte del *DBMS* richiedendo l'allocazione di strutture ausiliarie di accesso: ne descriveremo alcune nel capitolo 5. Anche nel *DBMS* contenuto nella libreria *OMSM* si puó forzare l'uso di un particolare indice spaziale: vedremo i dettagli di questo meccanismo nel capitolo 6.

4.2 L'accesso ad una base di dati

Uno degli scopi principali di un *DBMS* é la gestione degli accessi ad una base di dati, la quale puó essere interrogata contemporaneamente da vari utenti, diventando una risorsa condivisa fra piú processi. In questo modo si sposta la risoluzione del problema nell'ambito della programmazione concorrente, che é uno dei campi piú studiati dell'informatica: per approfondimenti rifarsi a [BA82], [Pap86], [And01] e [Pic02]. Molte tecniche risolutive per l'accesso ad una base di dati sono *ereditate* dagli studi sulla concorrenza: nella sezione 4.2.1 fisseremo il concetto di *transazione*, mentre vedremo come eseguirle nella sezione 4.2.2. Infine studieremo come garantirne le proprietá di *isolamento* nella sezione 4.2.3.

4.2.1 Le transazioni

Per mantenere consistenti le informazioni in una base di dati é opportuno controllare le sequenze di accesso, dette *transazioni*.

Definizione 4.8. Si dice *transazione* una sequenza di operazioni da eseguire sulla base di dati.

Una transazione deve soddisfare le proprietá di *Atomicitá*, di *Consistenza*, di *Isolamento* e di *Durabilitá*: le loro iniziali formano il termine *ACID* con il quale sono note in letteratura. Vediamo in cosa consistono.

- *Atomicitá* – tutte le operazioni di una transazione devono essere trattate come una singola unitá;
- *Consistenza* – una transazione deve agire correttamente sul database;
- *Isolamento* – ogni transazione non puó osservare i risultati intermedi di altre transazioni;
- *Durabilitá* – i risultati di una transazione terminata con successo devono essere resi permanenti nel database nonostante i possibili malfunzionamenti.

Il modello di transazione piú usato é quello a *transazione semplice*: esso prevede un solo livello di controllo a cui appartengono tutte le operazioni da eseguire. Vengono introdotte cinque operazioni:

- l'operazione *BeginTransaction* che dichiara l'inizio di una transazione;
- l'operazione *CommitTransaction* che segnala il raggiungimento di uno stato consistente da parte del sistema, situazione detta di *commit*;
- l'operazione *RollbackTransaction* che annulla gli effetti di una transazione, situazione detta di *rollback*;
- l'operazione di *lettura* di un certo dato dal database;
- l'operazione di *scrittura* di un certo dato nel database.

Vediamo in figura 4.2 la struttura tipica di una transazione con questo modello, descritta in uno pseudo–codice molto simile al linguaggio *C++*. Per comprenderne il significato é necessario ricordare la semantica di tale linguaggio: per approfondimenti rifarsi a [Str86], [Eck00], [CPP00] e [EA03]. Nella figura 4.2 sono ben visibili l'avvio, l'esecuzione e la fine di una transazione. L'istruzione *BeginTransaction* ne dichiara l'avvio: in questa fase vengono inizializzate le strutture dati necessarie a seconda delle varie implementazioni. Il corpo del blocco *try–catch* contiene le operazioni di lettura e di scrittura che devono essere eseguite dalla transazione. Ricordando il funzionamento di un blocco *try–catch* nel linguaggio *C++*, possiamo osservare che verrá eseguita l'istruzione *CommitTransaction* se non ci sono errori altrimenti quella *RollbackTransaction*: questo avviene in modo esclusivo. Nel primo caso la transazione potrá rendere persistenti i cambiamenti nella base di dati mentre nel secondo caso tutte le modifiche verranno annullate: nel paragrafo 4.3 vedremo come garantire la durabilitá delle modifiche.

```
// Parte la transazione
BeginTransaction

// Qui inizia il corpo della transazione
try {
        // Si eseguono le letture e le scritture
        ...

        // Si rendono persistenti le modifiche
        CommitTransaction }
catch( Errore ) {
        // Bisogna annullare le modifiche
        RollbackTransaction }
```

Figura 4.2: una transazione nel modello a transazione semplice

Il modello appena esposto non permette di operare su parti distinte di una transazione in quanto possiede una struttura di controllo troppo semplice per modellare applicazioni complesse. In letteratura ne sono state proposte estensioni: per approfondimenti rifarsi a [BOH+92], [Moh94] e [MAA+98].

4.2.2 Il controllo della concorrenza

Un *DBMS* garantisce l'integritá della base di dati sincronizzando l'esecuzione delle transazioni: per descrivere questo procedimento dobbiamo introdurre il concetto di *interleaving* e quello di *schedule*.

Definizione 4.9. L'esecuzione concorrente di due o piú transazioni genera un'alternanza di computazioni, nota come *interleaving*. La sequenza delle operazioni viene detta *schedule*.

L'interleaving puó generare uno stato scorretto della base di dati in quanto una transazione potrebbe *vedere* lo stato del sistema generato dalle altre computazioni, violando cosí la proprietá di isolamento. Si puó osservare che, se le transazioni venissero eseguite consecutivamente, si potrebbe ottenere uno stato corretto: ció porta alla definizione di *schedule seriale*.

Definizione 4.10. Uno *schedule seriale* é uno schedule in cui le transazioni vengono eseguite in sequenza.

Un *DBMS* assicura l'esecuzione dei soli *schedule serializzabili*.

Definizione 4.11. Uno schedule S é *serializzabile* se e solo se esiste uno schedule seriale S_1 tale che la sua esecuzione porta la base di dati in uno stato appartenente all'insieme degli stati ottenibili dall'esecuzione di S.

Nelle definizioni precedenti non abbiamo considerato che all'interno delle transazioni vi possono essere istruzioni operanti su dati condivisi, sulle quali non abbiamo imposto alcun ordine di esecuzione: questa scelta non garantisce l'univocitá delle computazioni e quindi degli stati ottenibili, che possono variare. Nella sezione 4.2.1 abbiamo introdotto le operazioni di lettura e di scrittura: si osserva che la lettura non é distruttiva in quanto non modifica il dato da leggere, mentre la scrittura lo é in quanto deve aggiornare il contenuto del database. Ovviamente non possono essere eseguite in modo concorrente se operanti sullo stesso dato, costituendo un esempio di *operazioni in conflitto.*

Definizione 4.12. Se due operazioni devono operare sullo stesso dato, si dicono in *conflitto.*

Questa definizione é molto forte: anche due operazioni di lettura sullo stesso dato sono in conflitto, ma visto che non sono distruttive, si potrebbe garantire l'accesso contemporaneo alla base di dati per velocizzarne le operazioni. Vedremo nella sezione 4.2.3 come indebolire questa definizione.

Per risolvere l'esecuzione di operazioni in conflitto bisogna usare un protocollo adatto: vi é un'ampia letteratura sull'argomento, per approfondimenti rifarsi ad esempio a [KR81] e [Fra97]. Il *DBMS* descritto in questa tesi utilizza la tecnica del *blocco a due fasi* (in inglese *two–phase locking*): nel seguito ne vedremo le caratteristiche fondamentali, rimandando a [TR91] per una trattazione piú completa. Questo protocollo si basa sul concetto di *blocco* di un dato (in inglese *lock*): vediamone le definizioni.

Definizione 4.13. Un blocco in modalitá *condivisa* su un certo dato Q (in inglese *shared lock*) permette la lettura, ma non la scrittura su Q. Un blocco di questo tipo puó essere concesso a piú transazioni concorrenti che intendano leggere Q.

Definizione 4.14. Un blocco in modalitá *esclusiva* su un certo dato Q (in inglese *exclusive lock*) permette sia la lettura e sia la scrittura su Q. Un blocco di questo tipo puó essere concesso ad una sola transazione.

Il protocollo *two–phase locking* prevede che ogni transazione richieda un blocco di tipo condiviso per ogni lettura ed uno di tipo esclusivo per ogni scrittura, da acquisire e rilasciare in due fasi distinte, la *fase di acquisizione* e quella di *rilascio.* La transazione richiede tutti i lock necessari nella prima fase e se non puó acquisirne uno essa viene bloccata: una volta terminata la sua esecuzione, essa rilascia i blocchi posseduti. In [ECLT76] si dimostra che questo protocollo garantisce la *serializzabilitá.* I blocchi vengono memorizzati in una tabella all'interno del *DBMS.*

Usando questa tecnica, possiamo peró avere situazioni di *stallo* (in inglese *deadlock*). É un problema molto noto in letteratura, nell'ambito della programmazione concorrente: per approfondimenti rifarsi a [Hav68], [CES71],

[Hal72], [IM80], [GHKO81], [Zob83] e [Tan92]. Vediamo come puó verificarsi questa situazione: supponiamo che due transazioni T_1 e T_2 debbano operare concorrentemente in scrittura sui dati X_1 e X_2. La transazione T_1 ottiene un blocco di tipo esclusivo su X_1 mentre la transazione T_2 ottiene un blocco di tipo esclusivo su X_2: in questo caso T_1 é in attesa del blocco su X_2, attualmente in possesso di T_2, la quale é a sua volta in attesa del blocco su X_1. Pertanto le due transazioni sono entrambe bloccate: questo ci permette di formalizzare la definizione di *deadlock*.

Definizione 4.15. Un sistema si dice in *deadlock* se esiste un insieme di transazioni tali che ognuna sia bloccata in attesa del rilascio di un blocco.

Per risolvere il problema dello stallo si possono usare protocolli di *prevenzione* oppure di *rilevazione e risoluzione*. Con i primi, ci si assicura che il sistema non entri mai in uno stato di deadlock, con i secondi si lascia avvenire lo stallo e poi si cerca di eliminarlo. Il *DBMS* usato in questa tesi sfrutta solamente un protocollo per la *rilevazione* del deadlock, come verrá descritto nella sezione 4.4.2. Di seguito vedremo le caratteristiche fondamentali di un algoritmo di rilevazione dello stallo: per una trattazione piú approfondita si rimanda a [New79]. Inoltre assumeremo note alcune nozioni di teoria dei grafi: per approfondimenti rifarsi a [BM76], [Bar04] e [Die05]. La situazione di *stallo* puó essere descritta in termini di un grafo diretto, detto *grafo delle attese*, il quale consiste in una coppia $G = (V, E)$, dove V é costituito da tutte le transazioni del sistema ed ogni elemento di E é una coppia ordinata (t_i, t_j). Se esiste una coppia (t_i, t_j), questo indica che t_i é in attesa che t_j rilasci un blocco su un dato necessario alla sua esecuzione. L'arco (t_i, t_j) é rimosso quando la transazione t_j rilascia il blocco. Come dimostrato in [New79], il sistema é in stallo se e solo se il grafo delle attese contiene un ciclo. Per rilevare lo stallo, il sistema gestisce il grafo delle attese e periodicamente invoca un algoritmo per rilevare la presenza di un ciclo nel grafo.

4.2.3 I livelli di isolamento

Un *DBMS* deve soddisfare la proprietá dell'*isolamento* dando l'illusione che ogni transazione venga eseguita come se nel sistema non ve ne fossero altre. Solitamente si forniscono questi quattro livelli di isolamento:

- il livello *Read Uncommitted* – Permette ad una transazione di leggere gli aggiornamenti non ancora persistenti nel database. Questa scelta puó provocare le *letture fantasma*: la transazione che ha modificato la base di dati potrebbe eseguire l'operazione di *rollback* e quindi le modifiche lette non avrebbero piú senso.

- il livello *Read Committed* – Non permette ad una transazione di leggere o sovrascrivere gli aggiornamenti non ancora persistenti nella base di dati: ogni aggiornamento diventa visibile dopo il *commit*.

- il livello *Repeteable Read* – Non permette l'aggiornamento dei dati mentre una transazione li sta leggendo: una modifica diventa visibile solamente dopo la terminazione della transazione impegnata nella lettura. Questa proprietá garantisce che la ripetizione di una lettura all'interno di una transazione dará sempre lo stesso risultato: di solito é il livello di default all'interno dei *DBMS*.

- il livello *Serializable* – Garantisce ad una transazione il diritto esclusivo di accesso ai dati, provocandone il blocco degli aggiornamenti. Questo corrisponde al livello massimo di sicurezza.

In questa tesi si é scelto l'uso del livello *Repeteable Read*, il quale garantisce un buon compromesso fra le prestazioni del *DBMS* e le proprietá di isolamento.

4.3 Le tecniche di ripristino

Un sistema informatico é soggetto a malfunzionamenti e le tecniche per il ripristino degli errori sono state ampiamente studiate in letteratura: per approfondimenti rifarsi ad esempio a [Nel90], [PBFO02], [PBB+02] e [FP03].

Per poter studiare uno schema di ripristino é necessario fissare un modello astratto per l'esecuzione delle transazioni, introdotte nella sezione 4.2.1. In questo modello una transazione é sempre in uno dei seguenti stati:

- lo stato *active* é quello iniziale;

- lo stato *partially committed* é quello dopo l'ultima istruzione prima di entrare nella fase di commit;

- lo stato *failed* é quello immediamente prima dell'operazione di rollback;

- lo stato *aborted* é quello immediatamente successivo ad un'operazione di rollback;

- lo stato *committed* é quello dopo l'esecuzione del commit.

Descriviamo brevemente i cambiamenti di stato di una transazione T. Essa parte nello stato *active* ed al termine della sua esecuzione entra nello stato *partially committed*: il suo output non é ancora permanente. Se si é certi del suo buon esito, T entra nello stato *committed* altrimenti T entra nello stato *failed* e quindi in quello *aborted*. Ma questo modello non garantisce la consistenza in quanto possiamo scrivere i dati letti sulla stampante o sullo schermo: in caso di rollback tali scritture non possono essere annullate. Si puó ovviare a questo problema memorizzando *temporaneamente* i valori associati alle scritture esterne e rendendole permanenti alla conferma del commit: in letteratura questo meccanismo é noto come la tecnica del *file di log*. Ne esistono due versioni: il *log incrementale con modifiche differite*

e quello *incrementale con modifiche immediate*, le quali verranno descritte rispettivamente nelle sezioni 4.3.1 e 4.3.2. Nella sezione 4.3.3 vedremo una tecnica per poter velocizzare le operazioni di ripristino.

4.3.1 Il log incrementale con modifiche differite

L'idea base consiste nello scrivere le modifiche nel file di log e differire le operazioni di scrittura fino a quando la transazione non entra nello stato *partially–committed*: vediamo come si articola questa tecnica. Quando inizia una transazione T, viene scritto nel log un record del tipo $< T\ start >$. Se bisogna eseguire una scrittura, viene scritto nel log un nuovo record che memorizza il nome del dato e della transazione ed il nuovo valore. Infine, quando T entra nello stato *partially committed*, viene scritto nel log il record $< T\ commit >$. Se T effettua il commit, si usa la procedura di ripristino $redo(T)$ che applica le modifiche contenute nel log alla base di dati. A seguito di un malfunzionamento, si consulta il log per determinare quali transazioni debbano essere rieseguite: in questo caso il log deve contenere i due record $< T\ start >$ e $< T\ commit >$.

4.3.2 Il log incrementale con modifiche immediate

Una tecnica alternativa prevede l'applicazione degli aggiornamenti direttamente sulla base di dati e la gestione di un *log incrementale* con le modifiche allo stato del sistema. Tutte le operazioni di scrittura vengono eseguite immediatamente e si aggiunge un nuovo record di log che contiene il nome della transazione, il nome, il vecchio ed il nuovo valore del dato su cui si opera. Quando T entra nello stato *partially committed*, viene scritto sul log il record $< T\ commit >$. Il meccanismo di ripristino utilizza le procedure *undo* e *redo*. La procedura $undo(T)$ ripristina i vecchi valori dei dati modificati da T, mentre quella $redo(T)$ memorizza i nuovi valori dei dati aggiornati da T. A seguito di un malfunzionamento, il meccanismo di ripristino consulta il log per determinare quali transazioni debbano essere rieseguite e quali disfatte: la transazione T é *disfatta* se il log contiene solamente il record $< T\ start >$, mentre viene *rieseguita* se li contiene entrambi.

4.3.3 La tecnica del checkpoint

Come abbiamo visto nelle sezioni 4.3.1 e 4.3.2, a seguito di una caduta del sistema, é necessario consultare il file di log per determinare quali transazioni devono essere rieseguite: in generale é necessaria la scansione dell'intero file di log e questa operazione comporta un tempo di ricerca elevato che puó incidere sulle prestazioni del sistema. Per risolvere questo problema, si introduce il meccanismo del *checkpoint*, definito come segue. Durante l'esecuzione, il sistema mantiene il log con una delle modalitá descritte e periodicamente forza la scrittura di tutti i record attualmente esistenti: questa sequenza di

operazioni é nota in letteratura come *checkpoint*. Per segnalare questo fatto, si aggiunge nel file di log il record $<$ *checkpoint* $>$: sulla base di questo meccanismo, possiamo raffinare gli schemi di ripristino delle sezioni 4.3.1 e 4.3.2. Dopo che si é verificato un malfunzionamento, si esamina il log per individuare l'insieme delle transazioni che non sono ancora state gestite da un checkpoint: ci basta trovare la prima transazione T_i che ha iniziato la propria esecuzione in un istante di tempo precedente l'ultimo checkpoint. Tale transazione puó essere individuata eseguendo una ricerca nel log dell'ultimo record $<$ *checkpoint* $>$ disponibile e quindi del successivo record $< T_i$ *start* $>$. Non appena identificata la transazione T_i, bisogna operare su T_i e su quelle seguenti T_k secondo questo schema di funzionamento:

- eseguire $redo(T_k)$ per ogni transazione T_k per la quale compare il record $< T_k$ *commit* $>$ nel log;
- eseguire $undo(T_k)$ per ogni transazione T_k per la quale non compare il record $< T_k$ *commit* $>$ nel log.

4.4 Il sistema Oracle Berkeley DB

In questo paragrafo verrá introdotto il concetto di *DBMS embedded* nella sezione 4.4.1 e verranno illustrate le caratteristiche principali del sistema software *Oracle Berkeley DB* nella sezione 4.4.2: esso é stato inglobato nel prototipo di *DBMS* spaziale realizzato in questa tesi con il compito di memorizzare i cluster di nodi dell'indice spaziale. Quindi fa parte di un'architettura piú generale, che descriveremo nel capitolo 6.

4.4.1 Introduzione

La fruizione delle informazioni in una base di dati é facilmente descrivibile attraverso un'architettura di tipo *client–server*. Solitamente vi é un solo server che offre come servizio l'accesso alla base di dati interfacciandosi con il *DBMS*, mentre i processi client interagiscono con il server per accedere al database. In letteratura questa situazione viene descritta come *Comunicazione tra i processi* (nota anche con il termine *IPC*, dall'inglese *Inter–Process Communication*): questo meccanismo introduce un overhead nella comunicazione che é intollerabile per certe esigenze (basti pensare alle computazioni in tempo reale) e richiede l'esecuzione contemporanea di almeno due processi. Di solito il processo server é quello che richiede piú risorse di sistema per la sua esecuzione: quest'architettura non é adatta per dispositivi mobili o per sistemi con poca memoria primaria. La soluzione comunemente accettata é quella di usare un *DBMS* di tipo *embedded*.

Definizione 4.16. Un *DBMS* di tipo *embedded* é un gestore di basi di dati, il quale lavora nello stesso spazio di memoria di una certa applicazione.

Questo tipo di *DBMS* non necessita della *Comunicazione fra Processi* in quanto é inglobato nello stesso spazio di memoria di una applicazione e quindi le operazioni sono implementate in maniera molto efficiente. Solitamente si presenta come una libreria che permette all'applicazione di interfacciarsi con i dati attraverso delle *API* (dall'inglese *Application Program Interface*) e senza utilizzare un linguaggio *DML*, concetto introdotto nella sezione 4.1.4. Questo campo di ricerca si é rivelato molto promettente: per approfondimenti rifarsi ad esempio a [CW00], [BHS02], [Sel05], [Col06] e [SW06].

4.4.2 Le caratteristiche principali

In questa sezione metteremo in evidenza le caratteristiche piú importanti del sistema software *Oracle Berkeley DB*: questo componente é un esempio di *DBMS* di tipo *embedded*, concetto introdotto nella sezione 4.4.1. É stato realizzato inizialmente dalla *Sleepycat Software*: tale azienda é stata inglobata dalla *Oracle Corporation* nel 2006, che ora lo distribuisce con il suo marchio. Per approfondimenti rifarsi a [Bdb06d]. É distribuito sotto forma di una libreria disponibile in doppia licenza, sia di tipo *open–source* e sia di tipo commerciale, per garantirne la massima diffusione. Tale libreria fornisce delle semplici API per l'accesso e la gestione dei dati: per approfondimenti rifarsi a [Bdb06a]. Ufficialmente i linguaggi supportati sono *C*, *C++*, *Java*, *Perl* e *Python*: nel nostro caso abbiamo usato la versione *C++* per poterla facilmente inglobare nella libreria *OMSM*, di cui parleremo nel capitolo 6. Questo *DBMS* é supportato sia dai sistemi *UNIX* e sia da quelli *Microsoft Windows* quindi é estremamente portabile. É un sistema altamente configurabile ed é composto da vari sottosistemi, ognuno dei quali puó essere abilitato o disabilitato a seconda di quello che si vuole ottenere: la figura 4.3 ne mostra la struttura generale.

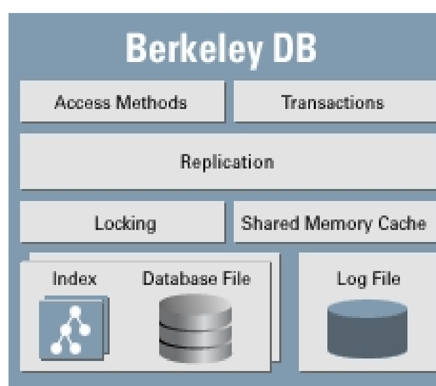


Figura 4.3: la struttura della libreria *Oracle Berkeley DB*

Il sottosistema *Access Methods* permette l'uso di particolari indici per velo-

cizzare l'accesso ai dati: abbiamo introdotto questo concetto nella sezione 4.1.3 e ne parleremo in maniera piú approfondita nel capitolo 5. Vi sono quattro metodi supportati, identificati da un certo flag:

- il metodo *B–tree* gestisce l'accesso ai dati tramite la struttura *B–albero* (in inglese *B–tree*), che descriveremo nel paragrafo 5.3: é identificato dal flag DB_BTREE;
- il metodo *Hash* gestisce l'accesso ai dati tramite un'implementazione dell'hashing lineare: é identificato dal flag DB_HASH;
- il metodo *Queue* gestisce l'accesso ai dati tramite la struttura *Coda* (in inglese *Queue*) assegnando come identificativo di ogni record il suo numero logico e richiedendo che tutti i record abbiano la stessa dimensione fissata: é identificato dal flag DB_QUEUE;
- il metodo *Recno* gestisce l'accesso ai dati in maniera molto simile a quello *Queue*, ma permette di avere record di dimensione variabile: é identificato dal flag DB_RECNO.

Il componente *Replication* fornisce il supporto per la replicazione dei dati: non lo abbiamo abilitato visto che l'uso delle tecniche di logging garantisce la tollerabilitá ai guasti del *DBMS* realizzato, il quale gestisce dati residenti sulla stessa macchina su cui viene eseguito. Invece la replicazione dei dati é particolarmente utile nell'approccio *client–server*. Per approfondimenti su questo componente rifarsi a [Bdb06b]. L'abilitazione degli altri componenti rende l'*Oracle Berkeley DB* un vero e proprio *DBMS* transazionale, proprietá ereditata anche dal sistema di memorizzazione dei dati sviluppato in questa tesi. Il componente *Transactions* si occupa dell'accesso concorrente al database offrendo il supporto per le transazioni semplici, definite dal modello che abbiamo introdotto nella sezione 4.2.1. Il componente *Locking* fornisce il supporto per il bloccaggio dei dati sfruttando la tecnica del *blocco a due fasi*, discussa nella sezione 4.2.2. Il componente *Shared Memory Cache* fornisce una cache in memoria primaria per velocizzare l'accesso ai dati, secondo lo schema che abbiamo presentato nel capitolo 3. Il componente *Log File* utilizza la tecnica del log con scritture differite e checkpoint per garantire la durabilitá delle modifiche: abbiamo parlato di queste tecniche rispettivamente nella sezione 4.3.1 ed in quella 4.3.3. Per approfondimenti su tali componenti rifarsi a [Bdb06c].

Questi componenti devono essere condivisi fra i processi che accedono al database: per semplificarne la gestione, viene introdotto il concetto di *ambiente* della base di dati (in inglese *database environment*). Questo concetto implica un cambiamento di prospettiva: non é importante solo la collezione di dati in sé, ma anche la tabella delle transazioni, quella dei lock, la memoria cache ed i file di log in modo da costituire un sistema il piú completo possibile. Ogni operazione viene effettuata sull'ambiente ed in questo modo

vengono condivisi i componenti citati. Nell'*Oracle Berkeley DB*, un ambiente della base di dati é costituito dal file che contiene il database vero e proprio, al quale si associa una directory dove vengono memorizzati i file di log e gli altri componenti. Per ragioni implementative, bisogna fornire il percorso assoluto di questa directory, la quale prende il nome di *environment directory*. Per garantire una maggiore probabilitá di successo nelle operazioni di ripristino, é buona norma fissare la directory di ambiente in un disco diverso da quello su cui si memorizza la base di dati vera e propria in quanto gli hard–disk hanno probabilitá diverse di subire un guasto.

Vediamo come gestire un ambiente nell'*Oracle Berkeley DB*: nella versione *C++* della libreria, un ambiente é descritto dalla classe *DbEnv*, la quale é definita nel file di intestazione *db_cxx.h*. Quest'ultimo contiene la definizione di tutte le classi usate dal *DBMS* e quindi deve essere incluso in ogni applicazione che ne fa uso. Per prima cosa dobbiamo creare un nuovo ambiente attraverso il metodo *DbEnv::open* che ha il seguente prototipo:

```
open( const char *envHome, u_int32_t flags, int mode )
```

Vediamo il significato dei parametri:

- *envHome* contiene il percorso assoluto dell'*environment directory*;
- *flags* contiene le impostazioni con cui creare l'ambiente, fornite attraverso l'operazione *or* bit a bit dei vari flag;
- *mode* fissa i permessi da attribuire all'*environment directory* attraverso le convenzioni di un sistema *UNIX*: in un sistema *Microsoft Windows* questo parametro viene banalmente ignorato.

In caso di errore, questo metodo solleva l'eccezione *DbException*. Le opzioni, fornite attraverso il parametro *flags*, abilitano i vari componenti:

- il flag *DB_INIT_LOCK* abilita il componente *Locking*;
- il flag *DB_INIT_LOG* abilita il componente *Log file*;
- il flag *DB_INIT_MPOOL* abilita il componente *Shared Memory Cache* e lo inizializza;
- il flag *DB_INIT_TXN* abilita il componente *Transactions*;
- il flag *DB_INIT_THREAD* abilita il supporto per il multi–threading: la sua presenza é necessaria per l'abilitazione degli altri componenti;

All'interno di un ambiente deve ovviamente trovare posto la base di dati, la quale viene memorizzata in un file binario con un certo finale, impostabile via software: abbiamo introdotto la definizione di *finale* nel paragrafo 3.2. Nella versione *C++* della libreria una base di dati é gestita dalla classe *Db*. Per prima cosa dobbiamo costruire un nuovo componente attraverso il costruttore:

```
Db ( DbEnv *dbenv, u_int32_t flags )
```

Vediamo il significato dei parametri:

- *dbenv* contiene l'ambiente a cui associare il database;
- *flags* contiene le impostazioni con cui creare questo componente: per gli usi piú comuni possiamo lasciarlo a 0.

Una volta creato il componente di classe *Db*, dobbiamo creare un database attraverso il metodo *Db::open* che ha il seguente prototipo:

```
open( DbTxn *txnid, const char *file, const char *database,
      DBTYPE type, u_int32_t flags, int mode )
```

In caso di errore, questo metodo solleva l'eccezione *DbException*. Vediamo il significato dei parametri:

- *txnid* contiene un riferimento alla transazione con la quale si vuole costruire il database in modo tale da garantire l'annullamento dell'operazione se questa non va a buon fine: se viene fornito il valore NULL viene disabilitata questa caratteristica;
- *file* contiene il nome del file binario che contiene il database: se non viene specificato il percorso assoluto, il file viene creato all'interno della directory di ambiente del database;
- *database* contiene il nome logico del database contenuto nel file binario in modo tale da memorizzare piú database nello stesso file: se viene fornito il valore NULL, si impone che un file contenga un solo database;
- *type* contiene il flag relativo al metodo di accesso scelto quindi sará uno fra DB_BTREE, DB_HASH, DB_QUEUE e DB_RECNO;
- *flags* contiene le molte opzioni con cui creare il database: per un elenco completo rifarsi a [Bdb06a];
- *mode* fissa i permessi da attribuire al file binario che contiene il database attraverso le convenzioni di un sistema *UNIX*: in un sistema *Windows* questo parametro viene banalmente ignorato.

La figura 4.4 mostra come sia possibile aprire un database all'interno di un ambiente, riassumendo i concetti descritti fino a questo punto. Per prima cosa impostiamo la directory */usr/envs/* come directory dell'ambiente *env* ed il flag risultante *envFlags*: questo permette l'attivazione dei componenti *Locking*, *Log File*, *Shared Memory Cache* e *Transactions* attraverso l'or bit a bit dei relativi flag. I permessi sulla directory di ambiente saranno quelli di default. Il primo blocco *try–catch* si occupa della creazione dell'ambiente

env: abbiamo tralasciato la gestione di un eventuale errore. Creato l'ambiente, dobbiamo aprire il database *db* all'interno di *env*. Impostiamo *prova.db* come il nome del file binario che dovrá memorizzare il database: visto che non abbiamo specificato il suo percorso assoluto allora sará creato all'interno della directory di ambiente. Tra le molte opzioni disponibili, scegliamo di forzare la creazione del database attraverso l'opzione DB_CREATE. Questo ci permette di creare il database: l'operazione viene svolta all'interno del secondo blocco *try–catch* dove abbiamo tralasciato la gestione di un eventuale errore. Come si puó notare, abbiamo disabilitato il rollback automatico della creazione del database ed abbiamo scelto di memorizzare un solo database logico nel file binario. Inoltre abbiamo forzato l'uso della struttura B–albero come metodo di accesso ai dati attraverso il flag DB_BTREE ed abbiamo fissato i permessi sul database ai valori di default.

```
#include <db_cxx.h>

// Percorso assoluto dell'ambiente
char *envHome = "/usr/envs/";

// Attivazione dei componenti
u_int32_t envFlags = DB_INIT_LOCK | DB_INIT_LOG |
                     DB_INIT_MPOOL | DB_INIT_THREAD |
                     DB_INIT_TXN;

DbEnv env(0);

// Costruzione dell'ambiente
try { env.open( envHome, envFlags, 0 ); }
catch( DbException dbe) { ... }

// Creazione del database
Db *db;
char *dbname = "prova.db";
u_int32_t dbFlags = DB_CREATE;

try {
     db = new Db( &env,0);
     db->open(NULL, dbname, NULL, DB_BTREE, dbFlags, 0); }
catch(DbException dbe) { ... }
```

Figura 4.4: costruzione di una base di dati all'interno di un ambiente

L'*Oracle Berkeley DB* non supporta un linguaggio di tipo *DML*, concetto introdotto nella sezione 4.1.4: il suo modello dei dati é molto semplice e

consiste in *record* formati da coppie del tipo *chiave/dato*. Questo schema é analogo ad una tabella del modello relazionale a due colonne, una per la chiave e l'altra per il dato: abbiamo accennato al modello relazionale nella sezione 4.1.2. Ognuna delle due parti del record é implementata dalla struttura *DBT* (dall'inglese *DataBase Thang*), descritta in figura 4.5.

```
typedef struct {
                    void * data;
                    u_int32_t size;
                    u_int32_t ulen;
                    u_int32_t dlen;
                    u_int32_t doff;
                    u_int32_t flags; } DBT;
```

Figura 4.5: la struttura *DataBase Thang* (*DBT*) in *C*

Come si puó notare dalla figura 4.5, il campo *data* puó essere di qualsiasi tipo mentre il campo *size* ne contiene la lunghezza totale in byte: in realtá la definizione vera e propria dei dati da memorizzare é lasciata al programmatore. Questi due campi sono piú che sufficienti alla gestione dei record, peró a volte ció non basta. Attraverso il campo *flags* possiamo agire su due aspetti della struttura: la gestione di una porzione di record e la memorizzazione dello stesso in un buffer fornito dall'utente. Possiamo abilitare la gestione di una porzione di record inserendo il flag DB_DBT_PARTIAL nel campo *flags*: in questo caso i campi *dlen* e *doff* contengono rispettivamente la lunghezza e l'offset della porzione interessata. L'*Oracle Berkeley DB* alloca automaticamente lo spazio necessario alla memorizzazione dei dati: possiamo forzarlo ad usare un buffer esterno di lunghezza *ulen* inserendo il flag DB_DBT_USERMEM nel campo *flags*. La struttura *DBT* é direttamente usata nel codice *C*: nella versione *C++* si usa un componente di classe *Dbt* la quale ne nasconde i dettagli implementativi. La gestione dei campi avviene attraverso dei metodi di tipo *get/set*: ad esempio il metodo *get_data* restituisce il campo *data* e quello *set_data* lo aggiorna. Gli altri metodi sono analoghi e possono essere studiati in [Bdb06a].

Le operazioni possibili sui record sono l'inserimento, la cancellazione e la ricerca dei dati: ognuna di queste puó avvenire all'interno di una *transazione*. Nella versione *C++* della libreria, una transazione é modellata dalla classe *DbTxn* e deve essere creata dall'ambiente della base di dati per garantire il controllo centralizzato sulla concorrenza. Possiamo avviare una transazione con il metodo *DbEnv::txn_begin* che ha il seguente prototipo:

```
txn_begin( DbTxn *parent, DbTxn **tid, u_int32_t flags )
```

Vediamo il significato dei parametri:

- *tid* contiene il riferimento alla nuova transazione che si sta cercando di avviare all'interno dell'ambiente;
- *parent* contiene il riferimento alla transazione di livello superiore rispetto a *tid* nel modello a transazioni nidificate, recentemente introdotto nell'*Oracle Berkeley DB*: il supporto é ancora a livello sperimentale e pertanto si consiglia di impostare questo parametro a NULL;
- *flags* contiene le impostazioni con cui avviare la nuova transazione: ad esempio possiamo impostarne il livello di isolamento, concetto che abbiamo introdotto nella sezione 4.2.3. Per default é attivo il livello 2 di isolamento, usato in questa tesi: gli altri vanno direttamente abilitati. Per un elenco completo delle impostazioni disponibili rifarsi a [Bdb06a].

In caso di errore, questo metodo solleva l'eccezione *DbException*. Riprendendo il modello a transazioni semplici studiato nella sezione 4.2.1, il metodo *txn_begin* fa le veci dell'istruzione *BeginTransaction*. Ovviamente una transazione puó eseguire l'istruzione *CommitTransaction* per consolidare le modifiche oppure la *RollbackTransaction* per annullarle. Nel primo caso si usa il metodo *DbTxn::commit()*, nel secondo quello *DbTxn::abort()*: entrambi i metodi sollevano l'eccezione *DbException* in caso di errore.

Le operazioni sui record sono implementate attraverso dei metodi invocabili su oggetti di classe *Db*: il metodo *Db::put* per l'inserimento di un nuovo record nel database, quello *Db::get* per la ricerca di un certo record e quello *Db:del* per la cancellazione di un certo record dal database. In caso di errore, questi metodi solleveranno l'eccezione *DbException*.

Il metodo *Db::put* permette l'inserimento nella base di dati di un nuovo record attraverso il seguente prototipo:

```
put ( DbTxn *txnid, Dbt *key, Dbt *data, u_int32_t flags )
```

Vediamo il significato dei parametri:

- *txnid* contiene la transazione che provvederá all'inserimento;
- *key* contiene la parte *chiave* del record da inserire;
- *data* contiene la parte *dato* del record da inserire;
- *flags* contiene le impostazioni con cui eseguire l'operazione. Ad esempio possiamo configurare l'*Oracle Berkeley DB* in modo da supportare record duplicati in quanto questi non sono supportati di default: per un elenco completo delle impostazioni rifarsi a [Bdb06a].

Useremo questo *DBMS* per memorizzare i cluster dei nodi dell'indice spaziale: in questo caso vogliamo che ogni cluster abbia un codice identificativo diverso quindi non abbiamo permesso record duplicati.

Il metodo *Db::get* permette la ricerca di un determinato record nel database attraverso il seguente prototipo:

```
get ( DbTxn *txnid, Dbt *key, Dbt *data, u_int32_t flags )
```

Vediamo il significato dei parametri:

- *txnid* contiene la transazione che provvederá alla ricerca;
- *key* contiene la parte *chiave* del record, sulla quale si basa la ricerca;
- *data* contiene la parte *dato* del record, che verrá caricata dal database;
- *flags* contiene le impostazioni con cui eseguire l'operazione di ricerca nel database: ad esempio possiamo forzare l'uso di un livello di isolamento diverso da quello della transazione *txnid*. Per un elenco completo delle impostazioni rifarsi a [Bdb06a].

Il metodo *Db::del* permette la cancellazione di un determinato record nel database attraverso il seguente prototipo:

```
del ( DbTxn *txnid, Dbt *key, Dbt *data, u_int32_t flags )
```

Vediamo il significato dei parametri:

- *txnid* contiene la transazione che provvederá alla cancellazione;
- *key* contiene la parte *chiave* del record da cancellare;
- *flags* deve essere impostato a 0 in quanto le versioni piú recenti della libreria non supportano piú alcuna impostazione.

La figura 4.6 mostra come poter inserire un record nel database sfruttando una transazione: per le altre operazioni la situazione é analoga. Per prima cosa, dobbiamo creare l'ambiente ed il database su cui inserire i dati: la figura 4.6 non spiega come farlo in quanto lo avevamo giá mostrato in quella 4.4. Dopo aver creato il database *db*, dobbiamo costruire entrambe le parti che compongono il nuovo record da inserire: l'oggetto *key* contiene la parte *chiave* del nuovo record (la stringa *"key"*), mentre l'oggetto *data* contiene la parte *dato* (la stringa *"data"*) del nuovo record. A questo punto dobbiamo impostare il contenuto e la lunghezza del buffer all'interno dei due componenti di classe *Dbt* usando rispettivamente i metodi *Db::set_data* e *set_size* sia sull'oggetto *key* e sia su quello *data*. Una volta impostato il nuovo record, dobbiamo creare la transazione che lo dovrá inserire nel database. La creazione e l'esecuzione della transazione seguono il modello che abbiamo introdotto nella sezione 4.2.1, opportunamente modificato con i metodi della libreria *Oracle Berkeley DB*. Per prima cosa dobbiamo creare la transazione *txn* attraverso l'invocazione del metodo *DbEnv::txn_begin*: una volta creata, invocheremo l'istruzione di inserimento del nuovo record. Se non ci sono errori durante questa operazione, verrá eseguita la fase di *commit* attraverso il metodo *DbTxn::commit*: in caso di errori verrá eseguita la fase di *rollback* attraverso il metodo *DbTxn::abort*.

```
#include <db_cxx.h>

   // Costruzione dell'ambiente
   DbEnv env(0);
   ...

   // Costruzione del componente Db
   Db *db;
   ...

   // Creazione degli oggetti Dbt
   Dbt key, data;
   key.set_data("key");
   key.set_size(strlen("key")+1);
   data.set_data("data");
   data.set_size(strlen("data")+1);

   // Apertura ed esecuzione della transazione
   DbTxn *txn;
   try {
          env.txn_begin( NULL, &txn, 0);
          db->put( txn, &key, &data, 0);
          txn->commit();
   }
   catch(DbException dbe) {
          // Bisogna annullare la transazione
          txn->abort();
   }
```

Figura 4.6: inserimento di un record all'interno di una transazione

Possiamo concludere che l'*Oracle Berkeley DB* é adatto quando si ha bisogno di gestire dati poco articolati ed in modalitá *embedded*: esso fornisce un motore transazionale di facile utilizzo, stabile e robusto. Per queste ragioni lo abbiamo usato in questa tesi: nel capitolo 6 descriveremo l'architettura in cui é stato inserito.

Capitolo 5

Le strutture di indicizzazione

> Il vero viaggio di scoperta non consiste nel cercare nuove terre, ma nell'avere nuovi occhi.
>
> *Voltaire*

Nella sezione 4.1.3 abbiamo accennato alla presenza di strutture dati, le quali facilitano la gestione delle informazioni contenute in un database: in letteratura sono note come *indici* o *strutture di indicizzazione*. Studieremo le caratteristiche generali di un indice nel paragrafo 5.1 in modo da essere in grado di studiarne alcuni esempi: nel seguito della trattazione illustreremo le caratteristiche degli indici che sono stati utilizzati durante la ricerca svolta in questa tesi. Nel paragrafo 5.2 vedremo alcune proprietá degli *Alberi Rosso–Neri*, indici particolarmente efficienti per la memorizzazione dei dati. Nel paragrafo 5.3 vedremo alcune caratteristiche dei *B–Alberi*, indici usati all'interno di un *DBMS* per l'accesso ai dati. Nel paragrafo 5.4 tratteremo le strutture dati *Octree*, un primo esempio di indici per la decomposizione spaziale di una certa entitá geometrica come una triangolazione, mentre nel paragrafo 5.5 studieremo le caratteristiche fondamentali della struttura *K–d tree*. I primi due indici sono di tipo tradizionale e quindi sono adatti alla memorizzazione di dati unidimensionali: in realtá non abbiamo implementato la struttura *B–Albero*, in quanto fornita dal sistema software *Oracle Berkeley DB*, del quale abbiamo parlato nella sezione 4.4.2. Invece, come vedremo nella sezione 6.3, abbiamo implementato una variante degli *Alberi Rosso–Neri*. Invece gli indici *Octree* e *K–d tree* sono indici spaziali e memorizzano dati multidimensionali e geometrici: come vedremo nel capitolo 6 essi sono stati implementati all'interno del prototipo di *DBMS* spaziale che abbiamo realizzato. Nella nostra ricerca useremo gli indici spaziali per memorizzare triangolazioni, ponendo particolare attenzione alle primitive di inserimento e di cancellazione in quanto siamo interessati a poter gestire in maniera dinamica i dati memorizzati attraverso delle modifiche successive. Tuttavia non ci interessa cancellare un singolo triangolo quanto piuttosto cancellarne un insieme per poi sostituirlo con altri complessi simpliciali: questo sará un fat-

to chiave quando andremo a parlare delle operazioni di cancellazione nei due indici spaziali. Le strutture dati *Octree* e *K–d tree* hanno un numero di nodi pari a quello degli elementi da indicizzare e quindi non sono adatti all'uso in memoria secondaria: per queste ragioni ne presenteremo delle varianti nel paragrafo 5.6, il cui utilizzo sará piú vantaggioso per i nostri scopi.

5.1 Introduzione

L'organizzazione *grezza* dei dati che abbiamo introdotto nella sezione 3.4.1 non é certamente adatta all'esecuzione di operazioni in quanto i dati non vengono organizzati in alcun modo. Il concetto di *database*, introdotto nel capitolo 4, costituisce una sua evoluzione: in entrambi i casi bisogna sviluppare tecniche che rendano piú efficiente la gestione di grosse quantitá di dati. L'operazione piú comune é l'*interrogazione*, un operazione che consente di individuare tutti i dati che soddisfano una certa proprietá: per risolverla piú facilmente, puó essere utile allocare delle strutture ausiliarie, note in letteratura con il nome di *indici* o *strutture di indicizzazione*: in questo paragrafo ne vedremo alcune proprietá. Per prima cosa forniremo una definizione generale di indice nella sezione 5.1.1, per poi occuparci di indici per dati spaziali nella sezione 5.1.2.

5.1.1 Una definizione di indice

Riprendiamo il modello dei dati introdotto nella sezione 4.1.2 dove abbiamo indicato un insieme di attributi delle entitá che sono di interesse durante l'operazione di ricerca con il termine *chiave di ricerca*. Solitamente, per migliorare l'accesso ai dati, questi ultimi vengono organizzati attraverso delle strutture ad albero: in questo modo si impone un criterio di allocazione dei dati e si gestiscono anche le operazioni di aggiornamento. L'idea che sta alla base di questo processo consiste nell'associare ai dati una sorta di *tabella* nella quale si memorizzano coppie del tipo (k_i, r_i) dove:

- k_i é un valore della chiave di ricerca su cui é costruito l'indice;
- r_i é un riferimento al record con valore di chiave k_i: puó essere l'indirizzo del blocco disco o della locazione di memoria primaria.

Le diverse tecniche di indicizzazione differiscono essenzialmente nel modo in cui gestiscono l'insieme delle coppie (k_i, r_i). Solitamente un indice si presenta come un albero: per fissarne la definizione dobbiamo assumere note alcune nozioni di teoria dei grafi, giá discusse nella sezione 4.2.2.

Definizione 5.1. Si dice *albero* un grafo G orientato nel quale due vertici qualsiasi sono connessi da un unico cammino.

In realtá ci interessa evidenziare un nodo in particolare che chiameremo *nodo radice*: questa esigenza ci conduce alla definizione di *albero radicato.*

Definizione 5.2. Si dice *albero radicato* una coppia $< T, r >$ dove T é un albero mentre r é un nodo di T. Solitamente il nodo r prende il nome di *radice* dell'albero T.

Dato un nodo N, é interessante definire i concetti di *padre* e di *figlio* di N.

Definizione 5.3. Siano $< T, r >$ un albero radicato, (u, v) un arco nell'albero T e γ il cammino da r a u. Se il cammino $\gamma^{'}$ da r a v si ottiene aggiungendo l'arco (u, v) a γ allora possiamo definire u come il nodo *padre* di v mentre v é il nodo *figlio* di u.

Bisogna osservare che la radice r non ha padre, mentre ogni nodo diverso da r ne ha uno solo. Inoltre ogni nodo puó avere un numero arbitrario di figli: si chiamano nodi *foglia* quelli privi di figli. Per mettere in relazione fra loro le chiavi di ricerca ed i riferimenti al relativo record abbiamo bisogno della nozione di *albero ordinato.*

Definizione 5.4. Un *albero ordinato* é un albero radicato nel quale é definito l'ordinamento totale dei figli di ogni suo nodo.

Particolarmente utile é la nozione di *sottoalbero* di un certo nodo.

Definizione 5.5. Siano $< T, r >$ un albero radicato, v un nodo di T diverso dalla radice r e u il nodo padre di v allora si dice *sottoalbero* del nodo v l'albero radicato $< v, T_v >$ dove T_v é un sottografo di T che contiene v e tutti i suoi nodi sono raggiungibili da v attraverso dei cammini che non contengono l'arco (u, v).

Nel seguito vedremo come l'efficienza delle operazioni sugli alberi dipenda dal bilanciamento della struttura: vediamo la definizione di *albero bilanciato.*

Definizione 5.6. Si dice *albero bilanciato* un albero ordinato nel quale la profonditá di ogni coppia dei sottoalberi dei suoi nodi differisce al piú di 1.

Queste definizioni ci permettono di fissare il concetto di *indice*: possiamo definirlo come un albero radicato ed ordinato in cui la relazione fra la chiave di ricerca contenuta in un certo nodo x e quelle memorizzate nei figli di x modella la tabella delle corrispondenze (k_i, r_i). Generalmente ogni nodo é implementato da una struttura dati che puó contenere un numero massimo di record: le posizioni inutilizzate potranno essere riempite in seguito all'aggiornamento dell'indice. Dunque possiamo utilizzare solamente una parte della memoria effettivamente allocata per gestire i record che si desiderano indicizzare. Gli indici devono garantire le seguenti proprietá:

- *bilanciamento* – l'indice deve essere un albero bilanciato in quanto questa proprietá garantisce l'efficienza delle interrogazioni;

- *occupazione minima* – é importante che si possa stabilire un limite inferiore alla quantitá di memoria allocata per ogni singolo nodo ed effettivamente usata per contenere i record che si vogliono indicizzare: questa proprietá evita una sottoutilizzazione delle risorse disponibili;

- *efficienza di aggiornamento* – i due requisiti precedenti devono essere soddisfatti garantendo al tempo stesso un costo limitato per le operazioni di aggiornamento.

5.1.2 Gli indici spaziali

Gli indici tradizionali, come gli *Alberi Rosso–Neri* ed i *B–alberi* che studieremo rispettivamente nelle sezioni 5.2 e 5.3, supportano dati unidimensionali mentre le recenti applicazioni, come quelle multidimediali o spaziali, trattano oggetti complessi e descrivibili come *punti* in uno spazio multidimensionale: questo fatto ha reso necessario lo sviluppo di nuove strutture di indicizzazione. In letteratura ne sono state sviluppate varie: per approfondimenti rifarsi a [TH81], [GG98], [Mar99], [SP00], [Kor00], [Sam03], [NS04] e [NS06].

In questa tesi focalizzeremo la nostra attenzione sugli *indici spaziali*, i quali gestiscono dati di tipo geometrico ad esempio punti nello spazio euclideo $\mathbb{E}^3$. Il loro campo di indicizzazione é legato alle caratteristiche spaziali degli oggetti memorizzati: un indice tradizionale si basa sull'*ordinamento totale* del dominio del campo *chiave*. Ad esempio, in un *Albero Rosso–Nero* non viene imposta alcuna *vicinanza spaziale* fra le chiavi: deve essere garantita soltanto la relazione d'ordine esistente fra la chiave memorizzata in un certo nodo e quelle memorizzate nei suoi sottoalberi, come vedremo nel paragrafo 5.2. Al contrario, in un indice spaziale é necessario utilizzare una relazione d'ordine fra le chiavi che preservi la prossimitá geometrica degli oggetti memorizzati: oggetti *vicini* nello spazio dovrebbero essere vicini anche nell'indice. Possiamo osservare che le varianti di indici spaziali si differenziano sulla relazione d'ordine usata per decomporre una certa entitá geometrica. Vi sono due approcci possibili:

- costruire una chiave unidimensionale a partire dall'oggetto geometrico ed usarla all'interno di un indice tradizionale, ad esempio in un *B–albero*, una struttura dati che descriveremo nel paragrafo 5.3;

- definire delle nuove strutture di accesso che garantiscano la relazione di vicinanza fra gli oggetti memorizzati.

In letteratura il secondo approccio si é dimostrato il piú promettente, dividendo le strutture di accesso in due categorie:

- strutture di tipo *space–driven* in cui lo spazio di riferimento é suddiviso in celle piú o meno regolari in maniera indipendente dalla distribuzione

degli oggetti, i quali vengono associati alle celle in base a qualche criterio geometrico: un esempio é dato dall'indice *Octree*, che studieremo nel paragrafo 5.4;

- strutture di tipo *data–driven* in cui si partiziona l'insieme degli oggetti tenendo conto della loro distribuzione nello spazio di riferimento: un esempio é dato dall'indice *K–d tree*, che studieremo nel paragrafo 5.5.

La maggior parte dei sistemi che forniscono un qualche tipo di accesso a dati spaziali supporta piú tipologie di strutture per garantire un'ampia gamma di scelte. Possiamo ricordare il sistema *Oracle Spatial*, il quale supporta la struttura dati *quadtree* e quella *R–tree*: per approfondimenti rifarsi a [FB74], [Gut84], [BGK04] e [OSP05]. Sui dati memorizzati possiamo eseguire alcune tipologie di interrogazioni geometriche, per le esigenze di questa tesi sono particolarmente importanti:

- l'interrogazione *Point–Query*, la quale controlla se un dato punto appartiene all'indice;
- l'interrogazione *Range–Query*, la quale restituisce tutti i punti dell'indice che appartengono ad una data finestra.

Queste primitive sono entrambe supportate dal prototipo di *DBMS* spaziale che descriveremo nel capitolo 6. Solitamente, l'esecuzione delle interrogazioni spaziali richiede l'esecuzione di operazioni geometriche complesse, le quali possono coinvolgere grandi quantitá di dati in memoria secondaria. L'indice dovrebbe garantire la proprietá di *dinamicitá*, cioé deve facilitare gli aggiornamenti e le interrogazioni adattandosi alle variazioni di numero, complessitá e distribuzione spaziale degli oggetti memorizzati. Dunque gioca un ruolo importante la distribuzione spaziale degli oggetti: é *difficile* ottenere strutture che operino in maniera dinamica con una qualunque distribuzione dei dati. La maggior parte degli indici garantiscono, in media, prestazioni ottimali cioé con un numero logaritmico di operazioni di I/O: questa proprietá non viene solitamente garantita nel caso peggiore a differenza di quanto avviene nella struttura tradizionale *B–Albero*, come vedremo nel paragrafo 5.3. In letteratura esistono delle strutture che garantiscono prestazioni ottimali anche nel caso peggiore ma sono *complicate* da gestire: un esempio di tali strutture é dato da quella *BK–d tree*, introdotta in [PAAV03].

La decomposizione spaziale trova applicazione in moltissimi campi: ad esempio puó migliorare gli algoritmi di routing in una rete wireless, per approfondimenti rifarsi a [XLT04] e [LZ05]. In questo capitolo vedremo alcune strutture di indicizzazione: per ulteriori esempi di indici spaziali rifarsi a [Gut84], [HSW89], [BKSS90], [HP94], [CPZ97], [CRZP97], [KS97a], [KS97b], [ZSAR98], [RMF^{+}00], [BNM03], [CLC03], [ASI05], [Mur05] e [Sam06]. In letteratura sono state sviluppate molte strutture di decomposizione spaziale,

ognuna con pregi e difetti. Questa proliferazione di strutture di indicizzazione ovviamente non facilita la modularitá nelle applicazioni in quanto un cambiamento di indice potrebbe condurre ad uno stravolgimento pressoché completo dell'applicazione stessa. Dunque l'obiettivo della ricerca attuale su questo argomento é la creazione di una architettura comune (solitamente si usa il termine inglese *framework* a tale proposito) con cui sia possibile implementare varie strutture di indicizzazione spaziale senza stravolgere l'intera applicazione. La struttura *GiST* é certamente l'esempio di framework per la gestione di indici spaziali piú noto in letteratura: essa é in grado di simulare il comportamento della maggior parte delle strutture di indicizzazione attraverso la modifica di una serie di metodi predefiniti. Lo studio approfondito di questa struttura esula dagli scopi di questa tesi: per maggiori dettagli rifarsi a [HNP95] e [Kor00]. Il *GiST Indexing Project* era un progetto promosso dalla *Berkeley University*, il quale si proponeva lo sviluppo di una libreria, nota come *libGiST*, in cui veniva implementata la struttura *GiST*: per approfondimenti rifarsi a [HKS+00]. Purtroppo questo progetto é stato abbandonato e la libreria prodotta, seppur molto valida, non é pertanto funzionante sui compilatori piú recenti, come il *GNU C++ Compiler 4.0* oppure il *Microsoft Visual Studio .NET*: per approfondimenti su questi compilatori rifarsi a [GCC87], [Net02], [MVS03] e [Mon04]. Per queste ragioni non abbiamo usato la libreria *libGiST* in questa tesi ed invece abbiamo sviluppato l'architettura per la memorizzazione di dati che descriveremo nel capitolo 6.

5.2 La struttura dati *Albero Rosso–Nero*

La struttura dati *Albero Rosso–Nero*, abbreviata nel seguito della trattazione con *RB–Albero* (dall'inglese *Red–Black Tree*), é una variante degli alberi binari di ricerca in cui si garantisce che l'altezza della struttura sia logaritmica nel numero di elementi memorizzati. Per una trattazione approfondita degli alberi binari di ricerca rifarsi a [Knu73] e [CLR90]: nel seguito li indicheremo con *BST* (dall'inglese *Binary Search Tree*) ed assumeremo che il loro funzionamento sia noto. In questo paragrafo vedremo alcuni aspetti della struttura dati *RB–Albero*: per approfondimenti rifarsi a [Bay72], [GS78] e [CLR90]. Nella sezione 5.2.1 ne fisseremo la definizione, mentre nella sezione 5.2.2 studieremo le operazioni di rotazione, le quali permettono il bilanciamento della struttura. Questa caratteristica permette l'implementazione efficiente delle operazioni di ricerca, inserimento e cancellazione, che analizzeremo rispettivamente nelle sezioni 5.2.3, 5.2.4 e 5.2.5.

5.2.1 Definizione della struttura dati

Le operazioni su una struttura *BST* di altezza h possono essere implementate in tempo $\mathcal{O}(h)$: nel caso peggiore le prestazioni di questo albero diventano

paragonabili a quelle di una lista concatenata. La struttura *RB–Albero* risolve questo problema: essa é un albero binario di ricerca in cui viene garantito il bilanciamento della struttura in modo tale che l'altezza h sia sempre logaritmica nel numero degli elementi memorizzati. Il concetto su cui si basa questa struttura é quello di *colore* di un certo nodo: é un campo binario che puó assumere i due valori *RED* e *BLACK*, dai quali deriva il nome di questo albero. Vediamo la definizione formale della struttura *RB–Albero*.

Definizione 5.7. Un *RB–Albero* é una struttura dati *BST* che soddisfa le seguenti proprietá:

- ciascun nodo σ dell'albero contiene i campi *color*, *key*, *left*, *right* e *parent* dove *color* memorizza il colore di σ, *key* contiene il dato memorizzato in σ, *left* e *right* sono rispettivamente i puntatori al sottoalbero sinistro e destro del nodo σ mentre *parent* é il puntatore al padre di σ;
- se un figlio o il padre di un nodo non esistono, allora il corrispondente campo puntatore assume il valore speciale *NIL*;
- il colore di ogni nodo é *RED* oppure *BLACK*;
- ciascuna foglia di tipo *NIL* ha colore *BLACK*;
- se un nodo ha colore *RED* allora tutti i suoi figli hanno colore *BLACK*;
- ogni cammino da un nodo ad una foglia contiene lo stesso numero di nodi di colore *BLACK*.

La figura 5.1 mostra un esempio di struttura *RB–Albero*: la colorazione dei nodi é ispirata dai valori comunemente usati nella definizione.

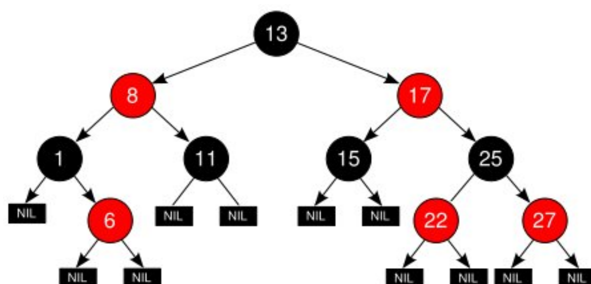


Figura 5.1: esempio di una struttura *RB–Albero*. In un'eventuale stampa in bianco e nero si osserva che i nodi con il colore piú chiaro sono quelli *RED*, quelli piú scuri sono i nodi *BLACK*: questa convenzione vale anche per le altre figure riguardanti gli *RB–Alberi*.

In [CLR90] viene dimostrato il seguente lemma che mostra la proprietá di bilanciamento della struttura *RB–Albero*.

Lemma 5.1. *Una struttura* RB–Albero *con n nodi interni possiede al piú un'altezza di* $2\log(n+1)$.

Nel seguito, assumeremo che un certo nodo sia modellabile come il record *RBNode*, il quale contiene i campi introdotti nella definizione 5.7. La figura 5.2 ne mostra la definizione: il campo *key* contiene il dato da memorizzare, *color* contiene il colore del nodo mentre *left*, *right* e *parent* contengono rispettivamente i puntatori al figlio sinistro, al figlio destro ed al padre del nodo modellato dal record *RBNode*.

```
record RBNode {
       RBNode left,right,parent;
       Key key;
       Color color; }
```

Figura 5.2: definizione in pseudocodice del record *RBNode*

Nel seguito della trattazione indicheremo con *RBTree* il record che contiene solamente il puntatore al nodo radice.

5.2.2 L'operazione di rotazione

In un *RB–Albero*, le operazioni di inserimento e di cancellazione ne modificano la struttura e quindi é possibile che l'albero prodotto non soddisfi piú le proprietá della definizione 5.7. In questo caso bisogna ripristinarne le proprietá attraverso la primitiva di *rotazione*, introdotta in [AVL62]: essa opera a livello locale sull'albero e non modifica l'ordinamento delle chiavi. Ve ne sono di due tipi, note in letteratura con il nome di *rotazione sinistra* e di *rotazione destra*: la figura 5.3 mostra come queste due operazioni siano simmetriche.

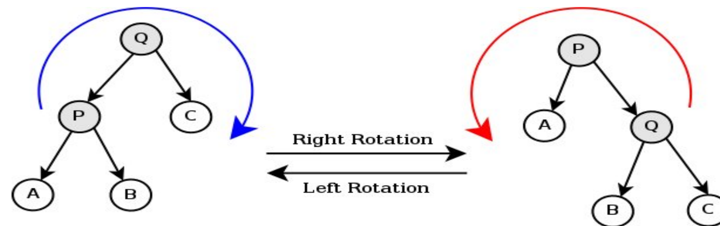


Figura 5.3: esempio di rotazione sinistra e destra

La figura 5.4 mostra lo pseudocodice della primitiva *leftRotate* che implementa la rotazione sinistra. La rotazione destra é implementata dalla primitiva *rightRotate* che si ottiene da quella *leftRotate* per simmetria, come dimostra

la figura 5.3. Entrambe le operazioni sono eseguite in tempo $\mathcal{O}(1)$ visto che modificano solo i puntatori all'interno del nodo.

```
void leftRotate( RBTree T, RBNode P ) {
     Q = P.right;
     P.right = Q.left;
     if ( Q.left != NIL ) Q.left.parent = P;
     Q.parent = P.parent;
     if ( P.parent == NIL ) T.root = Q;
     else if ( P == P.parent.left ) P.parent.left = Q;
     else P.parent.right = Q;
     Q.left = P;
     P.parent = Q; }
```

Figura 5.4: pseudo-codice per l'operazione di rotazione sinistra

5.2.3 L'operazione di ricerca

L'operazione di ricerca di un nodo in un *RB–Albero* non é altro che l'omonima operazione sulle strutture *BST*: come noto, in questo caso la complessitá di questa operazione vale $\mathcal{O}(h)$ dove h é l'altezza dell'albero. In un *RB–Albero* vale il lemma 5.1 e quindi la struttura é bilanciata: questo vuol dire che l'operazione di ricerca ha complessitá $\mathcal{O}(\log n)$.

5.2.4 L'operazione di inserimento

L'inserimento di un nuovo nodo in un *RB–Albero* puó essere eseguito in tempo $\mathcal{O}(\log n)$ dove n é il numero dei nodi memorizzati. L'idea é la seguente: si inserisce un nodo N in un *RB–Albero* come se quest'ultimo fosse un *BST* e si imposta il colore di N al valore *RED*. Questa operazione potrebbe non garantire le proprietá di un *RB–Albero*, violando la condizione che riguarda il colore dei figli di un nodo di colore *RED*: in tal caso é necessario sistemare l'albero modificato. Per poter gestire questa situazione, é necessario fissare i concetti di *fratello*, *zio* e *nonno* di un certo nodo.

Definizione 5.8. Dato un certo nodo N, diverso dalla radice, si definisce *fratello* di N un nodo S che abbia lo stesso padre di N: in realtá questo nodo puó non esistere ed in questo caso assume il valore speciale *NIL*.

Definizione 5.9. Dato un certo nodo N, diverso dalla radice, si definisce *zio* di N il fratello del padre del nodo N: in realtá questo nodo puó non esistere ed in questo caso assume il valore speciale *NIL*.

Definizione 5.10. Dato un certo nodo N, diverso dalla radice, si definisce *nonno* di N il padre del padre del nodo N, se esiste.

Questi concetti ci saranno molto utili nel seguito e per questa ragione é possibile definirli attraverso delle primitive. Il fratello di un nodo si puó trovare attraverso la primitiva *sibling*, il cui pseudocodice é mostrato in figura 5.5.

```
RBNode sibling ( RBNode N ) {
     if( N == N.parent.left ) return N.parent.right;
     else return N.parent.left }
```

Figura 5.5: pseudocodice della primitiva *sibling*

Invece lo zio di un nodo si puó trovare attraverso la primitiva *uncle*, il cui pseudocodice é mostrato in figura 5.6.

```
RBNode uncle ( RBNode N ) {
     if( N.parent == N.parent.parent.left )
             return N.parent.parent.right;
        else return N.parent.parent.left; }
```

Figura 5.6: pseudocodice della primitiva *uncle*

Infine il nonno di un nodo si puó trovare attraverso la primitiva *grandparent*, il cui pseudocodice é mostrato in figura 5.7.

```
RBNode grandparent ( RBNode N ) { return N.parent.parent; }
```

Figura 5.7: pseudocodice della primitiva *grandparent*

É banale osservare che queste primitive hanno complessitá $\mathcal{O}(1)$. Per semplicitá, dato un nodo N, indichiamo con la lettera P il padre di N (cioé $N.parent$), con G il nonno, con U lo zio e con S il fratello.

Fissati questi concetti, possiamo verificare le proprietá dell'*RB–Albero*: in letteratura si dimostra l'esistenza di cinque possibili casi da gestire. Non é detto che ognuno di questi casi contenga una violazione delle proprietá introdotte nella definizione 5.7: nel seguito ne vedremo una schema di risoluzione a cascata. Nel primo caso si ha che il nuovo nodo N é la radice dell'albero e come tale va colorata in $BLACK$. La figura 5.8 mostra lo pseudocodice della primitiva *insert_case1*, la quale si occupa della risoluzione di questo problema: come si puó notare, viene passato il controllo alla gestione del caso successivo se non ricadiamo in questa situazione. Nel secondo caso si ha che il padre P del nuovo nodo N é di colore $BLACK$ quindi sono banalmente verificate le proprietá introdotte nella definizione 5.6 e non vi é alcuna violazione. La figura 5.9 mostra lo pseudocodice della primitiva *insert_case2*, la quale in realtá passa il controllo alla gestione del caso successivo se non ricadiamo in questa situazione.

```
void insert_case1( RBNode N, RBTree T ) {
     if( P == NIL ) N.color = BLACK
     else insert_case2(N,T); }
```

Figura 5.8: pseudocodice della primitiva *insert_ case1*

```
void insert_case2(RBNode N,RBTree T) {
     if(P.color == RED) insert_case3(N,T); }
```

Figura 5.9: pseudocodice della primitiva *insert_ case2*

Nei casi seguenti, riassunti in figura 5.10, si assume che il nuovo nodo N abbia un *padre* P di colore *RED*.

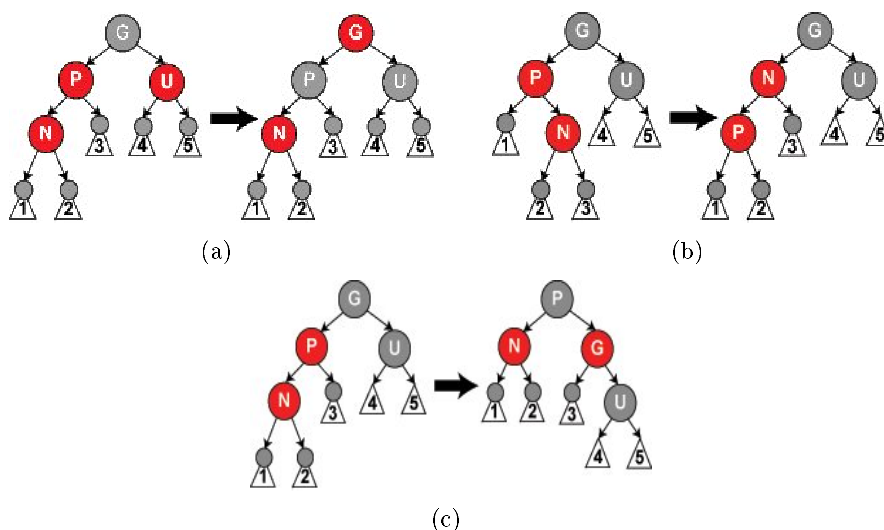


Figura 5.10: la gestione del terzo (a), del quarto (b) ed del quinto (c) caso nell'operazione di inserimento di un nuovo elemento in un *RB–Albero*.

Nel terzo caso i nodi P e U sono di colore *RED*, come mostra la figura 5.10(a). In questa situazione é possibile ricolorare P e U di colore *BLACK* e G di colore *RED*: in questo modo tutti i cammini a partire da un dato nodo verso le foglie contengono lo stesso numero di nodi di colore *BLACK*. Putroppo il nodo G puó violare le proprietá di un *RB–Albero* in quanto potrebbe avere un padre di colore *RED*: per risolvere questo problema, bisogna agire ricorsivamente sul nodo G. Viene attivata una serie di chiamate ricorsive alle primitive di gestione, la quale costituisce la parte piú *costosa* nell'algoritmo di inserimento di un nuovo elemento in un *RB–Albero*: ne parleremo in maniera approfondita quando discuteremo la complessitá di questa operazione. Questa situazione viene gestita dalla primitiva *insert_ case3*, il cui pseudoco-

dice é mostrato in figura 5.11: come si puó notare, viene passato il controllo alla gestione del caso successivo se non ricadiamo in questa situazione.

```
void insert_case3( RBNode N, RBTree T ) {
       if( U!= NIL and U.color == RED) {
               P.color = BLACK;
               P.color = BLACK;
               G.color = RED;
               insert_case1(G,T); }
       else insert_case4(N,T); }
```

Figura 5.11: pseudocodice della primitiva *insert_ case3*

Nel quarto caso il nodo P é di colore *RED*, ma il nodo U é di colore *BLACK*. Inoltre il nuovo nodo N é il figlio destro di P, il quale é a sua volta il figlio sinistro di G, come mostra la figura 5.10(b), quindi non é vero che tutti i figli di un nodo *RED* sono di colore *BLACK*. Per risolvere questo problema, bisogna effettuare una rotazione su G, a seconda della disposizione dei nodi, per scambiare di posto il nuovo nodo N ed il nodo padre P. Questa situazione viene gestita dalla primitiva *insert_ case4*, il cui pseudocodice é mostrato in figura 5.12: come si puó notare, viene passato il controllo alla gestione del caso successivo in quanto non é detto che siano ancora verificate le proprietá introdotte nella definizione 5.7.

```
void insert_case4( RBNode N, RBTree T ) {
      if( N == P.right and P == G.left ) {
                   leftRotate(T,P);
                   N = N.left; }
      else if( N == P.left and P == G.right ) {
                   rightRotate(T,P);
                   N = N.right;  }
      insert_case5( N,T ); }
```

Figura 5.12: pseudocodice della primitiva *insert_ case4*

Nel quinto caso si ha una situazione simile a quella precedente, come si puó notare dalla figura 5.10(c): il nuovo nodo N é il figlio sinistro di P, il quale é a sua volta il figlio sinistro di G. Per risolvere questo problema, bisogna effettuare una rotazione su G, a seconda della disposizione dei nodi, per scambiare di posto il nuovo nodo N ed il nodo padre P: se N é il figlio sinistro di P bisogna eseguire una rotazione destra su G altrimenti una rotazione sinistra su G. Questa situazione viene gestita dalla primitiva *insert_ case5*, il cui pseudocodice é mostrato in figura 5.13.

```
void insert_case5( RBNode N, RBTree T) {
          P.color = BLACK;
          G.color = RED;
          if( N == P.left and P == G.left) rightRotate(G,T);
          else leftRotate(G,T); }
```

Figura 5.13: pseudocodice della primitiva *insert_ case5*

Proviamo a calcolare la complessitá della primitiva di inserimento: il primo passo consiste nell'inserimento del nodo N come se fossimo in un BST quindi la complessitá deve essere almeno $\mathcal{O}(\log n)$ dove n é il numero degli elementi nell'albero. Inoltre dobbiamo valutare la complessitá delle operazioni necessarie alla risoluzione delle violazioni: l'unico caso non banale é il terzo, nel quale vengono effettuate alcune chiamate ricorsive necessarie a far *salire* un nodo fino alla radice quindi il numero massimo di chiamate é pari all'altezza dell'albero. Terminata questa fase, la violazione viene gestita da uno dei restanti casi, ognuno dei quali ha complessitá $\mathcal{O}(1)$: questo fatto ci permette di concludere che la complessitá dell'operazione é $\mathcal{O}(\log n)$.

5.2.5 L'operazione di cancellazione

La cancellazione di un nodo da una struttura *RB–Albero* puó essere descritta come una modifica della primitiva di cancellazione in un BST. Questa operazione potrebbe violare le proprietá di un *RB–Albero*, introdotte nella definizione 5.7 e quindi dobbiamo sistemare la struttura dell'albero in maniera analoga a quanto fatto nella sezione 5.2.4. Il caso piú semplice é quello in cui si cerca di cancellare un nodo di colore RED in quanto lo si puó sostituire con uno dei suoi figli di colore $BLACK$: questa operazione non modifica le proprietá di un *RB–Albero*. Un altro caso banale si ha quando il nodo da cancellare é di colore $BLACK$ mentre suo figlio é di colore RED: in questo caso é sufficiente ricolorare il nodo figlio di colore $BLACK$ per soddisfare ancora le proprietá della definizione 5.7. Ricadiamo nel caso piú complesso quando sia il nodo da cancellare e sia il nodo figlio sono di colore $BLACK$. Per prima cosa dobbiamo sostituire il nodo da eliminare con il nodo figlio: chiameremo N il figlio, S il suo nuovo fratello, P il nuovo padre di N, S_L il figlio sinistro di S ed S_R il figlio destro di S. In maniera analoga a quanto osservato nella sezione 5.2.4, potrebbe non essere soddifatta la proprietá sul numero dei nodi di colore $BLACK$ in tutti i cammini da un qualsiasi nodo alle foglie sottostanti: in letteratura si dimostra l'esistenza di sei possibili casi da gestire. Non é detto che ognuno di questi casi contenga una violazione delle proprietá introdotte nella definizione 5.7: nel seguito ne vedremo una schema di risoluzione a cascata.

Nel primo caso si ha che N é la nuova radice della struttura, la quale

é ancora un *RB–Albero* e quindi non vi é alcuna violazione. La figura 5.14 mostra lo pseudocodice della primitiva *remove_ case1*, la quale in realtá passa il controllo alla gestione del caso successivo se non ricadiamo in questa situazione.

```
void remove_case1( RBNode N, RBTree T) {
     if( P != NIL) remove_case2(N,T); }
```

Figura 5.14: pseudocodice della primitiva *remove_ case1*

I casi seguenti sono riassunti in figura 5.15.

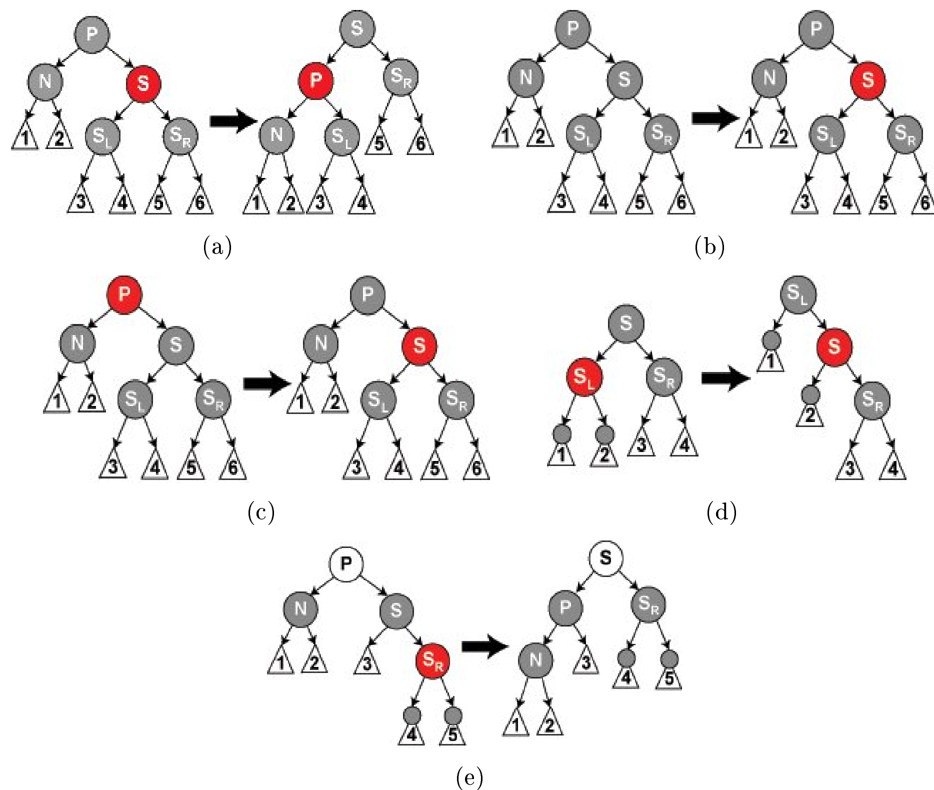


Figura 5.15: la gestione del secondo (a), del terzo (b), del quarto (c), del quinto (d) e del sesto (e) caso nell'operazione di cancellazione di un elemento da un *RB–Albero*

Nel secondo caso si ha che il nodo S é di colore *RED* e quello P é di colore *BLACK*, come mostra la figura 5.15(a). In questa situazione dobbiamo invertire i colori dei nodi P ed S ed eseguire una rotazione su P, a seconda della disposizione dei nodi. Questa situazione viene gestita dalla primitiva *remove_ case2*, il cui pseudocodice é mostrato in figura 5.16: come si puó

notare, viene passato il controllo alla gestione del caso successivo in quanto non é detto che siano ancora verificate le proprietá di un *RB–Albero*.

```
void remove_case2( RBNode N, RBTree T) {
     if( S.color == RED ) {
          P.color = RED;
          S.color = BLACK;
          if( N == P.left ) leftRotate(T,P);
          else rightRotate(T,P); }
     remove_case3(N,T); }
```

Figura 5.16: pseudocodice della primitiva *remove_ case2*

Nel terzo caso si ha che i nodi P, S ed i figli di S sono di colore *BLACK*, come mostra la figura 5.15(b). In questa situazione modifichiamo il colore del nodo S in *RED*: la proprietá sul numero dei nodi *BLACK* puó essere ancora violata e quindi dobbiamo far ripartire il processo di bilanciamento. Viene attivata una serie di chiamate ricorsive alle primitive di gestione che costituisce la parte piú *costosa* nell'algoritmo di cancellazione di un elemento da un *RB–Albero*: ne parleremo in maniera approfondita quando discuteremo la complessitá di questa operazione. Questa situazione viene gestita dalla primitiva *remove_ case3*, il cui pseudocodice é mostrato in figura 5.17: come si puó notare, viene passato il controllo alla gestione del caso successivo se non ricadiamo in questa situazione oppure viene fatto ripartire il processo di bilanciamento.

```
void remove_case3( RBNode N, RBTree T) {
     if( P.color == BLACK and S.color == BLACK and
         S.left.color == BLACK and S.right.color == BLACK) {
           S.color = RED;
           remove_case1( P, T); }
    else remove_case4(N); }
```

Figura 5.17: pseudocodice della primitiva *remove_ case3*

Nel quarto caso si ha che il nodo P é di colore *RED* mentre i nodi S ed i suoi figli sono di colore *BLACK*, come mostra la figura 5.15(c): in questo caso i cammini dalla radice ad un nodo foglia non contengono lo stesso numero di nodi di colore *BLACK*. Per risolvere questo problema dobbiamo scambiare le colorazioni dei nodi S e P. Questa situazione viene gestita dalla primitiva *remove_ case4*, il cui pseudocodice é mostrato in figura 5.18: come si puó notare, viene passato il controllo alla gestione del caso successivo se non ricadiamo in questa situazione.

```
void remove_case4( RBNode N, RBTree T) {
    if( P.color == RED and S.color == BLACK and
        S.left.color == BLACK and S.right.color == BLACK) {
            S.color = RED;
            P.color = BLACK; }
    else remove_case5(N); }
```

Figura 5.18: pseudocodice della primitiva *remove_ case4*

Nel quinto caso si ha che il nodo S ed il suo figlio destro S_R sono di colore *BLACK* mentre il figlio S_L é di colore *RED*, come mostra la figura 5.15(d). In questa situazione si opera una rotazione su S, a seconda della disposizione dei nodi: S_L diventa il padre di S e quindi possiamo scambiare le colorazioni di S e di S_L. Non viene modificato il numero di nodi di colore *BLACK*, ma dobbiamo ancora riordinare la struttura: in [CLR90] si dimostra che ció verrá fatto con la trattazione del sesto caso. Questa situazione viene gestita dalla primitiva *remove_ case5*, il cui pseudocodice é mostrato in figura 5.19.

```
void remove_case5( RBNode N, RBTree T) {
     if( N == P.left and S.color == BLACK and S.color == BLACK
         and S.left.color == RED and S.right.color == BLACK) {
               S.color = RED;
               S.left.color = BLACK;
               rightRotate(T,S); }
     else if( N == P.right and S.color == BLACK and
              S.right.color == RED and S.left.color == BLACK) {
                    S.color = RED;
                    S.right.color = BLACK;
                    leftRotate(T,S); }
     remove_case6(N,T); }
```

Figura 5.19: pseudocodice della primitiva *remove_ case5*

Nel sesto caso il nodo S é di colore *BLACK*, il figlio S_R é di colore *RED* mentre N é il figlio sinistro del nodo P il cui colore é indifferente: la figura 5.15(e) lo mostra in colore bianco. In questa situazione si opera una rotazione su P, a seconda della disposizione dei nodi: S diventa il padre di P e del figlio destro di S. Ora possiamo scambiare le colorazioni di P ed S, modificando il colore di S_R in *BLACK*. Cosí facendo, i cammini passanti per N incontrano un nodo di colore *BLACK* permettendo la cancellazione di un nodo in quei percorsi. Ma se un cammino non incontra il nodo N puó avvenire una delle due seguenti situazioni:

- viene attraversato il nuovo fratello di N e quindi anche i nodi S e P: in questo caso abbiamo semplicemente scambiato la colorazione di questi nodi e quindi il numero di nodi di colore $BLACK$ non varia;

- viene attraversato il nuovo zio di N cioé il nodo S_R e quindi anche il nodo S, il quale eredita il colore del padre. Inoltre abbiamo modificato la colorazione del nodo S_R da RED a $BLACK$ perció il numero di nodi di colore $BLACK$ non cambia.

Quindi le proprietá della definizione 5.7 sono ancora verificate: come si puó notare dalla figura 5.15(e), il colore del nodo P é irrilevante. Questa situazione viene gestita dalla primitiva *remove_ case6*, il cui pseudocodice é mostrato in figura 5.20.

```
void remove_case6( RBNode N, RBTree T) {
     S.color = P.color;
     P.color = BLACK;
     if( N == P.left) {
           S.right.color = BLACK;
           leftRotate( T,P ); }
     else {
           S.left.color = BLACK;
           rightRotate( T,P ); }
}
```

Figura 5.20: pseudocodice della primitiva *remove_ case6*

Proviamo a calcolare la complessitá dell'operazione di cancellazione in un *RB–Albero*: come abbiamo visto, la prima fase della primitiva consiste in una cancellazione del tutto analoga a quella effettuata in un BST quindi la sua complessitá é almeno $\mathcal{O}(\log n)$ dove n é il numero di elementi memorizzati nell'albero. Inoltre dobbiamo valutare la complessitá delle operazioni necessarie alla risoluzione delle violazioni: l'unico caso non banale é il terzo in cui vengono effettuate alcune chiamate ricorsive necessarie a far *salire* un nodo fino alla radice quindi il numero massimo di chiamate é pari all'altezza dell'albero. Terminata questa fase, la violazione viene gestita da uno dei restanti casi ognuno dei quali ha complessitá $\mathcal{O}(1)$. Questo ci permette di concludere che la complessitá dell'operazione di cancellazione di un elemento in un *RB–Albero* é $\mathcal{O}(\log n)$.

5.3 La struttura dati *B–Albero*

La struttura dati *B–Albero* (nota in letteratura con *B–tree*) é un albero bilanciato usato per facilitare l'accesso ai dati in memoria secondaria. In

questo paragrafo ne vedremo alcuni aspetti principali: per approfondimenti rifarsi a [BM72], [Com79] e [CLR90]. Nella sezione 5.3.1 ne fisseremo la definizione mentre studieremo l'operazione di ricerca di un elemento nella sezione 5.3.2, quella di inserimento nella sezione 5.3.3 ed infine quella di cancellazione nella sezione 5.3.4.

5.3.1 Definizione della struttura dati

Come giá accennato, un *B–tree* é un albero bilanciato che viene usato nell'indicizzazione di dati in memoria secondaria: vediamone la definizione.

Definizione 5.11. Un *B–Albero* di ordine m con $m \geq 3$ é un albero bilanciato che soddisfa le seguenti proprietá:

- ogni nodo contiene al piú $m - 1$ elementi;
- ogni nodo contiene almeno $\lceil \frac{m}{2} \rceil - 1$ elementi;
- il nodo radice puó contenere anche un solo elemento;
- ogni nodo contenente j elementi ha $j + 1$ figli;

Ogni nodo ha una struttura del tipo

$$c_0 \; k_1 \; c_1 \ldots \; c_{j-1} \; k_j \; c_j$$

dove j é il numero degli elementi contenuti nel nodo, $k_1, \; \ldots \; , k_j$ sono le chiavi ordinate e memorizzate nel nodo per le quali vale $k_1 < k_2 < \ldots < k_j$ e $c_0, \; \ldots \; , c_j$ sono i riferimenti ai nodi figli del nodo in questione. Questa struttura rimane invariata anche se il nodo in questione é una foglia: in questo caso cambia il ruolo di $c_1, \ldots, c_j$. Se il nodo non é una foglia, possiamo fissare $K(c_i)$ come l'insieme delle chiavi memorizzate nel sottoalbero di radice c_i con $i = 1, \ldots, j$ ed allora valgono le seguenti proprietá del nodo:

- $\forall y \in K(c_0).y < k_1$;
- $\forall y \in K(c_i).k_i < y < k_{i+1}, \; i = 1, \ldots, j - 1$
- $\forall y \in K(c_j).y > k_j$

Se il nodo é una foglia, ovviamente non si avranno figli ed i puntatori $c_1, \ldots, c_j$ verranno usati per accedere direttamente ai dati memorizzati nella memoria secondaria.

Dalle proprietá della definizione 5.11 é banale derivare che tutte le foglie compaiono allo stesso livello in un *B–Albero* e quindi l'albero é bilanciato. Inoltre, in [BM72] si dimostra il seguente lemma il quale esprime in maniera formale la proprietá di bilanciamento della struttura *B–Albero*.

Lemma 5.2. *Una struttura* B–Albero *di ordine* m *contenente* n *chiavi possiede un'altezza compresa fra* $\log_m(N+1)$ *e* $1+\log_{\lceil \frac{m}{2} \rceil}(\frac{N+1}{2})$.

Come conseguenza di questo lemma si avrá che una strutturá *B–Albero* sará sempre bilanciata e che la sua altezza sará logaritmica nel numero di chiavi. La figura 5.21 mostra una struttura *B–Albero* di ordine 5.

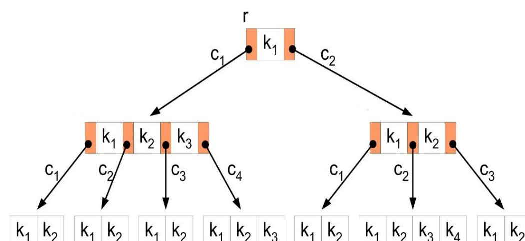


Figura 5.21: esempio di una struttura dati *B–Albero* di ordine 5.

Come abbiamo giá visto nella sezione 4.4.2, questa struttura viene usata dall'*Oracle Berkeley DB* per accedere alle coppie memorizzate nel database. In realtá ne viene usata una versione particolare, detta *B–Albero paginato*. L'implementazione delle primitive su un *B–Albero* puó essere onerosa e provocare l'aumento dell'altezza della struttura, un parametro critico nell'efficienza delle operazioni: approfondiremo questi aspetti nelle sezioni 5.3.2, 5.3.3 e 5.3.4. In caso di un insieme molto esteso di chiavi, la quantitá di memoria occupata é elevata e quindi un *B–Albero* non é completamente memorizzabile in memoria. Per ovviare a questo problema si suddivide l'albero in *pagine* di lunghezza fissata, le quali vengono memorizzate in memoria secondaria e caricate dinamicamente in *RAM*: ognuna di loro contiene contiene un *B–Albero*. In questo modo si potrá caricare direttamente il nodo radice in memoria primaria e poi caricare dinamicamente le pagine, se richiesto. Questo procedimento si puó estendere anche ad altri tipi di indici: l'architettura di memorizzazione dei dati spaziali, che descriveremo nel capitolo 6, sfrutta indici paginati. Nel seguito della trattazione, assumeremo che la struttura *B–Albero* sia completamente contenuta in memoria e che un suo generico nodo sia modellato dal record *BNode*, il quale contiene i campi introdotti dalla definizione 5.11. Inoltre verrá introdotto il record *BTree* per memorizzare solamente il puntatore al nodo radice del *B–Albero*.

5.3.2 L'operazione di ricerca

Lo scopo dell'operazione di ricerca é banalmente quello di controllare se una data chiave sia memorizzata o meno all'interno di un *B–Albero*: il record *Key* modella la chiave memorizzata all'interno dell'albero. La figura 5.22 mostra

lo pseudocodice della primitiva *search* che implementa questa operazione dove, come si puó notare, vengono usate alcune funzioni ausiliarie: la funzione *isLeaf(p)* controlla se un nodo p é una foglia, quella *belongs(y,p)* controlla se la chiave y é fra quelle memorizzate in un certo nodo p mentre la funzione *forward(p,y)* restituisce il nodo figlio c_i di un nodo p se esistono due chiavi k_i e k_{i+1} in $K(c)$ tali che $k_i < y < k_{i+1}$, il valore speciale *NIL* altrimenti. Inoltre si assume che le chiavi $k_1, \ldots, k_j$ ed i nodi figli $c_0, \ldots, c_j$ di un certo nodo siano memorizzate nei vettori c e k all'interno di p.

```
bool search( Key y, BNode p) {
        if( belongs(y,p) ) return true;
        else if( isLeaf(p) ) return false;
        else if( y < k[1] ) return search( y, p.c[0] );
        else if ( y > k[j] ) return search( y, p.c[j] );
        else return search( y,forward(p,y) ); }
```

Figura 5.22: l'algoritmo di ricerca di una certa chiave in un *B–Albero*

L'idea dell'algoritmo é la seguente: dato un nodo, si esegue la ricerca fra le chiavi contenute in esso fino a determinarne la presenza o l'assenza. Se la chiave non viene trovata, si prosegue nell'unico sottoalbero del nodo corrente che potrebbe contenere la chiave richiesta. La ricerca termina quando si raggiunge un nodo foglia: se la chiave non vi appartiene vuol dire che non é presente nell'albero. Se la chiave non appartiene alla struttura, bisogna visitare un cammino dalla radice ad una foglia prima di poter concludere: la lunghezza di questo cammino é l'altezza h dell'albero. Ricordando il lemma 5.2, possiamo concludere che la complessitá é logaritmica nel numero di elementi memorizzati nell'indice.

5.3.3 L'operazione di inserimento

Lo scopo dell'operazione di inserimento é banalmente quello di inserire una nuova chiave in una struttura dati *B–Albero*. Prima di descriverne l'algoritmo, dobbiamo fissare il concetto di *nodo pieno*.

Definizione 5.12. Un nodo di una struttura *B–Albero* di ordine m si dice *pieno* se contiene $m - 1$ elementi.

Come primo passo, dobbiamo verificare se la chiave é giá presente nell'albero attraverso la primitiva *search* che abbiamo introdotto nella sezione 5.3.2. Se la chiave é presente, non dobbiamo inserirla: in caso contrario dobbiamo inserirla nel nodo foglia P in cui é fallita la ricerca. Ricordiamo che, secondo la definizione 5.11, un nodo di un *B–Albero* di ordine m é una foglia se i suoi puntatori c_i con $i = 1, \ldots, j$ e $\lceil \frac{m}{2} \rceil - 1 \leq j \leq m - 1$ non memorizzano i riferimenti ai figli, ma servono per accedere direttamente a quanto memorizzato in memoria secondaria. Possiamo distinguere due casi:

- il nodo P non é pieno quindi possiamo inserire la nuova chiave;
- il nodo P é pieno e quindi dobbiamo attivare un processo di *suddivisione* (noto in letteratura con il termine inglese *splitting*) che puó propagarsi, nel caso peggiore, fino al nodo radice.

Proviamo a descrivere l'operazione di *splitting*: sia P il nodo pieno in cui dobbiamo inserire la nuova chiave. Supponiamo per un attimo che l'inserimento sia permesso e quindi il nuovo nodo P avrá la seguente struttura:

$$c_0 k_1 c_1 \ldots k_g c_g k_{g+1} c_{g+1} \ldots k_m c_m$$

dove $g = \lceil \frac{m}{2} \rceil - 1$ e la chiave k_{g+1} é detta *separatore*. Questo nodo dovrá essere suddiviso in due ulteriori nodi, come segue:

- nel nodo P rimangono gli elementi $c_0 k_1 c_1 \ldots k_g c_g$;
- si crea un nuovo nodo $P^{'}$ contenente gli elementi $c_{g+1} k_{g+2} c_{g+2} \ldots k_g c_g$.

A questo punto dobbiamo sistemare il separatore k_{g+1} a seconda della natura del nodo P. Se P non é il nodo radice, si inseriscono nel nodo padre di P (che indicheremo con Q) gli elementi $k_{g+1} p^{'}$, dove $p^{'}$ é il puntatore al nodo $P^{'}$. Ovviamente se Q é a sua volta pieno allora il processo di suddivisione deve essere ripetuto su Q. Se P era il nodo radice, allora si crea un nuovo nodo radice $P^{''}$ contenente $p k_{g+1} p^{'}$ dove p é il puntatore al nodo P.

Studiamo la complessitá di questa primitiva: il caso peggiore si verifica quando l'operazione di splitting si propaga fino alla radice. In questo caso dobbiamo attraversare un cammino dalla radice ad una foglia e poi ripercorrerlo nel senso opposto: ad ogni passo dobbiamo scrivere due nodi ed infine aggiungere la nuova radice. Dunque queste operazioni hanno complessitá lineare nell'altezza dell'albero e quindi logaritmica nella cardinalitá dell'indice, ricordando i risultati del lemma 5.2.

5.3.4 L'operazione di cancellazione

Lo scopo dell'operazione di cancellazione é banalmente quello di cancellare una chiave da una struttura *B–Albero*: per poterne descrivere l'algoritmo é necessario fissare il concetto di *nodo in underflow*.

Definizione 5.13. Un nodo di una struttura *B–Albero* di ordine m si dice *in underflow* se contiene meno di $\lceil \frac{m}{2} \rceil - 1$ elementi.

Come primo passo dobbiamo verificare, attraverso la primitiva *search* che abbiamo introdotto nella sezione 5.3.2, se la chiave k da cancellare é presente nell'albero. Se la chiave k é memorizzata nella struttura, dobbiamo cancellarla dal nodo Q che la contiene. Se Q é una foglia, possiamo attivare la cancellazione vera e propria: chiameremo questo nodo Q_k. Se Q non é

una foglia, dobbiamo sostituire k con una delle chiavi adiacenti che si trovano in una foglia (generalmente la chiave con cui sostituire k é quella successiva nell'ordinamento delle chiavi in un *B–Albero*) e ripetere ricorsivamente questo procedimento fino a quando la chiave k non viene memorizzata in una foglia. Alla fine di questa prima fase, la chiave k é memorizzata in una foglia Q_k, nella quale é possibile implementare l'operazione di cancellazione. Possono verificarsi due casi:

- la foglia Q_k non é in underflow dopo la cancellazione della chiave k;
- la foglia Q_k é in underflow dopo la cancellazione della chiave k e pertanto bisogna attivare il processo di *concatenazione* oppure quello di *bilanciamento* a seconda della situazione;

Descriviamo il processo di concatenazione, il quale opera in maniera esattamente inversa al processo di suddvisione: se Q_k é in underflow ed esiste un nodo adiacente $P^{'}$ con al piú $\lfloor \frac{m}{2} \rfloor$ chiavi allora si possono concatenare i due nodi Q_k e $P^{'}$, se questi contengono complessivamente meno di $m-1$ chiavi. Nel seguito assumeremo che P' sia il fratello destro di Q_k: se é quello sinistro l'implementazione delle operazioni é completamente analoga. Inoltre chiameremo R il padre dei nodi Q_k e $P^{'}$. L'operazione di concatenazione consiste nei seguenti passi:

- eliminare dal nodo R la chiave k_t di valore intermedio fra quelle di Q_k e quelle di $P^{'}$;
- sostituire i nodi Q_k e $P^{'}$ con un nuovo nodo P, il quale contiene tutte le chiavi di Q_k (dalle quali abbiamo cancellato k) seguite da quella k_t e poi tutte quelle di $P^{'}$: il numero totale delle chiavi di P é $m-1$.

L'eliminazione della chiave k_t dal nodo padre R puó innescare a sua volta una concatenazione che si puó propagare ricorsivamente fino alla radice la quale puó essere eliminata, riducendo l'altezza dell'albero.

Il processo di concatenazione non é sempre possibile in quanto potrebbe non esistere un fratello $P^{'}$ con le caratteristiche richieste: in questo caso bisogna attivare il processo di *bilanciamento*. Esso coinvolge i nodi Q_k e $P^{'}$, un fratello di Q_k, i quali possiedono complessivamente piú di $m-1$ chiavi: siccome Q_k é in underflow, possiamo eliminare una chiave da $P^{'}$ senza che questi vada in underflow. Nel seguito assumeremo che P' sia il fratello destro di Q_k: se é quello sinistro, l'implementazione delle operazioni é completamente analoga. Inoltre chiameremo R il padre dei nodi Q_k e $P^{'}$. L'operazione di bilanciamento consiste nei seguenti passi:

- individuare la chiave k_t del nodo R compresa fra i puntatori Q_k e $P^{'}$;
- cancellare la chiave k_t dal nodo R e spostarla nel nodo Q_k, dal quale abbiamo rimosso la chiave k;

- individuare la chiave di valore minimo del nodo P' e spostarla nel nodo R in modo da sostituire la chiave k_t.

Come si puó notare, il bilanciamento interessa anche il nodo R: in realtá non ne viene modificato il numero degli elementi quindi l'operazione non si propaga ai livelli superiori. Studiamo la complessitá di questa primitiva: il caso peggiore si verifica quando tutti i nodi del percorso di ricerca devono essere concatenati ad eccezione dei primi due. Inoltre il figlio del nodo radice coinvolto nel percorso viene modificato ed anche la radice dunque si leggono $2h - 1$ nodi e se ne riscrivono $h - 1$ perció la complessitá é $\mathcal{O}(h)$. Possiamo concludere che la complessitá della primitiva di cancellazione é logaritmica nella cardinalitá dell'indice, ricordando il lemma 5.2.

L'analisi di questi aspetti conclude lo studio dei due indici di tipo tradizionale usati in questa tesi: nei paragrafi 5.4 e 5.5 studieremo rispettivamente le proprietá degli indici *Octree* e *K–d tree*.

5.4 La struttura dati *Octree*

La struttura dati *Octree* é un esempio di indice spaziale di tipo *space–driven* ed é usata per memorizzare una decomposizione gerarchica di una porzione dello spazio euclideo $\mathbb{E}^3$, estendendo al caso tridimensionale la struttura *Quadtree*, introdotta in [FB74]. La struttura dati *Octree* costituisce un primo esempio di indice per dati multidimensionali ed é molto usata nelle applicazioni: nella sezione 2.4.2 abbiamo accennato al suo utilizzo nel calcolo *FEM*, per approfondimenti rifarsi a [TOL02], [TOL03] e [PAGL06]. Un'applicazione interessante é data dal rendering e dalla visualizzazione di volumi: per approfondimenti rifarsi a [IA95], [CGP03], [Kno06] e [KWPH06]. In questo paragrafo ne vedremo le caratteristiche piú importanti: per approfondimenti rifarsi a [Sam84], [Sam90b] e [Sam06]. Nella sezione 5.4.1 ne fisseremo la definizione mentre presenteremo gli algoritmi di inserimento e cancellazione rispettivamente nelle sezioni 5.4.2 e 5.4.3. Inoltre vedremo nella sezione 5.4.4 come implementare le interrogazioni spaziali supportate.

5.4.1 Definizione della struttura dati

L'idea che sta alla base di questa struttura é la suddivisione ricorsiva del dominio Γ di interesse in regioni disgiunte, ma che siano adiacenti fra loro. Ovviamente, possiamo avere molte tipologie di suddivisione e ció determina le differenti versioni della struttura: per una loro panoramica rifarsi a [Sam06]. In questa tesi useremo strutture dati *Octree* in cui la suddivisione di Γ verrá effettuata con piani paralleli a quelli cartesiani. Per poter descrivere questo tipo di struttura, dobbiamo introdurre la nozione di *rettangolo d–dimensionale*, noto anche come *iper–rettangolo*: vediamone la definizione.

Definizione 5.14. Si definisce *rettangolo d–dimensionale* o *iper–rettangolo* l'insieme R_d dato dal prodotto cartesiano di d intervalli del tipo $I_i = [a_i, b_i]$ dove $a_i, b_i \in \mathbb{R}$ per $i = 1, \ldots, d$.

Come si puó notare dalla definizione 5.14, la nozione di iper–rettangolo generalizza quelle di rettangolo e di parallelepipedo, rispettivamente nello spazio bidimensionale e tridimensionale. Ovviamente diremo che un punto d–dimensionale $(x_1, \ldots, x_d)$ appartiene a R_d se e soltanto se per ogni $i = 1, \ldots, d$ vale $x_i \in I_i$, cioé la i-esima coordinata di p appartiene all'i–esimo intervallo di R_d. Possiamo modellare l'insieme R_d con il record *Rectangle*, il quale contiene gli intervalli I_i per $i = 1, \ldots, d$.

Di grande importanza é poi la definizione del record *Point*, che serve un modellare un generico punto immerso in $\mathbb{E}^d$, con d generico: la figura 5.23 mostra la definizione di questo record in pseudo–codice dove il campo d indica la dimensione del punto, mentre quello *coords* ne contiene le coordinate euclidee.

```
record Point {
      int d;
      double coords[d]; }
```

Figura 5.23: pseudocodice del record *Point*, il quale modella un generico punto immerso nello spazio euclideo $\mathbb{E}^d$.

Inoltre, nel seguito, useremo la funzione:

```
bool contains( Rectangle r, Point p )
```

per controllare se un dato rettangolo r contiene un certo punto p. Questa operazione é definita se il punto p é d–dimensionale ed r é un rettangolo d–dimensionale. La complessitá di questa primitiva é $\mathcal{O}(d)$.

Vediamo ora come poter utilizzare questo concetto nella suddivisione del dominio Γ di interesse. Supponiamo di avere un dominio Γ ed un punto q appartenente a Γ, entrambi di dimensione generica d. L'idea consiste nel suddividere il dominio Γ secondo piani paralleli a quelli coordinati e passanti per il punto q: ovviamente qui si intendono *iperpiani* di dimensione d generica. Se il dominio di partenza Γ era a sua volta un iper–rettangolo di dimensione d si ottiene una decomposizione di Γ in rettangoli d–dimensionali disgiunti, ma che condividono una faccia. Ogni iper–rettangolo appartenente a questa decomposizione viene chiamato *quadrante d–dimensionale* o *iper–quadrante*. In letteratura, solitamente si indicano gli iper–quadranti bidimensionali semplicemente con il termine *quadrante* e quelli tridimensionali con il termine *ottante*. La figura 5.24 mostra un esempio di decomposizione di un certo dominio Γ nel caso bidimensionale: il dominio é un quadrato di lato 100 ed i punti sono interpretabili come note localitá geografiche. Questi dati sono

ottenuti tramite gli strumenti forniti dal progetto *Spatial Index Demos*, il quale permette lo studio di svariati indici spaziali nel caso bidimensionale attraverso delle applet *Java*: per approfondimenti si rimanda a [BS06].

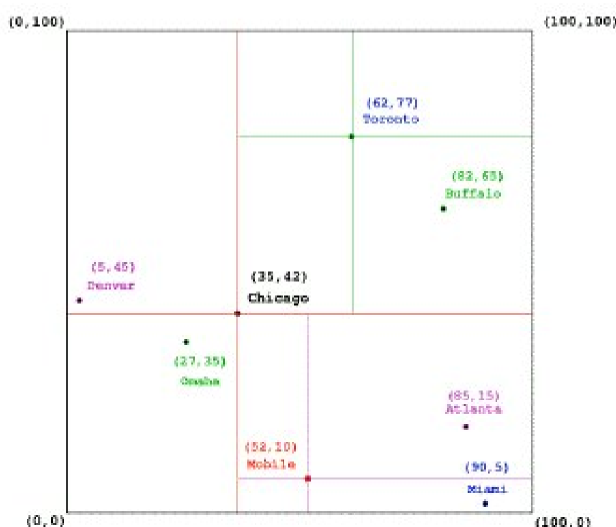


Figura 5.24: un esempio di struttura *Quadtree*.

La suddivisione che abbiamo appena descritto ci permette di definire un tipo di indice spaziale, il quale generalizza la struttura *Quadtree*, introdotta in [FB74] e definita nello spazio euclideo $\mathbb{E}^2$: vediamone la definizione.

Definizione 5.15. Supponiamo di voler decomporre un dominio $\Gamma \subseteq \mathbb{E}^d$ allora possiamo definire un *Octree* su Γ come un albero che soddisfa le seguenti proprietá:

- il nodo radice descrive il dominio Γ;
- ogni nodo N descrive un iper–rettangolo $\gamma_N \subseteq \Gamma$;
- ogni nodo contiene un punto $q \in \gamma_N$: di conseguenza $q \in \Gamma$;
- ogni nodo N non foglia contiene $l = 2^d$ puntatori ai figli $c_0, c_1, \ldots, c_l$, i quali descrivono rispettivamente gli iper–rettangoli $\gamma_0, \gamma_1, \ldots, \gamma_l$ appartenenti alla decomposizione di γ_N rispetto al punto q memorizzato nel nodo N. Questa decomposizione avviene secondo le regole che abbiamo discusso in precedenza.

Come si puó facilmente notare, ogni dominio Γ viene suddiviso ricorsivamente in 2^d iper–quadranti e quindi non é conveniente usare questa struttura per

dimensioni elevate in quanto l'occupazione spaziale del nodo cresce in maniera esponenziale. Tuttavia, nel prototipo di *DBMS* spaziale, che descriveremo nel capitolo 6, abbiamo implementato la versione generale di questo tipo di strutture. La figura 5.25(a) mostra la rappresentazione tridimensionale della decomposizione di tipo *Octree* di un certo dominio Γ: le linee in nero rappresentano il dominio iniziale mentre quelle in blu ne definiscono la decomposizione. Invece la figura 5.25(b) mostra l'albero che descrive questa decomposizione. Come abbiamo giá avuto modo di osservare, la decomposizione in figura 5.25(a) é stata effettuata rispetto al baricentro di Γ: in realtá la suddivisione puó avvenire rispetto ad un qualsiasi punto p che sia interno al dominio.

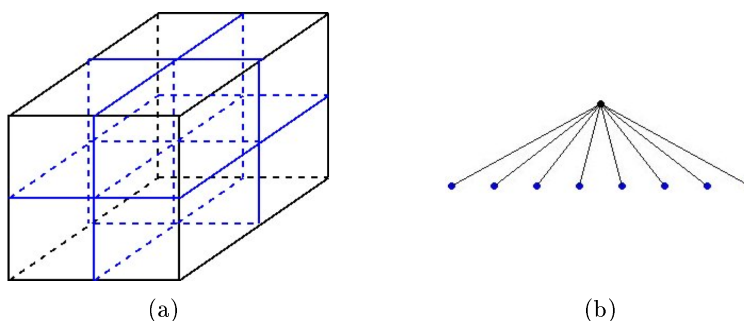


Figura 5.25: esempio di decomposizione in ottanti di una porzione di spazio euclideo $\mathbb{E}^3$ (a) e relativa struttura dati *Octree* (b). In questo caso specifico, la decomposizione viene effettuata rispetto al baricentro del dominio di partenza: in realtá puó avvenire rispetto un qualsiasi punto p del dominio.

In letteratura esistono molte varianti di questa struttura: se conosciamo a priori tutti i punti che vogliamo indicizzare, come in questa tesi, é utile usare come dominio il piú piccolo iper–rettangolo d–dimensionale che contiene al suo interno tutti i dati che vogliamo indicizzare: questo insieme é solitamente noto come *MBB* dei punti, dall'inglese *Minimal Bounding–Box*. Nel seguito del paragrafo assumeremo che un nodo di una struttura *Octree* venga descritto dal record *OctreeNode* il cui pseudocodice é mostrato in figura 5.26, la quale mostra la struttura nel generico spazio euclideo $\mathbb{E}^d$, ponendo $l = 2^d$ per indicare il numero dei figli.

```
record OctreeNode {
            Point q;
            OctreeNode children[l];
            Rectangle ret; }
```

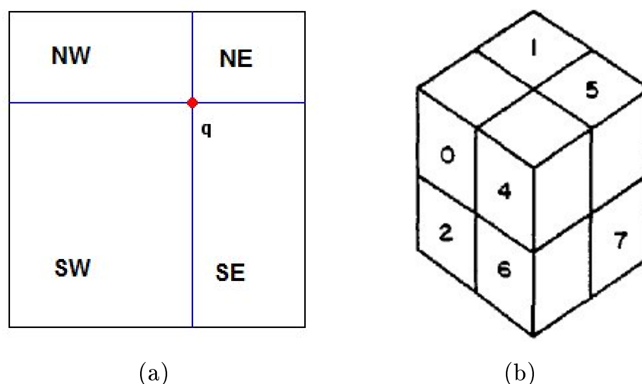
Figura 5.26: pseudocodice del record che modella un nodo di una struttura *Octree*, definita nello spazio euclideo $\mathbb{E}^d$

Nel record *OctreeNode* il campo q memorizza il punto contenuto nel nodo, il vettore *children* memorizza gli l figli del nodo mentre *ret* memorizza l'iper–quadrante sotto esame. Questo record modella direttamente un nodo non foglia: in maniera analoga a quanto fatto per gli *RB–alberi* nel paragrafo 5.2, possiamo usare il valore speciale *NIL* per segnalare l'assenza di un determinato figlio. In questo modo un nodo foglia avrá il puntatore $children[j]$ pari al valore speciale *NIL* per ogni indice $j = 0, \ldots, l-1$. Per creare un nodo foglia possiamo usare la primitiva

```
OctreeNode leafOctree( Point q )
```

la quale si occupa di memorizzare il punto q ed impostare correttamente i valori dei nodi figli: la complessitá di questa primitiva é $\mathcal{O}(2^d)$, dove d é la dimensione del punto q. Per i dati rappresentati in questa tesi vale $d = 3$ e quindi si ottiene banalmente che la sua complessitá é $\mathcal{O}(1)$.

A questo punto dobbiamo capire come identificare gli iper–quadranti: per semplicitá partiremo dall'analisi del caso bidimensionale per poi estenderla a quello generale. Nello spazio bidimensionale i quadranti vengono individuati rispetto ad un punto q di riferimento, il quale avrá coordinate (x_q, y_q): la figura 5.27(a) mostra i quattro quadranti in cui viene suddiviso un dominio $\Gamma \subseteq \mathbb{E}^2$ rispetto al punto q.



(a) (b)

Figura 5.27: suddivisioni dello spazio $\mathbb{E}^2$ in quadranti (a) e dello spazio $\mathbb{E}^3$ in ottanti (b). Nel caso tridimensionale la decomposizione é stata effettuata rispetto al baricentro del dominio Γ per ragioni di chiarezza figurativa: in realtá la suddivisione puó avvenire rispetto ad un qualsiasi punto q che sia interno al dominio Γ.

Per convenzione, ad ogni rettangolo bidimensionale viene associato un nome che richiama la posizione dei quadranti in una bussola e quindi sono ovvie le seguenti definizioni:

- il quadrante NW é dato da $NW(q) = \{\ (x, y) \in \Gamma.\ x < x_q \wedge y < y_q\ \}$;

- il quadrante *NE* é dato da $NE(q) = \{\ (x,y) \in \Gamma.\ x \geq x_q \wedge y \geq y_q\ \}$;
- il quadrante *SE* é dato da $SE(q) = \{\ (x,y) \in \Gamma.\ x > x_q \wedge y < y_q\ \}$;
- il quadrante *SW* é dato da $SW(q) = \{\ (x,y) \in \Gamma.\ x < x_q \wedge y < y_q\ \}$.

Questa numerazione non é l'unica possibile: ve ne sono altre, ugualmente valide. Le figure 5.25(a) e 5.27(b) mostrano invece la decomposizione del dominio $\Gamma \subseteq \mathbb{E}^3$ effettuata rispetto al baricentro di Γ per ragioni di chiarezza figurativa: in realtá la suddivisione puó avvenire rispetto ad un qualsiasi punto q, interno a Γ. Anche in questo caso non vi é un modo univoco per individuare gli iper–quadranti: la figura 5.27(b) mostra un possibile esempio di numerazione degli ottanti. L'iper–rettangolo con il codice 3 non é visibile perché si trova nella parte posteriore nel dominio Γ. Nel caso generale non esiste un'unica soluzione: l'idea é quella di numerare gli iper–quadranti secondo un qualche criterio che puó essere scelto a seconda delle varie esigenze, della facilitá e dell'efficienza dell'implementazione. Per le esigenze di questa tesi é sufficiente che un ottante abbia un certo codice identificativo: per semplicitá assumeremo che possa essere usato come indice all'interno del vettore *children* nel record *OctreeNode*. Dato un nodo N, é importante capire in quale ottante della decomposizione di γ_N possa appartenere un certo punto p: la primitiva

```
OctreeNode getNext( OctreeNode N, Point p )
```

risolve questo problema, assumendo che N non assuma il valore speciale *NIL*.

5.4.2 L'operazione di inserimento

Lo scopo dell'operazione di inserimento é banalmente quello di inserire un nuovo punto all'interno di una struttura dati *Octree*. La navigazione nell'albero é guidata dal fatto che un punto p di dimensione d appartenga o meno ad un certo iper–quadrante d–dimensionale γ: questa primitiva é implementata dalla funzione *contains*, introdotta nella sezione 5.4.1. Inoltre é importante, dato un nodo N, capire in quale ottante della decomposizione di γ_N possa appartenere un certo punto p: questa primitiva é implementata dalla funzione *getNext*, anch'essa definita nella sezione 5.4.1. Ora possiamo procedere nella descrizione dell'operazione di inserimento di un nuovo punto p in una struttura *Octree*, supponendo che p appartenga al dominio Γ descritto dall'indice in questione. Quest'operazione é implementata attraverso la primitiva di inserimento *octreeInsert*, la quale puó essere descritta dallo pseudocodice in figura 5.28. Come si puó notare, l'inserimento del nuovo punto p in una struttura *Octree* verrá effettuato creando un nuovo nodo foglia F che descrive l'ottante γ_F piú *piccolo* in grado di contenere p: la complessitá di questa operazione é $\mathcal{O}(2^d)$. La ricerca della posizione nel nuovo nodo F viene eseguita navigando fra i figli di un certo nodo attraverso la primitiva

getNext: uno dei figli descriverá sicuramente un ottante γ che contiene il punto p al suo interno visto che p appartiene a Γ. Quindi l'inserimento di un punto é equivalente alla visita di un cammino dalla radice ad un certo nodo foglia e quindi richiede h passi dove h é l'altezza dell'albero. Ogni passo richiede tempo $\mathcal{O}(2^d)$ e quindi la complessitá della primitiva *octreeInsert* é $\mathcal{O}(hl)$, dove $l = 2^d$.

```
void octreeInsert( OctreeNode N, Point p ) {
     OctreeNode aux;

     if(N == NIL) N = leafOctree(p);
     else if( N.q == p ) return;
     else {
          aux =  getNext( N,p );
          octreeInsert( aux,p ); }
}
```

Figura 5.28: pseudocodice della primitiva *octreeInsert*

Il problema é ora stabilire l'altezza di una struttura *Octree*, la quale dipende dall'ordine di inserimento dei nodi: il caso peggiore si ha quando i nodi vengono inseriti in un ordine tale che l'albero abbia altezza n dove n é il numero di punti memorizzati nell'albero. La figura 5.29 mostra la rappresentazione bidimensionale ed il relativo albero nel caso peggiore per una struttura *Quadtree*: nel caso tridimensionale la situazione é analoga.

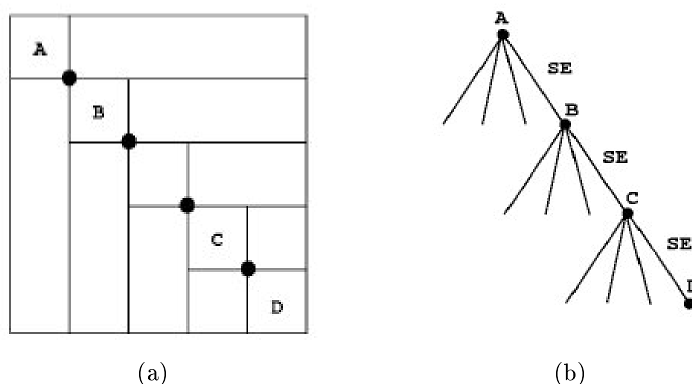


Figura 5.29: decomposizione bidimensionale (a) e relativa struttura dati (b) per un *Quadtree* sbilanciato.

Dunque la complessitá della primitiva di inserimento é $\mathcal{O}(nl)$ dove n é il numero di punti memorizzati nell'albero: la dimensione dello spazio euclideo $\mathbb{E}^d$ é solitamente molto minore di n e quindi la complessitá tende asintoticamente a $\mathcal{O}(n)$. Comunque nel caso *medio*, l'albero é all'incirca bilanciato e quindi

la complessitá é $\mathcal{O}(\log n)$: in letteratura si dimostra che questa situazione avviene quando l'ordine di inserimento dei punti é pressoché casuale. Per ovviare al problema dello sbilanciamento della struttura, in [FB74] e [OvL82] sono state sviluppate alcune tecniche per il caso bidimensionale, estendibili a quello generale: esse cercano di bilanciare la struttura e ridurre la complessitá a $\mathcal{O}(\log n)$: questi metodi richiedono la conoscenza a priori di tutti i punti che devono essere inseriti nell'albero dunque sono adatti alle esigenze di questa tesi: per un'analisi completa della complessitá di questi metodi rifarsi a [Moo95]. Come giá accennato nella sezione 5.4.1, in questa tesi verranno indicizzati i triangoli appartenenti ad una triangolazione immersa in $\mathbb{E}^3$, di cui conosciamo il dominio Γ. Piú precisamente gestiremo i loro centroidi, i quali avranno una distribuzione spaziale ovviamente limitata dall'estensione di Γ e quindi saranno disposti *abbastanza uniformemente* nello spazio in quanto appartengono ad una superficie finita. Dunque la struttura *Octree* risultante non sará *troppo* sbilanciata: questa intuizione é generalmente valida ma non copre alcuni casi particolari che possono ancora rendere sbilanciato l'albero. Il difetto principale delle strutture dati di tipo *Octree* é il fatto che un suo nodo possa contenere un solo punto multidimensionale (nel nostro caso un solo triangolo): questa caratteristica si rivela inefficiente nel caso di un numero importante di dati da indicizzare, in quanto l'altezza dell'albero diventa estremamente elevata. Pertanto questo tipo di struttura dati non é adatto all'utilizzo in memoria secondaria: nel paragrafo 5.6 ne vedremo una variante piú adatta a questo scopo.

5.4.3 L'operazione di cancellazione

Lo scopo dell'operazione di cancellazione é banalmente quello di cancellare una chiave esistente da una struttura *Octree*: non é un'operazione *facile* ed in letteratura si trovano molti tentativi di soluzione. Partiremo con l'analisi del caso bidimensionale e quindi dalle strutture *Quadtree* per poi estenderne le caratteristiche al caso generale. Un possibile metodo si articola in due fasi: nella prima fase si ricerca il nodo N da cancellare mentre nella seconda fase si analizza la natura del nodo vittima per capire come procedere. Se N é un nodo foglia allora possiamo cancellarlo altrimenti dobbiamo cercare un nodo foglia H che possa essere scambiato con N, soddisfando le proprietá di suddivisione della struttura. In [FB74] si dimostra che questo nodo puó non esistere e quindi bisogna reinserire tutti i nodi appartenenti al sottoalbero avente il nodo N come radice. Il caso peggiore si ha quando N era la radice dell'albero e quindi dobbiamo reinserire $n-1$ punti, dove n era il numero di elementi inizialmente contenuti nell'albero. Vista la complessitá dell'operazione di inserimento, allora possiamo concludere che la complessitá della primitiva di cancellazione di un punto in una struttura *Quadtree* é $\mathcal{O}(n^2)$: questo algoritmo é estendibile anche alle strutture *Octree*. In [Sam80] e [Sam06] sono state sviluppate delle euristiche efficienti per individuare il

nodo con cui sostituire il nodo N da cancellare in una struttura *Quadtree*: queste tecniche sono estendibili anche al caso tridimensionale e quindi alla struttura *Octree*. Non ne approfondiremo le caratteristiche visto che anche questo algoritmo richiede, nel caso peggiore, il reinserimento di elementi con complessitá $\mathcal{O}(n^2)$. Comunque, per le esigenze di questa tesi, é piú che sufficiente il primo algoritmo (quello senza euristiche ottimizzanti) in quanto non ci interessa tanto cancellare un singolo elemento quanto cancellare tutti i triangoli che hanno una intersezione non vuota con una certa finestra.

5.4.4 Le interrogazioni supportate

La struttura *Octree* supporta le interrogazioni spaziali *Point–query* e *Range–query*, introdotte nella sezione 5.1.2.

Vediamo come implementare l'interrogazione spaziale *Point–query* attraverso la primitiva *pointQuery*: la figura 5.30 ne mostra lo pseudocodice.

```
bool pointQuery( OctreeNode N, Point p ) {

      if( N == NIL ) return false;
      if( N.p == q ) return true;
      else return pointQuery( getNext(N,p),p ); }
```

Figura 5.30: lo pseudocodice della primitiva *pointQuery*

Come si puó notare, visitiamo un nodo N e controlliamo se questo contiene il punto p: in generale questo confronto ha complessitá $\mathcal{O}(d)$, dove d é la dimensione del punto q memorizzato nel nodo N. Se N non contiene il punto p e non é una foglia, dobbiamo procedere con la ricerca nei nodi figli altrimenti la ricerca termina: anche in questo caso, la navigazione nella struttura *Octree* é guidata dall'appartenenza di un punto ad un certo ottante. Lo schema di questa primitiva é molto simile a quello della primitiva di inserimento: nel caso peggiore, la ricerca termina una volta trovato il nodo foglia F che descrive l'ottante γ_F piú *piccolo* in grado di contenere p. Quindi il numero di passi effettuati é $\mathcal{O}(h)$ dove h é l'altezza dell'albero ed ogni passo ha un costo $\mathcal{O}(d)$. Per le considerazioni effettuate nella sezione 5.4.2, possiamo concludere che la complessitá della primitiva *pointQuery* é $\mathcal{O}(dn)$ dove n é il numero degli elementi memorizzati nella struttura. Solitamente la dimensione dei punti memorizzati é molto minore di n quindi la complessitá tende asintoticamente a $\mathcal{O}(n)$.

Vediamo ora come implementare l'interrograzione *Range–query*: per poterla descrivere, dobbiamo avere a disposizione la primitiva

```
bool overlap ( Rectangle a, Rectangle b)
```

la quale controlla se due rettangoli d–dimensionali hanno un'intersezione non vuota: la complessitá di questa operazione é $\mathcal{O}(d)$. Inoltre assumiamo di avere a disposizione una lista doppiamente linkata di record di tipo *Point*, che indicheremo con $list<Point>$: le operazioni di inserimento in testa e concatenazione fra liste sono indicate rispettivamente dagli operatori *::* e $+$ e la loro esecuzione richiede tempo $\mathcal{O}(1)$; il valore *empty* indica la lista vuota. Questa lista ci servirá per memorizzare il risultato dell'interrogazione *Range–Query*. La figura 5.31 mostra lo pseudocodice della primitiva *rangeQuery* che implementa questa interrogazione.

```
list<Point> rangeQuery( Rectangle w, OctreeNode N ) {
     list<Point> rq;

     if( N == NIL ) rq = empty;
     else if( overlap(w, N.ret) == false ) rq = empty;
     else {
             if(contains(w, N.q)) rq = N.q::rq;
             for j = 0,...,l-1
                     rq = rq + rangeQuery(w,N.children[j]); }
     return rq; }
```

Figura 5.31: pseudocodice della primitiva *rangeQuery*

In questo caso, la navigazione nella struttura é guidata dall'intersezione fra i rettangoli d–dimensionali: questa primitiva visita tutti gli iper–rettangoli che hanno intersezione non vuota con la finestra w e quindi eventualmente tutti gli iper–quadranti della struttura *Octree*. Per ogni iper–quadrante dobbiamo avviare il processo di ricerca nei nodi figli il quale termina ovviamente nelle foglie: nel caso peggiore si potrebbe attraversare l'intero albero. Ogni passo ha costo $\mathcal{O}(d)$ e quindi la complessitá della primitiva *rangeQuery* é $\mathcal{O}(dh)$ dove h é l'altezza dell'albero. Ricordando le osservazioni della sezione 5.4.2, possiamo concludere che la complessitá di questa primitiva diventa $\mathcal{O}(dn)$, dove n é il numero degli elementi memorizzati nell'albero. Solitamente la dimensione dei punti memorizzati é molto minore di n e quindi la complessitá tende asintoticamente a $\mathcal{O}(n)$. In [BSW77] e [LW77] si dimostra che, se i punti sono distribuiti in maniera uniforme, la complessitá si riduce a $\mathcal{O}(\sqrt{n}+F)$ dove F é il numero di punti che soddisfano questa interrogazione.

5.5 La struttura dati *K–d tree*

La struttura dati *K–d tree* é un esempio di indice spaziale di tipo *data–driven* e viene usata principalmente nelle applicazioni di tipo geometrico: ad esempio puó essere utilizzata per la realizzazione di un *GPU raytracer*, co-

me descritto in [FS05], [HSHH07a] e [HSHH07b]. Ma ci sono altri campi in cui questa struttura puó essere sfruttata, ad esempio nella descrizione degli algoritmi di routing dei pacchetti all'interno di una rete P2P: per approfondimenti rifarsi a [GYGM04]. In questo paragrafo ne vedremo le caratteristiche piú importanti: per approfondimenti rifarsi a [Ben75], [Sam90b] e [Sam06]. Nella sezione 5.5.1 ne fisseremo la definizione mentre presenteremo gli algoritmi di inserimento e cancellazione rispettivamente nelle sezioni 5.5.2 e 5.5.3. Infine vedremo nella sezione 5.5.4 come implementare le interrogazioni spaziali supportate.

5.5.1 Definizione della struttura dati

Come abbiamo visto nella sezione 5.4, l'occupazione spaziale di un nodo di una struttura *Octree* ha una crescita esponenziale rispetto alla dimensione d dei punti da indicizzare e le primitive *contains* e *leafOctree* hanno complessitá esponenziale in d: per dimensioni elevate, queste complessitá non sono piú gestibili ed accettabili. La struttura *K–d tree* risolve questo problema introducendo un metodo differente per la decomposizione del dominio di interesse Γ. Supponiamo di avere un dominio Γ ed un punto q appartenente a Γ, entrambi di dimensione generica d. L'idea consiste nel tagliare il dominio Γ con un iper–piano γ di dimensione d passante per il punto q, ottenendo due politopi d–dimensionali disgiunti, ma che condividono una faccia. La scelta di tale iperpiano determina il tipo di struttura: l'indice *K–d tree* appartiene alla famiglia delle strutture *data–driven*, il cui esponente certamente piú noto é la struttura *BSP–tree* (dall'inglese *Binary space partitioning*) dove viene usato un generico iper–piano γ per decomporre il dominio di interesse: per approfondimenti su questo tipo di strutture rifarsi a [FKN80] e [CG91].

Nel nostro caso, l'iper–piano γ sará parallelo ad uno degli iper–piani coordinati: la scelta avviene tramite una variante dello schema di indicizzazione delle strutture *BST*, di cui abbiamo parlato nel paragrafo 5.2. Come abbiamo giá avuto modo di osservare, l'iper–piano γ suddivide il dominio Γ in due politopi dunque é banale associare questa suddivisione a quella di una struttura *BST*: non ci resta che fissare il valore discriminante. La scelta di questo valore (e quindi dell'iper–piano γ) non dipende solamente dal dominio Γ in questione, ma anche dal livello di Γ nella decomposizione di tipo *K–d tree* di un certo dominio Γ_1 che contiene Γ al suo interno. Come livello del dominio Γ si intende il livello del nodo N_Γ all'interno di una struttura *K–d tree* che descrive la decomposizione di Γ: questo concetto sará piú chiaro una volta definito il nodo di una struttura *K–d tree*. Quindi, se il livello del nodo N_Γ é k allora l'equazione dell'iper–piano γ sará:

$$x_j = q_j \tag{5.1}$$

dove j é il resto della divisione del livello k per la dimensione d, x_j é la j–esima coordinata del sistema di assi cartesiani mentre q_j é la j–esima

coordinata del punto q. La coordinata q_j avrá il ruolo di valore discriminante: uno dei politopi della decomposizione sará caratterizzato da punti aventi la j–esima coordinata minore o uguale a q_j, l'altro sará caratterizzato da punti aventi la j–esima coordinata maggiore di q_j. Inoltre, se il dominio Γ di partenza é un iper–rettangolo di dimensione d, si ottiene una decomposizione di Γ in due rettangoli d–dimensionali disgiunti. In questa tesi conosciamo a priori tutti i punti che vogliamo indicizzare e quindi, per semplicitá, useremo come dominio la sua *MBB* perció tutti i politopi della decomposizione saranno iper–rettangoli.

A differenza della struttura *Octree*, discussa nel paragrafo 5.4, il principio di decomposizione é pressoché indipendente dalla dimensione dei punti da indicizzare: questa proprietá ci consente di affrontare la definizione di *K–d tree* nel generico spazio euclideo $\mathbb{E}^d$.

Definizione 5.16. Supponiamo di voler decomporre un dominio $\Gamma \subseteq \mathbb{E}^d$ allora possiamo definire un *K–d tree* su Γ come un albero binario che soddisfa le seguenti proprietá:

- il nodo radice descrive il dominio Γ;
- ogni nodo N descrive un rettangolo d–dimensionale $\gamma_N \subseteq \Gamma$;
- ogni nodo contiene un punto $q \in \gamma_N$: di conseguenza $q \in \Gamma$;
- ogni nodo N contiene due puntatori ai figli c_0 e c_1 che descrivono i rettangoli d–dimensionali, i quali costituiscono la decomposizione di γ_N rispetto al punto q memorizzato nel nodo N. Questa suddivisione avviene secondo le regole di cui abbiamo discusso in precedenza.

Nel seguito assumeremo che un nodo di una struttura *K–d tree* possa essere descritto dal record *KdtreeNode*, il cui pseudocodice é mostrato in figura 5.32. Nel record *KdtreeNode* il campo q memorizza il punto contenuto nel nodo, i campi c_0 e c_1 ne memorizzano i due nodi figli, *ret* rappresenta l'iper–rettangolo che si sta modellando mentre *level* contiene il livello del nodo N nell'albero.

```
record KdtreeNode {
            Point q;
            KdtreeNode c_0,c_1;
            Rectangle ret;
            int level; }
```

Figura 5.32: pseudocodice del record che modella un nodo di una struttura *K–d tree*, definita nello spazio euclideo $\mathbb{E}^d$, con d generico.

Questo record modella direttamente un nodo non foglia: in maniera analoga a quanto fatto per gli *RB–alberi* nel paragrafo 5.2, possiamo usare il valore

speciale *NIL* per segnalare l'assenza di un determinato figlio. In questo modo un nodo foglia avrá il puntatore c_j pari al valore speciale *NIL* per ogni indice $j = 0, 1$. Per creare un nodo foglia contenente un certo punto q possiamo usare la primitiva

```
KdtreeNode leafKdtree( Point q )
```

la quale si occupa di memorizzare il punto q ed impostare correttamente i valori dei nodi figli: la complessitá di questa primitiva é $\mathcal{O}(1)$. Nel prototipo di *DBMS* spaziale, che descriveremo nel capitolo 6, abbiamo implementato la versione generica della struttura *K–d tree*, immersa nello spazio euclideo $\mathbb{E}^d$ con d generico. La figura 5.33 mostra un esempio di questa struttura relativa al caso tridimensionale ed immmersa in $\mathbb{E}^3$.

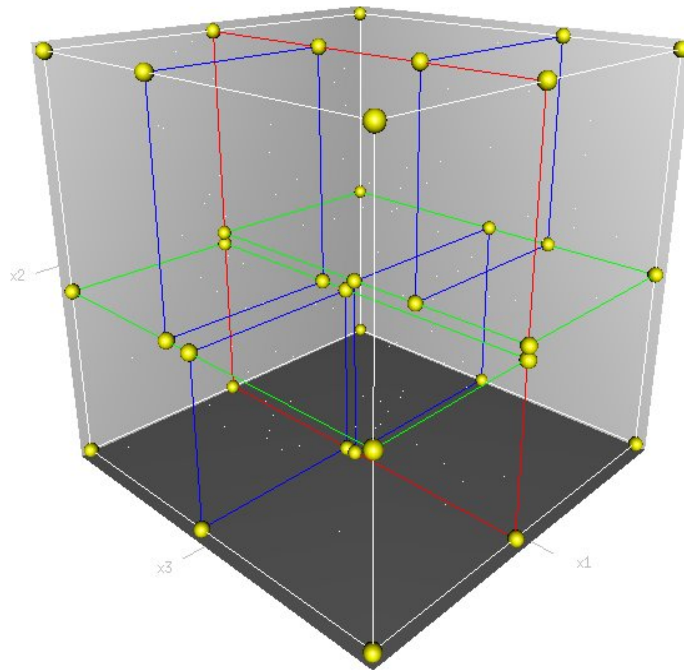


Figura 5.33: esempio di una struttura *k–d tree* in $\mathbb{E}^3$. La prima suddivisione (in rosso) divide il dominio iniziale Γ (in bianco) in due sottodominii lungo l'asse x_1, ognuno dei quali viene ancora suddiviso (in verde) in altre due sottocelle lungo l'asse x_2. Quest'ultime sono ulteriormente suddivise (in blu) lungo l'asse x_3. Le sfere gialle rappresentano i vertici della decomposizione.

5.5.2 L'operazione di inserimento

Lo scopo dell'operazione di inserimento é banalmente quello di inserire un nuovo punto all'interno di una struttura dati *K–d tree*. Come abbiamo visto

nella sezione 5.5.1, l'organizzazione di questa struttura é analoga a quella di un albero *BST*: la navigazione nella struttura dati *K–d tree* é guidata dall'individuazione del valore discriminante fra i nodi e quindi dalla scelta del figlio in cui proseguire la visita dell'albero. Quindi é necessario individuare il figlio in cui proseguire la navigazione nell'albero: la figura 5.34 mostra lo pseudocodice della primitiva *getNext*, la quale individua il figlio del nodo N sul quale proseguire la visita rispetto al punto in input.

```
KdtreeNode getNext( KdtreeNode N , Point p ) {
      int k = N.level MOD p.dim;
      if( p.coords[k] <= N.q.coords[k] ) return N.c0;
      else return N.c1; }
```

Figura 5.34: pseudocodice della primitiva *getNext*.

Questa primitiva assume che il nodo N non sia il valore speciale *NIL* e che il punto p abbia la stessa dimensione del punto memorizzato in N: inoltre l'operatore *MOD* calcola il resto della divisione fra due valori interi. É banale osservare che la complessitá di questa primitiva é $\mathcal{O}(1)$.

Dopo aver risolto il problema della navigazione negli iper–rettangoli della decomposizione, possiamo descrivere l'operazione di inserimento di un nuovo punto p in una struttura *K–d tree*. Come abbiamo avuto modo di osservare nella sezione 5.5.1, il principio di decomposizione di un nodo é pressoché indipendente dalla dimensione dei punti da indicizzare: questa proprietá ci consente di affrontare la descrizione di quest'operazione nello spazio euclideo $\mathbb{E}^d$, con d generico. La figura 5.35 mostra lo pseudocodice della primitiva *kdtreeInsert*, la quale si occupa di inserire un punto p in un nodo N, assumendo che p appartenga all'iper–quadrante Γ_N descritto da N.

```
void kdtreeInsert ( KdtreeNode N, Point p ) {

     if( N == NIL ) N = leafKdtree(p);
     else if( N.q == p ) return;
     else kdtreeInsert( getNext(N,p),p); }
```

Figura 5.35: pseudocodice della primitiva *kdtreeInsert*

Come si puó notare, l'inserimento del nuovo punto p in una struttura *K–d tree* verrá effettuato creando un nuovo nodo foglia F, il quale descrive l'iper–rettangolo γ_F piú *piccolo* in grado di memorizzare il punto p: in questo caso la complessitá é $\mathcal{O}(1)$. La ricerca della posizione del nuovo nodo F viene eseguita navigando fra i due figli di un certo nodo attraverso la primitiva *getNext*: uno dei figli descriverá sicuramente un iper–rettangolo γ che

contiene il punto p al suo interno visto che p appartiene a Γ. Quindi l'inserimento di un punto é equivalente alla visita di un cammino dalla radice ad un certo nodo foglia e quindi richiede h passi dove h é l'altezza dell'albero. Ogni passo richiede tempo $\mathcal{O}(1)$ e quindi la complessitá della primitiva *kd-treeInsert* é $\mathcal{O}(h)$. Il problema é ora stabilire l'altezza di un indice *K–d tree*, la quale dipende dall'ordine di inserimento dei nodi: il caso peggiore si ha quando i nodi vengono inseriti in un ordine tale che l'albero abbia altezza n dove n é il numero di punti memorizzati nell'albero. La figura 5.36 mostra la rappresentazione bidimensionale ed il relativo albero nel caso peggiore per una struttura *K–d tree*: nel caso tridimensionale la situazione é analoga.

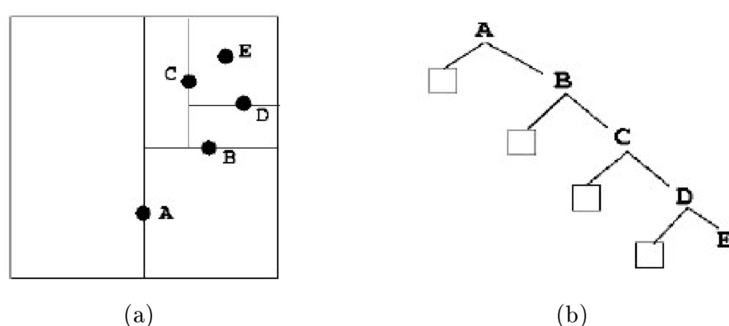


Figura 5.36: decomposizione bidimensionale (a) e relativa struttura dati (b) per un *K–d tree* sbilanciato.

Dunque la complessitá della primitiva di inserimento é $\mathcal{O}(n)$ dove n é il numero di punti memorizzati nell'albero. Comunque nel caso *medio*, l'albero é all'incirca bilanciato e quindi la complessitá é $\mathcal{O}(\log n)$: in letteratura si dimostra che questa situazione avviene quando l'ordine di inserimento dei punti é pressoché casuale. Come giá accennato nella sezione 5.4.1, in questa tesi verranno indicizzati i triangoli appartenenti ad una triangolazione immersa in $\mathbb{E}^3$, della quale conosciamo il dominio Γ. Piú precisamente gestiremo i loro centroidi, i quali avranno una distribuzione spaziale ovviamente limitata dall'estensione di Γ e quindi saranno disposti *abbastanza uniformemente* nello spazio in quanto appartengono ad una superficie finita. Dunque la struttura *K–d tree* risultante non sará *troppo* sbilanciata: questa intuizione é generalmente valida, ma non copre alcuni casi particolari che possono ancora rendere sbilanciato l'albero. Per ovviare al problema dello sbilanciamento della struttura, sono state sviluppate delle varianti attraverso la struttura dati *B–Albero*: gli indici *K–D–B–Tree* e *BK–d tree* ne costituiscono degli esempi, per approfondimenti rifarsi rispettivamente a [Rob81] e [PAAV03]. Anche negli indici di tipo *K–d tree* un nodo contiene un solo punto multidimensionale (nel nostro caso un solo triangolo) : questa caratteristica si rivela inefficiente nel caso di un numero importante di dati da indicizzare, in quanto l'altezza dell'albero diventa estremamente elevata. Pertanto questo

tipo di struttura dati non é adatto all'utilizzo in memoria secondaria: nel paragrafo 5.6 ne vedremo una variante piú adatta a questo scopo.

5.5.3 L'operazione di cancellazione

Lo scopo dell'operazione di cancellazione é banalmente quello di rimuovere un punto che appartiene ad una data struttura *K–d tree*: essa si presenta come una variante della primitiva di cancellazione di un elemento da un albero *BST*. Ne vedremo le idee principali in una struttura *K–d tree* relativa al caso bidimensionale per poi generalizzare quanto ottenuto.

Supponiamo di voler cancellare un punto bidimensionale p di coordinate (x_p, y_p) da una certa struttura *K–d tree*: il primo passo consiste nel verificare la presenza di p nell'albero attraverso la primitiva *pointQuery*, che descriveremo nella sezione 5.5.4. Se il punto p non é presente nella struttura abbiamo banalmente terminato la nostra analisi, altrimenti dobbiamo capire la natura del nodo N_p a cui appartiene p. Se N_p é una foglia possiamo cancellarlo ed eventualmente eliminare il fratello di N_p, se esiste ed é anch'esso una foglia: inoltre bisogna ovviamente aggiornare il nodo padre di N_p, se esiste. Se N_p é un nodo interno, la sua cancellazione é piú complessa: in questo caso almeno uno dei sottoalberi, la cui radice é data da uno dei due figli di N_p, é non vuoto. Nel seguito della trattazione indicheremo con N_l il figlio sinistro del nodo N_p (cioé il campo c_0 del record *KdtreeNode*) e con N_r il figlio destro del nodo N_p (cioé il campo c_1 del record *KdtreeNode*). Il processo di cancellazione puó essere suddiviso in tre fasi:

- trovare un nodo *vittima* R, appartenente a N_l o a N_r, che possa sostituire il nodo N_p senza perdite di dati nel sottoalbero in cui risiede;
- sostituire il nodo N_p con R;
- applicare ricorsivamente l'algoritmo di cancellazione sul nodo R fino ad arrivare ad una foglia, che puó essere direttamente cancellata.

Il passo critico di questo algoritmo é l'identificazione del nodo vittima R, il quale viene scelto in base alla natura di N_r: in generale si preferisce cercare R nel sottoalbero destro di N_p se questo non é vuoto visto che nel sottoalbero sinistro N_l puó esistere piú di un nodo con il valore massimo della coordinata x (o y, a seconda del livello di N_p) violando cosí la definizione della struttura dati *K–d tree*. Tuttavia, se N_r é vuoto, siamo costretti a scegliere, sotto opportune condizioni, un nodo R appartenente a N_l.

Vediamo ora cosa succede quando N_r non é vuoto: dobbiamo operare a seconda del livello del nodo N_p. Infatti possiamo considerare come nodo R:

- un qualsiasi nodo in N_r avente coordinata x piú piccola possibile, se il livello del nodo N_p é pari;

- un qualsiasi nodo in N_r avente coordinata y piú piccola possibile, se il livello del nodo N_p é dispari;

Le figure 5.37(b), 5.37(d) e 5.37(f) mostrano le scelte effettuate in situazioni in cui N_r non é vuoto.

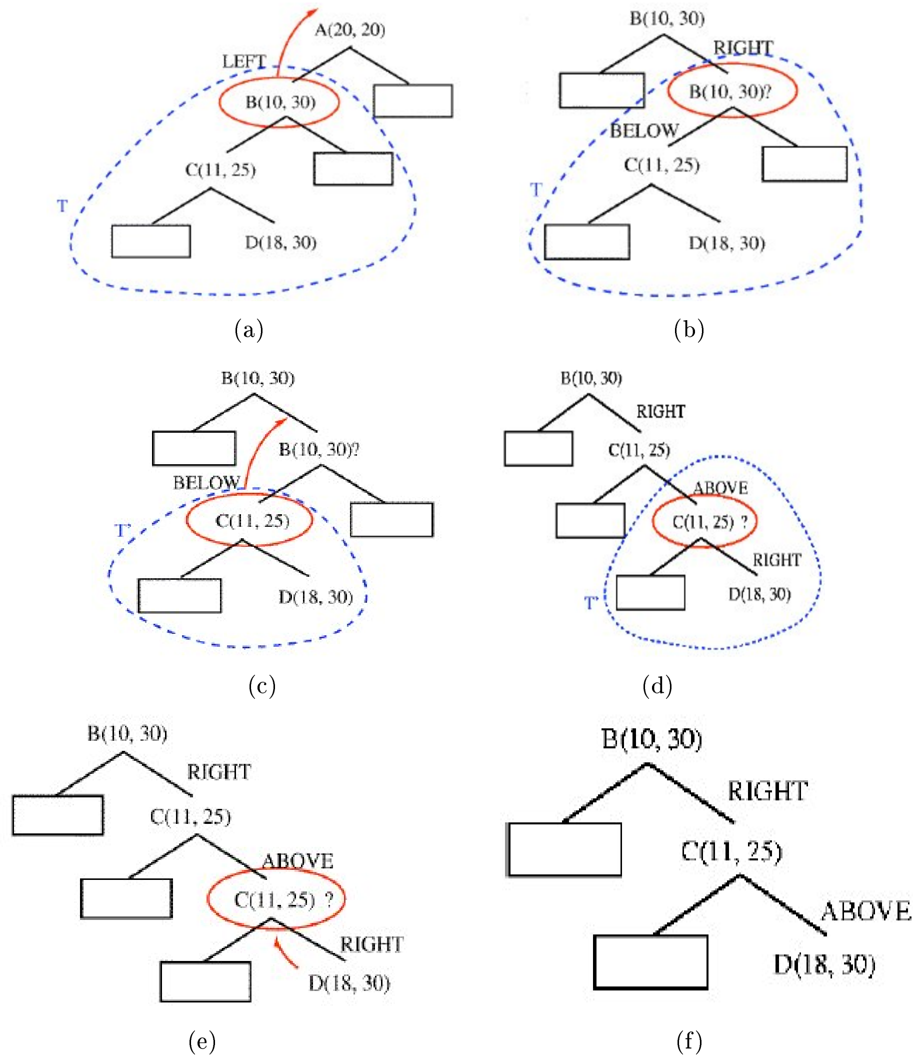


Figura 5.37: esecuzione dell'algoritmo di cancellazione del punto di coordinate $(20, 20)$ da una struttura *K–d tree*.

Se il sottoalbero N_r é vuoto, dobbiamo scegliere il sostituto di N_p in N_l, considerando come nodo R:

- un qualsiasi nodo in N_l avente coordinata x piú piccola possibile, se il livello del nodo N_p é pari;

- un qualsiasi nodo in N_l avente coordinata y piú piccola possibile, se il livello del nodo N_p é dispari.

A questo punto possiamo scambiare i figli di posto del nodo N_p in modo tale che venga analizzato nuovamente il figlio destro N_r in un'eventuale chiamata ricorsiva su N_p. Le figure 5.37(a), 5.37(c) e 5.37(e) mostrano le scelte effettuate in situazioni in cui N_r é vuoto.

Questo algoritmo si puó estendere al caso d–dimensionale: in [Ben75] si dimostra che la complessitá dell'operazione di ricerca di un nodo vittima R é $\mathcal{O}(n^{1-\frac{1}{d}})$. Da questo risultato possiamo concludere che la complessitá della primitiva di cancellazione di un nodo da una struttura *K–d tree* é $\mathcal{O}(n)$ nel caso peggiore, cioé quando si cerca di cancellare il nodo radice. In questa tesi non ci interessa tanto cancellare un singolo nodo quanto cancellare tutti i triangoli che hanno una intersezione non vuota con una certa finestra.

5.5.4 Le interrogazioni supportate

La struttura *K–d tree* supporta le interrogazioni spaziali *Point–query* e *Range–query*, introdotte nella sezione 5.1.2.

Vediamo come implementare l'interrogazione spaziale *Point–query* attraverso la primitiva *pointQuery*: la figura 5.38 ne mostra lo pseudocodice.

```
bool pointQuery( KdtreeNode N, Point p ) {

      if( N == NIL ) return false;
      if( N.p == q ) return true;
      else return pointQuery( getNext(N,p),p ); }
```

Figura 5.38: pseudocodice della primitiva *pointQuery*

Come si puó notare, visitiamo un nodo N e controlliamo se questo contiene il punto p: in generale questo confronto ha complessitá $\mathcal{O}(d)$, dove d é la dimensione del punto q memorizzato nel nodo N. Se N non contiene il punto p e non é una foglia, dobbiamo procedere con la ricerca nei nodi figli altrimenti la ricerca termina: anche in questo caso, la navigazione nella struttura *K–d tree* é guidata dall'appartenenza di un punto ad un certo iper–rettangolo della decomposizione. Lo schema di questa primitiva é molto simile a quello della primitiva di inserimento: nel caso peggiore, la ricerca termina una volta trovato il nodo foglia F che descrive l'iper–rettangolo γ_F piú *piccolo* in grado di contenere p. Quindi il numero di passi effettuati é $\mathcal{O}(h)$ dove h é l'altezza dell'albero ed ogni passo ha un costo $\mathcal{O}(d)$. Per le considerazioni effettuate nella sezione 5.5.2, possiamo concludere che la complessitá della primitiva *pointQuery* é $\mathcal{O}(dn)$ dove n é il numero degli elementi memorizzati

nella struttura. Solitamente la dimensione d dei punti memorizzati é molto minore del numero totale di elementi e quindi la complessitá tende a $\mathcal{O}(n)$.

Vediamo ora come implementare l'interrogazione spaziale *Range–query* attraverso la primitiva *rangeQuery*: la figura 5.39 ne mostra lo pseudocodice. Alcuni elementi che compaiono in questo pseudocodice sono stati introdotti nella sezione 5.4.4.

```
list<Point> rangeQuery( Rectangle w, KdtreeNode N ) {
     list<Point> rq;

     if( N == NIL ) rq = empty;
     else if( overlap(w, N.ret) == false ) rq = empty;
     else {
             if(contains(w, N.q)) rq = N.q::rq;
             rq = rq + rangeQuery(w,N.c0);
             rq = rq + rangeQuery(w,N.c1); }
     return rq; }
```

Figura 5.39: pseudocodice della primitiva *rangeQuery*

In questo caso, la navigazione nella struttura é guidata dall'intersezione fra i rettangoli d–dimensionali: questa primitiva visita tutti gli iper–rettangoli che hanno intersezione non vuota con la finestra w e quindi eventualmente tutti i politopi della decomposizone descritta dalla struttura *K–d tree*. Per ogni rettangolo dobbiamo avviare il processo di ricerca nei nodi figli, il quale termina ovviamente nelle foglie: nel caso peggiore si potrebbe attraversare l'intero albero. Ogni passo ha costo $\mathcal{O}(d)$, dove d é la dimensione dei punti memorizzati dunque la complessitá della primitiva *rangeQuery* é $\mathcal{O}(dh)$ dove h é l'altezza dell'albero. Ricordando le osservazioni della sezione 5.5.2, possiamo concludere che la complessitá di questa primitiva diventa $\mathcal{O}(dn)$, dove n é il numero degli elementi memorizzati nella struttura. Solitamente la dimensione d dei punti memorizzati é molto minore del numero totale di elementi e quindi la complessitá tende asintoticamente a $\mathcal{O}(n)$. In [LW77] si dimostra che se la struttura *K–d tree* é completa, cioé se ha un numero di nodi $n = 2^{h+1} - 1$, la complessitá diventa $\mathcal{O}(dn^{1-\frac{1}{d}} + F)$ dove F é il numero di punti che soddisfano questa interrogazione.

5.6 Le strutture dati *ibride*

In letteratura le strutture di indicizzazione spaziale solitamente si suddividono in indici *orientati ai punti* (in inglese *points–oriented*) ed *orientati alle regioni* (in inglese *regions–oriented*). Nel primo tipi di indici la suddivisione del dominio é guidata dai punti che si vogliono memorizzare, come avviene

nelle strutture *Octree* e *K–d tree* introdotte rispettivamente nei paragrafi 5.4 e 5.5: in questo caso un nodo della struttura dati memorizza un solo punto multidimensionale. Nel secondo caso la suddivisione é guidata dalle proprietá delle regioni che compongono la decomposizione del dominio, come avviene nella struttura dati *PR K–d tree*, introdotta in [Ore82]: in questo caso un nodo della struttura dati puó memorizzare diversi punti multidimensionali. In realtá é possibile realizzare delle versioni *ibride* di indici spaziali nei quali ogni nodo puó contenere diversi punti multidimensionali (caratteristica tipica degli indici *regions–oriented*) mentre la suddivisione del dominio avviene rispetto ad un punto caratteristico dell'insieme di punti memorizzato (caratteristica tipica degli indici *points–oriented*). Nella sezione 5.6.1 vedremo le proprietá delle strutture dati di tipo ibrido, il cui uso in memoria secondaria si rivela particolarmente vantaggioso. In questa tesi abbiamo implementato le varianti ibride delle strutture dati *Octree* e *K–d tree* delle quali parleremo nella sezione 5.6.2.

5.6.1 Aspetti introduttivi

In questa sezione vedremo su quali principi si basa il funzionamento di un indice spatiale di tipo *ibrido*.

Il primo aspetto da considerare é il fatto che un nodo possa contenere al piú c punti multidimensionali prima di essere ulteriormente suddiviso: questo valore viene detto *capacitá* del nodo e puó essere scelto a seconda delle diverse esigenze. In letteratura l'utilizzo di questa tecnica ha dato origine a varie tipologie di strutture, riassunte in [Sam06]: gli indici *PR K–d tree* e *PMR K–d tree* ne costituiscono degli esempi importanti, per approfondimenti rifarsi a [OS83a], [OS83b], [NS86a], [NS86b] e [NS87]. L'implementazione di queste strutture di indicizzazione in memoria secondaria é particolarmente efficace, come descritto in [Zha00].

In questa tesi abbiamo sviluppato degli indici in cui i nodi hanno una capacitá $c > 1$: per determinare il valore di c abbiamo sfruttato le proprietá riguardanti il numero di nodi di un *albero completo*.

Definizione 5.17. Un albero si dice *completo* se tutti i figli dei suoi nodi non foglia esistono e sono diversi dal valore speciale *NIL*.

Solitamente ogni nodo puó avere al piú m figli (com m fissato): ad esempio una struttura dati *K–d tree* ha sempre due figli. In queste ipotesi, un albero é completo se ogni nodo non foglia ha esattamente m figli. Fissata l'altezza h allora un'albero completo avrá un numero di nodi N_h pari a

$$N_h = \sum_{i=0}^{h-1} m^i = \frac{m^h - 1}{m - 1} \tag{5.2}$$

La dimostrazione di questa relazione si trova in [CR71]. Conoscendo il numero totale N di punti multidimensionali da indicizzare, una stima della

capacitá c di un certo nodo é data da:

$$c = \left\lceil \frac{N}{N_h} \right\rceil \tag{5.3}$$

In realtá questa stima é troppo restrittiva visto che l'indice sará completo (e quindi bilanciato) solamente nel caso migliore dove é ragionevole supporre che i nodi abbiano tutti la stessa capacitá. In generale, il valore di c deve garantire che il nodo contenga il maggior numero possibile di punti senza *sprecare* spazio: questo obiettivo é pressoché impossibile da raggiungere visto che richiederebbe la conoscenza a priori della distribuzione dei punti da indicizzare. Per ovviare a questo problema si puó cercare di variare la capacitá di ogni singolo nodo in maniera adattativa, ma questa soluzione non é praticabile e non supporta in maniera efficiente variazioni dinamiche nel numero di elementi memorizzati nella struttura: in letteratura il problema della scelta della capacitá c di un certo nodo é stato ampiamente trattato in [NS86b] e [NS87]. Una possibile soluzione é quella di fissare, in maniera empirica, la capacitá di ogni nodo in modo da minimizzarne il numero ed ovviare allo sbilanciamento della struttura: in questa tesi useremo la stima di c fornita dall'equazione 5.3 incrementata del 10% in modo da fornire nodi di capacitá maggiore rispetto a quelli di una struttura bilanciata.

Una volta fissata la capacitá di un certo nodo N, dobbiamo capire come suddividere l'insieme S_N degli elementi memorizzati se il loro numero supera il valore di c: in questo caso dobbiamo scegliere un punto discriminatore p_S con il quale decomporre l'insieme S, secondo una qualche tecnica (ad esempio una decomposizione di tipo *K–d tree*, descritta nella sezione 5.5.1). Una volta scelto p_s l'insieme S verrá suddiviso in due o piú nodi foglia e verrá creato un nodo non foglia contenente il punto p_S.

In letteratura sono state sviluppate delle tecniche adattative, introdotte in [FBF77] ed in [CK95], secondo le quali si sceglie come discriminatore un punto p_S memorizzato nella struttura tale da dividere l'insieme corrente di punti in sottoinsiemi composti all'incirca dallo stesso numero di elementi: come si puó intuire, questa scelta non supporta in maniera dinamica le modifiche alla struttura e non garantisce che un nodo contenga un almeno elemento visto che p_S potrebbe non esistere.

In realtá, scegliere il punto discriminatore p_S solamente in base al numero dei punti non é una tecnica adatta ai nostri scopi in quanto non tiene conto che la densitá degli elementi da indicizzare puó variare all'interno del dominio. In questa tesi useremo la regola di suddivisione detta *sliding midpoint*, la quale considera il baricentro dei punti dell'insieme S_N di un certo nodo N come punto discriminatore p_S: se uno dei punti in S_N coincide con il baricentro p_S, esso viene duplicato. La suddivisione del dominio operata da questa scelta si adatta alla distribuzione dei punti all'interno di N: inoltre garantisce che le regioni della suddivisione non siano vuote e che i nodi

risultanti contengano un numero di nodi minore di c. Per maggiori approfondimenti sulla tecnica *sliding midpoint* rifarsi a [MA97], [MM99a], [MM99b], [Man01]. Grazie all'uso combinato delle due tecniche appena descritte, é possibile modificare un qualsiasi indice spaziale in modo tale che:

- ogni nodo abbia una certa capacitá $c > 1$;
- la suddivisione del dominio avvenga rispetto a degli elementi discriminatori che non appartengono all'insieme dei punti da indicizzare: questa proprietá semplifica le operazioni di aggiornamento;
- i punti da indicizzare sono contenuti nelle foglie dell'albero.

In questo modo si ha un numero di nodi certamente minore rispetto alle strutture dati *canoniche* (come gli indici *Octree* e *K–d tree*, descritti rispettivamente nei paragrafi 5.4 e 5.5): controllando l'altezza h dell'albero si limita anche il numero massimo di accessi alla memoria secondaria e si garantisce che un nodo possa essere memorizzato in memoria primaria.

5.6.2 Le varianti ibride degli indici *Octree* e *K–d tree*

L'utilizzo delle strutture dati *Octree* e *K–d tree* in memoria secondaria non é efficiente in quanto il numero di nodi di queste strutture é pari al numero degli elementi da indicizzare: questo implica un elevato numero di richieste al supporto di memorizzazione il quale, come abbiamo visto nel capitolo 3, ha un tempo di risposta nettamente inferiore rispetto alla memoria primaria. Per questa ragione abbiamo implementato le varianti ibride degli indici *Octree* e *K–d tree*, nelle quali vengono utilizzate le tecniche introdotte nella sezione 5.6.1: nel seguito della trattazione vedremo cosa cambia rispetto alle strutture dati *canoniche*. Nella letteratura attuale non vi é una nomenclatura universalmente accettata per questi due indici: in [Sam06] si parla di indici *Bucket PR–Quadtree* e *Bucket PR k–d tree* riferendosi rispettivamente alle versioni ibride delle strutture *Octree* e *K–d tree*. Una nomenclatura alternativa consiste nel chiamarle rispettivamente indici *Hybrid Quadtrie* e *Hybrid K–d trie*: nel seguito della trattazione le adotteremo entrambe in maniera pressoché indifferente. Nel resto della sezione presenteremo solamente le proprietá della struttura dati *Bucket PR–Quadtree* in quanto l'indice *Bucket PR K–d tree* si puó ottenere da quello *K–d tree* applicando modifiche analoghe.

La struttura dati *Bucket PR–Quadtree* si ottiene dalle strutture della famiglia *Quadtree* introducendo le modifiche prospettate nella sezione 5.6.1, le quali conducono alla seguente definizione.

Definizione 5.18. Supponiamo di voler decomporre un dominio $\Gamma \subseteq \mathbb{E}^d$ allora possiamo definire un *Bucket PR–Quadtree* su Γ come un albero che soddisfa le seguenti proprietá:

- il nodo radice descrive il dominio Γ;

- ogni nodo N descrive un iper–rettangolo $\gamma_N \subseteq \Gamma$;
- ogni nodo non foglia contiene un punto p che ha il ruolo di discriminatore: inoltre vengono memorizzati $l = 2^d$ puntatori ai figli $c_0, c_1, \ldots, c_l$, i quali descrivono rispettivamente gli iper–rettangoli $\gamma_0, \gamma_1, \ldots, \gamma_l$ appartenenti alla decomposizione di γ_N rispetto al punto q;
- ogni nodo foglia F contiene una lista composta al massimo da c punti multidimensionali dove c é la capacitá di F.

La decomposizione del nodo avviene secondo le regole che abbiamo descritto nella sezione 5.4.1. Pertanto un nodo di una struttura dati *Bucket PR–Quadtree* viene descritto dal record *BPRQuadtreeNode*, il cui pseudocodice é mostrato in figura 5.40: in questo caso $l = 2^d$ dove d é la dimensione dello spazio euclideo $\mathbb{E}^d$ in cui é immerso l'indice.

```
record BPRQuadtreeNode {
            Point q;
            BPRQuadtreeNode children[l];
            Rectangle ret;
            int c;
            list<Point> plist; }
```

Figura 5.40: pseudocodice del record che modella un nodo di una struttura *Bucket PR–Quadtree*, definita nello spazio euclideo $\mathbb{E}^d$

Le uniche differenze rispetto al record *KdtreeNode* sono costituite dai campi c e *plist*, i quali contengono rispettivamente la capacitá e la lista dei punti contenuti in un certo nodo. Inoltre bisogna ricordare che i campi q e *children* sono significativi quando il nodo é interno, mentre quelli c e *plist* lo diventano quando il nodo é una foglia. Sono molto importanti le primitive:

- *bool isLeaf(BPRQuadtreeNode r)*, la quale controlla se un certo nodo r é una foglia: la complessitá di questa primitiva é $\mathcal{O}(1)$;
- *BPRQuadtreeNode (Point q)*, la quale crea un nodo foglia memorizzando il punto q ed impostando correttamente i valori dei nodi figli: la complessitá di questa primitiva é $\mathcal{O}(1)$.

Per quanto riguarda l'operazione di inserimento bisogna ricordare che é possibile aggiungere nuovi elementi solamente nelle foglie mentre nei nodi interni dobbiamo proseguire la navigazione, attraverso la primitiva *getNext*, fino a raggiungere una foglia. Una volta arrivati in un nodo foglia, dobbiamo capire se la capacitá del nodo é stata raggiunta altrimenti si puó aggiungere il nuovo punto nella lista degli elementi memorizzati. Se il nodo é pieno dobbiamo applicare la suddivisione di tipo *Quadtree* all'insieme dei punti S

rispetto al loro baricentro, creando 2^d nodi foglia mentre il nodo corrente diventa interno: questa operazione é svolta dalla primitiva *split* ed ha complessitá $\mathcal{O}(c)$. L'inserimento di un nuovo punto é implementato attraverso la primitiva *BPRQuadtreeInsert*, la quale puó essere descritta dallo pseudocodice in figura 5.41.

```
void BPRQuadtreeInsert( BPRQuadtreeNode N, Point p ) {

      if(N == NIL) N = BPRQuadtreeNode(p);
      else if( isLeaf(N) ) {
         if( (isIn( N.plist,p ) ) ;
         else if( length(N.plist)< N.c ) N.plist = p::N.plist;
         else split(N,p); }
      else BPRQuadtreeInsert( getNext(N,p),p ); }
```

Figura 5.41: pseudocodice della primitiva *BPRQuadtreeInsert*.

Nella primitiva *BPRQuadtreeInsert* abbiamo usato due funzioni ausiliarie: quella *isIn* controlla se una lista contiene un certo elemento mentre quella *length* restituisce il numero di elementi memorizzati in una lista.

Anche l'operazione di cancellazione si basa sul fatto che i punti indicizzati vengono memorizzati nelle foglie mentre i nodi interni servono per guidare la navigazione nell'albero. Una volta arrivati in una foglia F, dobbiamo eventualmente cancellare il punto desiderato (questa primitiva ha complessitá $\mathcal{O}(c)$) e controllare se é possibile fondere F con i nodi adiacenti che compongono la decomposizione del dominio: questa operazione é implementata attraverso la primitiva *fusion* e si puó propagare ai livelli superiori quindi puó essere ripetuta al piú h volte dove h é l'altezza della struttura. La cancellazione di un punto da una struttura *Bucket PR–Quadtree* é realizzata dalla primitiva *BPRQuadtreeRemove*, la quale puó essere descritta dallo pseudocodice in figura 5.42 dove la funzione *remove* cancella un elemento da una lista.

```
void BPRQuadtreeRemove( BPRQuadtreeNode N, Point p ) {

      if(N == NIL) ;
      else if( isLeaf(N) )
         if( (isIn( N.plist,p ) )  {
             remove( N.plist,p);
             fusion(N); }
      else BPRQuadtreeRemove( getNext(N,p),p ); }
```

Figura 5.42: pseudocodice della primitiva *BPRQuadtreeRemove*.

La struttura dati *Bucket PR–Quadtree* supporta supporta le interrogazioni spaziali *Point–query* e *Range–query*, introdotte nella sezione 5.1.2. L'idea che sta alla base di entrambe le interrogazioni é quella di operare la ricerca dei punti nelle foglie, le quali contengono gli elementi memorizzati.

Vediamo come implementare l'interrogazione spaziale *Point–query* attraverso la primitiva *pointQuery*: la figura 5.43 ne mostra lo pseudocodice.

```
void pointQuery( BPRQuadtreeNode N, Point p ) {
      BPRQuadtreeNode aux;

      if(N == NIL) return false;
      else if( isLeaf(N) ) return isIn(N.plist,p);
      else return pointQuery( getNext(N,p),p ); }
```

Figura 5.43: pseudocodice della primitiva *pointQuery*.

Come si puó notare, l'obiettivo della primitiva consiste nell'individuare il nodo foglia piú piccolo nel quale potrebbe essere memorizzato il punto p: in questo caso dobbiamo cercarlo nella lista dei punti memorizzati, lunga al piú c quindi l'operazione di ricerca di un punto all'interno della lista *plist* ha complessitá $\mathcal{O}(c)$.

Vediamo ora come implementare l'interrogazione spaziale *Range–query* attraverso la primitiva *rangeQuery*: la figura 5.44 ne mostra lo pseudocodice. In questo codice usiamo la funzione *extract(N,w)* la quale restituisce i punti memorizzati in *N.plist* che sono interni all'iper–rettangolo w: la complessitá di questa primitiva é $\mathcal{O}(c)$.

```
list<Point> rangeQuery( Rectangle w, BPRQuadtreeNode N ) {
     list<Point> rq;

     if( N == NIL ) rq = empty;
     else if( overlap(w, N.ret) == false ) rq = empty;
     else if( isLeaf(N) ) { rq = extract(N,w); }
     else
          for j = 0,...,l-1
               rq = rq + rangeQuery(w,N.children[j]);
     return rq; }
```

Figura 5.44: pseudocodice della primitiva *rangeQuery*.

Come possiamo notare, l'implementazione delle primitive appena descritte si compone di due fasi: la ricerca di un nodo foglia F e l'implementazione vera e propria sul nodo F: nella sezione 5.6.1 abbiamo descritto come determinare

la capacitá di un certo nodo a partire dall'altezza h in modo da permettere l'uso in memoria secondaria di queste strutture dati. Per raggiungere un nodo foglia a partire dal nodo radice bisognerá visitare un cammino di altezza all'incirca pari a h e quindi di lunghezza fissata: con questo tipo di indici il numero totale di nodi sará minore del numero degli elementi memorizzati, ma si potranno avere ancora alberi sbilanciati, a seconda della distribuzione dei dati. Quindi la complessitá di questa prima fase delle primitive é $\Omega(h)$ con $h < N$, dove N é il numero totale di elementi: nel caso migliore questa misura diventa $\mathcal{O}(\log N)$. La tecnica dello *sliding midpoint* garantisce che le regioni in cui si suddivide il dominio non siano vuote e quindi il grado di sbilanciamento sará nettamente inferiore a quello che si puó ottenere con le classiche strutture *Octree* e *K–d tree*. La seconda fase delle primitive é costituita solitamente da una modifica o da un'interrogazione della lista di punti memorizzati nel nodo foglia F e tale operazione ha complessitá $\mathcal{O}(c)$.

Capitolo 6

La libreria OMSM

L'arte non consiste nel rappresentare cose nuove,
bensí nel rappresentare con novitá.

Ugo Foscolo

La ricerca svolta in questa tesi ha richiesto un notevole lavoro implementativo in quanto é stato creato un framework generico per indicizzare modelli geometrici in memoria secondaria, caratteristica che ne fornisce il nome *OMSM*, dall'espressione inglese *Objects Management in Secondary Memory*: in questo paragrafo ne vedremo la definizione e le proprietá piú importanti. Inoltre ne descriveremo una particolare implementazione, adatta alla gestione di mappe poligonali. Come abbiamo visto nella sezione 2.3.2, un oggetto geometrico é modellato da un complesso simpliciale, solitamente una triangolazione. Nel nostro caso le dimensioni del modello eccedono quelle della memoria principale, pertanto sono necessarie delle tecniche ad hoc per gestire questa situazione. I risultati di questa ricerca sono confluiti nella creazione della libreria *OMSM*, la quale fornisce alcuni componenti utili per la gestione di modelli geometrici le cui dimensioni eccedono quella della memoria primaria. In questo capitolo vedremo nel dettaglio alcune parti della libreria che svolgono un ruolo particolarmente importante nell'economia di questa tesi: per i dettagli piú tecnici, che non era ragionevole includere in questa trattazione, si rimanda a [Can07a] e [Can07b].

Nel paragrafo 6.1 vedremo alcune caratteristiche generali della libreria, utili per comprendere la descrizione dei vari componenti. Nel paragrafo 6.2 studieremo le caratteristiche del componente *OperationTimer*, il quale scandisce il *tempo* all'interno delle applicazioni: il suo livello di precisione é il microsecondo e la sua efficienza gioca un ruolo fondamentale nelle prestazioni del sistema. Nel paragrafo 6.3 studieremo le caratteristiche della struttura dati *ERB–Albero*, la quale estende quella *RB–Albero*, introdotta nella sezione 5.2. Solitamente una triangolazione é fornita come un file scritto in un qualche formato e quindi bisogna gestirla in maniera opportuna vista la sua dimensione: nel paragrafo 6.4 descriveremo la soluzione offerta da questa

libreria. Infine studieremo nel paragrafo 6.5 le caratteristiche principali dell'architettura di memorizzazione dei dati spaziali, la quale verrá utilizzata per la semplificazione dei modelli geometrici.

6.1 Le caratteristiche generali

La libreria *OMSM* contiene l'ampia parte implementativa che costituisce l'attivitá di ricerca svolta in questa tesi ed é stata realizzata nel linguaggio *C++*: per approfondire le caratteristiche di tale linguaggio rifarsi a [Str86], [Eck00], [CPP00] e [EA03]. Tale scelta ha influito in maniera determinante sulla struttura e sull'organizzazione di tale libreria: ad esempio questo linguaggio non supporta nativamente alcun meccanismo di persistenza e questo ha condotto allo sviluppo del sistema di memorizzazione dei dati di cui parleremo nel paragrafo 6.5. Come si puó intuire, sono stati usati i principi della *programmazione ad oggetti*: nella sezione 6.1.1 vedremo una breve rassegna delle idee principali di questa tecnica di programmazione, necessarie per comprendere le problematiche e le soluzioni proposte. Nella sezione 6.1.2 vedremo alcune caratteristiche di base della libreria *OMSM*. La gestione ed il controllo degli errori costituiscono un elemento importante di qualunque sistema software: nella sezione 6.1.3 vedremo i meccanismi forniti da questa libreria per la risoluzione di questo problema.

6.1.1 Principi di programmazione ad oggetti

In questa sezione vedremo alcuni principi di programmazione ad oggetti: non si hanno pretese di completezza, per approfondimenti rifarsi a [Weg87], [Man05] e [Boo07]. Secondo la classificazione introdotta in [Weg87], un linguaggio *object–oriented* supporta le nozioni di *oggetto*, di *classe* e di *ereditarietá* fra classi (in inglese *inheritance*). Illustreremo brevemente nel seguito il significato di questi tre termini.

Partiamo dalla nozione di *oggetto*: solitamente lo si puó descrivere come un tipo di dato (spesso abbreviato con *ADT*, dall'inglese *Abstract Data Type*) dotato di uno stato interno. L'analogia con gli *ADT* nasce dal fatto che ogni oggetto ha un'interfaccia operazionale con la quale poter interagire con lo stato interno dell'oggetto: l'implementazione di queste operazioni é nascosta. In letteratura si indica un oggetto anche con il termine *attore*, enfatizzando il fatto che un oggetto si comporta secondo un *copione* prestabilito.

Possiamo concepire un certo programma come l'interazione fra piú oggetti e quindi é necessario descrivere lo schema di ogni singolo attore: questa funzione é svolta dal concetto di *classe*, la quale definisce un modello di oggetto che puó essere istanziato. Le componenti di una definizione di classe sono essenzialmente di due tipi: le *variabili di istanza* ed i *metodi*. Il primo tipo di componenti (note anche con il termine inglese *instance variables*) fa riferimento al fatto che esiste una copia di queste variabili per ogni istanza

della classe e questo vuol dire che ogni oggetto ha un proprio stato interno, in maniera indipendente dagli altri. I metodi sono funzioni (o procedure, a seconda dei casi) definite sullo stato dell'oggetto che istanzia la classe a cui essi appartengono: l'insieme dei metodi ne descrive l'*interfaccia* e permette l'interazione con il mondo esterno. Tutti gli attori modellati da una certa classe hanno la medesima interfaccia, ma l'invocazione di un metodo potrebbe produrre risultati diversi a seconda dello specifico oggetto sul quale si opera. Per poter gestire correttamente gli oggetti, una classe deve fornire almeno i metodi di tipo *costruttore* e quelli di tipo *distruttore*. I metodi del primo tipo creano un nuovo oggetto secondo lo schema definito nella classe, inizializzandone lo stato interno. I metodi del secondo tipo liberano la quantitá di memoria primaria attualmente occupata da un certo oggetto.

Il meccanismo dell'*ereditarietá* fra classi é particolarmente interessante per lo sviluppo incrementale ed il riutilizzo del software: vediamo su quali principi si basa. Data una classe $\mathcal{C}$ (detta classe *parent* o *antenata*) si definisce una classe $\mathcal{C}_1$ (detta classe *heir* o classe *erede*), ottenuta da $\mathcal{C}$ estendendone la definizione con nuove variabili di istanza e/o nuovi metodi. In questo modo la classe $\mathcal{C}_1$ riutilizza il codice della classe $\mathcal{C}$ e lo estende. Solitamente i metodi della classe antenata non sono duplicati nella classe erede, ma vengono invocati direttamente dalla classe $\mathcal{C}$ per ragioni di efficienza. Secondo questo schema, un oggetto di classe $\mathcal{C}_1$ puó comparire dove viene richiesto un oggetto di classe $\mathcal{C}$: solitamente non puó avvenire il contrario. In generale una classe erede puó anche ridefinire i metodi della classe antenata: in questo caso le due classi non condivideranno il codice per questi metodi. Con queste modifiche sorge il problema di capire quale sia la versione del metodo da invocare. Ci sono almeno due approcci risolutivi possibili:

- il *binding di tipo statico* (noto in inglese come *compile–time binding*), dove la versione del metodo da invocare é determinata dalla classe $\mathcal{C}_S$ della variabile in cui é contenuto l'oggetto sul quale si invoca il metodo: $\mathcal{C}_S$ é ottenuta esaminando il codice sorgente;

- il *binding di tipo dinamico* (noto in inglese come *run–time binding*), dove la versione del metodo da invocare é determinata dalla classe $\mathcal{C}_D$ di cui é istanza l'oggetto contenuto nella variabile sulla quale si invoca il metodo: $\mathcal{C}_D$ é ottenuta durante l'esecuzione del programma.

Il linguaggio *C++* utilizza nativamente il primo tipo di approccio: per abilitare il binding di tipo dinamico nei riguardi di un certo metodo bisogna usare la parola chiave *virtual* nella sua definizione. La maggior parte dei metodi della libreria *OMSM* sfruttano questa tecnica.

Un'altra caratteristica importante dei linguaggi *object–oriented* é costituita dall'*overloading*, la quale permette ad una classe di avere piú metodi (direttamente dichiarati o ereditati) con lo stesso nome, ma dotati di una diversa lista dei parametri: tale proprietá vale anche per i costruttori. Per

queste ragioni é importante definire delle regole che permettano di determinare il metodo da associare ad una chiamata: questo problema é noto come *risoluzione dell'overloading* e garantisce che, una volta risolto, i metodi vengano considerati indipendenti fra loro. Solitamente si considera il tipo statico degli argomenti e non si permette la definizione di metodi che differiscano solamente per il tipo di ritorno, come avviene nel linguaggio *C++*.

6.1.2 Aspetti introduttivi

La libreria *OMSM* contiene moltissimi componenti utili alla gestione di insiemi di oggetti geometrici, la cui dimensione eccede quella della memoria primaria disponibile: questo sistema fornisce due contributi alla discussione sull'argomento, il quale é molto importante nella ricerca attuale.

- Il primo contributo consiste nella creazione di un ambiente completo in grado di gestire in maniera trasparente triangolazioni che solitamente vengono fornite in differenti formati. Inoltre questo componente si occupa di analizzare ed eventualmente correggere i modelli geometrici in input in modo tale da operare su triangolazioni: fra le altre caratteristiche, é in grado di triangolare le facce del modello in input, se queste sono dei poligoni semplici. Parleremo in maniera piú approfondita di questo componente nel paragrafo 6.4.

- Il secondo contributo consiste nella creazione di un framework generico per l'indicizzazione spaziale di oggetti geometrici in maniera indipendente dalla loro dimensione euclidea e dalla loro topologia. Questo componente é facilmente configurabile ed ha una struttura modulare: ció permette di variare in maniera molto semplice alcuni aspetti del suo comportamento rispetto al tipo di indice spaziale usato, alla politica di paginazione ed alla disposizione dei dati memorizzati. Parleremo in maniera piú approfondita di questo componente nel paragrafo 6.5.

I contributi appena descritti sono i piú importanti fra quelli forniti dalla libreria *OMSM*, ma ve ne sono altri di minore importanza la cui utilitá verrá messa in luce durante la descrizione del sistema realizzato.

La libreria *OMSM* é organizzata come una collezione di *classi C++*: la definizione e l'implementazione di questi componenti sono organizzate su due file indipendenti in modo da permetterne la modifica in maniera modulare. Solitamente la definizione di classe é contenuta nel *file di intestazione* (noto in inglese come *header file* e caratterizzato dall'estensione *".h"*), mentre l'implementazione é contenuta nel *file sorgente* (noto in inglese come *source file* e caratterizzato dall'estensione *".cpp"*). All'interno della libreria vi é una famiglia di componenti che sono stati realizzati attraverso tecniche di *programmazione generica*: si tratta di uno stile di programmazione basato sull'utilizzo di classi in cui il tipo di alcuni attributi é un parametro. Il linguaggio *C++* supporta classi generiche, dette *template*: inoltre si possono

definire anche funzioni template in cui il tipo degli argomenti o del valore di ritorno é un parametro. Le classi generiche sono particolarmente utili per definire contenitori di dati arbitrari e non noti durante la compilazione, mentre le funzioni template permettono di realizzare algoritmi generici in grado di operare su collezioni di dati arbitrari. La libreria *STL* (dall'inglese *Standard Template Library*) é ormai entrata a far parte delle *API* normalmente fornite da un compilatore *C++* e fornisce classi ed algoritmi basati sui template: per approfondimenti rifarsi a [STL94] e [CPP00]. L'uso della programmazione generica permette lo sviluppo di componenti generali e di facile utilizzo: la definizione e l'implementazione delle delle classi template sono tuttavia contenute in un unico file, come richiesto dagli standard *ANSI* del linguaggio *C++*. Nel paragrafo 6.3 descriveremo il componente *ERB-Tree*, che é una variante della struttura *RB–albero* introdotta nella sezione 5.2, in cui si é fatto largo uso dei template per memorizzare coppie di dati.

Il codice della libreria é ampiamente commentato in modo da facilitarne la comprensione e si puó generare la sua documentazione in maniera automatica attraverso il toolkit *Doxygen*, distribuito con licenza *GPL*: per approfondimenti sull'utilizzo di questo sistema software nella libreria *OMSM* rifarsi a [vH97] e [Can07b]. Durante lo sviluppo si é fatta molta attenzione alla compatibilitá del codice prodotto con gli standard *POSIX* in modo tale che la libreria potesse essere il piú portabile possibile e che il codice potesse essere lo stesso sui diversi compilatori utilizzati. Questo obiettivo ha richiesto la riformulazione di alcuni problemi in modo tale che questi fossero indipendenti dal tipo di piattaforma utilizzata e che la loro risoluzione fosse indipendente dalla presenza di una determinata *API*, anche se questa scelta ha spesso causato dei rallentamenti. In realtá ci sono due componenti della libreria in cui non é stato possibile garantire questa proprietá in quanto fruitori di servizi esterni: uno di questi é il timer per le operazioni, il quale si appoggia al clock della macchina su cui é attualmente in esecuzione, che descriveremo nel paragrafo 6.2. L'altro riguarda la gestione avanzata dei files dove viene sfruttata l'esecuzione di comandi nella shell fornita dal sistema operativo ed ovviamente ció non puó essere simulato con tecniche alternative. La piattaforma principale di sviluppo della libreria é stata la distribuzione *GNU/Linux Open SUSE 10.1* con l'ausilio del *GNU C++ Compiler 4.0*: per una trattazione approfondita di questo sistema software rifarsi a [Sta85], [GCC87], [TD01], [TAW+02], [Esu04] e [Osu05]. Su questa piattaforma la libreria é compilabile sia in forma statica e sia in forma dinamica: per approfondimenti rifarsi a [Can07b]. Grazie all'aderenza agli standard *POSIX*, il codice prodotto é ampiamente supportato dalla piattaforma *Microsoft Windows*, dove si é utilizzato l'ambiente integrato *Microsoft Visual Studio .NET*: per una trattazione approfondita di questo sistema software rifarsi a [Net02] ed a [MVS03]. Su questa piattaforma la libreria é usabile solamente in forma statica in quanto non si é voluto appesantire il sistema realizzato con l'interfaccia necessaria allo sviluppo delle *DLL* (dall'inglese *Dynamic Loading*

Library) per non peggiorare ulteriormente le prestazioni. Come vedremo nel capitolo 7, l'implementazione della libreria *OMSM* con questa piattaforma é piú lenta ed inefficiente, a paritá di codice e di macchina utilizzati: questo vuol dire che le implementazioni delle *API* sono meno efficaci su questa piattaforma. Inoltre bisogna osservare che non si é usato il framework *.NET*, ma solamente alcuni elementi di quella *Win32* in modo tale da garantire la compatibilitá della libreria *OMSM* con versioni precedenti di questo compilatore: questo fatto puó aver influito sull'efficienza delle operazioni eseguite.

Per il funzionamento della libreria *OMSM* é necessaria la presenza del toolkit *Oracle Berkeley DB*, del quale abbiamo ampiamente parlato nella sezione 4.4.2. Il prototipo realizzato ha una struttura modulare e, come vedremo nel paragrafo 6.5, sfrutta questo componente per la memorizzazione fisica dei modelli geometrici: in realtá il design della libreria *OMSM* consente la sostituzione di questo dispositivo senza stravolgere l'intera architettura.

La libreria sviluppata in questa tesi non contiene alcuno strumento per la visualizzazione dei modelli geometrici che é in grado di memorizzare, anche se puó essere utilizzata a tali scopi, vista la sua particolare struttura. Per approfondimenti sulle tecniche di visualizzazione di modelli geometrici in memoria secondaria rifarsi a [FS01], [CKS02], [SCC+02], [Lin03], [CGG+04], [RTBS05] e [CPK+05].

6.1.3 La gestione degli errori

Durante una computazione puó avvenire una situazione di errore, la quale deve essere gestita in maniera opportuna: in informatica, si usa il termine *eccezione* per descrivere l'occorrenza di diversi tipi di condizioni che alterano il normale flusso di controllo e l'esecuzione di un microprocessore o di un programma software. Tradizionalmente, il termine eccezione si applica a condizioni hardware: per esempio una divisione per zero causa un'eccezione hardware che interrompe l'esecuzione del programma. In realtá questo termine é stato riutilizzato in molti linguaggi di programmazione ad alto livello, come ad esempio il linguaggio *C++*, per indicare un meccanismo che interrompa l'esecuzione di un sottoprogramma e propaghi l'errore ad un livello piú alto del programma stesso. Quindi é necessario creare un'infrastruttura in grado di riconoscere la segnalazione dell'errore: per la descrizione dettagliata dei diversi tipi di eccezioni e degli usi impropri del termine *eccezione* rifarsi a [MV96].

Per le esigenze di questa tesi ci limiteremo alla gestione delle eccezioni di tipo software ed useremo i costrutti forniti dal linguaggio *C++*, come descritto in [Str86], [Eck00], [CPP00] e [EA03]: nella libreria *OMSM*, ogni eccezione é modellata dalla classe *OmsmException*, la cui implementazione é contenuta nel file *omsmexception.h* ed é riassunta dalla figura 6.1. Come si puó notare il suo stato interno contiene solamente la stringa *msg*, la quale memorizza la descrizione dell'errore che si vuole gestire.

```
class OmsmException{
 public:
 OmsmException(string msg);
 string getMessage();
 protected:
 string msg; };
```

Figura 6.1: l'implementazione della classe *OmsmException*

Vediamo ora le proprietá dei metodi che costituiscono l'interfaccia esterna della classe:

- il metodo *OmsmException(string msg)* é il costruttore di questa classe, il quale crea una nuova eccezione a partire dalla stringa *msg* contenente la descrizione dell'errore;
- il metodo *getMessage* restituisce la descrizione dell'errore.

Per semplificare la gestione degli errori, sono state definite delle sottoclassi di quella *OmsmException*: per brevitá non ne presenteremo le caratteristiche, rimandando a [Can07a] e [Can07b] per maggiori approfondimenti.

6.2 Il componente *OperationTimer*

Questo componente modella un timer, il quale scandisce il *tempo* all'interno delle applicazioni: il suo livello di precisione é il microsecondo e la sua efficienza gioca un ruolo fondamentale nelle prestazioni del sistema. Esso si basa sulla lettura del numero di cicli di clock di un elaboratore in un determinato istante di tempo t_0. Tale misura é nota in letteratura come *RDTSC*, dall'inglese *Read Time Stamp Counter*: storicamente é stata introdotta nei processori di classe *Pentium*, ma attualmente si é diffusa negli altri tipi di *CPU*. Per poter utilizzare questa misura per misurare il tempo necessario all'esecuzione di una certa operazione $\mathcal{O}$ é necessario eseguire questi passi:

- leggere il valore corrente dell'*RDTSC*, detto *valore iniziale*: nel seguito del paragrafo lo indicheremo con c_i;
- eseguire l'operazione $\mathcal{O}$;
- leggere il valore corrente dell'*RDTSC*, detto *valore finale*: nel seguito del paragrafo lo indicheremo con c_f;

A questo punto la differenza $c = c_f - c_i$ fornirá il tempo di esecuzione di $\mathcal{O}$, espresso in cicli di clock: questa misura é relativa alla specifica macchina sulla quale stiamo operando e potrebbe cambiare se usiamo un elaboratore con una frequenza di clock diversa. Inoltre il dato fornito non é significativo per l'utente in quanto egli potrebbe essere interessato ad una stima in secondi: vediamo come poter ottenere questa stima, che indicheremo nel seguito con S. Supponiamo che Q sia il numero di cicli del clock eseguiti in un secondo quindi il tempo di esecuzione S espresso in secondi sará:

$$S = \frac{c}{Q} \tag{6.1}$$

Questa tecnica fornisce una soluzione semplice ed elegante alla misura del tempo di esecuzione di $\mathcal{O}$, ma in realtá essa soffre di un problema estremamente insidioso: vediamo quale. Abbiamo assunto che i valori dell'*RDTSC* siano crescenti in maniera monotona, cioé $c_f > c_i$: il numero dei cicli del clock viene memorizzato in una variabile di tipo intero ad alta precisione (solitamente 64 bit), la quale potrá contenere un certo intervallo finito di valori, seppur molto *ampio*. Dunque siamo in presenza di un *contatore ciclico* e pertanto puó accadere che $c_f < c_i$. La probabilitá con cui avviene questo fenomeno dipende dalla velocitá della macchina utilizzata e dal tipo di operazione $\mathcal{O}$ eseguita: si puó intuire che questo fatto avviene piú frequentemente con macchine estremamente veloci e computazioni che richiedono *molto* tempo di calcolo. Quindi possiamo concludere che questa tecnica non fornisce una stima *robusta* del tempo di esecuzione per una certa operazione $\mathcal{O}$. Per le esigenze di questa tesi, la tecnica dell'*RDTSC* é piú che sufficiente: come vedremo nel capitolo 7, i tempi di calcolo non sono *molto* elevati e questo garantisce una certa stabilitá dei risultati.

Vediamo ora come é stata implementata la tecnica dell'*RDTSC* all'interno della libreria *OMSM*: il componente di riferimento é quello descritto dalla classe *OperationTimer*, la cui definizione é contenuta nel file *timerop.h* mentre la sua implementazione é descritta dal file *timerop.cpp*. Visto che l'operazione di lettura del clock non é comtemplata dalle librerie standard del *C++*, quella *OperationTimer* é l'unica classe della libreria *OMSM* che possiede diverse implementazioni, ognuna delle quali é adatta ad uno specifico calcolatore. Nel seguito del paragrafo vedremo come sia possibile creare due versioni di questo componente, una per il *GNU C++ Compiler 4.0* e l'altra per l'ambiente di sviluppo *Microsoft Visual Studio .NET*: queste piattaforme sono le uniche sulle quali é stato testato il funzionamento della libreria *OMSM*. Bisogna osservare che, nelle due versioni sviluppate, l'interfaccia esterna del componente é rimasta inalterata ed é composta dai seguenti metodi:

- il metodo *OperationTimer()* é il costruttore per questa classe, il quale si occupa anche di far partire il conteggio del timer;

- il metodo ~*OperationTimer()* é il metodo distruttore per questa classe, il quale si occupa anche di fermare il conteggio del timer;

- il metodo *startTimer()* si occupa delle operazioni necessarie a far partire il conteggio del timer: il suo compito é quello di leggere il valore c_i dell'*RDTSC*, sovrascrivendo l'eventuale valore precedente;

- il metodo *getTime()* fornisce la stima del tempo corrente a partire dall'ultima volta in cui é si é invocata la partenza del timer.

Vediamo ora l'implementazione adatta al *GNU C++ Compiler 4.0*: in questa piattaforma viene fornita la funzione

```
int gettimeofday( struct timeval *tp, void* tzp)
```

la quale é definita nel file di intestazione *sys/time.h*. Essa fornisce il numero di secondi e microsecondi che intercorrono fra l'istante corrente e la mezzanotte del 1 Gennaio 1970: questa data é nota in letteratura come l'inizio della *EPOCH UNIX* in quanto é il giorno in cui é stata rilasciata la prima versione del sistema operativo *UNIX*. Al suo interno sfrutta delle chiamate del sistema operativo per conoscere il valore corrente dell'*RDTSC* e quindi opera ad un livello piú astratto rispetto a quello che abbiamo descritto in precedenza. Questa funzione restituisce un valore nullo per segnalare l'assenza di errori durante l'operazione, un valore non nullo per segnalarne la presenza. Il risultato dell'invocazione viene salvato nel record *tp* di tipo *timeval*, il quale contiene i campi *tv_sec* per i secondi e *tv_usec* per i microsecondi: l'uso del puntatore *tzp* é rimasto per ragioni di compatibilitá in quanto obsoleto. Per ulteriori approfondimenti su questa funzione rifarsi a [Pic02]. La figura 6.2 mostra l'implementazione della classe *OperationTimer* con la piattaforma *GNU/Linux*.

Lo stato interno della classe é composto dalle strutture *_tstart* e *_tend*: la prima memorizza l'istante iniziale in cui é stato fatto partire il timer, mentre la seconda l'istante corrente in cui invochiamo *getTime*. É importante mettere in luce alcuni aspetti del codice in figura 6.2: come si puó notare, il metodo costruttore avvia il conteggio del timer, mentre quello distruttore non effettua alcuna operazione sullo stato interno del componente *OperationTimer*. Inoltre il metodo *startTimer* non fa altro che invocare la funzione *gettimeofday* per ottenere un'approssimazione dell'istante iniziale t_0, memorizzando il risultato nella variabile di istanza *_tstart* e facendo ripartire il conteggio del timer. Il metodo *getTime* é quello piú importante all'interno della classe: come prima operazione viene invocata la funzione *gettimeofday* per ottenere un'approssimazione dell'istante corrente t_c, memorizzando il risultato nella variabile di istanza *_tend*. A questo punto viene calcolata la differenza fra *_tstart* e *_tend*, la quale deve essere espressa in secondi per fornire una stima comprensibile per l'utente.

```
class OperationTimer {

public:

OperationTimer() { this->startTimer(); }

~OperationTimer() { ; }

void startTimer() { gettimeofday(&(this->_tstart), NULL); }

double getTime() {

  gettimeofday(&(this->_tend),NULL);
  double t1 = this->_tstart.tv_sec + this->_tstart.tv_usec;
  double t2 = this->_tend.tv_sec + this->_tend.tv_usec;
  return (t2-t1)/(1000*1000); }

protected:

struct timeval _tstart, _tend; };
```

Figura 6.2: l'implementazione della classe *OperationTimer* nella piattaforma di sviluppo *GNU/Linux*.

Vediamo ora l'implementazione adatta all'ambiente integrato *Microsoft Visual Studio .NET*: in questa piattaforma vengono fornite le funzioni

- `BOOL QueryPerformanceCounter( LARGE_INTEGER *tim)`
- `BOOL QueryPerformanceFrequency ( LARGE_INTEGER *frq)`

per estrarre rispettivamente il valore corrente dell'*RDTSC* e la frequenza del clock. Sono definite nel file di intestazione *windows.h* ed entrambe salvano il risultato in una struttura *LARGE_INTEGER*, la quale memorizza un intero a 64 bit. Inoltre restituiscono un valore nullo per segnalare la presenza di errori durante l'operazione ed un valore non nullo per segnalarne l'assenza: per ulteriori approfondimenti su queste funzioni rifarsi a [MSD07]. La figura 6.3 mostra l'implementazione della classe *OperationTimer* con l'ambiente integrato *Microsoft Visual Studio .NET*. Come si puó notare da un rapido confronto con la figura 6.2, questa implementazione opera in maniera analoga alla precedente, anche se con alcune lievi differenze dovute all'utilizzo delle funzioni *QueryPerformanceCounter* e *QueryPerformanceFrequency*.

In questo paragrafo abbiamo illustrato le caratteristiche principali della classe *OperationTimer*: per maggiori dettagli rifarsi a [Can07a] e [Can07b].

```
class OperationTimer {

public:

OperationTimer() {
  QueryPerformanceFrequency(&(this->freq));
  this->startTimer(); }

~OperationTimer() { ; }

void startTimer() { QueryPerformanceCounter(&this->_tstart); }

double getTime() {

  QueryPerformanceCounter(&(this->_tend));
  double d1=(double)this->_tend.QuadPart;
  double d2=(double)this->_tstart.QuadPart;
  double d3=(double)this->freq.QuadPart;
  return (d1 - d2)/d3; }

protected:

LARGE_INTEGER _tstart,_tend,freq; };
```

Figura 6.3: l'implementazione della classe *OperationTimer* nella piattaforma di sviluppo *Microsoft Visual Studio .NET*. Come si puó notare, le operazioni svolte ricalcano completamente lo schema dell'equazione 6.1.

É interessante valutare l'efficienza delle diverse implementazioni: nella sezione 7.1.2 descriveremo il programma *Timers*, il quale chiarisce come la versione *GNU/Linux* sia quella piú efficace.

6.3 La struttura dati *ERB–Albero*

Nella sezione 5.2 abbiamo parlato delle proprietá della struttura *RB–albero*, la quale garantisce prestazioni ottimali in quanto viene attivato un processo di bilanciamento dell'albero durante le operazioni di inserimento e di cancellazione. In questa tesi ne abbiamo implementato una variante che chiameremo *ERB–albero* (dall'inglese *Extended Red–Black Tree*): non si tratta di uno stravolgimento della struttura iniziale, quanto piuttosto del risultato di alcune modifiche alla definizione 5.5 in modo da estenderne le possibilitá di utilizzo. Nella sezione 6.3.1 vedremo la definizione e l'implementazione di questa struttura all'interno della libreria *OMSM*. Questa struttura sta alla base di altri due componenti molto importanti nell'ambito del siste-

ma realizzato come la cache per la memorizzazione di elementi e la tabella delle corrispondenze. Parleremo di questi componenti rispettivamente nelle sezioni 6.3.2 e 6.3.3.

6.3.1 Le proprietá della struttura dati *ERB–Albero*

La struttura dati *RB–albero* che abbiamo descritto nel paragrafo 5.2 garantisce l'efficienza delle operazioni di modifica, ma non é adatta ai nostri scopi in quanto:

- non gestisce in maniera flessibile le informazioni visto che il record *RBTreeNode* fornisce solo il campo *key* per memorizzare i dati;
- non facilita l'enumerazione dei dati memorizzati in quanto é supportata solamente la definizione ricorsiva di *visita*;
- non supporta una marca temporale per individuare l'ultimo accesso ai dati: questa proprietá é molto utile nel caching.

La struttura *ERB–albero* risolve questi problemi, aggiungendo le funzionalitá richieste alla struttura *RB–albero*: nel seguito vedremo quali modifiche verranno introdotte.

Per risolvere il primo problema ogni record viene suddiviso in due parti: una che lo identifica all'interno del sistema (detta parte *chiave* del record), mentre l'altra contiene l'informazione vera e propria (detta parte *dato*). Le operazioni di ricerca, inserimento e cancellazione vengono implementate sulla parte chiave di ogni record: un esempio di questa suddivisione é fornito dalla gestione dei dati all'interno del sistema *Oracle Berkeley DB*, del quale abbiamo parlato nella sezione 4.4.2. Per risolvere il secondo problema tutti i nodi della struttura vengono collegati fra loro da una lista doppiamente concatenata, la quale contiene l'insieme ordinato delle chiavi in senso crescente: in questo modo viene creato un indice secondario sulle informazioni memorizzate, permettendo di effettuare ricerche anche sulla parte dato del record. La figura 6.4 mostra un esempio di struttura *ERB–Albero*: le linee rosse continue rappresentano la lista doppiamente concatenata.

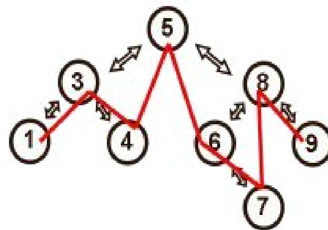


Figura 6.4: un esempio della struttura dati *ERB–albero*.

Per risolvere il terzo problema si puó introdurre un timer all'interno della struttura, il quale puó generare delle marche temporali, se necessario: solitamente si utilizza come tempo di riferimento l'istante in cui é stata creata la struttura *ERB–albero*. Per evitare il rischio di collisioni abbiamo deciso di aggiornare la marca temporale solamente per il nuovo nodo creato nell'operazione di inserimento e per il nodo obiettivo in quella di ricerca. Una volta messe in luce le caratteristiche di una struttura *ERB–albero*, é interessante capire come si modificano le primitive di ricerca, inserimento e cancellazione rispetto ad una classica struttura *RB–albero*. La presenza della lista non influenza l'operazione di ricerca di un certo elemento p e quindi questa primitiva rimane quella descritta nella sezione 5.2.3: l'unica differenza sta nella memorizzazione della marca temporale associata al nodo N_p, il quale contiene l'elemento p, se presente nell'albero. L'operazione di inserimento va modificata: per prima cosa si aggiunge il nuovo elemento p nell'albero secondo la primitiva descritta nella sezione 5.2.4, avendo cura di capire se la foglia N nella quale é stato inserito p sia il figlio sinistro o il figlio destro di un certo nodo P. Questa informazione serve ad aggiornare la lista doppiamente concatenata, eseguendo i seguenti passi:

- se il nodo N é il figlio sinistro di P, allora dobbiamo inserirlo nella lista come l'elemento immediatamente precedente rispetto a P;

- se il nodo N é il figlio destro di P, allora dobbiamo inserirlo nella lista come l'elemento immediatamente successivo rispetto a P.

Una volta aggiornata la lista doppiamente concatenata, dobbiamo memorizzare la nuova marca temporale associata al nodo N. L'operazione di cancellazione va anch'essa modificata: per prima cosa si cancella l'elemento p dall'albero secondo la primitiva descritta nella sezione 5.2.5 e poi si cancella l'elemento p dalla lista: non ci interessa modificare la marca temporale associata a p visto che l'elemento deve essere rimosso. Come si puó notare dalle nuove primitive, l'overhead necessario al mantenimento della lista doppiamente concatenata é trascurabile visto che la complessitá delle operazioni di gestione della lista hanno costo $\mathcal{O}(1)$ e ció garantisce l'efficienza delle primitive.

Vediamo ora come é stata implementata la struttura *ERB–albero* all'interno della libreria *OMSM*: le classi di riferimento sono quelle *ERBTreeNode* e *ERBTree*. Questi componenti sono in grado di memorizzare coppie generiche di dati usando i due template *Key* e *Data*: il primo template modella la parte *chiave* del record, mentre il secondo la parte *dato* del record. Una volta scelti quali ADT (o classi) associare a *Key* ed a *Data*, questi oggetti saranno descritti come l'istanziazione rispettivamente delle classi template *ERBTreeNode<Key,Data>* e *ERBTree<Key,Data>*, secondo la notazione del linguaggio *C++*. Tale linguaggio richiede che sia la definizione e sia l'im-

plementazione di una classe template siano contenute all'interno dello stesso file: per ulteriori approfondimenti rifarsi a [EA03].

```
template <class Key,class Data> class ERBTreeNode {

public:

 ERBTreeNode();

 ERBTreeNode(double luse);

 double getLastUseTime();

 void setLastUseTime(double luse);

 Key getKey();

 Data getData();

 void setData(Data newdata);

 ERBTreeNode<Key,Data> *getPrev();

 ERBTreeNode<Key,Data> *getNext();

protected:

 Key item;

 Data info;

 ERBTreeNode<Key,Data> *next, *prev, *left, *right, *parent;

 int colorNode;

 double luse; };
```

Figura 6.5: la definizione della classe template *ERBTreeNode*

Vediamo ora le caratteristiche della classe *ERBTreeNode*, la quale implementa un nodo della struttura *ERB–albero*. É interessante analizzare le caratteristiche dei vari campi dello stato interno, mostrati in figura 6.5:

- il campo *item* contiene la parte *chiave* del record che si vuole memoriz-

zare nel nodo: il suo tipo é dato dal template *Key*;

- il campo *info* contiene la parte *dato* del record che si vuole memorizzare nel nodo: come si puó notare, il suo tipo é dato dal template *Data*;

- i campi *next* e *prev* puntano rispettivamente al nodo successivo e a quello precedente il nodo corrente nella lista della struttura *ERB–albero*;

- i campi *left*, *right* e *parent* puntano rispettivamente al figlio sinistro, al figlio destro ed al padre del nodo corrente in accordo con la definizione 5.7, introdotta nella sezione 5.2.1;

- il campo *colorNode* contiene il colore del nodo corrente in accordo con la definizione 5.7, introdotta nella sezione 5.2.1;

- il campo *luse* contiene la marca temporale associata a questo nodo fornendo una stima dell'ultimo accesso effettuato.

La maggior parte dei metodi dell'interfaccia esterna serve per leggere e per modificare i campi dello stato interno del componente pertanto l'implementazione della classe *ERBTreeNode* non presenta aspetti di particolare interesse: per ulteriori approfondimenti su questi metodi si rimanda a [Can07a] e a [Can07b]. Inoltre bisogna osservare che questa classe puó essere considerata di *servizio*: non a caso nella sua definizione, la classe *ERBTree* é indicata come *classe friend*. Ricordiamo che nel linguaggio *C++* se una classe C_0 é dichiarata *friend* di quella C_1, allora i metodi di C_0 possono accedere alle variabili di istanza di C_1 senza restrizioni.

Il componente *ERBTree* implementa direttamente la struttura *ERB–Albero* ed é anch'essa una classe *generica*, definita sui template *Key* e *Data* con le stesse modalitá di quella *ERBTreeNode*. La classe *ERBTree* memorizza solamente il nodo radice della struttura: anche la definizione di questa classe, riassunta in figura 6.6, é memorizzata nel file *erbtree.h*. Nello stato interno della classe *EBTree* possiamo individuare le variabili di istanza:

- i campi *first* e *last* sono rispettivamente i puntatori al primo ed all'ultimo nodo della lista doppiamente concatenata all'interno della struttura dati *ERB–Albero*;

- i campi *root* e *NIL* sono rispettivamente i puntatori al nodo radice ed al nodo terminatore: per nodo terminatore si intende il nodo che sostituisce il puntatore speciale *NIL*, introdotto nella sezione 5.2.1 all'interno della definizione 5.7, in modo tale che ogni nodo abbia lo stesso numero di figli, garantendo le proprietá della struttura *RB–Albero*;

- il campo *mCount* contiene il numero di elementi memorizzati nella struttura *ERB–Albero*;

- il campo *ot* contiene il timer usato per generare le marche temporali, le quali segnalano l'ultimo accesso ad un nodo: questo componente viene attivato una volta creata la struttura dati *ERB–Albero*.

```
template <class Key,class Data> class ERBTree {

public:

 ERBTree() throw(AllocationErrorException);

 ~ERBTree();

 void makeEmpty();

 unsigned long getCount();

 ERBTreeNode<Key,Data> *getFirst();

 ERBTreeNode<Key,Data> *getLast();

 ERBTreeNode<Key,Data> *find(Key item);

 ERBTreeNode<Key,Data> *insert(Key item,Data data)
 throw(AllocationErrorException);

 ERBTreeNode<Key,Data> *remove(Key item);

 void remove(ERBTreeNode<Key,Data> *onode);

 ERBTreeNode<Key,Data> *getOldestNode();

protected:

 ERBTreeNode<Key,Data> *first,*last,*root,*NIL;

 unsigned long mCount;

 OperationTimer *ot; };
```

Figura 6.6: la definizione della classe template *ERBTree*.

Vediamo ora una breve descrizione dei metodi dell'interfaccia esterna:

- il metodo *ERBTree()* é il costruttore per questa classe, il quale crea una struttura dati *ERB–Albero* vuota;

- il metodo *~ERBTree()* é il distruttore per questa classe, il quale cancella anche tutti i record della struttura dati *ERB–Albero*;
- il metodo *makeEmpty()* cancella tutti i record presenti nella struttura dati *ERB–Albero*;
- il metodo *getCount()* restituisce il numero di record presenti nella struttura dati *ERB–Albero*;
- il metodo *getFirst()* restituisce il primo nodo della lista doppiamente concatenata nella struttura dati *ERB–Albero*;
- il metodo *getLast()* restituisce l'ultimo nodo della lista doppiamente concatenata nella struttura dati *ERB–Albero*;
- il metodo *find(Key item)* effettua la ricerca del record identificato dalla chiave *item*: se questo record appartiene alla struttura dati viene restituito il nodo che lo contiene, altrimenti il valore speciale *NULL*;
- il metodo *insert(Key item,Data data)* implementa l'inserimento nella struttura dati *ERB–Albero* del record identificato dalla chiave *item* e contenente l'oggetto *data*, restituendo il nuovo nodo creato;
- il metodo *remove(Key item)* implementa la cancellazione dalla struttura dati *ERB–Albero* del record identificato dalla chiave *item*, restituendo il nodo cancellato;
- il metodo *remove(ERBTreeNode<Key,Data> *onode)* implementa la cancellazione dalla struttura dati *ERB–Albero* del record contenuto nel nodo *onode*;
- il metodo *getOldestNode()* restituisce il nodo della struttura dati *ERB–Albero* che é stato acceduto meno di recente.

Alcuni di questi metodi sollevano l'eccezione *AllocationErrorException*, la quale segnala il verificarsi di un errore durante l'allocazione della memoria richiesta. Non presenteremo l'implementazione della classe *ERBTree* in quanto é piuttosto onerosa: per ulteriori approfondimenti sui metodi di questa classe si rimanda a [Can07a] e a [Can07b].

6.3.2 Il componente *Cache*

Nella sezione 3.4.2 abbiamo introdotto la tecnica del *caching* dei blocchi disco per ridurre il numero di accessi alla memoria secondaria: in questa sezione vedremo come questa tecnica possa essere generalizzata alla gestione di record modellati da coppie del tipo *chiave/dato* e resi persistenti in un supporto di memorizzazione.

L'idea é molto semplice: la cache contiene un insieme limitato di record in modo tale che l'operazione di lettura possa essere soddisfatta senza accedere alla memoria secondaria, se il record richiesto é salvato nella cache. La capacitá di questo componente é limitata e quando viene raggiunta bisogna sostituire un record attualmente memorizzato per far posto ad uno nuovo. Le politiche di sostituzione sono del tutto simili a quelle usate nella paginazione della memoria e sono state analizzate nella sezione 3.4.2. Per le esigenze di questa tesi useremo la politica di sostituzione *LRU* (dall'inglese *Least–Recently–Used*), la quale viene usata per sostituire il record acceduto meno recentemente fra quelli presenti nella cache.

```
template <class Key,class Data> class Cache {

public:

 Cache() throw(AllocationErrorException);

 ~Cache();

 unsigned long getDimension();

 unsigned long getStoredElementsNumber();

 bool contains(Key item);

 bool getData(Key item, Data &info);

 bool add(Key item,Data info) throw(AllocationErrorException);

 bool forceZombieKill(Key item);

 void getKeys(vector<Key> &keyset);

 void getDatas(vector<Data> &dataset);

protected:

 unsigned long dim;

 ERBTree<Key,Data> * keys; };
```

Figura 6.7: la definizione della classe template *Cache<Key,Data>*.

L'uso di questo componente deve essere trasparente per l'utente quindi le

operazioni di inserimento, cancellazione e ricerca devono essere implementate in maniera efficiente: nella libreria *OMSH* abbiamo sviluppato una cache basata sulla struttura dati *ERB–Albero* in grado di memorizzare coppie di tipo generico: la classe template *Cache<Key,Data>* é quella di riferimento ed é anch'essa definita tramite i template *Key* e *Data*, in maniera analoga a quella *ERBTree<Key,Data>*, introdotta nella sezione 6.3.1. La sua definizione (e quindi anche la sua implementazione) é contenuta nel file *cache.h* ed é riassunta nella figura 6.7. Lo stato interno di questa classe é composto dai seguenti campi:

- il campo *dim* contiene il numero massimo di elementi memorizzabili nella cache: solitamente questo valore viene detto *dimensione*;
- il campo *keys* contiene la struttura dati *ERB–Albero* che abbiamo introdotto nella sezione 6.3.1: useremo un componente di classe *ERBTree<Key,Data>* per memorizzare i record all'interno della cache.

Vediamo ora una breve descrizione dei metodi dell'interfaccia esterna:

- il metodo *Cache(unsigned long dim)* é il costruttore per questa classe il quale crea una cache, inizialmente vuota, di dimensione *dim*, sollevando l'eccezione *AllocationErrorException* in caso di errori durante l'allocazione della memoria;
- il metodo ∼*Cache* é il distruttore per questa classe, il quale elimina tutti i record memorizzati nella cache;
- il metodo *getDimension()* restituisce il numero massimo di record memorizzabili nella cache;
- il metodo *getStoredElementsNumber()* restituisce il numero corrente di record memorizzati nella cache;
- il metodo *contains(Key item)* controlla se la cache contiene il record identificato dalla chiave *item*;
- il metodo *getData(Key item,Data &info)* cerca di estrarre dalla cache il record identificato dalla chiave *item*: se il record richiesto é memorizzato nella cache allora nella variabile *info* ne verrá salvata la parte *dato* corrispondente e verrá restituito il valore di veritá *true*, altrimenti verrá restituito *false*, senza modificare il valore del parametro *info*;
- il metodo *add(Key item,Data info)* aggiunge il nuovo record identificato dalla chiave *item* e contenente l'oggetto *info*, attivando la politica di sostituzione *LRU* se la cache é piena: in caso di errori di memoria durante la creazione del nuovo record viene sollevata l'eccezione *AllocationErrorException*;

- il metodo *forceZombieKill(Key item)* forza la cancellazione del record identificato dalla chiave *item*: in questo modo si viola il principio di funzionamento di una cache il quale non prevede che i record vengano cancellati dall'esterno, ma solo internamente attraverso le scelte operate dalla politica di sostituzione;

- il metodo *getKeys(vector<Key> &keyset)* restituisce l'insieme ordinato in senso crescente delle chiavi delle coppie memorizzate nella cache;

- il metodo *getDatas(vector<Key> &dataset)* restituisce l'insieme delle parti *dato* dei record ordinati in senso crescente rispetto al valore delle chiavi memorizzate nella cache.

Come si puó intuire, l'implementazione dei metodi della classe *Cache* é concettualmente semplice in quanto ogni operazione deve essere eseguita, con le opportune modifiche, sulla struttura *ERB–Albero*. In realtá, essa contiene una serie di dettagli implementativi poco interessanti che non riporteremo: per approfondimenti rifarsi a [Can07a] e [Can07b].

6.3.3 La tabella delle corrispondenze

In questa tesi abbiamo l'esigenza di memorizzare una *tabella delle corrispondenze*, una struttura in grado di gestire un numero arbitrario di coppie di tipo generico, ognuna delle quali é identificata da una chiave. Il contenuto di questa struttura potrá variare in maniera dinamica, attraverso inserimenti e cancellazioni di record. Questo componente sará particolarmente utile nella memorizzazione dei nodi all'interno dei cluster, come vedremo nella sezione 6.5.4: a seconda della politica di suddivisione dei nodi dell'indice spaziale, ogni cluster potrá contenere un numero arbitrario di nodi, i quali potranno variare in maniera dinamica, a seconda delle operazioni di aggiornamento dell'indice. Per risolvere questo problema, nella libreria *OMSM* é stata sviluppata la classe template *CorrespondencesTable<Key,Data>*, la quale é anch'essa definita tramite i template *Key* e *Data* in maniera analoga a quella *ERBTree<Key,Data>*, introdotta nella sezione 6.3.1. La sua definizione (e quindi anche la sua implementazione) é contenuta nel file *corrstable.h* ed é riassunta nella figura 6.8. Lo stato interno di questa classe é composto solamente dal campo *keys*, il quale contiene la struttura dati *ERB–Albero*, introdotta nella sezione 6.3.1, che verrá utilizzata per la memorizzazione dei vari record. Dalla definizione della classe, si intuisce che ogni operazione deve essere eseguita, con le opportune modifiche, sulla struttura *ERB–Albero* contenuta nel campo *keys*. Per queste ragioni non ne mostreremo l'implementazione in quanto priva di motivi di particolare interesse, ma presenteremo solamente le caratteristiche piú importanti dell'interfaccia esterna della classe template *CorrespondencesTable<Key,Data>*.

```
template <class Key,class Data> class CorrespondencesTable {

public:

 CorrespondencesTable() throw(AllocationErrorException);

 ~CorrespondencesTable();

 unsigned long getStoredElementsNumber();

 bool contains(Key item);

 bool getData(Key item, Data &info);

 bool add(Key item,Data info) throw(AllocationErrorException);

 bool remove(Key item);

 void getKeys(vector<Key> &keyset);

 void getDatas(vector<Data> &dataset);

protected:

 ERBTree<Key,Data> * keys; };
```

Figura 6.8: la definizione della classe template *CorrespondencesTable*.

Vediamo ora una breve descrizione dei metodi appartenenti all'interfaccia esterna della classe:

- il metodo *CorrespondencesTable()* é il costruttore per questa classe, il quale crea una tabella delle corrispondenze inizialmente vuota, sollevando l'eccezione *AllocationErrorException* in caso di errori durante l'allocazione della memoria;
- il metodo ~*CorrespondencesTable()* é il distruttore per questa classe, il quale elimina tutte le corrispondenze memorizzate nella tabella;
- il metodo *getStoredElementsNumber()* restituisce il numero delle corrispondenze memorizzate nella tabella;
- il metodo *contains(Key item)* controlla se la tabella delle corrispondenze contiene il record identificato dalla chiave *item*;
- il metodo *getData(Key item,Data &info)* cerca di estrarre dalla tabella la coppia identificata dalla chiave *item*: se la corrispondenza richiesta

é memorizzata nella tabella allora nella variabile *info* ne verrá salvata la parte *dato* e verrá restituito il valore di veritá *true*, altrimenti verrá restituito *false*, senza modificare il valore del parametro *info*;

- il metodo *add(Key item,Data info)* aggiunge nella tabella la nuova coppia identificata dalla chiave *item* e contenente il dato *info*: in caso di errori di memoria durante la creazione della nuova corrispondenza viene sollevata l'eccezione *AllocationErrorException*;

- il metodo *remove(Key item)* rimuove dalla tabella la corrispondenza identificata dalla chiave *item*;

- il metodo *getKeys(vector<Key> &keyset)* restituisce l'insieme ordinato in senso crescente delle chiavi delle coppie memorizzate nella tabella;

- il metodo *getDatas(vector<Key> &dataset)* restituisce l'insieme delle parti *dato* dei record ordinati in senso crescente rispetto al valore delle chiavi memorizzate nella tabella.

Per ulteriori approfondimenti su questo componente rifarsi a [Can07a].

6.4 La gestione delle triangolazioni

Il primo contributo interessante fornito dalla libreria *OMSM* consiste nella creazione di un ambiente completo in grado di gestire mappe poligonali, solitamente salvate su file in differenti formati. Il framework realizzato ha quindi bisogno di una architettura di I/O in grado di trattare sia caratteri *ASCII* e sia dati binari: nella sezione 6.4.1 vedremo le caratteristiche della soluzione proposta in questa libreria. Una volta realizzata l'infrastruttura di I/O, bisogna gestire i modelli geometrici in input: per risolvere questo problema, é stata realizzata una architettura operante su due livelli in grado di operare su mappe poligonali a prescindere dal loro formato ed in maniera trasparente per l'utente, la quale verrá descritta nella sezione 6.4.2. Caricato un modello $\mathcal{M}$, possiamo:

- verificare le proprietá della mappa poligonale Γ che lo approssima, trasformandola eventualmente in una triangolazione;

- convertire la triangolazione di $\mathcal{M}$ in un insieme non organizzato di triangoli (in inglese *triangles soup*);

- decomporre l'insieme non organizzato di triangoli in un indice spaziale.

Queste operazioni devono essere eseguite in sequenza: le descriveremo rispettivamente nelle sezioni 6.4.3, 6.4.4 e 6.4.5.

6.4.1 L'architettura di I/O su file

L'architettura di I/O che abbiamo realizzato estende le caratteristiche dell'infrastruttura per la gestione dei file fornita dal linguaggio *C++*, introducendo un livello di astrazione, il quale rende piú semplici ed intuitive le operazioni di I/O sui tipi predefiniti del linguaggio utilizzato. Sono stati realizzati due componenti, uno in grado di gestire le operazioni di input mentre l'altro quelle di output ed entrambi possono essere abilitati alla gestione dei caratteri *ASCII* o dei dati binari, a seconda delle esigenze. Prima di fissarne le caratteristiche, é necessario descrivere in cosa si differenzia la gestione dei dati binari da quella dei caratteri nel sistema realizzato.

La gestione dei dati binari é piuttosto semplice in quanto bisogna operare su una sequenza di byte di una certa lunghezza, a seconda del tipo di operazione richiesta. Ad esempio, se vogliamo leggere un valore di tipo *double* dobbiamo estrarre otto byte ed interpretarli come un unico dato: nella sezione 7.1.1 presenteremo le caratteristiche del programma *Types*, il quale si occupa di visualizzare il numero di byte ed il dominio dei valori rappresentabili con i tipi predefiniti del linguaggio *C++*. In questo ambito, non ci occuperemo dei *finali* dei byte in quanto accettiamo passivamente l'allineamento della sequenza sulla quale operare, delegandone la gestione all'applicazione che sfrutta i dati binari.

Operare sui caratteri richiede invece una sequenza di istruzioni piú articolata: uno dei problemi piú importanti é la corretta gestione delle terminazioni di linea (spesso indicate con *EOL*, dall'inglese *End–Of–Line*). Ogni sistema operativo sfrutta una propria combinazione di caratteri per indicare la terminazione di una linea e questo fatto puó essere problematico per la condivisione di file scritti su diverse piattaforme. Per ovviare a questo problema, si possono gestire le linee di caratteri contenute in un file attraverso la libreria *Getline*, definita nello standard *POSIX*: per approfondimenti su questo strumento software rifarsi a [Pic02]. In questo modo la gestione dei terminatori é demandata alla versione di tale libreria presente sulla piattaforma di sviluppo: é importante capire quali conseguenze comporta questa scelta rispetto alle operazioni sui dati. Per quanto riguarda la gestione dell'output di caratteri, questa scelta comporta la scrittura di una linea alla volta, la quale é solitamente rappresentata con una stringa e ció non pone particolari problemi. Invece la gestione dell'input di caratteri prevede la presenza di un buffer dove memorizzare la linea appena letta e la conseguente suddivisione in *token*. Per le esigenze di questa tesi, un token é una sequenza non vuota di caratteri diversi da quelli di controllo e dal carattere *blank* (il carattere “ ”). L'utilizzo di questo buffer é piuttosto semplice: si possono consumare i token attualmente memorizzati ed una volta che il buffer é vuoto, si legge una nuova linea, se presente nel file. Dunque la gestione del buffer é trasparente per l'utente, il quale non é interessato alla provenienza del token. Nella libreria *OMSM* una linea é memorizzata in una stringa e

la suddivisione in token é effettuata dal componente modellato dalla classe *StringHandler*, la quale si occupa della gestione delle stringhe. La definizione di questa classe é descritta nel file *stringhandlers.h* mentre la sua implementazione é contenuta nel file *stringhandlers.cpp*: essa non presenta particolari motivi di interesse e per questa ragione si rimanda a [Can07a] per maggiori approfondimenti su questo componente.

Ora possiamo descrivere come vengono implementati i due diversi componenti dell'architettura di I/O su file, quello di input e quello di output.

Il componente di input é modellato dalla classe *DataReader*, la cui definizione é contenuta nel file *datareader.h* mentre la sua implementazione é memorizzata nel file *datareader.cpp*: questo componente é configurabile per supportare in maniera esclusiva la gestione dei caratteri *ASCII* o dei dati binari utilizzando le tecniche introdotte in precedenza. Per queste ragioni, la sua interfaccia esterna é dotata di moltissimi metodi e quindi, per brevitá, vedremo nella figura 6.9 solamente la definizione di quelli piú significativi, mentre per gli altri si rimanda a [Can07a] e [Can07b].

```
DataReader(string fname,bool bin)
throw(NoAvailableDataSourceException);

string getData(disk_address &pos)
throw(ForbiddenOperationException, NoDataAvailableException,
DataReadingErrorException);

void readValue(double &d) throw(ForbiddenOperationException,
NoDataAvailableException,DataReadingErrorException);
```

Figura 6.9: i metodi piú significativi della classe *DataReader*

Vediamo alcune caratteristiche dei metodi della figura 6.9:

- il metodo *DataReader(string fname,bool bin)* crea un nuovo componente per leggere i dati dal file *fname* ed abilita la gestione dei dati binari se *bin* assume il valore di veritá *true* e la gestione dei caratteri se *bin* vale *false*: se non é possibile leggere dal file richiesto viene sollevata l'eccezione *NoAvailableDataSourceException*;
- il metodo *getData(disk_ address &pos)* estre un token da un file contenente caratteri *ASCII*, memorizzandone in *pos* la posizione all'interno del file: solleva l'eccezione *ForbiddenOperationException* se questo componente supporta la gestione dei dati binari, l'eccezione *NoDataAvailableException* se nel file non ci sono nuovi token e l'eccezione *DataReadingErrorException* se avviene un errore durante la lettura;
- il metodo *readValue(double &d)* legge otto byte da un file contenente

dati binari e li interpreta come un valore di tipo *double*, salvandolo in *d*: solleva l'eccezione *ForbiddenOperationException* se questo componente supporta la gestione dei caratteri *ASCII*, l'eccezione *NoDataAvailableException* se nel file non ci sono nuovi byte e l'eccezione *DataReadingErrorException* se avviene un errore durante la lettura.

Come si puó osservare dalla figura 6.9, il metodo *readValue* legge un valore di tipo *double*: per gli altri tipi predefiniti del linguaggio *C++*, esistono dei metodi con lo stesso nome, i quali si occupano di leggere i valori del tipo voluto sotto forma di dati binari.

Il componente di output é modellato dalla classe *DataWriter*, la cui definizione é contenuta nel file *datawriter.h* mentre la sua implementazione é contenuta nel file *datawriter.cpp*: questo componente é molto simile a quello *DataReader*, come testimonia la figura 6.10, la quale mostra la definizione dei metodi piú significativi fra quelli dell'interfaccia esterna della classe *DataWriter*. Per gli altri metodi si rimanda a [Can07a] e [Can07b].

```
DataWriter(string fname,bool bin,bool app)
throw(NoAvailableDataDestinationException);

void writeData(string line) throw(ForbiddenOperationException,
DataWritingErrorException);

void writeValue(double d) throw(ForbiddenOperationException,
DataWritingErrorException);
```

Figura 6.10: i metodi piú significativi della classe *DataWriter*.

Vediamo le caratteristiche dei metodi piú importanti della figura 6.10:

- il metodo *DataWriter(string fname, bool bin, bool app)* crea un nuovo componente per scrivere dati sul file *fname*, abilitando la gestione dei dati binari se *bin* assume il valore *true* oppure la gestione dei caratteri se *bin* vale *false* ed attivando la modalitá *append* se *app* vale *true* oppure quella di sovrascrittura se *app* vale *false*: se non é possibile scrivere sul file richiesto solleva l'eccezione *NoAvailableDataDestinationException*;

- il metodo *writeData(string line)* scrive una nuova linea *line* su un file contenente caratteri *ASCII*, aggiungendo il carattere di fine linea relativo al sistema operativo che si sta usando: solleva l'eccezione *ForbiddenOperationException* se questo componente supporta la gestione dei dati binari e quella *DataWritingErrorException* se avviene un errore durante la scrittura della linea;

- il metodo *writeValue(double d)* scrive gli otto byte che compongono il valore *d*, di tipo *double*, su un file contenente dati binari: solleva l'eccezione *ForbiddenOperationException* se questo componente supporta la gestione dei caratteri e solleva l'eccezione *DataWritingErrorException* se avviene un errore durante la scrittura.

Come si puó osservare dalla figura 6.10, il metodo *writeValue* scrive un valore di tipo *double*: per gli altri tipi predefiniti del linguaggio *C++*, esistono dei metodi con lo stesso nome che si occupano di scrivere i valori del tipo voluto sotto forma di dati binari.

6.4.2 Il caricamento delle mappe poligonali

La prima fase della nostra ricerca si basa sul caricamento di un certo modello geometrico $\mathcal{M}$, il quale é solitamente fornito tramite una triangolazione (piú in generale una mappa poligonale) memorizzata su un file. Quindi la lettura del modello in input dipende dal formato in cui é codificato il complesso simpliciale: nella libreria *OMSM* é stato realizzato un ambiente completo in grado di gestire differenti formati di mappe poligonali in maniera trasparente per l'utente. Questo ambiente lavora su due livelli:

- un livello esterno, detto *MeshHandler*, il quale fornisce agli utenti le primitive con cui gestire una certa mappa poligonale;
- un livello interno, detto *MeshFormatHandler*, nel quale vengono implementate le primitive del livello superiore a seconda dello specifico formato della mesh.

Questo tipo di architettura fornisce un approccio modulare alla risoluzione del problema ed é facilmente estendibile in quanto l'aggiunta di un nuovo formato non interessa (o quasi) il livello *MeshHandler*, ma solamente quello *MeshFormatHandler*. Questi due livelli sono stati realizzati attraverso due classi che, per semplicitá, hanno lo stesso nome dei layer: la classe *MeshHandler* realizza il livello esterno, mentre la classe *MeshFormatHandler* quello interno. La classe *MeshFormatHandler* ha una sottoclasse per ogni formato della mesh ed é implementata come una *classe astratta*, cioé una classe che non puó avere istanze dirette, ma che serve a definire i metodi comuni delle sottoclassi: per poter attivare il binding dinamico é necessario che i suoi metodi siano *virtuali*, cioé contrassegnati dalla parola chiave *virtual*, come abbiamo accennato nella sezione 6.1.1. I formati supportati sono conosciuti a priori dal livello *MeshHandler*, il quale li identifica attraverso la loro estensione e decide quale sottoclasse del componente *MeshFormatHandler* utilizzare. I formati riconosciuti sono:

- il formato *OFF* (dall'espressione inglese *Object File Format*), il quale é definito in [Sha01] e [Att06]: le mesh memorizzate in questo formato hanno estensione *"off"*;

- il formato *PLY* (noto anche come *Stanford Triangle Format*), il quale é definito in [Wal01] e [Att06]: le mesh memorizzate in questo formato hanno estensione *"ply"*;

- il formato *VER–TRI*, il quale é un formato proprietario utilizzato all'*IMATI–CNR* di Genova e definito in [Att06]: le mesh memorizzate in questo formato hanno estensione *"tri"*;

- il formato *TSOUP* (dall'espressione inglese *Triangles Soup*), il quale é stato introdotto in questa tesi per gestire un insieme non organizzato di triangoli: come vedremo nella sezione 6.4.4 le mesh memorizzate in questo formato hanno estensione *"tsoup"*.

Vediamo come viene implementato il livello *MeshHandler* all'interno della libreria *OMSM*: come abbiamo giá accennato, viene modellato dalla classe *MeshHandler*, la cui definizione é contenuta all'interno del file *mshandler.h* mentre la sua implementazione é memorizzata nel file *mshandler.cpp*. Questa classe fornisce all'utente tre operazioni eseguibili su una certa mesh e ad ogni primitiva corrisponde un metodo della sua interfaccia esterna, come riassunto dalla figura 6.11.

```
class MeshHandler {

 public:

 MeshHandler();

 ~MeshHandler();

 void checkMesh(string fname) throw(AllocationErrorException,
 MeshAnalysisErrorException);

 void toSoup(string fname) throw(AllocationErrorException,
 MeshAnalysisErrorException);

 void decompose(string meshname,string optsname)
 throw(AllocationErrorException, MeshAnalysisErrorException);

 protected:

 MeshFormatHandler *mfh; };
```

Figura 6.11: la definizione della classe *MeshHandler*.

É interessante riassumere brevemente il funzionamento di questi metodi:

- il metodo *MeshHandler()* é il costruttore per questa classe;
- il metodo *~MeshHandler()* é il distruttore per questa classe;
- il metodo *checkMesh(string fname)* controlla le proprietá della mappa poligonale contenuta nel file *fname*, sollevando l'eccezione *AllocationErrorException* se avviene un errore durante l'allocazione della memoria oppure quella *MeshAnalysisErrorException* se la mesh in input non soddisfa le proprietá richieste: parleremo dei requisiti di una mesh e del funzionamento di questo metodo nella sezione 6.4.3;
- il metodo *toSoup(string fname)* converte la triangolazione contenuta nel file *fname* in un insieme non organizzato di triangoli (definito dal formato *TSOUP*), sollevando l'eccezione *AllocationErrorException* se avviene un errore durante l'allocazione della memoria oppure quella *MeshAnalysisErrorException* se il processo di conversione fallisce: il risultato della conversione verrá memorizzato in un file secondo un formato introdotto in questa tesi, come vedremo nella sezione 6.4.4;
- il metodo *decompose(string meshname,string optsname)* decompone la triangolazione contenuta nel file *meshname* in un'istanza del framework *OMSM* modellata dalle impostazioni contenute nel file *optsname*, sollevando l'eccezione *AllocationErrorException* se avviene un errore durante l'allocazione della memoria oppure quella *MeshAnalysisErrorException* se il processo di decomposizione fallisce: approfondiremo il funzionamento di questo metodo nella sezione 6.4.5.

Come si puó notare dalla figura 6.11, lo stato interno della classe *MeshHandler* é costituito dal solo componente *MeshFormatHandler* e questa proprietá é particolarmente importante per l'implementazione delle tre primitive. I tre metodi dell'interfaccia esterna della classe *MeshHandler* si occuperanno di creare la sottoclasse di quella *MeshFormatHandler* piú adatta alla gestione della mesh in input, memorizzandola nel campo *mfh*: a questo punto l'esecuzione del metodo verrá propagata sul nuovo oggetto creato. Il componente *MeshFormatHandler* (piú precisamente una sua sottoclasse) contiene una serie di sottocomponenti, compresa una rappresentazione della mappa poligonale, sui quali verranno eseguite le operazioni richieste in modo tale da poterle implementare direttamente sul formato richiesto.

Vediamo ora come viene gestito il livello *MeshFormatHandler* all'interno della libreria *OMSM*: come abbiamo giá accennato viene modellato dalla classe astratta *MeshFormatHandler*, la cui definizione é contenuta all'interno del file *meshformathandler.h*. La figura 6.12 mostra l'interfaccia esterna della classe *MeshFormatHandler*.

```
class MeshFormatHandler {

 public:

 MeshFormatHandler();

 ~MeshFormatHandler();

 virtual void checkMesh(string fname)
 throw(AllocationErrorException, MeshAnalysisErrorException);

 virtual void toSoup(string fname)
 throw(AllocationErrorException, MeshAnalysisErrorException);

 virtual void decompose(string meshname,string optsname)
 throw(AllocationErrorException, MeshAnalysisErrorException);

 virtual bool handleTsoup(); };
```

Figura 6.12: l'interfaccia esterna della classe *MeshFormatHandler*

Come si puó notare dalla figura 6.12, l'interfaccia esterna di questo componente é simile a quella della classe *MeshHandler*: l'unico metodo che non ha un corrispettivo nel livello superiore é quello *handleTsoup()*, il quale controlla se questo componente é in grado di gestire mappe poligonali codificate nel formato *TSOUP*: parleremo di questo formato nella sezione 6.4.4. Ogni sottoclasse di quella *MeshFormatHandler* gestisce uno specifico formato: la figura 6.13 mosta le classi attualmente disponibili.

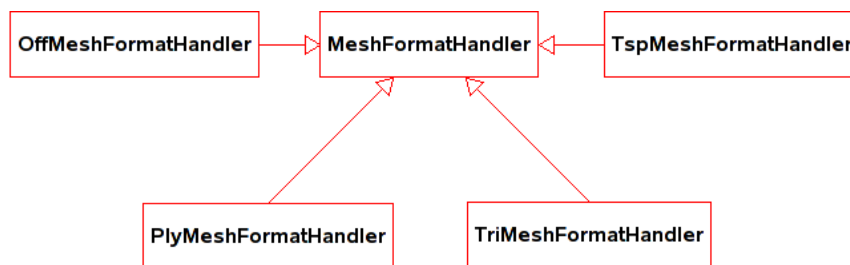


Figura 6.13: il diagramma di ereditarietá per la classe *MeshFormatHandler*.

Riassumiamo brevemente le caratteristiche delle sottoclassi di quella *MeshFormatHandler*:

- la classe *OffMeshFormatHandler* gestisce le mappe poligonali in forma-

to *OFF*: la sua definizione é contenuta nel file *offhandler.h*, mentre la sua implementazione é memorizzata nel file *offhandler.cpp*;

- la classe *PlyMeshFormatHandler* gestisce le mappe poligonali in formato *PLY*: la sua definizione é contenuta nel file *plyhandler.h*, mentre la sua implementazione é memorizzata nel file *plyhandler.cpp*;
- la classe *TriMeshFormatHandler* gestisce le triangolazioni in formato *VER–TRI*: la sua definizione é contenuta nel file *trihandler.h*, mentre la sua implementazione é memorizzata nel file *trihandler.cpp*.
- la classe *TspMeshFormatHandler* gestisce le triangolazioni in formato *TSOUP*: la sua definizione é contenuta nel file *tsphandler.h*, mentre la sua implementazione é memorizzata nel file *tsphandler.cpp*.

Nel seguito della trattazione assumeremo note le specifiche dei formati supportati perché sono molto noti ed utilizzati nelle applicazioni: per approfondimenti rifarsi alla vastissima letteratura sull'argomento, come ad esempio [Sha01], [Wal01] e [Att06]. É importante osservare che i primi tre formati supportati sono molto simili in quanto modellano le mappe poligonali attraverso la struttura dati *indicizzata*, descritta in [PS85]: con questa tecnica viene fornito l'elenco completo dei vertici della suddivisione simpliciale in input, ognuno dei quali é identificato da un certo indice. Di conseguenza, é naturale descrivere ogni faccia della mesh come la sequenza degli indici che identificano i vertici appartenenti al suo contorno. Lo scopo di ogni formato é quello di codificare questa struttura dati attraverso particolari convenzioni, definite nelle sue specifiche. Pertanto la corretta implementazione delle primitive nelle sottoclassi del componente *MeshFormatHandler* deve tenere conto della variante della struttura indicizzata, a seconda del formato gestito: come si puó intuire, l'implementazione dei quattro dispositivi in figura 6.13 é piuttosto onerosa, anche se la maggior parte del codice puó essere riutilizzato. Per queste ragioni non vedremo nel dettaglio il funzionamento di ogni singolo componente: per approfondimenti sull'argomento rifarsi a [Can07a] e [Can07b]. Tuttavia é possibile osservare che ogni sottoclasse sfrutta una piattaforma comune di sottocomponenti, rappresentati in figura 6.14: con la notazione *DataWriter(3)* vogliamo esprimere il fatto che ci sono tre componenti di questo tipo in questa classe. Nel seguito riassumeremo brevemente il funzionamento dei sottocomponenti della classe *MeshFormatHandler*.

Il componente *DataReader*, introdotto nella sezione 6.4.1, viene utilizzato per leggere dal file gli elementi sintattici che descrivono la mappa poligonale in input: puó essere abilitato alla gestione dei dati binari o dei caratteri a seconda dello specifico formato.

Due dei tre componenti di tipo *DataWriter* (che abbiamo introdotto nella sezione 6.4.1) sono adibiti alla scrittura della triangolazione che si ottiene dalla mappa poligonale in input: come giá accennato in precedenza, é possibile triangolare le facce non triangolari del complesso simpliciale in modo

da ottenere una griglia di triangoli. Il formato della mesh puó prevedere solamente una parte *ASCII* (come il formato *OFF*), solamente una parte binaria oppure entrambe (come nel formato *PLY*) e quindi la presenza di questi due componenti permette di adattare la scrittura alle varie esigenze. Nella sezione 6.4.3 approfondiremo il funzionamento di questo meccanismo.

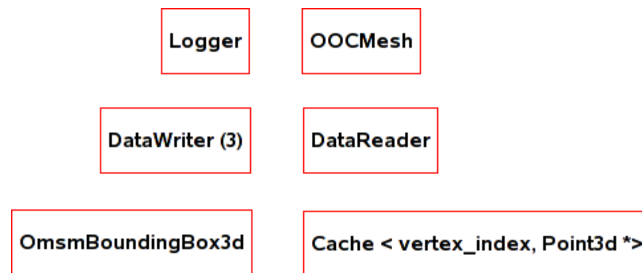


Figura 6.14: la piattaforma comune di sottocomponenti per le sottoclassi del dispositivo *MeshFormatHandler*. Con la notazione *DataWriter(3)* vogliamo esprimere il fatto che ci sono tre componenti di questo tipo.

Il rimanente componente di tipo *DataWriter* é adibito alla scrittura del risultato della conversione di una mesh in un insieme non organizzato di triangoli sul file di destinazione e quindi alla scrittura di una mesh in formato *TSOUP*. Questo componente supporta la gestione dei caratteri *ASCII*: vedremo le caratteristiche dell'operazione di conversione nella sezione 6.4.4.

Il componente *OOCMesh* memorizza una mappa poligonale in memoria secondaria: visto che i modelli utilizzati in questa tesi eccedono la dimensione della memoria primaria, non é possibile salvarli interamente in *RAM*. Per minimizzare l'occupazione spaziale, non vengono memorizzate informazioni riguardanti le facce visto che non é prevista la possibilitá di agire localmente sulla mesh in tempi differenti: le primitive del componente *MeshHandler* operano sull'intera mappa poligonale, caricando le facce in maniera sequenziale e quindi é necessario conoscere soltanto le coordinate dei punti. Per ogni vertice viene memorizzato l'indirizzo disco dove é possibile leggerne le coordinate cartesiane in modo da poterle eventualmente recuperare. La definizione della classe *OOCMesh* é contenuta nel file *oocmesh.h*, mentre la sua implementazione é memorizzata nel file *oocmesh.cpp*: non presenta particolari motivi di interesse pertanto si rimanda a [Can07a] per ulteriori approfondimenti.

Il componente *Logger* é una variante del dispositivo *DataWriter* la quale si occupa di mantenere traccia su un file, detto *file di log*, delle operazioni svolte in modo da fornirne un resoconto in una forma comprensibile per l'utente: questa tecnica é nota con il termine inglese *logging* ed é del tutto analoga a quella illustrata nella sezione 4.3. In questo modo, in caso di errore,

se ne possono capire le ragioni analizzando il file di log. La definizione di questo dispositivo é contenuta nel file *logger.h* mentre la sua implementazione é memorizzata nel file *logger.cpp*. L'unico metodo offerto da questa classe é

```
void logMessage(string log) throw(ImpossibleLoggingException)
```

il quale si occupa di scrivere un messaggio nel file di log, sollevando l'eccezione *ImpossibleLoggingException* se avviene un errore durante la scrittura.

Il componente *OmsmBoundingBox3d* implementa un *iper–rettangolo* nello spazio euclideo $\mathbb{E}^3$: esso memorizza la *MBB* della mappa poligonale. La definizione di questa classe é contenuta nel file *omsmbbox3d.h*, mentre la sua implementazione é memorizzata nel file *omsmbbox3d.cpp*: questa classe non presenta particolari motivi di interesse pertanto si rimanda a [Can07a] e [Can07b] per ulteriori approfondimenti.

Il componente *Cache<vertex_index,Point3d *>* é una istanza della classe template *Cache*, della quale abbiamo parlato nella sezione 6.3.2. Come abbiamo visto in precedenza, il componente *OOCMesh* tiene traccia dell'indirizzo disco dei vertici in modo tale da poterne recuperare le coordinate, le quali potranno essere memorizzate nella cache in modo da evitare delle operazioni di lettura dal disco fisso. In questa cache ogni vertice é identificato da un indice, memorizzato in un valore di tipo *vertex_index*, mentre le sue coordinate euclidee sono memorizzate in un oggetto modellato dalla classe *Point3d*, la cui definizione é contenuta nel file *geom3d.h*, mentre la sua implementazione é memorizzata nel file *geom3d.cpp*: questa classe non presenta particolari motivi di interesse pertanto si rimanda a [Can07a] e [Can07b] per ulteriori approfondimenti.

6.4.3 La verifica delle triangolazioni

Il primo passo nell'analisi di un certo modello geometrico $\mathcal{M}$ consiste nel controllare che $\mathcal{M}$ verifichi alcune proprietá importanti nell'economia di questa tesi in modo da poter essere correttamente analizzato all'interno della libreria *OMSM*. Per avviare il processo di verifica di una certa mappa poligonale bisogna invocare il metodo:

```
void checkMesh(string fname,string optsname)
throw(AllocationErrorException, MeshAnalysisErrorException)
```

della classe *MeshHandler*, definito su tutti i formati supportati. La fase di verifica di una mappa poligonale si occupa di tre aspetti:

- il controllo sulla natura dei valori memorizzati all'interno del file in quanto vogliamo trattare solamente dati rappresentabili;
- l'analisi degli elementi sintattici in accordo con le regole stabilite dal formato della mesh memorizzata in un certo file;

- l'analisi delle facce della mappa poligonale in modo tale che siano triangoli non degeneri.

Il controllo dei valori numerici é necessario visto che la libreria *OMSM* supporta differenti piattaforme, le quali potrebbero fornire diversi insiemi di valori rappresentabili: queste verifiche garantiscono di poter operare con dati corretti per l'architettura in questione. Nella sezione 7.1.1 vedremo le caratteristiche del programma *Types*, il quale si occupa di visualizzare il numero di byte ed il dominio dei valori rappresentabili con i tipi predefiniti del linguaggio *C++*: in questo modo si ha una descrizione completa dell'architettura corrente. Nella sezione 6.4.1 abbiamo visto come sia possibile estrarre token e dati binari da un file utilizzando i componenti della libreria *OMSM*: tuttavia dobbiamo controllare l'allineamento dei byte per i dati binari visto che i componenti dell'architettura di I/O non lo gestiscono.

Come prima operazione bisogna comprendere il tipo di finale della piattaforma utilizzata: nella libreria *OMSM* é stata sviluppata la classe *PlatformDescription*, la quale si occupa della descrizione del sistema software corrente. La definizione di questa classe é contenuta nel file *os.h*, mentre la sua implementazione é contenuta nel file *os.cpp*: la figura 6.15 mostra l'implementazione dei due metodi piú significativi.

```
bool isBigEndian() {

  long x = 0x41424344;
  char *s = (char *)&x;
  if(s[0]=='D') return true;
  else return false; }

bool isLittleEndian() {

  long x = 0x41424344;
  char *s = (char *)&x;
  if(s[0]=='A') return true;
  else return false; }
```

Figura 6.15: l'implementazione dei due metodi piú significativi della classe *PlatformDescription*.

Il metodo *isBigEndian()* controlla se la piattaforma corrente é a finale grande, mentre il metodo *isLittleEndian()* controlla se la piattaforma corrente é a finale piccolo. Nella figura 6.15 il valore x non é altro che la rappresentazione numerica della stringa *"ABCD"*: in una macchina big–endian i byte meno significativi vengono memorizzati negli indirizzi piú piccoli quindi nell'ordine *"DCBA"*, mentre in una macchina little–endian avviene il contrario.

Una volta capito il finale della macchina sulla quale si sta operando e supponendo noto il finale dei dati da gestire, bisogna invertire l'ordinamento della sequenza di byte sotto esame, se i due finali risultano differenti. Nella libreria *OMSM* é stata sviluppata la classe *BytesOrdering*, la quale si occupa della gestione di questo problema: la sua definizione é contenuta nel file *border.h* mentre la sua implementazione é memorizzata nel file *border.cpp*. Questa classe fornisce il metodo

```
void reverseOrder(byte_array val,unsigned long nbytes)
throw(NullPointerException)
```

per invertire la sequenza binaria *val* di lunghezza *nbytes*, sollevando l'eccezione *NullPointerException* se avvengono violazioni di memoria: per maggiori approfondimenti su questa classe rifarsi a [Can07a]. L'operazione di allineamento dei byte é importante sia per la fase di lettura e sia per la fase di scrittura e verrá usata anche nel sistema di persistenza degli oggetti che studieremo nella sezione 6.5.1. Nella sezione 7.1.3 vedremo il funzionamento dei programmi *Cubel* e *Cubeb*, i quali forniscono un metodo per valutare l'overhead introdotto da questa primitiva.

Una volta corretto l'allineamento dei byte, possiamo attivare il processo di verifica: limitiamo la nostra analisi ai tipi predefiniti del linguaggio *C++*. Per risolvere questo problema é stata sviluppata la classe *CppTypeStringHandler*, la cui definizione é contenuta nel file *cpparser.h* mentre la sua implementazione é memorizzata nel file *cpparser.cpp*. Come si puó intuire, questa classe contiene moltissimi metodi: nel seguito vedremo una breve descrizione di quelli piú importanti, riassunti nella figura 6.16, mentre per gli altri si rimanda a [Can07a] e [Can07b].

```
double string2double(string str,bool bin)
throw(NoDoubleStringException, DoubleUnderflowValueException,
DoubleOverflowValueException);

string double2string(double d,bool bin);
```

Figura 6.16: i due metodi piú significativi della classe *CppTypeStringHandler*.

Il metodo *string2double* converte la stringa *str* in un valore di tipo *double*: bisogna interpretare il contenuto della stringa come byte se *bin* vale *true*, altrimenti come caratteri. La conversione puó fallire per vari motivi:

- la stringa *str* non puó essere convertita in un valore valido: in questo caso viene sollevata l'eccezione *NoDoubleStringException*;
- la stringa *str* contiene un valore troppo piccolo per essere rappresentato, situazione detta di *underflow*: in questo caso viene sollevata l'eccezione *DoubleUnderflowValueException*;

- la stringa *str* contiene un valore troppo grande per essere rappresentato, situazione detta di *overflow*: in questo caso viene sollevata l'eccezione *DoubleOverflowValueException*.

Invece il metodo *double2string* converte un valore d in una stringa la quale sará composta da byte se *bin* vale *true* altrimenti da caratteri.

L'analisi degli elementi sintattici memorizzati nel file in input procede in accordo con le regole stabilite dal formato della mesh e quindi varia a seconda delle sottoclassi introdotte nella sezione 6.4.2. Tuttavia é possibile scomporre la risoluzione di questo problema in diverse parti, risolvibili piú facilmente in maniera indipendente: ad esempio possiamo individuare la lettura dell'intestazione del file, la lettura del numero di vertici, del numero delle facce, di un certo indice, di una coordinata e via dicendo. Per risolvere ognuna di queste sottoparti del problema é stato sviluppato un metodo opportuno, non appartenente all'interfaccia esterna, in modo da semplificarne la soluzione e garantire un approccio modulare: nelle successive operazioni sulla mesh possiamo riutilizzare il codice prodotto. Il funzionamento generale di questi metodi é quello di estrarre dal file un elemento sintattico di riferimento (un token o una stringa binaria), controllare se questo é rappresentabile nella piattaforma corrente (secondo quanto discusso in precedenza) e controllare se soddisfa i requisiti richiesti dallo specifico formato supportato. Ad esempio é interessante controllare che:

- l'intestazione del file sia coerente con la sua estensione;
- la mesh contenga almeno tre vertici ed una faccia;
- ogni elemento sintattico sia di tipo corretto rispetto alla sua funzione: ad esempio un indice sulle facce deve essere un intero positivo ed indicare una faccia valida.

Come si puó intuire, l'implementazione di questi metodi é piuttosto onerosa e dipende dal formato che vogliamo gestire: per brevitá non la presenteremo, rimandando a [Can07a] e [Can07b] per ulteriori approfondimenti.

L'analisi delle facce della mappa poligonale in modo tale che siano triangoli non degeneri é un'importante funzionalitá fornita dalla libreria *OMSM*: vediamone le idee principali. Una volta lette le coordinate euclidee di una faccia γ della mappa poligonale, dobbiamo verificare che γ sia un poligono planare e semplice: in caso contrario viene sollevata l'eccezione *MeshAnalysisErrorException*. Nei formati supportati il contorno di un poligono γ é composto da un'unica componente connessa, descritta dai suoi vertici (dagli indici dei suoi vertici, nel caso indicizzato) e l'ordine in cui compaiono determina l'orientamento di γ. In queste ipotesi, un politopo:

- é *semplice* se la sequenza dei suoi vertici forma una catena semplice e senza autointersezioni, se non nei punti comuni fra gli spigoli;

- é *planare* se tutti i suoi vertici appartengono ad uno stesso piano.

Nella libreria *OMSM* é stata realizzata una serie di classi che descrivono le entitá geometriche immerse negli spazi euclidei $\mathbb{E}^2$ e $\mathbb{E}^3$:

- la classe *Point2d* modella un punto immerso in $\mathbb{E}^2$;
- la classe *Triangulator* modella un componente per la triangolazione di un poligono immerso in $\mathbb{E}^2$;
- la classe *Point3d* modella un punto immerso in $\mathbb{E}^3$;
- la classe *Vector3d* modella un vettore immerso in $\mathbb{E}^3$
- la classe *Segment3d* modella un segmento immerso in $\mathbb{E}^3$;
- la classe *Face3d* modella un politopo immerso in $\mathbb{E}^3$.

Le loro definizioni si trovano nei file *geom2d.h* e *geom3d.h*, mentre le loro implementazioni nei file *geom2d.cpp* e *geom3d.cpp*: queste classi implementano alcuni concetti fondamentali della geometria computazionale, ampiamente trattati in [PS85], [Hil94] e [O'R94]. Questi componenti non presentano particolari motivi di interesse pertanto si rimanda a [Can07a] e [Can07b] per ulteriori approfondimenti. Una volta verificato che un poligono γ sia planare e semplice, dobbiamo controllare che γ sia un triangolo ed in caso contrario bisogna imporre questa condizione. Ovviamente questa operazione non sará supportata dai formati *VER–TRI* e *TSOUP* in quanto possono modellare solamente triangoli e non poligoni qualsiasi.

Il primo aspetto da controllare é la triangolabilitá di γ: in [BDE98] si dimostra che questo problema é *NP–completo* nello spazio euclideo $\mathbb{E}^3$. Tuttavia vengono fornite alcune euristiche per semplificarne la risoluzione, le quali si basano sul fatto che nello spazio $\mathbb{E}^2$ il concetto di *poligono triangolabile* é ben definito ed esiste un'ampia letteratura a riguardo: per approfondimenti rifarsi ad esempio a [FM84], [Cha91], [Sei91], [Tou91], [NM94], [AGR01] e [Hel01]. L'idea chiave consiste nel proiettare un politopo tridimensionale γ nello spazio $\mathbb{E}^2$, ottenendo un poligono γ': γ sará triangolabile se γ' é semplice. É sufficiente che esista una proiezione $\varphi : \mathbb{E}^3 \rightarrow \mathbb{E}^2$, che garantisca le proprietá richieste.

Visto che i poligoni in input sono semplicemente connessi, semplici e planari possiamo utilizzare la proiezione ortogonale su un certo piano cartesiano, scelto in modo tale che il poligono non degeneri in un segmento oppure non ci siano vertici coincidenti: questa situazione avviene, ad esempio, se il poligono giace sul piano XY e si cerca di proiettare sul piano XZ. Per semplicitá non valuteremo le proprietá del poligono γ' in quanto é sufficiente che esista almeno un piano cartesiano che ci garantisca che il poligono γ' sia semplice. I seguenti metodi della classe *Face3d*

```
bool canProjectXYplane(vector<Point3d> vet)

bool canProjectXZplane(vector<Point3d> vet)

bool canProjectYZplane(vector<Point3d> vet)
```

controllano se una catena di punti tridimensionali memorizzata in *vet* (e quindi un poligono immerso in $\mathbb{E}^3$) possa essere proiettata rispettivamente sui piani cartesiani XY, XZ e YZ.

Una volta individuato un piano cartesiano adatto ai nostri scopi, é possibile proiettare il poligono di partenza attraverso i metodi della classe *Face3d*

```
void onXYplane(vector<Point3d> vet,vector<Point2d> &vet2d)

void onXZplane(vector<Point3d> vet,vector<Point2d> &vet2d)

void onYZplane(vector<Point3d> vet,vector<Point2d> &vet2d)
```

i quali proiettano il politopo memorizzato in *vet* rispettivamente sui piani cartesiani XY, XZ e YZ, memorizzando il risultato in *vet2d*.

Una volta che ci siamo ridotti al caso bidimensionale, possiamo triangolare il poligono: l'algoritmo di triangolazione utilizzato é quello *ear–cutting*, introdotto in [PS85]. Questa tecnica é molto semplice da implementare, tuttavia la sua complessitá é quadratica nel numero di vertici in input: il componente che si occupa della sua realizzazione é quello modellato dalla classe *Triangulator*, la cui definizione é contenuta nel file *geom2d.h*, mentre la sua implementazione é memorizzata nel file *geom2d.cpp*. Il metodo

```
triangulate(vector<Point2d> &cntr,vector<Point2d> &triangles)
```

si occupa di triangolare un poligono, il cui contorno é definito da una lista di punti bidimensionali contenuta in *cntr*. Il risultato di questa operazione é memorizzato nella lista *triangles*, nella quale ogni triangolo é identificato da una tripla di punti bidimensionali consecutivi.

Una volta triangolato il poligono bidimensionale γ', bisogna riportare i triangoli ottenuti nello spazio metrico euclideo $\mathbb{E}^3$, ottenendo la triangolazione del politopo γ di partenza. In realtá non é necessario applicare la proiezione inversa: stiamo usando delle mesh codificate in formato indicizzato e quindi ogni punto tridimensionale di partenza avrá un codice identificativo, il quale viene memorizzato anche nella sua proiezione bidimensionale. In questo modo é possibile estrarre da ogni triangolo la tripla degli indici dei punti tridimensionali di partenza.

Dopo aver triangolato una faccia, bisogna apportare le modifiche alla mappa poligonale di partenza: la risoluzione di questo problema offre alcuni spunti di riflessione interessanti. La scansione delle facce avviene in maniera sequenziale e, per non sovrascrivere dati non ancora analizzati, si é scelto

di riscrivere in un file distinto la mappa poligonale, opportunamente revisionata, man mano che ne vengono verificati gli elementi sintattici. Questa operazione viene gestita dai due componenti *DataWriter*, introdotti nella figura 6.14 a questo proposito: nel seguito vedremo su quali principi si basa questo processo. Il problema piú importante é causato dalle proprietá dell'infrastruttura di I/O fornita dal linguaggio *C++*, la quale non prevede la possibilitá di poter scrivere, su un file giá esistente, nuovi dati in posizione arbitraria e senza distruggerne il contenuto: questa limitazione é importante per i nostri scopi perché non ci permette di aggiornare correttamente il numero delle facce in quei formati composti da due parti di tipo differente (una parte binaria ed una a caratteri *ASCII*), dove é necessario riaprire piú volte il file modificando il tipo di dati gestibili, come avviene nel formato *PLY*. Dovendo gestire questo tipo di formato, la riscrittura della mappa poligonale viene suddivisa in due parti, ognuna delle quali avviene su un file diverso. Nel primo file viene scritta l'intestazione della mesh nella quale aggiorniamo il numero totale di facce dopo aver terminato la fase di analisi e quindi contiene la parte *ASCII* della mappa. Nel secondo file vengono scritti i vertici e le facce in accordo con il formato definito dagli elementi sintattici *vertex* e *face* e quindi il file contiene la parte binaria della mappa. Una volta che tutte le facce sono state verificate, i due file vengono concatenati formando la triangolazione risultante. Anche i formati *OFF* e *PLY* (nella loro versione *ASCII*) supportano questo tipo di correzioni: in questo caso l'analisi ne risulta ovviamente facilitata visto che non dobbiamo modificare le modalitá di apertura del file di partenza.

Proviamo a determinare la complessitá dell'operazione di verifica di una mappa poligonale: supponiamo di partire da una mesh avente N vertici e F facce: a prescindere dal suo formato dobbiamo leggere tutti i vertici e controllare la validitá di tutte le facce. Nel caso migliore, non dobbiamo triangolare alcuna faccia e quindi la complessitá diventa $\mathcal{O}(N + F)$: se il dominio della triangolazione é una varietá allora vale $F \sim 2N$ (come dimostrato in [Ede87]) e quindi la complessitá diventa $\mathcal{O}(N)$. Nel caso peggiore dobbiamo triangolare tutte le F facce: a priori non conosciamo il numero di vertici di ogni faccia e quindi é complicato stabilire la complessitá di questa operazione. In realtá le mappe poligonali solitamente utilizzate nelle applicazioni sono generate tramite un processo di acquisizione e quindi le loro facce saranno triangoli oppure quadrilateri: facce con un numero maggiore di lati sono poco frequenti e quindi in questo caso la complessitá sará ancora $\mathcal{O}(N + F)$, visto che la triangolazione di un quadrilatero si puó ottenere attraverso una sua diagonale.

In base a queste considerazioni é consigliabile eseguire questa operazione una volta sola in una fase di preprocessing e solamente in quei casi in cui non si é sicuri che il modello da gestire sia rappresentabile sulla piattaforma corrente. Nella sezione 7.2.1 descriveremo il funzionamento del programma *Check*, il quale si occupa di controllare che il modello in input verifichi le pro-

prietá richieste: in caso contrario questo programma cercherá di correggere le violazioni, se possibile.

6.4.4 La conversione delle triangolazioni

Il secondo passo dell'analisi di un certo modello geometrico $\mathcal{M}$ consiste nel convertire la mappa poligonale γ che lo approssima in un insieme non organizzato di triangoli in modo da poterli facilmente decomporre nell'indice spaziale: si assume che γ abbia superato con successo i controlli effettuati dalla primitiva *checkMesh*. Il formato che descrive il risultato di questa fase é detto *TSOUP* (dalla contrazione dell'espressione inglese *Triangles Soup*), introdotto per le esigenze di questa tesi. I formati *OFF*, *VER–TRI* e *PLY* sfruttano la struttura indicizzata e quindi non permettono l'accesso diretto alle coordinate euclidee di ogni faccia. Invece, nel formato *TSOUP*, ogni triangolo viene descritto dalle coordinate euclidee dei suoi tre vertici: ovviamente ogni vertice compare nella definizione di tutti i triangoli che lo condividono nel loro contorno e quindi la rappresentazione della mesh non é piú compatta. Inoltre viene memorizzato l'indice di ogni vertice nella mesh di partenza (in forma indicizzata) in modo da poterla ricostruire, se necessario. La figura 6.17 mostra la rappresentazione di un triangolo nello spazio euclideo $\mathbb{E}^3$ in questo formato, permettendoci di studiarne le caratteristiche.

```
TSOUP
 NVERTICES 3
 NTRIANGLES 1
 BBOX
 0.0 1.0
 0.0 1.0
 0.0 1.0
 0 1.0 0.0 0.0
 1 0.0 1.0 0.0
 2 0.0 0.0 1.0
```

Figura 6.17: esempio di mesh in formato *TSOUP*.

Come si puó notare, si tratta di un formato *ASCII*: l'intestazione contiene la parola chiave *TSOUP* che ne dichiara il tipo. Segue la dichiarazione del numero di vertici, introdotta dalla parola chiave *NVERTICES* (nel nostro caso ci sono tre vertici), poi quella del numero di triangoli, introdotta dalla parola chiave *NTRIANGLES* (nel nostro caso vi é un solo triangolo), ed infine la dichiarazione della *MBB* della mesh, cioé il piú piccolo rettangolo tridimensionale che contiene al suo interno tutti i vertici della triangolazione. Questa definizione é introdotta dalla parola chiave *BBOX* e richiede gli intervalli dei

valori sui tre assi cartesiani a cui appartengono le coordinate dei vertici: nel nostro caso le coordinate appartengono all'intervallo $[0, 1]$ su ogni asse. Infine trovano posto le dichiarazione dei triangoli, ognuno dei quali é descritto dai suoi tre vertici: ogni vertice contiene il suo indice nella mesh di partenza e poi le sue tre coordinate cartesiane. Questo formato fornisce informazioni sulla mesh che ci saranno molto utili nella fase di decomposizione.

Per attivare la conversione di una generica mesh nel formato *TSOUP* bisogna invocare il metodo

```
void toSoup(string fname) throw(AllocationErrorException,
MeshAnalysisErrorException)
```

della classe *MeshHandler*: ovviamente questa operazione non viene eseguita se la mesh é giá in questo formato. Per semplificarne l'implementazione, il metodo *toSoup* condivide con il metodo *checkMesh* gran parte del codice necessario al parsing di uno specifico formato, discusso nella sezione 6.4.3. Anche in questa primitiva viene utilizzata la tecnica del logging per documentare le operazioni svolte attraverso il componente *Logger*.

Proviamo a valutare la complessitá di questa operazione: supponiamo di partire da una triangolazione avente N vertici e T triangoli scritta in un qualche formato indicizzato. Per poterla convertire nel formato *TSOUP*, dobbiamo leggere le coordinate dei vertici, memorizzarne gli indirizzi disco ed infine analizzare tutti i triangoli, scrivendone le rappresentazioni. Quindi in generale la complessitá di questa operazione é $\mathcal{O}(N + T)$: se il dominio della triangolazione é una varietá allora vale $T \sim 2N$ e quindi la complessitá diventa $\mathcal{O}(N)$. Anche questa operazione é piuttosto onerosa: in realtá puó essere eseguita una volta sola nella fase di preprocessing. Nella sezione 7.2.2 vedremo il funzionamento del programma *ToSoup*, il quale converte il formato di una mesh in quello *TSOUP*.

6.4.5 La decomposizione delle triangolazioni

Il terzo passo dell'analisi di un certo oggetto $\mathcal{M}$ prevede la decomposizione di una sua griglia di triangoli all'interno di un certo indice spaziale in modo da semplificarne le interrogazioni geometriche. Nel capitolo 5 abbiamo visto come gli indici spaziali usino le coordinate euclidee di un punto multidimensionale per capire dove memorizzarlo quindi é necessario poter accedere in maniera efficiente alla geometria di $\mathcal{M}$. Nel nostro caso vogliamo indicizzare triangoli e quindi possiamo considerare un punto rappresentativo dell'entitá spaziale da memorizzare (solitamente il centroide del triangolo) in modo tale da poter utilizzare le tecniche descritte nel capitolo 5. Quindi i formati *OFF*, *VER–TRI* e *PLY* non sono adatti ai nostri scopi, vista la loro struttura indicizzata: invece quello *TSOUP*, introdotto nella sezione 6.4.4, ne facilita l'implementazione. Per avviare il processo di decomposizione bisogna invocare il metodo:

```
void decompose(string meshname,string optsname)
throw(AllocationErrorException, MeshAnalysisErrorException)
```

della classe *MeshHandler* e ció avviene solamente se il nome del file della mesh, contenuto nella stringa *meshname*, ha estensione *"tsoup"*: non é possibile attivarlo con gli altri formati. La stringa *optsname* contiene il nome del file, il quale contiene le impostazioni con cui personalizzare il funzionamento dell'architettura per la memorizzazione dei dati che descriveremo nel paragrafo 6.5. Anche in questa primitiva viene utilizzata la tecnica del logging per documentare le operazioni svolte.

Ricordando la struttura di un file scritto nel formato *TSOUP*, come quello in figura 6.17, é possibile effettuarne l'analisi utilizzando gran parte dei componenti sviluppati nella sezione 6.4.3. Il processo di decomposizione avviene in tre fasi distinte:

- nella prima fase bisogna prevedere la lettura dell'intestazione del file e quindi del numero dei vertici e dei triangoli;
- nella seconda fase bisogna estrarre la *MBB* della mesh, in quanto la presenza di questo componente facilita la gestione degli iper–quadranti per gli indici spaziali sviluppati;
- nella terza fase bisogna leggere tutti i triangoli appartenenti alla mesh ed inserirli nell'indice.

Visto che abbiamo a disposizione tutti i triangoli della mesh, sarebbe estremamente utile una tecnica di tipo *a blocco* (in letteratura si parla di tecniche di tipo *bulk*) per decomporre il modello geometrico $\mathcal{M}$ in un'unica operazione. Putroppo non tutti gli indici spaziali prevedono tecniche di questo tipo, anche se in letteratura sono presenti dei tentativi di creazione di framework in grado di gestire queste problematiche, come quelli definiti in [AYCD98], [AAPV01] e [vdBS01]. Quindi l'implementazione della primitiva *decompose* prevede l'inserimento iterativo dei triangoli del modello geometrico $\mathcal{M}$ nell'indice spaziale. Il tipo della struttura dati utilizzata per decomporre il modello geometrico $\mathcal{M}$ é del tutto trasparente per questa primitiva quindi non é possibile stabilirne a priori la complessitá in quanto essa dipende dalle caratteristiche dell'indice spaziale usato per memorizzare i triangoli. Infatti nel prototipo del *DBMS* realizzato in questa tesi sono stati implementate varie strutture di indicizzazione come quelle *Octree*, *K–d tree*, *Hybrid PR Quadtree* e *Hybrid PR K–d tree*: nel capitolo 5 ne abbiamo delineato le caratteristiche fondamentali.

Nella sezione 7.3.1 mostreremo le proprietá del programma *Omsmconf*, il quale facilita la configurazione del sistema realizzato: nella sezione 7.3.2 vedremo le caratteristiche del programma *Decompose*, il quale si occupa di decomporre una certo insieme non ordinato di triangoli attraverso un certo indice spaziale.

6.5 La memorizzazione dei dati spaziali

Il secondo contributo di una certa rilevanza fornito dalla libreria *OMSM* consiste nella definizione di un prototipo per un *DBMS* di tipo embedded in grado di memorizzare molte tipologie di componenti geometrici.

In letteratura sono state sviluppate una serie di tecniche, le quali gestiscono la memorizzazione dei dati geometrici in memoria secondaria. Questo problema é stato ampiamente trattato in letteratura e le soluzioni proposte sono divenute ormai di uso comune: sono stati sviluppati esempi di sistemi per la gestione di dati spaziali come quelli descritti in [IBM01], [BGK04] e [OSP05]. Nei capitoli 3 e 4 abbiamo delineato rispettivamente le proprietá piú importanti delle tecniche utilizzate nella memoria secondaria e l'organizzazione di una base di dati.

Possiamo osservare che il funzionamento di tali architetture si basa sull'integrazione di questi aspetti:

- l'utilizzo di una struttura ausiliaria di accesso ad albero, detta *indice spaziale*, per facilitare le operazioni di aggiornamento e di interrogazione sui dati: alcuni esempi di indici sono descritti in [Sam06];

- la suddivisione dei nodi dell'indice spaziale in cluster secondo una certa politica in modo da minimizzare il numero di accessi alla memoria secondaria, la quale é piú lenta di quella primaria: per *cluster* si intende un gruppo di nodi, scelti in base ad un certo criterio, i quali possono essere considerati un'unica entitá;

- la gestione dinamica dei cluster in memoria secondaria: questo aspetto puó essere gestito attraverso varie tecniche di memorizzazione, le quali dipendono anche dalla distribuzione fisica dei dati.

Come si puó notare, questi tre aspetti possono essere considerati indipendenti fra loro e le tecniche utilizzate per la loro gestione possono essere combinate in maniera ortogonale. Molte architetture di memorizzazione prevedono una serie di scelte fissate a priori, soprattutto per quanto riguarda gli ultimi due aspetti, mentre é oramai consuetudine fornire la possibilitá di scegliere fra piú indici spaziali. Ad esempio, l'architettura *GiST*, introdotta in [HNP95], permette di variare solamente il tipo di indice spaziale, mentre la sua politica di clustering é quella introdotta in [DRSS96]: inoltre questo framework é capace di gestire solamente un database locale.

In questo paragrafo viene introdotta la struttura del framework *OMSM* (dall'espressione inglese *Objects Management in Secondary Memory*) per la gestione di grosse quantitá di dati geometrici in memoria secondaria: questa architettura puó integrare fra loro le diverse tecniche sviluppate in letteratura e garantisce la massima flessibilitá possibile nella risoluzione di questo problema. Questo framework é in grado di memorizzare un certo insieme di

entitá geometriche di dimensione arbitraria: l'utilizzo di una tecnica di persistenza indipendente dal tipo di oggetto rende possibile rappresentare oggetti geometrici di dimensione topologica diversa, ma immersi nello stesso spazio metrico euclideo. Descriveremo le tecniche di persistenza utilizzate nella sezione 6.5.1 e quelle di rappresentazione degli oggetti geometrici nella sezione 6.5.2. L'insieme dei dati da memorizzare puó essere di dimensioni elevate ed eccedere la quantitá di *RAM* presente in un calcolatore, pertanto il sistema *OMSM* cerca di fornire un'architettura che faciliti la risoluzione di queste problematiche: la sua struttura ricalca quella del modello *PDM*, introdotto nella sezione 3.4.3. Infatti si compone di tre livelli, i quali interagiscono fra loro come riassunto dalla figura 6.18.

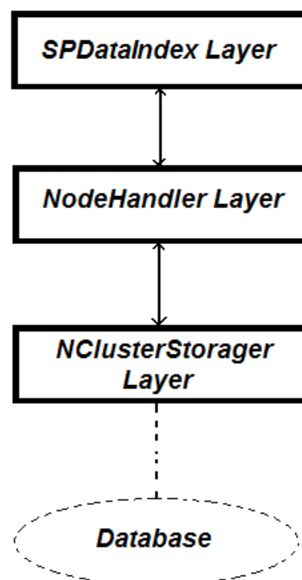


Figura 6.18: le relazioni che intercorrono fra i moduli del framework *OMSM*.

Riassumiamo brevemente le caratteristiche piú importanti dei tre livelli del framework *OMSM*:

- il livello *SPDataIndex* fornisce l'interfaccia esterna fra l'architettura *OMSM* e l'utente, permettendo di interagire con i dati memorizzati: nella sezione 6.5.3 ne discuteremo le proprietá in maniera approfondita;

- il livello *NodeHandler* si occupa della gestione dei nodi dell'indice spaziale, suddividendoli in cluster in maniera tale da minimizzare il numero delle operazioni da eseguire sul supporto di memorizzazione: discuteremo le proprietá di questo livello nella sezione 6.5.4;

- il livello *NClusterStorager* si occupa della gestione a basso livello dei cluster sul supporto di memorizzazione: discuteremo le proprietá di questo livello nella sezione 6.5.5.

La linea tratteggiata in figura 6.18 enfatizza il fatto che non é nota a priori la tecnologia utilizzata per la memorizzazione fisica dei cluster quindi non é definita la relazione esistente fra il livello *NClusterStorager* ed il *database*: per memorizzare i cluster possiamo usare una soluzione di tipo embedded oppure una gerarchia di database (come quella introdotta in [CMZ04a] e [CMZ04b]) e quindi le tecniche utilizzate sono diverse a seconda dei casi.

Come abbiamo giá accennato nella sezione 1.2, l'architettura *OMSM* fornisce una piattaforma comune con la quale implementare le operazioni sui dati, utilizzando le funzionalitá fornite dai tre livelli appena descritti. A priori non viene garantita l'efficienza delle operazioni sui dati in quanto essa é la conseguenza delle scelte effettuate in fase di configurazione: nella sezione 6.5.6 vedremo i meccanismi che permettono di modificare il comportamento del framework *OMSM*.

6.5.1 La persistenza degli oggetti

Uno dei problemi piú importanti all'interno del sistema di memorizzazione che abbiamo realizzato é quello della *persistenza* degli oggetti da indicizzare: come abbiamo ampiamente discusso nei capitoli 3 e 4 vogliamo rendere permanenti le modifiche apportate al database in modo da poterle riutilizzare in successive computazioni.

Dunque il nostro obiettivo é quello di memorizzare in maniera efficiente gli oggetti geometrici, modellati da una classe del linguaggio *C++*: questo linguaggio non offre alcun supporto per la persistenza degli oggetti quindi bisogna affidarsi a librerie esterne come possono essere quelle che descrivono i framework *CORBA* (dall'inglese *Common Object Request Broker Architecture*) e *Microsoft .NET*: per approfondimenti su questi software rifarsi rispettivamente a [Cor91] e [Net02]. Il loro funzionamento esula dagli scopi di questa tesi, anche perché entrambi i sistemi non forniscono una soluzione al problema della persistenza degli oggetti che sia compatibile con le diverse piattaforme: ad esempio la tecnologia *Corba* é ormai obsoleta e ne esistono varie implementazioni, incompatibili fra loro. Diverso é il discorso per la piattaforma *Microsoft .NET*: essa costituisce una parte importante del sistema operativo *Microsoft Windows*, ma ne esiste una versione anche per *GNU/Linux*, la quale peró supporta una porzione ridotta delle funzionalitá offerte. Questa versione é realizzata dal *Progetto Mono*: per approfondimenti rifarsi a [Mon04]. Per queste ragioni abbiamo sviluppato un sistema per la persistenza degli oggetti basato sulla rappresentazione di un certo componente attraverso una sequenza di byte.

Ogni oggetto $\mathcal{O}$ possiede uno stato interno, composto da un certo numero di campi di diverso tipo: la sequenza di byte che rappresenta $\mathcal{O}$ é la concate-

nazione delle rappresentazioni binarie dei campi interni. La sequenza di byte che descrive un campo interno dipende dal tipo della variabile di istanza:

- se la variabile di istanza ha un tipo predefinito, cioé fornito dal linguaggio *C++*, la sua sequenza di byte sará banalmente quella che ne descrive il valore;

- se la variabile di istanza é a sua volta un oggetto si applica ricorsivamente questo schema di rappresentazione.

L'ordine di concatenazione delle sequenze binarie dei vari campi interni di un certo oggetto $\mathcal{O}$ é noto a priori e gli elementi della rappresentazione sono a finale piccolo: per velocizzare le operazioni di estrazione, la rappresentazione a byte viene mantenuta all'interno di $\mathcal{O}$ ed aggiornata in presenza di modifiche allo stato interno. Quindi ogni oggetto $\mathcal{O}$ é memorizzato come una sequenza di byte all'interno del database: vediamo ora come vengono implementate le operazioni di lettura e di scrittura di $\mathcal{O}$ in una base di dati.

L'operazione di scrittura é banale: dato un certo oggetto $\mathcal{O}$ viene estratta la sequenza di bytes che lo rappresenta, la quale viene memorizzata nel database, assieme ad un certo codice identificativo.

L'operazione di lettura é piú interessante: la prima operazione da compiere é il caricamento di una sequenza binaria dal database attraverso il codice identificativo dell'oggetto richiesto. A questo punto bisogna ricostruire l'oggetto a seconda della sua classe. In questo ambito sono particolarmente importanti alcune proprietá tra le quali ricordiamo:

- il numero di byte che descrivono ogni campo: questa proprietá puó cambiare a seconda della piattaforma;

- la gestione dell'allineamento della sequenza: i dati vengono memorizzati a finale piccolo e quindi bisogna allinearli correttamente a seconda della piattaforma, usando gli strumenti descritti nella sezione 6.4.3;

- la dimensione della sequenza di byte da gestire, la quale deve garantire una rappresentazione minimale dell'oggetto in modo da facilitare le operazioni di I/O nel database e la ricostruzione dell'oggetto in memoria primaria: deve essere un compromesso fra le due esigenze.

Un oggetto contiene la rappresentazione binaria del suo stato interno: questa scelta é dovuta al fatto che le operazioni di I/O sul database sono piú costose di quelle in memoria primaria e quindi si é scelto di facilitarne l'esecuzione, vista anche la natura dinamica del framework *OMSM*. É ragionevole pensare che le operazioni di scrittura di un certo oggetto nel database siano piú frequenti rispetto a quelle che ne modificano lo stato interno quindi é necessario velocizzarle. L'utilizzo di questa tecnica di persistenza richiede all'incirca il doppio della memoria primaria per la gestione di un oggetto $\mathcal{O}$

in quanto bisogna mantenere ed aggiornare la sequenza di byte che descrive $\mathcal{O}$: l'overhead puó essere ridotto con un'attenta scelta degli elementi da rappresentare. Inoltre questa scelta implementativa permette al sottosistema di memorizzazione di essere indipendente dal tipo di componente che si sta gestendo, a prescindere dalla sua complessitá: un oggetto sará sempre descritto da una sequenza di byte, la quale puó essere facilmente inviata sulla rete, compressa o cifrata in modo da adattarsi alle diverse esigenze.

Vediamo come viene implementato questo meccanismo all'interno della libreria *OMSM*: ogni oggetto che deve memorizzato nel database, deve estendere la classe *RawData*, la cui definizione é contenuta nel file *raw.h*. Si tratta di una classe astratta, la quale definisce i servizi che un oggetto deve fornire in modo da poter essere gestito all'interno di un database: la figura 6.19 ne mostra la definizione.

```
class RawData {

public:

 RawData();

 ~RawData();

 virtual byte_array getBytes()
 throw(AllocationErrorException);

 virtual unsigned long getBytesNumber()
 throw(AllocationErrorException);

protected:

 byte_array ba;

 unsigned long nbytes; };
```

Figura 6.19: la definizione della classe *RawData*.

Nello stato interno della classe troviamo:

- il campo *ba*, il quale contiene la rappresentazione binaria dell'oggetto;
- il campo *nbytes*, il quale contiene il numero di byte necessari a rappresentare lo stato dell'oggetto.

Vediamo ora una breve descrizione dei metodi dell'interfaccia esterna:

- il metodo *RawData* é il costruttore di questa classe;

- il metodo ∼*RawData* é il distruttore di questa classe;
- il metodo *getBytes* restituisce la sequenza binaria, la quale descrive lo stato interno di questo oggetto: in caso di errori durante l'allocazione della memoria viene sollevata l'eccezione *AllocationErrorException*;
- il metodo *getBytesNumber* restituisce il numero di byte necessari a rappresentare lo stato interno di questo oggetto: in caso di errori durante l'allocazione della memoria viene sollevata l'eccezione *AllocationError-Exception*.

Come si puó notare, la conversione dello stato interno di un certo oggetto in una sequenza di byte viene delegata alla sottoclasse che modella una specifica entitá da memorizzare all'interno del database. Nelle sezioni seguenti verranno presentate le sottoclassi di quella *RawData* che svolgono un ruolo importante nell'economia dell'architettura *OMSM*.

6.5.2 La rappresentazione degli oggetti geometrici

Nell'architettura *OMSM* é possibile memorizzare oggetti geometrici, immersi in un certo spazio metrico euclideo di dimensione arbitraria, quindi deve essere possibile:

- capire la dimensione d dello spazio metrico in cui sono immersi;
- estrarre la loro geometria, cioé le coordinate euclidee dei loro vertici;
- estrarre un punto d–dimensionale che sia rappresentativo (secondo un qualche criterio) dell'entitá spaziale stessa in modo da poterlo utilizzare come chiave nell'indice spaziale;
- applicare le tecniche di persistenza introdotte nella sezione 6.5.1.

Queste proprietá permettono di modellare un certo oggetto spaziale attraverso la classe *OmsmSpatialObject*, la cui definizione é contenuta nel file *omsmspobject.h*. Anche in questo caso si tratta di una classe astratta, la quale definisce i servizi che un oggetto geometrico deve fornire in modo da poter essere gestito all'interno di un database: la figura 6.20 ne mostra la definizione. Come si puó notare, questa classe estende quella *RawData* in maniera tale da poter essere memorizzata nella struttura *OMSM*.

Vediamo ora una breve descrizione dei metodi dell'interfaccia esterna:

- il metodo *OmsmSpatialObject* é il costruttore di questa classe;
- il metodo ∼*OmsmSpatialObject* é il distruttore di questa classe;
- il metodo *getSpatialObjectClass* restituisce un codice che identifica il tipo di entitá spaziale modellata: é estremamente utile per ricostruire questo oggetto a partire da una sequenza di byte;

```
class OmsmSpatialObject : public RawData {

 public:

 OmsmSpatialObject();

 ~OmsmSpatialObject();

 virtual omsmspobj_class getSpatialObjectClass();

 virtual unsigned long getTopologicalDimension();

 virtual unsigned long getSpaceDimension();

 virtual void getDescriptivePoint(vector<vcoord> &cens);

 virtual void getDescriptivePoint(Point **op)
 throw(AllocationErrorException);

 virtual void getEuclideanVertices(vector<vcoord> &cens);

 virtual void getEuclideanVertices(vector<Point *> &cens)
 throw(AllocationErrorException);

 virtual void info();

 virtual byte_array getBytes()
 throw(AllocationErrorException);

 virtual unsigned long getBytesNumber()
 throw(AllocationErrorException);

 virtual bool sameObject(OmsmSpatialObject *oso)
 throw(NullPointerException,ForbiddenOperationException); };
```

Figura 6.20: la definizione della classe *OmsmSpatialObject*.

- il metodo *getTopologicalDimension* restituisce la dimensione topologica di questo oggetto geometrico: per la definizione di questo concetto rifarsi alla sezione 2.1.1;

- il metodo *getSpaceDimension* restituisce lo dimensione dello spazio euclideo in cui é immerso l'oggetto geometrico;

- i metodi *getDescriptivePoint* restituiscono il punto rappresentativo dell'oggetto geometrico sotto esame: la prima versione ne salva le coordinate nel vettore *cens*, mentre il secondo nell'oggetto *op* di classe *Point* sollevando l'eccezione *AllocationErrorException* in caso di errori durante l'allocazione della memoria;

- i metodi *getEuclideanVertices* restituiscono le coordinate geometriche di questo oggetto geometrico: la prima versione le salva come tuple consecutive di valori di tipo *vcoord*, mentre la seconda versione le salva nel vettore di oggetti di classe *Point* sollevando l'eccezione *AllocationErrorException* in caso di errori durante l'allocazione della memoria;

- il metodo *info* stampa sullo schermo una descrizione dell'oggetto geometrico in questione;

- i metodi *getBytes* e *getBytesNumber* sono ereditati dalla classe *RawData*, definita nella sezione 6.5.1;

- il metodo *sameObject* controlla se i due oggetti geometrici (quello corrente e quello *oso*) descrivono la stessa entitá spaziale, sollevando l'eccezione *NullPointerException* se il parametro non é valido e l'eccezione *ForbiddenOperationException* se cerchiamo di confrontare fra loro due oggetti di tipo diverso o immersi in spazi metrici euclidei con dimensione differente.

Tutti gli oggetti geometrici memorizzati nell'architettura *OMSM* devono estendere la classe *OmsmSpatialObject*: come abbiamo visto nei metodi dell'interfaccia esterna, ogni specifica sottoclasse é identificata da un certo codice in modo tale da poter ricostruire correttamente un certo oggetto a partire da una sequenza di byte. Questo problema si pone quando viene ricostruito un certo nodo in memoria primaria, il quale puó contenere alcuni oggetti geometrici: per semplificare questo processo é stata realizzata la classe *OmsmSpatialObjectFactory*, la cui definizione é contenuta nel file *omsmspobjfactory.h*, mentre la sua implementazione é mantenuta nel file *omsmspobjfactory.cpp*. Questo é l'unico componente in grado di creare correttamente tutte le tipologie di oggetti geometrici disponibili attraverso l'utilizzo del metodo

```
void createSpatialObject(omsmspobj_class otype, byte_array seq,
unsigned long lg, OmsmSpatialObject **newobj)
```

dove:

- *otype* indica il tipo dell'oggetto spaziale da creare;

- *seq* contiene la sequenza di byte per costruire l'oggetto spaziale;

- *lg* contiene la lunghezza della sequenza *seq*;
- *newobj* é il puntatore al nuovo oggetto spaziale.

Inoltre il metodo *createSpatialObject* solleva le sequenti eccezioni:

- *NullPointerException*, se uno dei parametri ha un valore non valido;
- *AllocationErrorException*, se avviene un errore durante l'allocazione del nuovo oggetto spaziale;
- *WrongSPBSequenceException*, se la sequenza *seq* non descrive un oggetto di tipo *otype*;
- *UnkwnownSpatialObjectException*, se il tipo *otype* é scorretto.

Tutte le sottoclassi attualmente disponibili della classe *OmsmSpatialObject* sono riassunte nella figura 6.21, la quale mostra anche la relazione di ereditarietá con quella *RawData*.

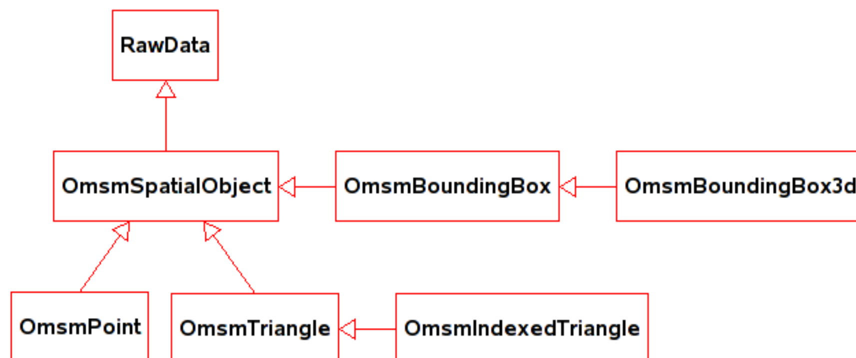


Figura 6.21: il diagramma delle classi eredi di quella *OmsmSpatialObject*.

Vediamo una breve descrizione delle sottoclassi e quindi degli oggetti geometrici attualmente disponibili:

- la classe *OmsmBoundingBox* modella un iper–rettangolo in dimensione arbitraria: la sua definizione é contenuta nel file *omsmbbox.h* mentre la sua implementazione é mantenuta nel file *omsmbbox.cpp*;
- la classe *OmsmBoundingBox3d* modella un parallelepipedo nello spazio euclideo tridimensionale: ad esempio serve a descrivere la *MBB* all'interno del componente *MeshFormatHandler*, come descritto nella sezione 6.4.2;

- la classe *OmsmPoint* modella un punto in dimensione arbitraria: la sua definizione é contenuta nel file *omsmpoint.h*, mentre la sua implementazione é mantenuta nel file *omsmpoint.cpp*;
- la classe *OmsmTriangle* modella un triangolo in dimensione arbitraria, visto come tripla di oggetti di classe *OmsmPoint*: la sua definizione é contenuta nel file *omsmtriangle.h*, mentre la sua implementazione é mantenuta nel file *omsmtriangle.cpp*;
- la classe *OmsmIndexedTriangle* modella un triangolo in dimensione arbitraria, nel quale i tre vertici e lo stesso oggetto geometrico possiedono un codice identificativo: viene utilizzato in questa tesi per memorizzare i triangoli che provengono dalla versione indicizzata della mesh, convertita in formato *TSOUP*. La sua definizione é contenuta nel file *omsmitriangle.h*, mentre la sua implementazione é mantenuta nel file *omsmitriangle.cpp*.

Per approfondimenti sul funzionamento delle classi citate in questa sezione rifarsi a [Can07a] e [Can07b].

6.5.3 Il livello *SPDataIndex*

Questo livello é quello piú esterno dell'architettura *OMSM*, il quale permette all'utente di interagire con i dati memorizzati, fornendo una serie di servizi: inoltre nasconde i dettagli implementativi e le specifiche tecniche utilizzate per la gestione degli oggetti. Considerando il modello del framework *OMSM* in figura 6.18, si puó comprendere come questa architettura sia facilmente modificabile ed espandibile: una volta creata una particolare istanza non é piú interessante ricordarne le caratteristiche e si puó utilizzare questo componente senza conoscerne la struttura. Questa scelta progettuale facilita la costruzione di applicazioni che si interfacciano con i dati memorizzati.

In questo livello é importante stabilire quale indice spaziale bisogna utilizzare, a seconda delle impostazioni: questo aspetto é del tutto trasparente per i livelli inferiori del modello *OMSM*. Per memorizzare l'indice si é scelto di mantenere in memoria primaria solamente il nodo radice: gli altri nodi verranno caricati in maniera dinamica, a seconda della navigazione nella struttura. Nella libreria *OMSM* un nodo di un generico indice spaziale (e quindi anche il nodo radice) viene modellato dalla classe astratta *OmsmSpatialIndexNode*, la cui definizione é contenuta nel file *spindexnode.h*. Tra i metodi forniti da questa classe possiamo ricordare quelli piú interessanti per i nostri scopi:

```
bool addSpatialObject(OmsmSpatialObject *oso, NodeHandler *nh,
unsigned long lev) throw(NullPointerException,
ForbiddenOperationException,AllocationErrorException,
SPDataIndexInsertFailureException);
```

```
bool objectQuery(OmsmSpatialObject *oso)
throw(NullPointerException,AllocationErrorException,
SPDataIndexResearchFailureException);

unsigned long rangeQuery(OmsmBoundingBox *bbox,
vector<OmsmSpatialObject *> &objs) throw(NullPointerException,
AllocationErrorException,SPDataIndexResearchFailureException);
```

Vediamone una breve descrizione:

- il metodo *addSpatialObject* inserisce un certo oggetto geometrico *oso* nel nodo corrente, a livello *lev* nell'albero, utilizzando il componente *nh* per il caricamento dinamico dei nodi: solleva le eccezioni *NullPointerException* se uno dei parametri ha un valore non valido, *ForbiddenOperationException* se le dimensioni euclidee dell'oggetto *oso* e del dominio descritto da questo nodo non coincidono, *AllocationErrorException* se avvengono degli errori durante l'allocazione della memoria e *SPDataIndexInsertFailureException* se avviene un errore durante l'utilizzo del componente *nh*;

- il metodo *objectQuery* controlla se l'oggetto *oso* é memorizzato in questo nodo oppure nei suoi figli: solleva l'eccezione *NullPointerException* se il parametro assume un valore non valido, *AllocationErrorException* se avviene un errore durante l'allocazione della memoria e quella *SPDataIndexResearchFailureException* se avviene un errore durante l'esecuzione dell'interrogazione spaziale;

- il metodo *rangeQuery* restituisce tutti gli oggetti geometrici memorizzati in questo nodo e nei suoi figli che abbiano un intersezione non vuota con l'iper–rettangolo *bbox*: solleva l'eccezione *NullPointerException* se il parametro assume un valore non valido, *AllocationErrorException* se avviene un errore durante l'allocazione della memoria e quella *SPDataIndexResearchFailureException* se avviene un errore durante l'esecuzione dell'interrogazione spaziale. Inoltre restituisce il numero di oggetti con le caratteristiche richieste, memorizzati all'interno del vettore *objs*.

Gli altri metodi di questa classe non presentano particolari motivi di interesse pertanto si rimanda a [Can07a] per approfondimenti.

Anche questa classe é *astratta* e non puó essere istanziata: l'effettiva implementazione dei metodi sará contenuta nelle sottoclassi. Nell'architettura *OMSM* i nodi degli indici spaziali devono estendere la classe *OmsmSpatialIndexNode*: come vedremo nella sezione 6.5.6, ogni indice spaziale é identificato da un certo codice in modo tale da poter ricostruire correttamente un certo nodo a partire da una sequenza di byte o da un oggetto spaziale

esistente. Per semplificare la gestione di questo problema é stata realizzata la classe *SpatialIndexLeavesFactory*, la cui definizione é contenuta nel file *spindexfactory.h*, mentre la sua implementazione é mantenuta nel file *spindexfactory.cpp*. Questo é l'unico componente in grado di creare correttamente tutte le tipologie di foglie di indici spaziali disponibili attraverso l'utilizzo del metodo

```
void createLeaf(node_type ntype,node_id nid,
OmsmSpatialObject *spobj,OmsmBoundingBox *bbox,
unsigned long cap, OmsmSpatialIndexNode **osin)
```

dove:

- *ntype* contiene il tipo di indice spaziale da creare;
- *nid* é il codice identificativo della nuova foglia
- *spobj* é l'oggetto geometrico da memorizzare nella nuova foglia;
- *bbox* é il dominio gestito dalla nuova foglia;
- *cap* é il numero massimo di oggetti geometrici memorizzabili nella nuova foglia: questo parametro verrá ignorato se non si sta creando una struttura *ibrida*, come quelle descritte nel paragrafo 5.6;
- *osin* é il puntatore alla nuova foglia.

Inoltre questo metodo solleva le seguenti eccezioni:

- *NullPointerException*, se uno dei parametri ha un valore non valido;
- *ForbiddenOperationException*, se le dimensioni euclidee di *spobj* e di *bbox* non coincidono;
- *AllocationErrorException*, se avviene un errore durante l'allocazione della nuova foglia;
- *NoBuildableNodeException*, se il tipo della foglia da creare non é valido.

La creazione dei nodi a partire da una sequenza di byte é stata inglobata nella gestione di oggetti di classe *Node*, il cui utilizzo sará piú chiaro una volta studiate le caratteristiche del livello *NodeHandler*, nella sezione 6.5.4. Tutti i tipi di indici spaziali attualmente disponibili nel prototipo iniziale del framework *OMSM* sono riassunti nella figura 6.22: come si puó notare é necessario estendere la classe *RawData* per poter usufruire dei servizi di persistenza, come descritto nella sezione 6.5.1.

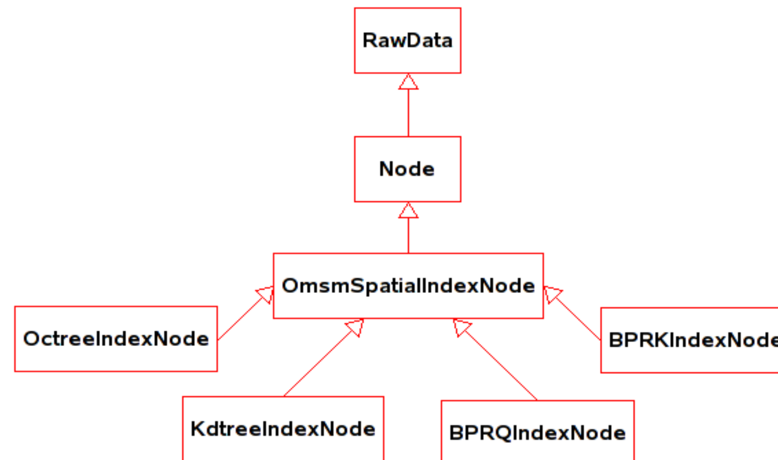


Figura 6.22: il diagramma delle classi eredi di quella *OmsmSpatialIndexNode*.

Vediamo una breve descrizione delle sottoclassi e quindi degli indici spaziali attualmente disponibili:

- la classe *OctreeIndexNode* modella l'indice spaziale *Octree*, discusso nella sezione 5.4: la sua definizione é contenuta nel file *octreenode.h*, mentre la sua implementazione é mantenuta nel file *octreenode.cpp*;
- la classe *KdtreeIndexNode* modella l'indice spaziale *K–d tree*, discusso nella sezione 5.5: la sua definizione é contenuta nel file *kdtreenode.h*, mentre la sua implementazione é mantenuta nel file *kdtreenode.cpp*;
- la classe *BPRQIndexNode* modella l'indice spaziale *Bucket PR–Quadtree*, discusso nella sezione 5.6.2: la sua definizione é contenuta nel file *bprqnode.h*, mentre la sua implementazione é mantenuta nel file *bprqnode.cpp*;
- la classe *BPRKIndexNode* modella l'indice spaziale *Bucket PR k–d tree*, discusso nella sezione 5.6.2: la sua definizione é contenuta nel file *bprknode.h*, mentre la sua implementazione é mantenuta nel file *bprknode.cpp*.

Abbiamo ora tutti gli elementi per studiare come viene rappresentato questo livello. Nel framework *OMSM* questo livello é modellato dalla classe *SPDataIndex*, la cui definizione é contenuta nel file *spdataindex.h*, mentre la sua implementazione é mantenuta nel file *spdataindex.cpp*: a differenza degli altri due livelli, che studieremo rispettivamente nelle sezioni 6.5.4 e 6.5.5, possiamo istanziare direttamente la classe *SPDataIndex*: la figura 6.23 ne mostra la definizione.

```
class SPDataIndex {

public:

 SPDataIndex(string dbname,bool createOn,bool rdonly,
 SPDataOptions *opts) throw(NullPointerException,
 AllocationErrorException,InvalidSPDIException);

 ~SPDataIndex();

 void close(bool rem);

 void addSpatialObject(OmsmSpatialObject *oso)
 throw(NullPointerException,AllocationErrorException,
 SPDataIndexInsertFailureException);

 bool objectQuery(OmsmSpatialObject *oso)
 throw(NullPointerException,AllocationErrorException,
 SPDataIndexResearchFailureException);

 unsigned long rangeQuery(OmsmBoundingBox *bbox,
 vector<OmsmSpatialObject *> &objs) throw(NullPointerException,
 AllocationErrorException,SPDataIndexResearchFailureException);

protected:

 node_type ntype;

 bool rdonly;

 string dbname;

 NodeHandler *nh;

 SuperNode *snode;

 OmsmSpatialIndexNode *rnode;

 OmsmBoundingBox *bbox; };
```

Figura 6.23: la definizione della classe *SPDataIndex*, la quale modella il livello piú esterno del framework *OMSM*.

Vediamo ora una breve descrizione dello stato interno della classe *SPDataIn-*

dex:

- il campo *ntype* contiene il codice identificativo dell'indice spaziale da gestire in questa istanza del framework *OMSM*;
- il campo *rdonly* contiene l'indicazione se il database supporta solamente le operazioni di lettura;
- il campo *dbname* memorizza il nome del database sul quale operare;
- il campo *nh* memorizza il componente che modella il livello *NodeHandler*: ne parleremo in maniera piú approfondita nella sezione 6.5.4;
- il campo *snode* contiene il *super–nodo*, cioé un particolare tipo di nodo che contiene la descrizione completa dell'architettura *OMSM* attualmente in esecuzione: é una istanza della classe *SuperNode*, la quale verrá descritta nella sezione 6.5.6;
- il campo *rnode* contiene il nodo radice dell'indice spaziale ed é una istanza della classe *OmsmSpatialIndexNode*: se l'indice é vuoto questo puntatore assumerá il valore speciale *NULL*;
- il campo il campo *bbox* contiene l'iper–rettangolo che modella il dominio descritto dal nodo.

Vediamo ora una breve descrizione dei metodi dell'interfaccia esterna:

- il metodo *SPDataIndex* crea un componente con il quale accedere al database *dbname* attraverso le impostazioni contenute in *opts* (che approfondiremo nella sezione 6.5.6), creando fisicamente il database a seconda del valore del parametro *createOn* e supportando solamente operazioni di lettura in base al valore di *rdonly*: questo metodo solleva l'eccezione *NullPointerException* se un parametro assume un valore non valido, *AllocationErrorException* se avviene un errore durante l'allocazione della memoria e quella *InvalidSPDIException* se avviene un errore durante l'esecuzione delle operazioni richieste;
- il metodo $\sim$ *SPDataIndex* é il distruttore per questa classe;
- il metodo *close*, dopo la sua esecuzione, non permette piú l'accesso al database, mantenendone consistente lo stato, come descritto nel capitolo 4: se il parametro *rem* vale *true* allora il database viene fisicamente cancellato;
- il metodo *addSpatialObject* inserisce l'oggetto geometrico *oso* nel database: solleva l'eccezione *NullPointerException* se il parametro assume un valore non valido, *AllocationErrorException* se avviene un errore durante l'allocazione della memoria e quella *SPDataIndexInsertFailureException* se avviene un errore durante l'inserimento dell'oggetto geometrico;

- il metodo *objectQuery* controlla se l'oggetto *oso* é memorizzato nel database: solleva l'eccezione *NullPointerException* se il parametro assume un valore non valido, *AllocationErrorException* se avviene un errore durante l'allocazione della memoria e quella *SPDataIndexResearchFailureException* se avviene un errore durante l'esecuzione dell'interrogazione spaziale;

- il metodo *rangeQuery* restituisce tutti gli oggetti geometrici, salvandoli in *objs*, che abbiano un intersezione non vuota con l'iper–rettangolo *bbox*: solleva l'eccezione *NullPointerException* se il parametro assume un valore non valido, *AllocationErrorException* se avviene un errore durante l'allocazione della memoria e quella *SPDataIndexResearchFailureException* se avviene un errore durante l'esecuzione dell'interrogazione spaziale. Inoltre restituisce il numero di oggetti con le caratteristiche richieste, memorizzati all'interno di *objs*.

Come si puó intuire, l'esecuzione dei metodi offerti dalla classe *SPDataIndex* dipende dallo stato interno dell'indice spaziale e quindi dal nodo radice *rnode*: infatti, se l'indice spaziale non é vuoto, l'esecuzione dei vari metodi viene propagata sul nodo radice, utilizzando i corrispettivi metodi della classe *OmsmSpatialIndexNode*. Grazie a queste caratteristiche é possibile fissare in maniera automatica la dimensione dello spazio euclideo nel quale sono immersi gli oggetti geometrici da memorizzare quando si inserisce la prima entitá spaziale nel database, mentre la loro dimensione topologica puó variare. Secondo la definizione di dimensione fornita nella sezione 2.1.1, la dimensione topologica di un oggetto é il numero di parametri indipendenti necessari alla sua descrizione. Inoltre, grazie al metodo costruttore, é possibile creare un nuovo database oppure aprire un database esistente a seconda del parametro *createOn*. Nel secondo caso si cerca di caricare l'istanza del framework *OMSM* attraverso le impostazioni di input: una volta caricata, verranno ricostruiti i livelli secondo la *vera* struttura memorizzata.

Per approfondimenti sul funzionamento delle classi citate in questa sezione rifarsi a [Can07a] e [Can07b].

6.5.4 Il livello *NodeHandler*

Il livello *NodeHandler* é forse quello piú importante nell'economia dell'architettura *OMSM* in quanto si occupa della gestione dei nodi dell'indice spaziale e di quello di configurazione (detto anche *super–nodo*), a seconda delle richieste del livello *SPDataIndex*. Inoltre raggruppa questi nodi secondo una certa politica di clustering per minimizzare il numero di operazioni di I/O da eseguire sul database: la memorizzazione fisica di ogni cluster viene demandata al livello *NClusterStorager*, di cui parleremo nella sezione 6.5.5, in maniera tale che la distribuzione dei dati e la tecnica utilizzata per memorizzarli siano trasparenti per il componente *NodeHandler*.

A questo punto é interessante osservare che i nodi gestiti da questo livello possono appartenere a due diverse categorie:

- i nodi degli indici spaziali, i quali vengono modellati dalla classe *OmsmSpatialIndexNode* e dalle sue sottoclassi, introdotte nella sezione 6.5.3;
- un nodo speciale, detto *super–nodo*, il quale contiene la descrizione completa dell'architettura *OMSM*, attualmente in esecuzione: ne discuteremo le caratteristiche principali nella sezione 6.5.6.

Nel framework *OMSM* la gestione di queste due entitá viene unificata dall'introduzione della classe astratta *Node*, la cui definizione é memorizzata nel file *node.h*: le classi *OmsmSpatialIndexNode* e *SuperNode* (la quale modella il *super–nodo*) estendono quella *Node*, come dimostra la figura 6.24.

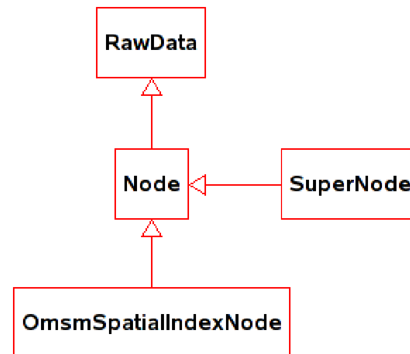


Figura 6.24: il diagramma delle classi eredi di quella *Node*. Le sottoclassi di quella *OmsmSpatialIndexNode* sono mostrate in figura 6.22.

Questa classe descrive un oggetto che puó essere memorizzato all'interno di un database e quindi deve essere una sottoclasse di quella *RawData*, in modo da poter utilizzare i meccanismi di persistenza all'interno del sistema *OMSM*, introdotti nella sezione 6.5.1. Le altre funzionalitá della classe *Node* non sono di particolare interesse: per questa ragione rimandiamo a [Can07a] e [Can07b] per eventuali approfondimenti.

Come dimostra la figura 6.24 ci sono varie sottoclassi di quella *Node*, ognuna delle quali é identificata da un certo codice: per semplificarne la creazione a partire da una particolare sequenza di byte é stata realizzata la classe *NodesFactory*, la cui definizione é contenuta nel file *nfactory.h*, mentre la sua implementazione é mantenuta nel file *nfactory.cpp*. Questo é l'unico componente in grado di creare correttamente tutte le sottoclassi di quella *Node* attraverso il metodo:

```
void createNode(node_type ntype,byte_array ba,unsigned long lg,
Node **newnode)
```

dove:

- il parametro *ntype* contiene il tipo di nodo da costruire;
- *seq* contiene la sequenza di byte per costruire il nuovo nodo;
- *lg* contiene la lunghezza della sequenza *seq*;
- il parametro *newnode* contiene il puntatore al nuovo nodo da costruire.

Inoltre questo metodo solleva le sequenti eccezioni:

- *NullPointerException* se uno dei parametri ha un valore non valido;
- *AllocationErrorException* se avviene un errore durante l'allocazione del nuovo nodo;
- *WrongSPBSequenceException* se la sequenza *seq* non descrive un nodo di tipo *ntype*;
- *NoBuildableNodeException* se il tipo di nodo *ntype* é scorretto.

Attraverso questa tecnica, il livello *NodeHandler* puó gestire solamente oggetti di classe *Node*, senza preoccuparsi del loro effettivo contenuto. L'unico vincolo da soddisfare coinvolge il *super–nodo*: il cluster adibito alla sua memorizzazione non puó contenere altri oggetti *Node* e viene gestito in maniera differente dagli altri. Questa scelta implementativa permette di gestire in maniera efficiente la descrizione di un'architettura *OMSM*.

Ogni oggetto di classe *Node* puó essere inserito in un cluster, il quale viene modellato dalla classe *NodeCluster*: la sua definizione é contenuta nel file *ncluster.h*, mentre la sua implementazione é mantenuta nel file *ncluster.cpp*. Anche questa classe descrive un oggetto che puó essere memorizzato all'interno di un database e quindi deve estendere quella *RawData*, in modo da poter utilizzare i meccanismi di persistenza all'interno del sistema *OMSM*, introdotti nella sezione 6.5.1. Le altre funzionalitá della classe *NodeCluster* non sono di particolare interesse: per questa ragione rimandiamo a [Can07a] e [Can07b] per eventuali approfondimenti.

La suddivisione in cluster puó avvenire secondo una qualsiasi politica di decomposizione, la quale puó garantire o meno l'efficenza delle funzionalitá offerte da questo livello, a seconda delle impostazioni volute. In [DdFPS05] vengono analizzate varie politiche di clustering, chiarendo come si possano ottenere risultati migliori raggruppando nello stesso cluster i nodi che contengono oggetti spazialmente vicini, ad esempio utilizzando indici spaziali come quelli *R–tree* e *PK–tree*: per approfondimenti su queste strutture dati rifarsi a [Gut84], [YWM97] e [WYM98]. Un'altra suddivisione interessante é quella descritta in [CBD+07], la quale é particolarmente indicata per una piattaforma multi–processore: questa tecnica permette di creare una suddivisione

gerarchica di una certa quantitá di dati in cluster in modo tale da assegnarne una porzione ad ogni processore, bilanciando in questo modo il carico computazionale. In [Pro97a] vengono descritte ulteriori politiche di clustering, sviluppate nell'ambito dell'intelligenza artificiale e dell'apprendimento automatico. In generale, ognuna di queste tecniche puó essere integrata all'interno del livello *NodeHandler*, il quale non assume l'implementazione di alcuna politica di clustering in particolare.

Vediamo ora come viene gestito questo livello all'interno dell'architettura *OMSM*: esso é modellato dalla classe astratta *NodeHandler*, la cui definizione é contenuta nel file *nodehandler.h*. Tra i metodi forniti da questa classe possiamo ricordare quelli piú interessanti:

```
void getNode(node_id nid,Node **newnode)
throw(AllocationErrorException, NoNodeErrorException,
NodeLoadingErrorException)

void addNode(Node *newnode) throw(NullPointerException,
AllocationErrorException, NodeAddingErrorException)
```

Vediamone una breve descrizione:

- il metodo *getNode* carica il nodo avente *nid* come codice identificativo, salvandolo in *newnode*: solleva le eccezioni *AllocationErrorException* se avviene un errore durante l'allocazione della memoria, *NoNodeErrorException* se non é memorizzato un nodo con il codice richiesto e *NodeAddingErrorException* se avviene un errore durante il caricamento del nodo;

- il metodo *addNode* memorizza il nodo *newnode* avente *nid* come codice identificativo in un cluster, scelto in base alla politica di suddivisione dei nodi: solleva le eccezioni *AllocationErrorException* se avviene un errore durante l'allocazione della memoria e *NodeAddingErrorException* se avviene un errore durante l'aggiunta del nuovo nodo.

Come si puó notare, l'operazione piú importante é quella di individuare il cluster che contiene un certo nodo γ, solitamente a partire dal codice identificativo di γ: per i nostri scopi sono particolarmente indicate tecniche di tipo *lineare*, nelle quali l'operazione di suddivisione si basa sulla generazione di codici locazionali, i quali identificano la posizione del cluster in maniera efficiente. Per approfondimenti su queste tecniche rifarsi a [Mor66], [Gar82], [AS83], [FR89], [Jag90], [Sag94], [Jaf03] e [TO04].

Inoltre, per minimizzare il numero degli accessi al supporto di memorizzazione viene mantenuta una cache di dimensione fissata, la quale utilizza la politica di sostituzione *LRU*, in modo da mantenere in memoria primaria i cluster usati piú di recente: per modellare questo componente usiamo un'istanza della classe template *Cache*, definita nella sezione 6.3.2. Il

template *Key* sará istanziato dal tipo *cluster_identifier*, il quale modella il codice identificativo di un cluster: invece il template *Data* sará istanziato dal tipo *NodeCluster**, il quale contiene il puntatore ad un oggetto di classe *NodeCluster*.

Per semplicitá, nella versione preliminare del framework *OMSM* supportiamo solamente la politica di clustering *singola*, secondo la quale ogni cluster puó contenere un solo nodo: in questo modo il codice del cluster sará identico a quello del nodo, facilitandone la ricerca. Purtroppo questa tecnica puó essere molto inefficiente in quanto potrebbe richiedere un'operazione di I/O per ogni nodo sul quale si vuole operare. Questo specifico componente é modellato dalla classe *SingleNodeHandler*, la cui definizione é contenuta nel file *singlenodehandler.h* mentre la sua implementazione é mantenuta nel file *singlenodehandler.cpp*.

Come vedremo nella sezione 6.5.6, ogni sottoclasse di quella *NodeHandler* é identificata da un certo codice in modo da facilitare il processo di personalizzazione dell'architettura *OMSM*: per capire quale versione del componente bisogna creare, é necessario conoscere tutte le componenti attualmente supportate. Per questi motivi é stata realizzata la classe *NodeHandlerFactory*, la cui definizione é contenuta nel file *nhfactory.h*, mentre la sua implementazione é mantenuta nel file *nhfactory.cpp*. Questo é l'unico componente in grado di creare correttamente tutte le sottoclassi di quella *NodeHandler* attraverso il metodo:

```
void createNodeHandler(string sname,bool con,
SPDataOptions *opts, NodeHandler **nh)
```

dove:

- *sname* é il nome del database da gestire;
- *con* é un valore binario, il quale indica se dobbiamo creare un nuovo database (in tal caso ha valore di veritá *true*) o se dobbiamo caricare un database esistente (in tal caso ha valore di veritá *false*);
- *opts* é il componente che modella le impostazioni con cui gestire l'architettura *OMSM*, il quale contiene anche la politica di clustering da applicare: nella sezione 6.5.6 verrá approfondito il meccanismo di configurazione;
- *nh* é il puntatore al nuovo oggetto *NodeHandler*.

Inoltre questo metodo solleva le seguenti eccezioni:

- *NullPointerException*, se uno dei parametri ha un valore non valido;
- *AllocationErrorException*, se avviene un errore durante l'allocazione del nuovo componente modellato dalla sottoclasse di quella *NodeHandler*;

- *InvalidNHandlerException*, se non é possibile creare il nuovo componente modellato dalla sottoclasse di quella *NodeHandler*.

Per approfondimenti sul funzionamento delle classi citate in questa sezione rifarsi a [Can07a] e [Can07b].

6.5.5 Il livello *NClusterStorager*

Il livello *NClusterStorager* si occupa della gestione a basso livello dei cluster di nodi, definiti nella sezione 6.5.4, operando in maniera efficiente su un supporto di memorizzazione. Le tecniche utilizzate in questo livello dipendono dalla distribuzione dei dati: ad esempio é possibile supportare l'accesso a database remoti oppure memorizzati localmente. Per semplificare le operazioni di I/O, il modello dei dati gestiti da questo livello é definito dalla coppia formata dal codice identificativo del cluster e dalla sequenza di byte che lo rappresenta: in questo modo é possibile tralasciare i dettagli implementativi della struttura dati da gestire e permettere l'utilizzo di ulteriori trasformazioni sulla sequenza di byte, ad esempio é possibile cifrarla o estrarre una sua versione compressa, a seconda delle varie esigenze.

Nel framework *OMSM* questo livello é implementato dalla classe *NClusterStorager*, la cui definizione é contenuta nel file *ncstorager.h*. Si tratta di una classe astratta, la quale definisce i servizi che un componente di questo tipo deve fornire per poter implementare le operazioni di I/O dei cluster. Tra i metodi forniti da questa classe possiamo ricordare:

```
void addCluster(cluster_id cid,byte_array ba,unsigned long lg)
throw((NullPointerException,AllocationErrorException,
WrongClusterStoringException);

void getCluster(cluster_id cid,byte_array *ba,unsigned long
&lg) throw(AllocationErrorException,NoClusterErrorException,
WrongClusterReadingException);

bool searchCluster(cluster_id cid)
throw(WrongClusterReadingException);
```

Vediamo ora una breve descrizione di questi metodi:

- il metodo *addCluster* memorizza un cluster avente *cid* come codice identificativo e descritto dalla sequenza *ba* composta da *lg* byte: solleva le eccezioni *NullPointerException* se uno dei parametri non é valido, *AllocationErrorException* se avviene un errore durante l'allocazione della memoria e *WrongClusterStoringException* se avviene un errore durante la scrittura nel database;

- il metodo *getCluster* carica dal database un cluster identificato dal codice *cid* e ne memorizza in *ba* la sequenza che lo descrive, composta da *lg* byte: solleva le eccezioni *AllocationErrorException* se avviene un errore durante l'allocazione della memoria, *NoClusterErrorException* se il cluster specificato non é attualmente memorizzato e *WrongClusterReadingException* se avviene un errore durante la lettura del cluster dal database.

- il metodo *searchCluster* controlla se il cluster identificato dal codice *cid* é memorizzato nel database: solleva l'eccezione *WrongClusterReadingException* se avviene un errore durante la ricerca dei cluster.

Visto che la classe *NClusterStorager* é astratta e quindi non puó essere istanziata direttamente, l'implementazione vera e propria di questo livello sará fornita da una specifica sottoclasse: nella versione preliminare della libreria *OMSM* supportiamo solamente la gestione di un database memorizzato nella stessa macchina che stiamo utilizzando. Questo specifico componente é modellato dalla classe *LNCStorager*, la cui definizione é contenuta nel file *lncstorager.h*, mentre la sua implementazione é mantenuta nel file *lncstorager.cpp*: per memorizzare in maniera efficiente i dati é stato utilizzato il sistema software *Oracle Berkeley DB*, che abbiamo descritto nella sezione 4.4.2. Si tratta di un esempio di *DBMS* di tipo embedded, il quale supporta le proprietá descritte nel capitolo 4: questa é una possibile soluzione al problema della memorizzazione dei dati: in realtá é possibile sostituirlo con un altro componente che sia in grado di eseguire le stesse operazioni senza compromettere le funzionalitá dell'intera architettura *OMSM*, come quelli descritti in [BP95], [GHSS95] e [NGHS97].

Come vedremo nella sezione 6.5.6 ogni sottoclasse di quella *NClusterStorager* é identificata da un certo codice in modo da facilitare il processo di personalizzazione dell'architettura *OMSM*: peró, per capire quale versione del componente bisogna creare, é necessario conoscere tutte le componenti attualmente supportate. Per questi motivi é stata realizzata la classe *ClusterStoragerFactory*, la cui definizione é contenuta nel file *csfactory.h*, mentre la sua implementazione é mantenuta nel file *csfactory.cpp*. Questo é l'unico componente in grado di creare correttamente tutte le sottoclassi di quella *NClusterStorager* attraverso il metodo:

```
void createClusterStorager(string sname,bool con,
SPDataOptions *opts, NClusterStorager **stor)
```

dove:

- *sname* é il nome del database da gestire;

- *con* é un valore binario, il quale indica se dobbiamo creare un nuovo database (in tal caso ha valore di veritá *true*) o se dobbiamo caricare un database esistente (in tal caso ha valore di veritá *false*);

- *opts* é il componente che modella le impostazioni con cui gestire l'architettura *OMSM*, il quale contiene anche il codice della sottoclasse da creare ed un parametro ausiliario, che svolge un ruolo importante nella creazione del componente *NClusterStorager*: nella sezione 6.5.6 verrá approfondito il meccanismo di configurazione;
- *stor* é il puntatore al nuovo oggetto *NClusterStorager*.

Inoltre questo metodo solleva le seguenti eccezioni:

- *NullPointerException*, se uno dei parametri ha un valore non valido;
- *AllocationErrorException*, se avviene un errore durante l'allocazione del nuovo componente modellato dalla sottoclasse di quella *NClusterStorager*;
- *InvalidNCSException*, se non é possibile creare il nuovo componente modellato dalla sottoclasse di quella *NClusterStorager*.

Per approfondimenti sul funzionamento delle classi citate in questa sezione rifarsi a [Can07a] e [Can07b].

6.5.6 La configurazione del sistema

Come abbiamo spiegato nelle sezioni precedenti, il framework *OMSM* ha una struttura modulare ed i vari livelli, riassunti nella figura 6.18, possono gestire una particolare tecnica, in maniera indipendente dagli altri. In questa sezione vedremo i meccanismi forniti dalla libreria *OMSM* per variare le impostazioni dell'architettura corrente.

Dall'analisi dei tre livelli *SPDataIndex*, *NodeHandler* e *NClusterStorager* é chiaro che bisogna poter variare almeno:

- il tipo di indice spaziale da utilizzare: questo aspetto interessa il livello *SPDataIndex* ed in particolare la sottoclasse di quella *OmsmSpatialIndexNode*, introdotta nella sezione 6.5.3;
- la politica di clustering dei nodi: questo aspetto interessa il livello *NodeHandler* ed in particolare la scelta della sottoclasse da usare, come descritto nella sezione 6.5.4;
- la distribuzione fisica dei dati: questo aspetto interessa il livello *NClusterStorager* ed in particolare la scelta della sottoclasse da usare, come descritto nella sezione 6.5.5.

Inoltre vi sono altri due parametri importanti nell'economia del framework *OMSM*: il numero di livelli ed un'informazione ausiliaria per il livello *NClusterStorager*, detta *parametro ausiliario*. Il numero di livelli é importante per definire la capacitá di un certo nodo, secondo il processo descritto nella

sezione 5.6.1: ovviamente il suo valore non é importante per quei indici spaziali, come le strutture k–d tree, i cui nodi hanno una capacitá fissata. Invece il significato del parametro ausiliario varia a seconda della distribuzione dei dati: ad esempio con un database locale, usato nella versione preliminare della libreria, é interessante memorizzare il percorso della directory di ambiente del database, concetto introdotto nella sezione 4.4.2. Per unificare la gestione di queste impostazioni, abbiamo sviluppato la classe *SPDataOptions*, la cui definizione é contenuta nel file *spdataoptions.h*, mentre la sua implementazione é mantenuta nel file *spdataoptions.cpp*: come si puó intuire, vogliamo memorizzare le impostazioni all'interno del database e quindi questa classe estende quella *RawData*, per poter usufruire dei servizi di persistenza, introdotti nella sezione 6.5.1. Il comportamento di questa classe non offre spunti di particolare interesse, perció si rimanda a [Can07a] e [Can07b] per maggiori approfondimenti.

Nella libreria *OMSM* ogni possibile scelta é stata codificata attraverso l'introduzione di una serie di codici identificativi, i quali appartengono ai seguenti dominii:

- quello del tipo *spindex_type*, usato per memorizzare la tipologia di indice spaziale all'interno del database;
- quello del tipo *nodes_clustering_pp*, usato per memorizzare la politica di clustering dei nodi dell'indice spaziale;
- quello del tipo *storager_location*, usato per memorizzare la distribuzione fisica dei dati all'interno del database;
- quello del tipo *levels_number*, usato per memorizzare il numero dei livelli dell'indice spaziale.

In realtá questi quattro tipi sono delle *abbreviazioni* di quello predefinito *unsigned long*, fornito dal linguaggio *C++*.

Dunque dobbiamo associare un codice identificativo ad ogni tecnica realizzata nel framework *OMSM* in modo da poterla specificare nelle impostazioni e quindi all'interno del componente *SPDataOptions*: nel seguito faremo riferimento alla versione preliminare della libreria *OMSM*. In questo prototipo, per semplificarne la gestione, sono state fissate alcune impostazioni predefinite:

- l'indice spaziale da utilizzare é quello *K–d tree*: questa scelta implementativa é identificata dal codice *KDTREE_INDEX*, memorizzato in un valore di tipo *spindex_type*;
- la politica di clustering dei nodi da utilizzare é quella *singola*, cioé quella secondo la quale un cluster puó contenere un solo nodo: questa scelta implementativa é identificata dal codice *SINGLE_CLUSTERING*, memorizzato in un valore di tipo *nodes_clustering_pp*;

- la distribuzione fisica dei dati da utilizzare é quella *locale*, nella quale i cluster vengono memorizzati nella stessa macchina sulla quale si sta operando: questa scelta implementativa é identificata dal codice *LOCAL_STORAGER*, memorizzato in un valore di tipo *storager_location*;

- il parametro ausiliario contiene il percorso della directory corrente;

- il numero dei livelli é indifferente, visto che l'indice spaziale *K–d tree* non supporta questo parametro.

Per l'elenco completo dei codici identificativi rifarsi a [Can07a] e [Can07b].

Per facilitarne le modifiche, si possono memorizzare le impostazioni su un file, in modo da poterle riutilizzare. Il formato del file é a caratteri *ASCII* e si basa sulla memorizzazione di coppie del tipo chiave/dato, codificate secondo il seguente formato

```
< parola chiave > = < dato >
```

in modo da poter essere gestito attraverso il componente *DataReader*, introdotto nella sezioni 6.4.1. La parte *chiave* identifica l'aspetto del framework da modificare, mentre quella *dato* specifica quale tecnica utilizzare. Quindi é necessario fissare una corrispondenza fra i token del file e le varie tecniche, rendendo flessibile questo processo in maniera tale da supportare l'aggiunta di nuove funzionalitá. Vediamo un esempio di file di configurazione:

```
spindex_type = kdtree
spindex_clustering = single
spindex_location = local
spindex_auxpar = /home/david/Tesi/gargoyle/
spindex_lnumbers = 18
```

Vediamo il significato delle impostazioni contenute in questo file:

- il token *"spindex_type"* specifica che si vuole modificare il tipo di indice spaziale: in questo caso si vuole utilizzare un indice *K–d tree*, identificato dal token *"kdtree"*;

- il token *"spindex_clustering"* specifica che si vuole modificare la politica di clustering dei nodi: in questo caso si vuole utilizzare quella *singola*, identificata dal token *"single"*;

- il token *"spindex_location"* specifica che si vuole modificare la distribuzione fisica dei dati: in questo caso si vuole utilizzare un database locale;

- il token *"spindex_ auxpar"* specifica che si vuole modificare il parametro ausiliario: visto che il database é locale, questo parametro dovrá contenere il percorso della directory di ambiente del database, cioé quella *"/home/david/Tesi/gargoyle/"*;
- il token *"spindex_ lnumbers"* specifica che si vuole modificare il numero di livelli dell'indice: in questo caso l'indice spaziale scelto é quello *K–d tree* quindi questo valore é indifferente.

Tuttavia questo approccio non é intuitivo per l'utente, in quanto deve ricordare i token che identificano le varie scelte implementative: per evitare questo inconveniente si puó utilizzare un programma con interfaccia grafica come quello *Omsmconf*, del quale parleremo nella sezione 7.3.1. L'introduzione di questo programma richiede un'ulteriore corrispondenza fra gli elementi sintattici dell'interfaccia grafica e quelli del file di configurazione: queste informazioni sono mantenute all'interno della classe *OptionsManager*, la cui definizione é contenuta nel file *optionsmanager.h*, mentre la sua implementazione é mantenuta nel file *optionsmanager.cpp*. Inoltre questo componente restituirá i codici identificativi a partire dagli elementi sintattici del file di configurazione o dell'interfaccia grafica. Pertanto per ogni funzionalitá della libreria sono necessarie le seguenti informazioni, da memorizzare all'interno della classe *OptionsManager*:

- i token che individuano una determinata sottoclasse ed il loro codice rappresentativo;
- una stringa che descriva la tecnica utilizzata in maniera significativa per l'utente;
- una stringa che possa essere utilizzata in un'interfaccia grafica.

Grazie a questa organizzazione, l'insieme delle tecniche supportate da una specifica versione della libreria *OMSM* é completamente descritta dal componente *OptionsManager*, favorendone la gestione: ad esempio il programma *Omsmconf*, del quale parleremo nella sezione 7.3.1 é indipendente dalla versione della libreria *OMSM* utilizzata e si adatta in maniera dinamica alle tecniche attualmente disponibili. Riassumiamo ora le funzionalitá implementate nel prototipo del framework *OMSM*, associandole alle informazioni memorizzate nella classe *OptionsManager*:

- l'indice spaziale *K–d tree*
  - é identificato dal codice KDTREE_INDEX;
  - é identificato dal token *"kdtree"*;
  - é associato alla stringa *"k–d tree index"*, da utilizzare in un'interfaccia grafica per la configurazione del framework *OMSM*;

  - é associato alla stringa *"a k–d tree index"*, la quale ne fornisce una descrizione significativa per l'utente;

- l'indice spaziale *Octree*
  - é identificato dal codice OCTREE_INDEX;
  - é identificato dal token *"octree"*;
  - é associato alla stringa *"octree index"*, da utilizzare in un'interfaccia grafica per la configurazione del framework *OMSM*;
  - é associato alla stringa *"an octree index"*, la quale ne fornisce una descrizione significativa per l'utente;

- l'indice spaziale *Hybrid Quadtrie*
  - é identificato dal codice HYBRID_QTRIE_INDEX;
  - é identificato dal token *"hybrid_ quadtrie"*;
  - é associato alla stringa *"hybrid quadtrie index"*, da utilizzare in un'interfaccia grafica per la configurazione del framework *OMSM*;
  - é associato alla stringa *"an hybrid quadtrie index"*, la quale ne fornisce una descrizione significativa per l'utente;

- l'indice spaziale *Hybrid K–d trie*
  - é identificato dal codice HYBRID_KDTRIE_INDEX;
  - é identificato dal token *"hybrid k–d trie"*;
  - é associato alla stringa *"hybrid k–d trie index"*, da utilizzare in un'interfaccia grafica per la configurazione del framework *OMSM*;
  - é associato alla stringa *"an hybrid k-d trie index"*, la quale ne fornisce una descrizione significativa per l'utente;

- la politica di clustering dei nodi *singola*
  - é identificata dal codice SINGLE_CLUSTERING;
  - é identificata dal token *"single"*;
  - é associata alla stringa *"a single node in a cluster"*, da utilizzare in un'interfaccia grafica per la configurazione del framework *OMSM*;
  - é associato alla stringa *"a single node in a cluster"*, la quale ne fornisce una descrizione significativa per l'utente;

- la distribuzione *locale* dei dati
  - é identificata dal codice LOCAL_STORAGER;
  - é identificato dal token *"local"*;

– é associato alla stringa *"local"*, da utilizzare in un'interfaccia grafica per la configurazione del framework *OMSM*;
– é associato alla stringa *"a local storager"*, la quale ne fornisce una descrizione significativa per l'utente;

Pertanto, per aggiungere una qualsiasi funzionalitá nei tre livelli del framework *OMSM*, é necessario:

- fornirne l'implementazione completa come sottoclasse di quella relativa al livello voluto, utilizzando la classe astratta giá esistente: ad esempio se vogliamo inserire un nuovo indice spaziale, dobbiamo creare un'opportuna sottoclasse di quella *OmsmSpatialIndexNode*, introdotta nella sezione 6.5.3;
- fissarne i relativi codici identificativi ed i token nella classe *OptionsManager*, secondo quanto descritto in precedenza;
- inserire l'operazione di creazione all'interno del componente per la generazione unificata delle sottoclassi relative al livello voluto a partire dal suo codice identificativo: ad esempio se stiamo inserendo un nuovo indice spaziale, dobbiamo inserire la creazione del nuovo tipo di nodo nel metodo *createLeaves* all'interno della classe *SpatialIndexLeavesFactory*, introdotta nella sezione 6.5.3.

La classe *SPDataOptions* memorizza le informazioni strutturali dell'architettura, in realtá durante l'utilizzo del framework possono variare in maniera dinamica altri parametri quali:

- il numero di nodi e di cluster attualmente memorizzati;
- i codici dei nodi attualmente non assegnati;
- il codice del nodo radice.

Queste informazioni devono essere mantenute all'interno del database in modo da poterle riutilizzare quindi é necessario integrarle con quelle strutturali. Per risolvere questo problema é stata sviluppata la classe *SuperNode*, la cui definizione é contenuta nel file *supernode.h*, mentre la sua implementazione é mantenuta nel file *supernode.cpp*. Questa classe estende quella *Node*, in modo tale da poter essere gestita dal livello *NodeHandler*, come descritto nella sezione 6.5.4. Il suo compito é quello di memorizzare le informazioni strutturali dell'architettura, descritte dalla classe *SPDataOptions*, integrandole con quelle *dinamiche*: il funzionamento di questa classe non offre spunti di particolari interesse, ad eccezione della gestione dei codici dei nodi attualmente non assegnati.

I codici dei nodi attualmente memorizzati sono gestiti attraverso la classe *NodesNamesHandler*: la definizione di questa classe é contenuta nel file

nngenerator.h, mentre la sua implementazione é mantenuta nel file *nngenerator.cpp*. Il suo funzionamento si basa sul fatto che i codici dei nodi appartengono ad un intervallo noto a priori, quindi possiamo mantenere la lista degli intervalli di quelli attualmente non assegnati. Ogni intervallo dei nomi dei nodi é modellato dalla classe *NodeNameInterval*, la cui definizione é contenuta nel file *nngenerator.h* mentre la sua implementazione é memorizzata nel file *nngenerator.cpp*. Su questa lista di intervalli possiamo eseguire le seguenti operazioni:

- richiedere un codice attualmente non assegnato: se la lista non é vuota, viene restituito l'estremo inferiore del primo intervallo disponibile, altrimenti viene sollevato un errore;
- rilasciare un codice attualmente assegnato: questo codice viene inserito nella lista ordinata in senso crescente ed eventualmente si possono fondere intervalli che diventano consecutivi dopo l'inserimento.

Sapere quali sono i codici dei nodi disponibili é un'informazione importante per i nostri scopi quindi questi componenti devono essere resi persistenti, estendendo la classe *RawData*. In realtá é stata realizzata la classe astratta *OmsmGenericComponent*, la cui definizione é contenuta nel file *omsmgcomp.h*, in modo da *raggruppare* tutti i componenti che possono essere resi persistenti, ma che non sono legati fra loro da una qualche relazione. La figura 6.25 riassume le relazioni di ereditarietá fra queste classi.

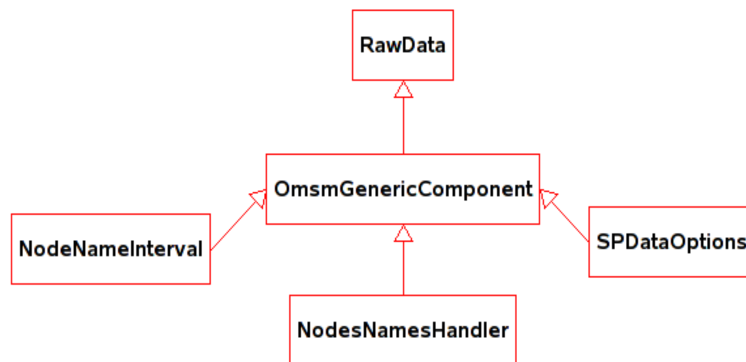


Figura 6.25: il diagramma delle classi eredi di quella *OmsmGenericComponent*. Queste classi modellano oggetti che devono essere resi persistenti, ma che non sono legati fra loro da una qualche relazione.

Come si puó intuire dall'analisi di questo paragrafo, le classi che estendono quella *RawData* sono parecchie: sebbene si possa ricostruire il diagramma di ereditarietá completo a partire dall'analisi di quelli che abbiamo presentato precedentemente, puó essere interessante analizzare la figura 6.26,

la quale riassume il diagramma completo della classi di base: le sottoclassi di *OmsmSpatialObject* e *OmsmSpatialIndexNode* non vengono mostrate perché possono variare a seconda delle differenti versioni della libreria *OMSM*. Per maggiori approfondimenti sull'utilizzo di queste sottoclassi rifarsi rispettivamente alle sezioni 6.5.2 e 6.5.3.

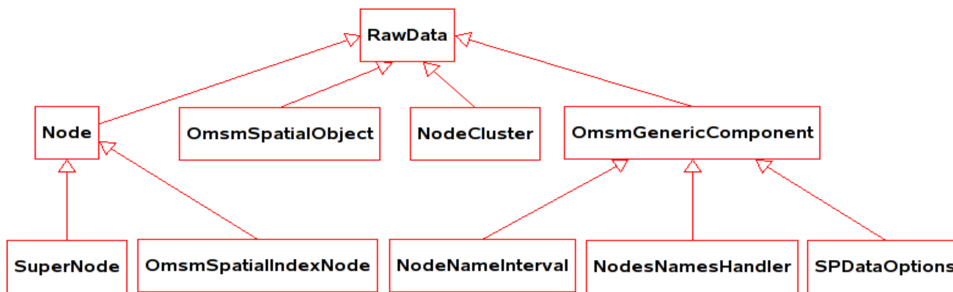


Figura 6.26: il diagramma delle classi eredi di quella *RawData*. Le sottoclassi di *OmsmSpatialIndexNode* e *OmsmSpatialObject* non vengono mostrate perché possono variare a seconda delle differenti versioni della libreria *OMSM* che stiamo utilizzando.

Riassumiamo brevemente il ruolo delle varie classi:

- la classe *Node* modella la descrizione degli oggetti gestibili dal livello *NodeHandler*, come descritto nella sezione 6.5.4;
- la classe *NodeCluster* modella i cluster di nodi, come descritto nella sezione 6.5.4;
- la classe *OmsmSpatialObject* modella il generico oggetto spaziale memorizzabile nell'architettura sviluppata;
- la classe *OmsmGenericComponent* modella un generico componente che deve essere persistente nell'architettura *OMSM*;
- la classe *SuperNode* contiene le impostazioni strutturali e dinamiche dell'architettura *OMSM*;
- la classe *OmsmSpatialIndexNode* modella il nodo di un generico indice spaziale, come descritto nella sezione 6.5.3;
- la classe *NodeNameInterval* modella un intervallo di codici identificativi dei nodi, il quale puó essere memorizzati in una lista;
- la classe *NodesNamesHandler* modella il gestore dei codici identificativi dei nodi che non sono ancora stati assegnati;

- la classe *SPDataOptions* modella un componente per la gestione delle impostazioni strutturali dell'architettura *OMSM*.

Per approfondimenti sul funzionamento di queste classi rifarsi alle relative sezioni oppure a [Can07a] e [Can07b].

Tutti i diagrammi delle classi eredi riportati in questo capitolo sono stati realizzati attraverso il programma *Umbrello*, un modellatore per il linguaggio *UML*, il quale facilita la progettazione logica di un sistema software. Non abbiamo utilizzato le tecniche di modellazione *UML* all'interno dei diagrammi per non appesantire ulteriormente la notazione: per maggiori approfondimenti su questo software e sul linguaggio *UML* rifarsi a [Rid01].

Capitolo 7

I risultati ottenuti

Asserzioni notevoli richiedono prove notevoli.
Carl Sagan

In questo capitolo la nostra analisi verterá sulla presentazione dei risultati sperimentali ottenuti, ponendo maggior enfasi sui programmi sviluppati attraverso la libreria *OMSM* e gli altri sistemi software di cui si é fatto uso. Per verificare il funzionamento dei sistemi realizzati é stato seguito un approccio modulare: ad ogni passo della nostra ricerca corrisponde uno specifico programma in modo da facilitare eventuali variazioni. Il primo aspetto importante da discutere é la verifica di alcune caratteristiche della libreria *OMSM*, le quali dipendono dalla piattaforma utilizzata: ne discuteremo le caratteristiche nel paragrafo 7.1. Una volta verificato il corretto funzionamento della libreria *OMSM*, ci occuperemo della gestione dei complessi simpliciali, i quali forniscono le rappresentazioni degli oggetti geometrici che vogliamo memorizzare, come abbiamo visto nella sezione 2.3.2. Nel paragrafo 7.2 mostreremo i modelli geometrici che sono stati oggetto della nostra indagine e gli strumenti software che abbiamo utilizzato per gestirli. Infine, vedremo come poter utilizzare il framework *OMSM* per decomporre griglie di triangoli immerse nello spazio euclideo $\mathbb{E}^3$ attraverso l'utilizzo di un certo indice spaziale: nel paragrafo 7.3 vedremo quali programmi abbiamo sviluppato per risolvere questo problema, verificando il diverso comportamento di alcuni indici in memoria secondaria.

7.1 I test preliminari

In questa prima fase, la nostra analisi verterá sullo studio delle caratteristiche del sistema realizzato, le quali possono dipendere dalla piattaforma di sviluppo: ad esempio puó variare il numero dei byte necessari per codificare valori di un certo tipo. Il primo passo della nostra analisi consiste nel fissare le piattaforme che useremo per la fase di verifica: la macchina utilizzata é

un computer portatile con processore *Mobile AMD Athlon XP2500+* avente 512 *Mb* di memoria primaria, sul quale vengono eseguiti il sistema operativo *GNU/Linux* e quello *Microsoft Windows*. Come esponente del sistema operativo *GNU/Linux*, useremo la distribuzione *Open SUSE 10.1* avente la versione 2.6.16 del kernel: su questa piattaforma il compilatore di riferimento é il *GNU C++ Compiler 4.0*. Per maggiori approfondimenti su questa piattaforma rifarsi a [GCC87], [TAW+02], [Esu04] e [Osu05]. Invece, come esponente del sistema operativo *Microsoft Windows*, utilizzeremo *Microsoft Windows XP Home Edition* con il *Service Pack 2*: su questa piattaforma il compilatore di riferimento é il *Microsoft Visual Studio .NET*. Per maggiori approfondimenti su questo sistema software rifarsi a [Net02], [MVS03] e [MSD07]. Come abbiamo avuto modo di osservare nella sezione 6.1.2, entrambe le piattaforme supportano la libreria *OMSM* quindi é interessante controllarne il funzionamento.

Il primo aspetto che é importante verificare é la codifica dei valori all'interno delle due piattaforme: nella sezione 7.1.1 presenteremo le caratteristiche del programma *Types*, il quale si occupa di visualizzare il numero di byte ed il dominio dei valori rappresentabili con i tipi predefiniti del linguaggio *C++*. Nel paragrafo 6.2 abbiamo visto come l'oggetto modellato dalla classe *OperationTimer* sia l'unico componente la cui implementazione dipende dalla piattaforma in esame: nella sezione 7.1.2 descriveremo l'utilizzo del programma *Timers*, il quale si occupa di confrontare le prestazioni e di calcolare il rallentamento introdotto dal componente *OperationTimer* sulle due diverse piattaforme. Durante il nostro lavoro di ricerca riveste un ruolo importante il tipo di finale dei dati binari: basti pensare all'allineamento dei byte nella rappresentazione binaria di un certo oggetto, aspetto che abbiamo discusso nella sezione 6.5.1. Pertanto é importante verificare il corretto riconoscimento del tipo di piattaforma ed applicare l'eventuale operazione di allineamento dei byte. Descriveremo i programmi *Cubel* e *Cubeb* nella sezione 7.1.3: essi si occupano di scrivere la descrizione binaria di una particolare mesh su un certo file in formato *PLY*. Il programma *Cubel* si occupa dell'allineamento dei byte a finale piccolo, *Cubeb* di quello a finale grande. Non tutti i programmi realizzati sfruttano la libreria *OMSM*: il programma *Types* non utilizza alcuna libreria aggiuntiva oltre a quelle standard del linguaggio *C++* mentre gli altri programmi sfruttano le funzionalitá offerte dalla libreria *OMSM*, collegandola in maniera sia statica e sia dinamica sulla piattaforma *GNU/Linux* e solamente in maniera statica per la piattaforma *Microsoft Windows*: abbiamo giá espresso le motivazioni di questa scelta implementativa nella sezione 6.1.2.

7.1.1 Il programma *Types*

Come abbiamo visto nel paragrafo 6.4, é importante fissare in maniera univoca il tipo di un certo dato in quanto possiamo essere interessati al numero

di byte effettivamente occupati e quindi all'insieme dei valori rappresentabili con quel tipo di dato: per queste ragioni abbiamo sviluppato il programma *Types*, il quale si occupa di estrarre e visualizzare sullo schermo il numero di byte ed il dominio dei valori rappresentabili con i tipi predefiniti del linguaggio *C++*. *Types* é un semplice programma a linea di comando, contenuto nel file *types.cpp* ed invocabile dalla shell del sistema di operativo digitando:

```
user@host:path > types
```

La tabella 7.1 riassume i risultati ottenuti i quali coincidono per le piattaforme di sviluppo della libreria *OMSM*: inoltre, per i valori in virgola mobile, gli estremi del dominio si riferiscono al valore assoluto del dato rappresentato: se il dominio é $[a, b]$ allora i valori rappresentabili appartengono a $[-b, -a] \cup \{0\} \cup [a, b]$.

Tipo	**Dimensione**	**Dominio**
char	1 byte	$-127, \ldots, 128$
unsigned char	1 byte	$0, \ldots, 255$
short int	2 byte	$-32768, \ldots, 32767$
unsigned short int	2 byte	$0, \ldots, 65535$
int	4 byte	$-2147483648, \ldots, 2147483647$
unsigned int	4 byte	$0, \ldots, 4294967295$
long int	4 byte	$-2147483648, \ldots, 2147483647$
unsigned long int	4 byte	$0, \ldots, 4294967295$
float	4 byte	$1.17549e-38, \ldots, 3.40282e+38$
double	8 byte	$2.22507e-308, \ldots, 1.79769e+308$

Tabella 7.1: i risultati forniti dall'esecuzione del programma *Types*: si ottengono gli stessi risultati sia sulla piattaforma *GNU/Linux* e sia su quella *Microsoft Windows*.

Le informazioni fornite da questo programma sono state codificate dagli standard *ANSI* e dovrebbero essere oramai supportate da quasi tutti i compilatori in circolazione: in realtá vecchie versioni dell'ambiente integrato *Microsoft Visual Studio* sembrano non rispondere correttamente a questi standard e possono fornire dei risultati differenti. I risultati riassunti dalla tabella 7.1 coincidono con quelli definiti dallo standard *ANSI*: per maggiori informazioni sul funzionamento di questo programma rifarsi a [Can07c].

7.1.2 Il programma *Timers*

Nel paragrafo 6.2 abbiamo visto come l'oggetto modellato dalla classe *OperationTimer* sia il componente dedicato alla misura del tempo nelle applicazioni: esso viene usato anche per la generazione di una marca temporale

all'interno della cache, come abbiamo avuto modo di descrivere nel paragrafo 6.3. Inoltre questo componente puó essere utilizzato per misurare il tempo di esecuzione, ad esempio nei programmi di verifica che descriveremo nel resto del capitolo: ovviamente il suo funzionamento introduce un rallentamento nell'esecuzione delle operazioni. Questa penalizzazione é nota in letteratura con il termine inglese *overhead* ed é opportuno misurarla: per fornire una stima dell'overhead intodotto dall'utilizzo del componente *OperationTimer* abbiamo sviluppato il programma *Timers*. Si tratta di un semplice programma a linea di comando, contenuto nel file *timers.cpp* ed invocabile dalla shell del sistema operativo, digitando:

```
user@host:path > timers
```

La figura 7.1 ne mostra il codice, scritto nel linguaggio *C++*.

```
#include "os.h"
#include "timerop.h"
#include "memoryexception.h"
#include <cstdlib>
#include <iostream>
using namespace std;
using namespace omsm;

#define TIMES 10000000

int main() {
    OperationTimer *ot;
    double aux;

    if((ot = new OperationTimer())==NULL) { ... }
    ...
    for(unsigned long j=0;j<TIMES;j++) { aux = ot->getTime(); }
    ...
    cout<<"\tThe overhead time is: "<<aux/TIMES<<endl<<endl;
    cout.flush();

    delete ot;
    return EXIT_SUCCESS; }
```

Figura 7.1: il codice *C++* del programma *Timers*

Come si puó notare, l'utilizzo di una specifica piattaforma é trasparente per questo programma, in quanto questo aspetto é completamente gestito dal componente *OperationTimer*, quindi possiamo dimenticare, per un attimo,

le differenze esistenti fra le diverse implementazioni di questo oggetto, concentrandoci solamente sull'interfaccia esterna. La prima parte del codice si occupa di creare i componenti richiesti mentre il ciclo *for* é la parte piú significativa del programma: eseguendo un numero elevato di invocazioni del metodo *getTime* si ottiene una stima S_t del tempo necessario ad eseguirle. Quindi, dividendo S_t per il numero di invocazioni (10 milioni nel nostro caso), si puó stimare il tempo di esecuzione S_g del metodo *getTime*: questa stima misura il rallentamento introdotto dall'utilizzo del componente *OperationTimer*. Per rendere piú stabili i risultati ottenuti é possibile ripetere piú volte l'esecuzione di questo programma e poi considerare il valore medio della stima S_g: la tabella 7.2 riassume i risultati ottenuti.

Ripetizioni	S_g **con *GNU/Linux***	S_g **con *Microsoft Windows***
10	$0,4413\ \mu s$	$0,8446\ \mu s$
20	$0,4397\ \mu s$	$0,8469\ \mu s$
30	$0,4411\ \mu s$	$0,8435\ \mu s$

Tabella 7.2: i risultati dell'esecuzione del programma *Timers* nelle due piattaforme di sviluppo.

Come si puó notare, l'implementazione del programma *Timers* é piú lenta all'incirca di un fattore due se si utilizza la piattaforma *Microsoft Windows*: questo vuol dire che la funzione *QueryPerformanceCounter* é meno efficiente di quella *gettimeofday*, presente nelle *API* del kernel *Linux*. Per i nostri scopi, una granularitá di $0,84\ \mu s$ é comunque sufficiente: per maggiori informazioni sul funzionamento di questo programma rifarsi a [Can07c].

7.1.3 I programmi *Cubel* e *Cubeb*

Nel paragrafo 3.2 abbiamo fissato il concetto di *finale* per un dato binario e nella sezione 6.4.3 abbiamo visto quanto sia importante verificare il corretto riconoscimento del tipo di piattaforma nel parsing di un file in formato *PLY* di tipo binario. Inoltre abbiamo descritto il sistema di persistenza degli oggetti nella sezione 6.5.1 ed abbiamo chiarito in che modo possa essere organizzata la sequenza di byte che descrive un qualsiasi componente all'interno della libreria *OMSM*. Per verificare questi aspetti abbiamo realizzato i programmi *Cubel* e *Cubeb*, i quali si occupano di scrivere su un certo file la descrizione binaria della mesh *cubo canonico* in formato *PLY* rispettivamente a finale piccolo ed a finale grande. Per *cubo canonico* si intende la mappa poligonale che descrive il contorno del cubo nello spazio euclideo $\mathbb{E}^3$ definibile come il prodotto cartesiano $[0,1] \times [0,1] \times [0,1]$: in questo caso elenchiamo le sei facce quadrate del cubo. *Cubel* e *Cubeb* sono dei semplici programmi a linea di comando, invocabili dalla shell del sistema operativo digitando rispettivamente:

```
user@host:path > cubel meshfile
user@host:path > cubeb meshfile
```

dove *meshfile* é il percorso del file in cui scrivere la rappresentazione binaria della mesh richiesta, per entrambi i programmi realizzati. I programmi *Cubel* e *Cubeb* sono contenuti rispettivamente nei file *cubel.cpp* e *cubeb.cpp*: non ne mostreremo il codice in quanto non presentano proprietá di particolare interesse. L'unica caratteristica importante é la capacitá di adattamento rispetto al finale dei dati binari: ogni sequenza viene invertita solamente se l'ordinamento dei byte nella piattaforma é di tipo diverso da quello dell'output in modo da allinearli correttamente. Quindi *Cubel* invertirá le stringhe di byte da scrivere solamente se la piattaforma é di tipo *big–endian* mentre *Cubeb* solamente se la piattaforma é di tipo *little–endian*: la macchina che abbiamo adoperato per la fase di verifica é a finale piccolo quindi l'operazione di allineamento verrá effettuata solamente dal programma *Cubeb*. Questa proprietá ci permette di misurare la complessitá dell'operazione di inversione: vediamo come. Il primo passo consiste nel misurare il tempo di esecuzione dei due programmi attraverso il componente *OperationTimer*, che abbiamo descritto nel paragrafo 6.2: indicheremo con T_l e T_b rispettivamente i tempi di esecuzione dei programmi *Cubel* e *Cubeb*. Per rendere piú stabili le misurazioni é possibile ripetere piú volte l'esecuzione di ogni programma e poi considerare il valore medio dei tempi ottenuti, a seconda delle varie piattaforme di sviluppo. Le tabelle 7.3 e 7.4 mostrano rispettivamente i risultati ottenuti dal programma *Cubel* e da quello *Cubeb*.

Ripetizioni	T_l **con *GNU/Linux***	T_l **con *Microsoft Windows***
10	$0,4563\ \mu s$	$0,9037\ \mu s$
20	$0,4627\ \mu s$	$0,8840\ \mu s$
30	$0,4664\ \mu s$	$0,8724\ \mu s$

Tabella 7.3: i risultati dell'esecuzione del programma *Cubel* nelle due diverse piattaforme di sviluppo: le sequenze di byte da scrivere non vengono invertite perché la macchina utilizzata é a finale piccolo.

Ripetizioni	T_b **con *GNU/Linux***	T_b **con *Microsoft Windows***
10	$0,5135\ \mu s$	$1,1285\ \mu s$
20	$0,5144\ \mu s$	$1,1179\ \mu s$
30	$0,5142\ \mu s$	$1,1208\ \mu s$

Tabella 7.4: i risultati dell'esecuzione del programma *Cubeb* nelle due diverse piattaforme di sviluppo: le sequenze di byte da scrivere vengono invertite perché la macchina utilizzata é a finale piccolo.

A prima vista, l'implementazione *GNU/Linux* dei due programmi sembrerebbe nettamente piú veloce di quella *Microsoft Windows*, tuttavia bisogna tenere presente che i tempi forniti dalle tabelle 7.3 e 7.4 sono comprensivi dell'overhead introdotto dall'utilizzo del componente *OperationTimer* e quindi questo risultato é una conseguenza di quanto visto nella sezione 7.1.2. Ricordando i risultati della tabella 7.2, é facile osservare che i tempi di esecuzione di ogni programma sono all'incirca gli stessi su entrambe le piattaforme in quanto il tempo di esecuzione *reale* é nettamente inferiore rispetto a quello dell'invocazione del metodo *getTime*. Una volta misurati i tempi di esecuzione T_l e T_b, dobbiamo stimare la complessitá dell'operazione di inversione delle sequenze di byte, eseguita dal programma *Cubeb*: supponiamo che questa operazione richieda tempo t_c. Entrambi i programmi scrivono la stessa quantitá di dati sul file ed interrogano il componente *OperationTimer*: per semplicitá si suppone che il tempo necessario ad eseguire queste due operazioni sia lo stesso per i due programmi, nel seguito lo indicheremo con T_0. In queste ipotesi valgono le seguenti relazioni:

- $T_l = T_0$;
- $T_b = T_0 + t_c$.

Pertanto si puó utilizzare la differenza ΔT fra i tempi T_b e T_l definita da

$$\Delta T = \|T_b - T_l\| = \|T_0 + t_c - T_0\| = t_c$$

per misurare il tempo necessario al programma *Cubeb* per allineare i byte. La tabella 7.5 mostra le differenze ΔT ottenute a partire dai tempi di esecuzione dei programmi *Cubel* e *Cubeb*, contenuti nelle tabelle 7.3 e 7.4: come si puó notare dai risultati della tabella 7.5, l'operazione di allineamento dei byte é circa quattro volte piú lenta nella piattaforma *Microsoft Windows* e questo potrebbe significare che la gestione della memoria é piú efficiente in un sistema *GNU/Linux*.

Ripetizioni	**ΔT con *GNU/Linux***	**ΔT con *Microsoft Windows***
10	$0,0572\ \mu s$	$0,2248\ \mu s$
20	$0,0517\ \mu s$	$0,2339\ \mu s$
30	$0,0478\ \mu s$	$0,2484\ \mu s$

Tabella 7.5: il tempo di esecuzione dell'operazione di allineamento dei byte nelle due diverse piattaforme di sviluppo.

Inoltre possiamo osservare che, ricordando i dati riassunti dalla tabella 7.2, la misura dell'overhead introdotto dall'operazione di allineamento dei byte ha un peso diverso nelle due piattaforme utilizzate per lo sviluppo della libreria *OMSM*. Nella piattaforma *GNU/Linux* il valore ΔT é all'incirca un

ottavo del tempo di esecuzione del metodo *getTime* e circa un decimo rispetto ai tempi T_l e T_b quindi l'overhead introdotto é trascurabile. Nella piattaforma *Microsoft Windows* il valore ΔT é all'incirca un quarto del tempo di esecuzione del metodo *getTime* e circa un quinto rispetto ai tempi T_l e T_b e quindi l'overhead introdotto ha sicuramente un maggior peso. Tuttavia dobbiamo ricordare che non abbiamo usato componenti del framework *.NET* e questo potrebbe aver influenzato le prestazioni: non si esclude che questo framework possa essere utilizzato in future versioni della libreria *OMSM*, delle quali parleremo nel capitolo 8.

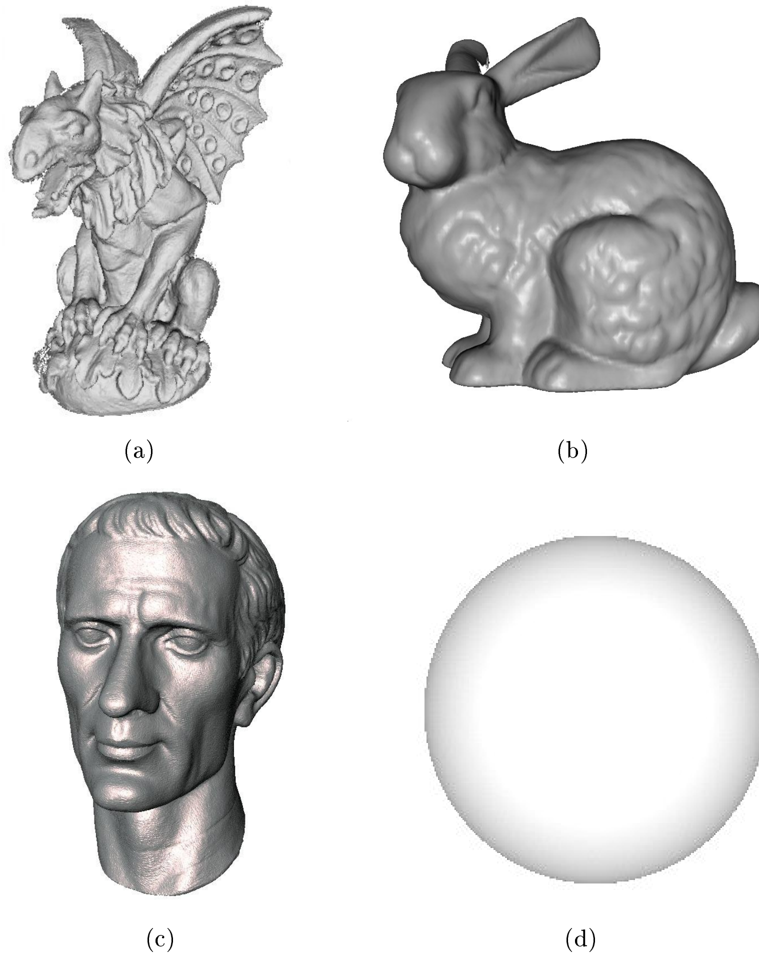
7.2 La gestione dei modelli geometrici

L'obiettivo primario della seconda fase della nostra ricerca é quello di ottenere una rappresentazione adatta ai nostri scopi, a partire da un certo modello geometrico $\mathcal{M}$ sul quale vogliamo operare.

Come abbiamo giá visto nel paragrafo 6.4, i modelli indicizzati non sono adatti ai nostri scopi visto che non forniscono un accesso diretto alle coordinate geometriche dei simplessi che vogliamo studiare: per queste ragioni abbiamo definito il formato *TSOUP*, del quale abbiamo parlato nella sezione 6.4.4. Pertanto dobbiamo convertire un modello indicizzato in un insieme non organizzato di triangoli (noto in letteratura come *triangles soup*), ma prima di compiere quest'operazione bisogna accertarsi che il modello $\mathcal{M}$ soddisfi le proprietá richieste e che i suoi simplessi siano triangoli, in modo da poterlo convertire in maniera corretta. Nella sezione 7.2.1 descriveremo il programma *Check*, il quale si occupa di controllare che il modello in input verifichi le proprietá richieste: in caso contrario questo programma cercherá di correggere le violazioni, se possibile. Nella sezione 7.2.2 descriveremo il programma *ToSoup*, il quale si occupa di convertire un modello geometrico $\mathcal{M}$ in una *triangles soup*, assumendo che $\mathcal{M}$ abbia superato i controlli del programma *Check*. Entrambi i programmi realizzati sfruttano le funzionalitá offerte dalla libreria *OMSM*, collegandola in maniera sia statica e sia dinamica sulla piattaforma *GNU/Linux* mentre il collegamento sará solamente di tipo statico sulla piattaforma *Microsoft Windows*: abbiamo giá espresso le motivazioni di questa scelta implementativa nella sezione 6.1.2.

Abbiamo verificato il funzionamento di tali programmi utilizzando vari modelli geometrici: la figura 7.2 ne mostra alcune rappresentazioni artistiche, mentre la tabella 7.6 ne riassume le caratteristiche. I modelli *Gargoyle* e *Caesar* sono liberamente scaricabili dall'archivio del progetto *AIM@SHAPE*: per approfondimenti rifarsi a [Aim04]. Il modello *Bunny* é scaricabile dall'archivio di modelli geometrici presso lo *Stanford Computer Graphics Laboratory*: per approfondimenti rifarsi a [Sta94]. Inoltre abbiamo usato il modello *Cube* (un semplice cubo) per verificare le capacitá di correzione del sistema realizzato mentre il modello *Icosphere* é l'approssimazione di una sfe-

ra che si ottiene attraverso la bisezione ricorsiva delle facce di un icosaedro: per la sua definizione formale rifarsi a [Oli05] e [SWND07].



(a) (b)

(c) (d)

Figura 7.2: rappresentazioni artistiche del modello *Gargoyle* (a), del modello *Bunny* (b), del modello *Caesar* (c) e di quello *Icosphere* (d). Il modello *Cube*, che non rappresentiamo, é un semplice cubo con le sei facce quadrate.

Modello	**Formato**	**Numero di vertici**	**Numero di facce**
Cube	*PLY*	8	6
Gargoyle	*PLY*	25038	50000
Bunny	*TRI*	34834	69451
Caesar	*OFF*	387900	774164
Icosphere	*PLY*	655362	1310720

Tabella 7.6: caratteristiche dei modelli usati nella nostra ricerca.

7.2.1 Il programma *Check*

Nella sezione 6.4.3 abbiamo delineato quali proprietá debba possedere una mappa poligonale per poter essere adeguatamente gestita dalla libreria *OMSM*: inoltre abbiamo fissato le caratteristiche dei componenti software che si occupano della gestione dei modelli in input e della loro trasformazione in triangolazioni. Per verificare questi aspetti, abbiamo realizzato il programma *Check*, il quale é un semplice programma a linea di comando, contenuto nel file *check.cpp* ed invocabile dalla shell del sistema operativo, digitando:

```
user@host:path > check meshfile
```

dove *meshfile* é il percorso del file che contiene la mesh da analizzare. La figura 7.3 ne mostra il codice, scritto nel linguaggio *C++*.

```
#include "os.h"
#include "timerop.h"
#include "meshandler.h"
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;
using namespace omsm;

int main(int argc, char **argv) {

 MeshHandler* mh;
 OperationTimer *ot;

 if(argc!=2) { ... }
 if((mh = new MeshHandler())==NULL) { ... }
 if((ot = new OperationTimer())==NULL) { ... }

 try { mh->checkMesh(string(argv[1])); }
 catch(AllocationErrorException aee) { ... }
 catch(MeshAnalysisErrorException maee) { ... }

 cout<<"\tThis program runs for: "<<ot->getTime()<<" seconds";
 delete mh;
 delete ot;
 return EXIT_SUCCESS; }
```

Figura 7.3: il codice *C++* del programma *Check*.

Come si puó notare, il tempo di esecuzione viene misurato dal componente *OperationTimer*, descritto nella sezione 6.2. Ma la caratteristica piú im-

portante del programma *Check* risiede nell'utilizzo del componente *MeshHandler*, introdotto nella sezione 6.4.2: questa scelta ha delle implicazioni importanti, vediamo quali.

Il formato di una mesh é identificato dall'estensione del file che la contiene: l'utilizzo del componente *MeshHandler* permette al programma *Check* di poter gestire i formati supportati dalla libreria *OMSM* in maniera trasparente per l'utente. Inoltre questa scelta permette un approccio modulare e piú flessibile alla risoluzione del problema: in future versioni della libreria *OMSM* potrebbe essere ampliato l'insieme dei formati supportati e con questo approccio il programma *Check* sará in grado di gestire direttamente i nuovi formati, senza alcuna modifica. Questa scelta progettuale é stata proposta piú volte all'interno della libreria *OMSM* allo scopo di migliorare la modularitá e l'estendibilitá del sistema realizzato.

Questo programma supporta anche l'operazione di logging e documenta le operazioni svolte su un file, che indicheremo come *file di log*: il percorso di questo file é dato dalla concatenazione del parametro di input *meshfile* con la stringa *"_check.log"*: ad esempio se la mesh é contenuta nel file *"cube.ply"* allora il percorso del file di log sará *"cube.ply_check.log"*.

Il programma *Check* fornisce anche la versione verificata della mesh in cui tutte le facce sono triangoli: questa operazione non é supportata dai formati *VER–TRI* e *TSOUP*, come abbiamo giá accennato nella sezione 6.4.3. Il percorso della versione corretta della mesh é ottenuto frapponendo la stringa *"_errata"* fra il nome e l'estensione del file, contenuti nella stringa *meshfile*: se la mesh é contenuta nel file *"cube.ply"* allora il percorso della mesh modificata sará *"cube_errata.ply"*. Se la mesh é in formato *OFF*, la versione verificata viene cancellata se non vi sono correzioni da apporre, mentre con il formato *PLY* verrá sempre mantenuta in quanto vengono memorizzati solamente gli elementi sintattici *vertex* (con le proprietá x, y e z) e *face* (con la proprietá *vertex_indices*) per velocizzare le future analisi su questa mesh: per approfondimenti sulla struttura di un file *PLY* rifarsi a [Wal01] e [Att06].

Puó essere interessante valutare l'efficienza del programma *Check* sui modelli geometrici che abbiamo descritto in precedenza e capire se ci sono differenze importanti fra le due piattaforme di sviluppo. Il tempo di esecuzione viene misurato attraverso il componente *OperationTimer*: nella sezione 7.1.2 abbiamo valutato l'overhead dovuto all'utilizzo di questo oggetto. Per rendere piú stabili i risultati é possibile ripetere piú volte l'esecuzione di questo programma e poi considerare il valore medio della stima del tempo di esecuzione T_C: la tabella 7.7 riassume i tempi ottenuti, espressi in secondi. I tempi di esecuzione forniti dalla tabella 7.7 comprendono non solo la fase di verifica, ma anche le altre operazioni da eseguire sulla mesh come la documentazione delle operazioni svolte e la creazione di una versione corretta della mesh: inoltre vi é l'overhead dovuto all'utilizzo del componente *OperationTimer*. In generale i tempi di esecuzione ottenuti dal programma *Check* concordano con quelli presentati nel paragrafo 7.1, dove si sono analizzate le

prestazioni di alcuni componenti della libreria *OMSM* rispetto alle diverse piattaforme utilizzate.

Modello	T_C **con** ***GNU/Linux***	T_C **con** ***Microsoft Windows***
Cube	$0,041\ s$	$0,43\ s$
Gargoyle	$17,303\ s$	$147,35\ s$
Bunny	$15,775\ s$	$154,69\ s$
Caesar	$194,83\ s$	$1846,86\ s$
Icosphere	$471,32\ s$	$4678,33s$

Tabella 7.7: i tempi di esecuzione del programma *Check*, espressi in secondi, sulle due piattaforme di sviluppo, utilizzando i modelli descritti nel paragrafo 7.2. É interessante osservare la crescita del tempo di esecuzione T_C in funzione della complessitá del modello.

L'esempio piú significativo é fornito dal modello *Icosphere*, il quale é memorizzato sotto forma di un file binario a finale grande: la macchina utilizzata per le nostre verifiche é di tipo little–endian perció é necessario allineare correttamente le sequenze di byte. Come abbiamo ampiamente discusso nel paragrafo 7.1, l'efficienza delle tecniche sviluppate nella libreria *OMSM* puó dipendere dalla piattaforma utilizzata e questa proprietá puó spiegare, in parte, la discrepanza dei tempi di esecuzione fra le due diverse piattaforme: come si puó notare l'esecuzione del programma *Check* é circa dieci volte piú lenta con la piattaforma *Microsoft Windows*. Questo fenomeno puó essere dovuto anche alle diverse tecniche di allocazione della memoria nei due sistemi operativi, visto che durante l'esecuzione vengono continuamente creati e distrutti oggetti: anche negli altri programmi realizzati é stato riscontrato questo problema. Per ulteriori approfondimenti sul funzionamento del programma *Check* rifarsi a [Can07c].

7.2.2 Il programma *ToSoup*

Nella sezione 6.4.4 abbiamo visto come i formati indicizzati non permettano l'accesso diretto alle coordinate euclidee di una faccia e quindi non sono indicati per i nostri scopi, a differenza del formato *TSOUP*. Pertanto é necessario convertire una mesh in formato indicizzato in un insieme non organizzato di triangoli. Per risolvere questo problema abbiamo sviluppato il programma *ToSoup*, il quale é un semplice programma a linea di comando, contenuto nel file *tosoup.cpp* ed invocabile dalla shell del sistema operativo, digitando:

```
user@host:path > tosoup meshfile
```

dove *meshfile* é il percorso del file che contiene la mesh da convertire. La figura 7.4 ne mostra il codice, scritto nel linguaggio *C++*: come si puó notare la sua struttura é molto simile a quella del programma *Check*.

```
#include "os.h"
#include "timerop.h"
#include "meshandler.h"
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;
using namespace omsm;

int main(int argc, char **argv) {

 MeshHandler* mh;
 OperationTimer *ot;

 if(argc!=2) { ... }
 if((mh = new MeshHandler())==NULL) { ... }
 if((ot = new OperationTimer())==NULL) { ... }

 try { mh->tosoup(string(argv[1])); }
 catch(AllocationErrorException aee) { ... }
 catch(MeshAnalysisErrorException maee) { ... }

 cout<<"\tThis program runs for: "<<ot->getTime()<<" seconds";
 delete mh;
 delete ot;
 return EXIT_SUCCESS; }
```

Figura 7.4: il codice *C++* del programma *ToSoup*.

Il programma *ToSoup* delega la gestione dei formati supportati al componente *MeshHandler*, ereditando in maniera completamente trasparente i vantaggi di questa scelta progettuale, descritti nella sezione 7.2.1. Anche questo programma supporta l'operazione di logging e documenta le operazioni svolte su un file, che indicheremo come *file di log*: il percorso di questo file é dato dalla concatenazione del parametro di input *meshfile* con la stringa *"_ tosoup.log"*: ad esempio se la mesh é contenuta nel file *"cube.ply"* allora il percorso del file di log sará *"cube.ply_ tosoup.log"*.

Il programma *ToSoup* produce come risultato la versione della mesh in input convertita in formato *TSOUP*: questa operazione non é ovviamente definita per una mesh giá in questo formato, come giá accennato nella sezione 6.4.4. Il percorso della mesh convertita in formato *TSOUP* si puó ottenere dalla stringa *meshfile*, la quale contiene il percorso della triangolazione originale: vengono estratti il nome e l'estensione del file, divisi dalla stringa

"_", il risultato viene poi concatenato con l'estensione *"tsoup"*. Se la mesh é contenuta nel file *"cube.ply"* allora il risultato sará memorizzato nel file *"cube_ply.tsoup"*. Il programma *ToSoup* assume che il modello geometrico in input abbia superato i controlli eseguiti da *Check*, in modo tale che ogni faccia sia un triangolo non degenere. In realtá questo programma condivide gran parte del codice sviluppato per la verifica delle mappe poligonali e quindi é in grado di riconoscere una faccia che non sia un triangolo valido: in questo caso l'esecuzione del programma *ToSoup* viene fermata.

Puó essere interessante valutare l'efficienza del programma *ToSoup* sui modelli geometrici che abbiamo descritto in precedenza e capire se ci sono differenze importanti fra le due piattaforme di sviluppo. Anche in questo caso abbiamo utilizzato il componente *OperationTimer* per valutare il tempo di esecuzione T_S: le misure sono state effettuate con la stessa metodologia utilizzata per il programma *Check*. La tabella 7.8 riassume i tempi ottenuti, espressi in secondi.

Modello	T_S **con *GNU/Linux***	T_S **con *Microsoft Windows***
Cube	$0,033\ s$	$0,35\ s$
Gargoyle	$17,43\ s$	$169,25\ s$
Bunny	$19,69\ s$	$185,46\ s$
Caesar	$246,16\ s$	$2328,23\ s$
Icosphere	$554,25\ s$	$5127,23\ s$

Tabella 7.8: i tempi di esecuzione del programma *ToSoup*, espressi in secondi, sulle due piattaforme di sviluppo, utilizzando i modelli descritti nel paragrafo 7.2. É interessante osservare la crescita del tempo di esecuzione T_S in funzione della complessitá del modello.

Come si puó facilmente notare, i risultati ottenuti sono in accordo con le osservazioni generali sul funzionamento della libreria nelle sue diverse implementazioni: anche in questo caso é presente la differenza di funzionamento sulla piattaforma *Microsoft Windows*, giá evidenziata nella sezione 7.2.1. Per queste ragioni, nelle prossime sezioni non presenteremo i risultati in entrambe le piattaforme, visto che é sufficiente presentare quelli ottenuti con il sistema operativo *GNU/Linux*: in realtá non é tanto interessante misurare l'effettivo peggioramento del tempo di esecuzione, quanto conoscere l'esistenza di questo problema. Per ulteriori approfondimenti sul funzionamento del programma *ToSoup* rifarsi a [Can07c].

7.3 La decomposizione dei modelli geometrici

Per poter applicare una qualsiasi tecnica di semplificazione iterativa, é necessario decomporre un modello geometrico attraverso un indice spaziale, il

quale sará implementato nell'architettura di memorizzazione dei dati geometrici, discussa nel paragrafo 6.5. Il framework *OMSM* richiede un'adeguata fase di configurazione prima di poter essere utilizzato. Nella sezione 6.5.6 abbiamo discusso i principi sui quali si basa questo processo: come si puó notare non é un'operazione intuitiva per l'utente. Per facilitarne la configurazione, abbiamo sviluppato il programma *Omsmconf*, che descriveremo nella sezione 7.3.1. Una volta aggiornate le impostazioni del sistema possiamo attivare il processo di decomposizione di un complesso simpliciale, utilizzando il programma *Decompose*, che descriveremo nella sezione 7.3.2.

Il programma *Omsmconf* é stato realizzato attraverso il toolkit *FLTK*, un framework di ridotte dimensioni per la creazione di interfacce grafiche: questo sistema software é pensato come una libreria collegabile solamente in modo statico, viste le sue ridotte dimensioni. Quindi, per evitare comportamenti inefficienti ed instabili, il programma *Omsmconf* richiede il collegamento di tipo statico di tutte le librerie necessarie, sia per la piattaforma *GNU/Linux* e sia per quella *Microsoft Windows.* Invece il programma *Decompose* si comporta in maniera simile ai programmi *Check* e *ToSoup*, descritti rispettivamente nelle sezioni 7.2.1 e 7.2.2.

7.3.1 Il programma *Omsmconf*

Nel paragrafo 6.5 abbiamo descritto le caratteristiche dell'architettura di memorizzazione dei dati spaziali contenuta nella libreria *OMSM*, mettendo in luce la sua natura modulare e l'ampia gamma di scelte disponibili. Tuttavia la gestione del file di configurazione, necessaria per modificare le impostazioni del sistema, non é intuitiva per l'utente in quanto egli deve conoscerne il formato: in caso di nuove aggiunte alle funzionalitá della libreria *OMSM* questo meccanismo mostra tutti i suoi limiti.

Per semplificare la modifica delle impostazioni del sistema é stato sviluppato il programma *Omsmconf*, dotato di una comoda interfaccia grafica attraverso la quale l'utente puó scegliere le opzioni volute. Per la sua realizzazione, é stato utilizzato il toolkit *FLTK* (dall'inglese *Fast Light ToolKit*), disponibile in licenza *LGPL*: si tratta di un framework di ridotte dimensioni in grado di realizzare interfacce grafiche su diverse piattaforme, per approfondimenti su questo sistema software rifarsi a [SES98a] e [SES98b]. Il programma *Omsmconf* é invocabile dalla shell del sistema di operativo, senza alcun parametro di input, digitando:

```
user@host:path > omsmconf
```

Il codice di questo programma é molto complesso e quindi non lo presenteremo: per ulteriori approfondimenti rifarsi ai file *omsmconf.cpp*, *omsmconfigurator.h* e *omsmconfigurator.cpp* oppure a [Can07c]. Il programma *Omsmconf* utilizza il componente *OptionsManager*, introdotto nella sezione 6.5.6, in modo tale da garantire l'aderenza dell'interfaccia grafica ai servizi

offerti dalla specifica versione della libreria che stiamo utilizzando. La figura 7.5 mostra come si presenta la sua interfaccia grafica per la versione preliminare della libreria *OMSM*, nella quale non possiamo modificare la politica di paginazione e la distribuzione dei dati.

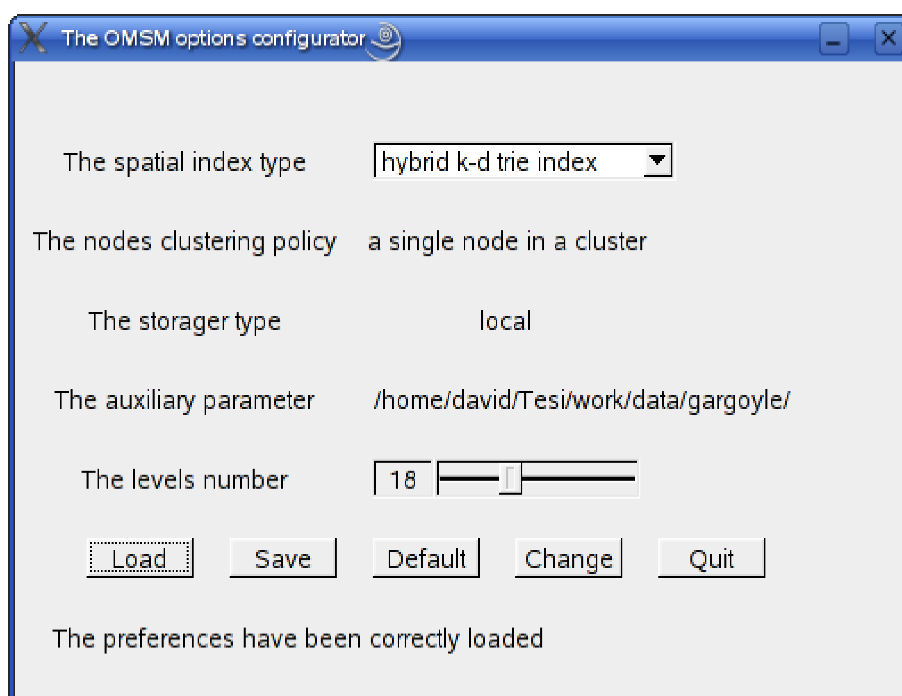


Figura 7.5: l'interfaccia grafica del programma *Omsmconf* adatta al prototipo iniziale della libreria *OMSM*.

Il componente etichettato con *"The spatial index type"* permette la scelta del tipo di indice spaziale che si vuole utilizzare, scegliendo fra gli indici attualmente supportati. Nella versione corrente della libreria *OMSM* possiamo scegliere fra:

- l'elemento *"octree index"* per poter utilizzare l'indice spaziale *Octree*, descritto nella sezione 5.4;
- l'elemento *"k–d tree index"* per poter utilizzare l'indice spaziale *K–d tree*, descritto nella sezione 5.5;
- l'elemento *"hybrid quadtrie index"* per poter utilizzare la versione ibrida della struttura *Octree*, discussa nella sezione 5.6.2;
- l'elemento *"hybrid k–d trie index"* per poter utilizzare la versione ibrida della struttura *K–d tree*, discussa nella sezione 5.6.2.

Il componente etichettato con *"The nodes clustering policy"* permette la scelta del tipo di politica di paginazione che si vuole utilizzare, mentre quello etichettato con *"The storager type"* permette di impostare il tipo di memorizzazione dei dati in base alla loro dislocazione fisica. Nel prototipo realizzato non vi sono altre scelte se non memorizzare un solo nodo in un cluster e salvare i dati nella stessa macchina sulla quale é in esecuzione il sistema *OMSM* e quindi non viene data la possibilitá di modificare questi due aspetti. Future versioni della libreria *OMSM* potrebbero supportare altre politiche di clustering dei nodi e la memorizzazione di dati remoti quindi questi componenti forniranno gli strumenti per modificare le impostazioni, adattandosi automaticamente alla situazione corrente.

Il componente etichettato con *"The auxiliary parameter"* mostra il valore corrente del parametro ausiliario. Come abbiamo visto nella sezione 6.5.6, il significato di questo parametro cambia a seconda della dislocazione fisica dei dati, infatti esso contiene

- la directory di ambiente del database, se i dati sono memorizzati nella stessa macchina sulla quale é in esecuzione il sistema;
- una stringa da interpretare a seconda delle esigenze, negli altri casi: ad esempio puó contenere l'indirizzo remoto dei dati.

Possiamo modificare il valore del parametro ausiliario attraverso il bottone *"Change"*, del quale parleremo nel seguito.

Il componente etichettato con *"The levels number"* permette la modifica del numero di livelli h, che abbiamo utilizzato nella sezione 5.6.1 per stimare la capacitá di un nodo foglia di una struttura *ibrida*: questa impostazione non ha senso con le strutture dati di tipo *tradizionale* come quelle k–d tree e octree. Questo elemento sará visualizzato dal programma *Omsmconf* solamente con gli indici spaziali che supportano nodi con capacitá $c > 1$, come avviene nella figura 7.5.

Il bottone *"Load"* permette il caricamento di impostazioni precedentemente salvate su un certo file, mostrando una finestra di dialogo attraverso la quale scegliere l'utente puó scegliere il file su cui operare. Se le impostazioni correnti non sono state salvate, viene chiesto all'utente se desidera salvarle prima di attivare il processo di caricamento dal nuovo file.

Il bottone *"Save"* permette il salvataggio delle impostazioni correnti su un certo file, mostrando una finestra di dialogo attraverso la quale l'utente puó scegliere il file su cui operare.

Il bottone *"Default"* riporta i valori delle impostazioni correnti a quelli di default. Quindi il tipo di indice spaziale sará un *K–d tree*, la politica di paginazione sará quella *"singola"*, la dislocazione fisica dei dati sará quella *"locale"* mentre il parametro ausiliario sará la directory corrente. Bisogna notare che in questo prototipo possiamo modificare solamente il tipo di indice spaziale, il numero massimo dei livelli ed il parametro ausiliario.

Il bottone *"Change"* permette di modificare il valore del parametro ausiliario attraverso una finestra di dialogo il cui comportamento dipende dalla dislocazione fisica dei dati. In questo prototipo i dati vengono salvati localmente e quindi la finestra di dialogo permette di scegliere la directory nella quale salvare l'ambiente del database. L'implementazione attuale fornisce una finestra di dialogo con un campo editabile dall'utente per modificare il valore corrente del parametro ausiliario, se la dislocazione dei dati non é quella *"locale"*: questa scelta puó rivelarsi troppo generica rispetto a delle specifiche esigenze. Ad esempio, se vogliamo inserire un indirizzo *IP* sarebbe piú opportuno creare un componente che ne faciliti la gestione: si puó concludere che questa é l'unica parte dell'interfaccia del programma *Omsmconf* che andrebbe rivista in presenza di nuove funzionalitá nella libreria *OMSM*.

Il bottone *"Quit"* permette di chiudere il programma: se le impostazioni correnti non sono state salvate, viene chiesto all'utente se desidera salvarle.

7.3.2 Il programma *Decompose*

Nella sezione 6.4.5 abbiamo delineato quali siano gli aspetti piú importanti nella decomposizione di un certo complesso simpliciale, utilizzando le funzionalitá offerte dal framework *OMSM*. Inoltre nella sezione 6.5.6 abbiamo capito come configurare l'architettura di memorizzazione, modificandone il funzionamento a seconda delle varie esigenze.

Per risolvere questi problemi abbiamo sviluppato il programma *Decompose*, il quale é un semplice programma a linea di comando, contenuto nel file *decompose.cpp* ed invocabile dalla shell del sistema operativo digitando:

```
user@host:path > decompose meshfile [ -f optsfile ]
```

dove *meshfile* é il percorso del file che contiene la mesh in formato *TSOUP* da decomporre secondo le impostazioni volute. Il parametro *optsfile* é opzionale: se non é presente, vengono applicate le impostazioni di default, descritte nella sezione 6.5.6, altrimenti *optsfile* viene interpretato come il percorso del file di configurazione, il quale potrebbe essere stato realizzato attraverso il programma *Omsmconf*, descritto nella sezione 7.3.1.

Anche il programma *Decompose* supporta l'operazione di logging e documenta le operazioni svolte su un file, che indicheremo come *file di log*: il percorso di questo file é dato dalla concatenazione del parametro di input *meshfile* con la stringa *"_ decompose.log"*: ad esempio se la mesh é contenuta nel file *"cube.tsoup"* allora il percorso del file di log sará *"cube.tsoup_ decompose.log"*.

Nella versione preliminare della libreria *OMSM*, i dati vengono memorizzati in un database locale: il percorso di questo file si puó ottenere dalla stringa *meshfile*, la quale contiene il percorso della mesh da decomporre. Come primo passo viene estratto il nome del file, il quale viene poi concatenato con la stringa *"_ tsoup.db"*. Se la mesh é contenuta nel file *"cube.tsoup"* allora

il risultato sará memorizzato nel file *"cube_ tsoup.db"*. In questo caso, il parametro ausiliario deve memorizzare il percorso della directory di ambiente del database: si consiglia di mantenere il database ed il suo ambiente su due dischi differenti, visto che ogni supporto di memorizzazione ha una probabilitá indipendente di fallimento, come abbiamo visto nella sezione 4.4.2.

La figura 7.6 mostra il codice del programma *Decompose*, scritto nel linguaggio *C++*.

```
#include "os.h"
#include "timerop.h"
#include "meshandler.h"
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;
using namespace omsm;

int main(int argc, char **argv) {

 MeshHandler* mh;
 OperationTimer *ot;
 string meshfile,optsfile;

 extractParameters(argc,argv,meshfile,optsfile);

 if((mh = new MeshHandler())==NULL) { ... }
 if((ot = new OperationTimer())==NULL) { ... }

 try { mh->decompose(meshfile,optsfile); }
 catch(AllocationErrorException aee) { ... }
 catch(MeshAnalysisErrorException maee) { ... }

 cout<<"\tThis program runs for: "<<ot->getTime()<<" seconds";
 delete mh;
 delete ot;
 return EXIT_SUCCESS; }
```

Figura 7.6: il codice *C++* del programma *Decompose*.

Come si puó notare dalla figura 7.6, la struttura del programma *Decompose* é molto simile a quella degli altri programmi, delegando la gestione dei formati supportati al componente *MeshHandler*. La funzione *extractParameters* si occupa di estrarre i parametri *meshfile* ed *optsfile* dalla lista degli argomenti forniti dalla shell del sistema operativo. Attualmente l'ope-

razione di decomposizione é definita solamente sul formato *TSOUP*, ma in futuro potrebbe essere ampliato l'insieme dei formati che supportano questa operazione: usando il componente *MeshHandler* si eredita in maniera completamente trasparente il supporto dei nuovi formati, come descritto nella sezione 7.2.1.

Puó essere interessante valutare l'efficienza del programma *Decompose* sui modelli geometrici che abbiamo descritto in precedenza: anche in questo caso abbiamo utilizzato il componente *OperationTimer* per valutare il tempo di esecuzione T_D e le misure sono state effettuate con la stessa metodologia utilizzata per gli altri programmi. In questo ambito presenteremo solamente i risultati ottenuti con la piattaforma *GNU/Linux*, viste le problematiche sollevate dall'utilizzo della libreria *OMSM* sulla piattaforma *Microsoft Windows*, giá riscontrate nell'analisi dei programmi *Check* e *ToSoup*.

Visto che il framework ci permette di variare l'indice spaziale utilizzato, possiamo verificare l'efficienza del meccanismo di decomposizione a seconda della struttura di indicizzazione. La tabella 7.9 mostra i tempi di esecuzione T_D del programma *Decompose* utilizzando le strutture dati *Octree* e *K–d tree*, descritte rispettivamente nei paragrafi 5.4 e 5.5: i tempi riportati sono espressi in secondi e, come si puó notare, sono estremamente elevati.

Modello	T_D **con *Octree***	T_D **con *K–d tree***
Cube	$0,55\ s$	$0,53\ s$
Gargoyle	$9100,85\ s$	$9085,24\ s$
Bunny	$9563,56\ s$	$9335,67\ s$
Caesar	$28123,45\ s$	$27617,23\ s$
Icosphere	$46234,37\ s$	$45055,23\ s$

Tabella 7.9: i tempi di esecuzione del programma *Decompose*, espressi in secondi, sulla piattaforma *GNU/Linux*, utilizzando i modelli descritti nel paragrafo 7.2.

É evidente come il comportamento di questi indici sia estremamente inefficiente: ogni nodo contiene un solo elemento e quindi il numero di nodi é pari al numero dei triangoli della mesh, che puó essere molto elevato. Inoltre, ricordando le proprietá del prototipo che stiamo utilizzando, si osserva che un cluster puó mantenere un solo nodo e quindi la quantitá di informazione trasferita sul supporto di memorizzazione impone un numero elevato di operazioni di I/O, le quali sono particolarmente inefficienti: nel capitolo 3 abbiamo analizzato gli effetti di questa situazione. Pertanto i risultati ottenuti concordano con quelli attesi.

Per queste ragioni puó essere interessante verificare l'efficienza del meccanismo di decomposizione con altre tipologie di indici spaziali come le strutture dati *Bucket PR Quadtree* e *Bucket PR K–d tree*, descritte nella sezione

5.6.2: data la loro definizione é lecito aspettarsi un comportamento migliore rispetto a quello degli indici *Octree* e *K–d tree*. A differenza di queste strutture dati che abbiamo appena analizzato, ogni nodo foglia puó memorizzare un certo numero di oggetti geometrici: tale valore é detto *capacitá* del nodo ed é stato calcolato secondo la regola descritta nella sezione 5.6.1, cercando di fissare l'altezza h dell'albero. Con questa regola si riduce certamente il numero di nodi dell'indice spaziale e questo fatto garantisce una migliore velocitá di trasferimento rispetto al caso precedente: nella sezione 3.4.2 abbiamo visto come l'occupazione spaziale e la velocitá di trasferimento su un disco fisso siano grandezze inversamente proporzionali. Nel nostro caso avviene una situazione simile: ogni operazione di I/O interessa un singolo cluster, il quale avrá un'occupazione spaziale maggiore rispetto all'utilizzo degli indici *K–d tree* e *Octree*. Inoltre il numero di operazioni di I/O richieste per il caricamento dell'indice é minore rispetto a quanto avviene nel caso precedente: secondo quanto descritto nella sezione 5.6.2, la primitiva di inserimento di un certo oggetto geometrico $\mathcal{O}$ é equivalente alla visita di un cammino dalla radice al nodo piú piccolo in grado di contenere $\mathcal{O}$ al suo interno. Grazie alla tecnica di suddivisione *sliding midpoint*, introdotta in [MA97], [MM99a], [MM99b] e [Man01], abbiamo fissato l'altezza h dell'albero e quindi dobbiamo visitare al piú h nodi. Vista la politica di suddivisione dei nodi, dobbiamo eseguire h operazioni di I/O per caricare i cluster corrispondenti. Dovendo inserire N oggetti geometrici allora il numero di operazioni di I/O é pari a $\mathcal{O}(hN)$: utilizzando l'indice spaziale *K–d tree* sono necessarie $\mathcal{O}(N^2)$ operazioni di I/O. Questa proprietá puó spiegare la maggior efficienza di queste strutture.

Le tabelle 7.10 e 7.11 mostrano i tempi di esecuzione T_D del programma *Decompose* utilizzando rispettivamente la struttura dati *Bucket PR Quadtree* e quella *Bucket PR K–d tree*: i risultati ottenuti concordano con quelli presentati in [MM99a] e [MM99b]. Dal nostro punto di vista non é interessante misurare il tempo di esecuzione in maniera precisa, quanto verificare il miglioramento delle prestazioni ottenute utilizzando indici spaziali di tipo *ibrido* rispetto a quelle *K–d tree* e *Octree*.

Modello	***Bucket PR Quadtree***	***Numero massimo dei livelli***
Cube	$0,45\ s$	1
Gargoyle	$1167,30\ s$	3
Bunny	$1380,45\ s$	3
Caesar	$2714,53\ s$	8
Icosphere	$5479,93\ s$	9

Tabella 7.10: i tempi di esecuzione del programma *Decompose*, espressi in secondi, utilizzando i modelli descritti nel paragrafo 7.2 e l'indice spaziale *Bucket PR Quadtree* e fissando il numero massimo dei livelli dell'indice.

Modello	***Bucket PR K–d tree***	***Numero massimo dei livelli***
Cube	$0,30\ s$	1
Gargoyle	$1080,36\ s$	3
Bunny	$1324,45\ s$	3
Caesar	$2578,32\ s$	8
Icosphere	$5157,33\ s$	9

Tabella 7.11: i tempi di esecuzione del programma *Decompose*, espressi in secondi, utilizzando i modelli descritti nel paragrafo 7.2 e l'indice spaziale *Bucket PR K–d tree* e fissando il numero massimo dei livelli dell'indice.

La discrepanza di funzionamento fra i due diversi indici spaziali é una banale conseguenza dell'equazione 5.3, che abbiamo introdotto nella sezione 5.6.1 per fissare la capacitá di un nodo. Questa equazione si basa sul numero di nodi di un albero completo, nel quale ogni nodo ha m figli. Ogni nodo della struttura dati *Bucket PR K–d tree* ha due figli mentre quello della struttura dati *Bucket PR Quadtree* ne ha otto: di conseguenza, mantenendo fissato il numero massimo di livelli, la versione completa della struttura *Bucket PR Quadtree* avrá un numero maggiore di nodi di capacitá inferiore rispetto a quella dei nodi della struttura *Bucket PR K–d tree.* Di conseguenza la velocitá di trasferimento sará inferiore. In realtá é importante notare che l'utilizzo di queste due strutture é certamente preferibile rispetto a quello degli indici *K–d tree* e *Octree*, come si puó facilmente dedurre dai tempi di esecuzione del programma *Decompose.* In questa versione preliminare del framework *OMSM* possiamo utilizzare solamente la politica *"singola"* per la suddivisione dei nodi di cluster: con questa scelta un cluster potrá contenere solamente un nodo. In realtá il framework *OMSM* puó supportare diverse politiche di clustering e quindi si potranno ottenere risultati differenti, a seconda delle scelte effettuate.

Per ulteriori approfondimenti sul funzionamento del programma *Decompose* rifarsi a [Can07c].

Capitolo 8

Conclusioni e sviluppi futuri

> Il pregiudizio umano, secondo cui ció che si conosce basta a comprendere tutti i misteri dell'universo, oppone resistenza alle teorie innovative. Nonostante ció, le nuove idee finiscono comunque per farsi strada e nel modo meno prevedibile. É l'aspetto appassionante di questo lavoro: le meraviglie inaspettate sono piú meravigliose di quelle attese.
>
> *Jayant Vishnu Narlikar*

Il lavoro di ricerca svolto in questa tesi si inquadra nella risoluzione del problema della semplificazione di un complesso simpliciale immerso in uno spazio metrico euclideo $\mathbb{E}^d$, con d generico: la dimensione di questa mesh puó eccedere quella della memoria primaria comunemente disponibile in un calcolatore pertanto le uniche tecniche adatte ai nostri scopi sono quelle che mantengono parzialmente la mesh in memoria secondaria, caricandone in maniera dinamica le porzioni volute in memoria primaria.

Il punto di partenza della nostra analisi é costituito dalla tecnica di semplificazione descritta in [CMRS03], la quale scompone una triangolazione immersa nello spazio metrico euclideo $\mathbb{E}^3$ attraverso l'indice spaziale *Octree*, introdotto in [FB74]. Ogni blocco Γ della suddivisione puó essere semplificato attraverso l'applicazione iterativa dell'operatore di collassamento degli spigoli, non modificando quelli sul contorno di Γ: si possono ottenere degli sgradevoli artefatti, ben visibili sul modello semplificato in quanto sono a risoluzione maggiore rispetto al resto. Per evitare questo problema, durante l'operazione di modifica su un certo blocco Γ vengono caricati i nodi adiacenti in modo da poter operare sugli spigoli appartenenti al contorno della porzione di mesh contenuta nell'ottante descritto dalla foglia sotto analisi. Una volta terminata l'analisi di una singola porzione, é possibile fonderla con quelle adiacenti e ripetere ricorsivamente il processo di semplificazione: lo scopo di questa tecnica é quello di ottenere una rappresentazione semplificata della mesh di partenza, memorizzata su un numero ridotto di blocchi e direttamente gestibile in memoria primaria.

Nella ricerca svolta in questa tesi abbiamo generalizzato questo algoritmo sia riguardo al tipo di indice spaziale utilizzato per la decomposizione della mesh e sia riguardo all'algoritmo di semplificazione iterativa dei blocchi della suddivisione. La tecnica descritta in [CMRS03] nasce per la gestione di triangolazioni, ma puó essere estesa anche al caso di griglie di tetraedri in maniera analoga: é ovvio che in questo caso bisogna utilizzare un qualsiasi algoritmo di semplificazione per tetraedralizzazioni, come quelli descritti in [dFD06]. Pertanto questo processo di semplificazione deve avvenire secondo queste due fasi, che devono essere eseguite in sequenza:

- la decomposizione della mesh con un indice spaziale in memoria secondaria: si puó utilizzare una qualsiasi struttura di indicizzazione, come quelle descritte in [Sam06];
- l'applicazione di un qualsiasi algoritmo di semplificazione iterativa sulla decomposizione a seconda del tipo di complesso simpliciale memorizzato: ad esempio possiamo utilizzare le tecniche descritte in [Gar99a], [Pri00] e [CMRS03] se stiamo utilizzando triangolazioni oppure quelle descritte in [dFD06] se dobbiamo gestire tetraedralizzazioni.

Quindi é ovvio che per poter decomporre una mesh, é necessaria un'architettura di memorizzazione, la quale permetta di poter gestire complessi simpliciali di differente dimensione e di poter variare in maniera intuitiva il tipo di indice spaziale da utilizzare. Attualmente non esistono framework per l'indicizzazione spaziale, liberamente utilizzabili, con le caratteristiche richieste, sebbene in letteratura ci siano stati dei tentativi come quelli descritti in [HNP95], [Kor99], [AI01a], [AI01b], [AI01c], [Ily02] e [AV05], i quali non sono adatti ai nostri scopi. Per queste ragioni abbiamo sviluppato il framework *OMSM* (dall'espressione inglese *Objects Management in Secondary Memory*) per la gestione di grosse quantitá di dati geometrici in memoria secondaria: questa architettura si caratterizza per l'elevato grado di modularitá e di flessibilitá. Nel paragrafo 8.1 ne riassumeremo le caratteristiche principali, delineandone alcune possibili estensioni.

Per dimostrare la fattibilitá di questa soluzione, é stata sviluppata una versione preliminare dell'architettura *OMSM* in grado di decomporre una griglia di triangoli, immersa nello spazio metrico euclideo $\mathbb{E}^3$: questo prototipo supporta gli indici spaziali *Octree*, *K–d tree*, *Bucket PR Quadtree* e *Bucket PR K–d tree*. Bisogna ricordare che la versione realizzata é un prototipo iniziale quindi potrá essere ulteriormente esteso in futuro: questa soluzione potrá essere utilizzata per la gestione di una griglia di triangoli, mantenuta in memoria secondaria, utilizzando un qualsiasi algoritmo di semplificazione come ad esempio quelli descritti in [Gar99a], [Pri00] e [CMRS03].

Come si puó intuire, l'approccio risolutivo scelto é molto generale e puó essere facilmente esteso alle griglie di tetraedri, le quali stanno divenendo sempre piú importanti nell'ambito della visualizzazione scientifica. Inoltre

potrebbe essere applicato alla creazione di un modello multi–risoluzione in memoria secondaria a partire da griglie di triangoli o di tetraedri, come vedremo nel paragrafo 8.2. Ricordiamo che un modello multi–risoluzione fornisce una gamma di rappresentazioni di un dato oggetto a differenti livelli di dettaglio ed é un argomento di grande interesse nella ricerca attuale, come descritto in [DdFM+06].

8.1 Un framework utile a molti scopi

Il framework *OMSM* é stato introdotto in questa tesi per gestire grandi quantitá di dati geometrici, mantenuti in memoria secondaria ed é in grado di memorizzare complessi simpliciali a prescindere dalla loro dimensione e dallo spazio metrico euclideo nel quale sono immersi grazie all'utilizzo del concetto di *punto rappresentativo* di un dato simplesso. In questo modo é necessario gestire solamente punti multi–dimensionali, astraendo dal tipo di simplesso da memorizzare.

A differenza di molti sistemi per la gestione di dati geometrici, il framework *OMSM* puó integrare fra loro, in maniera dinamica, le diverse tecniche sviluppate in letteratura per la gestione dei seguenti aspetti:

- l'utilizzo di una struttura ausiliaria di accesso ad albero, detta *indice spaziale*, per facilitare le operazioni di aggiornamento e di interrogazione sui dati: alcuni esempi di indici sono descritti in [Sam06];
- la suddivisione dei nodi dell'indice spaziale in *cluster* secondo una certa politica in modo da minimizzare il numero di accessi alla memoria secondaria, la quale é piú lenta di quella primaria: per *cluster* si intende un gruppo di nodi, scelti in base ad un certo criterio, i quali possono essere considerati un'unica entitá;
- la gestione dinamica dei cluster in memoria secondaria: questo aspetto puó essere gestito attraverso varie tecniche di memorizzazione, le quali dipendono anche dalla distribuzione fisica dei dati.

La sua struttura é modulare ed ognuno dei tre aspetti appena delineati viene gestito da uno specifico componente senza assumere l'utilizzo di una particolare tecnica: in questo modo il funzionamento di un livello puó essere considerato indipendente da quello degli altri ed é possibile modificarne il funzionamento. Per dimostrare un possibile utilizzo di questo framework, ne é stata implementata una versione preliminare, nella quale é possibile utilizzare una vasta gamma di scelte possibili. Vengono supportati gli indici spaziali k–d tree, octree, PR k–d trie di tipo ibrido e PR quadtrie di tipo ibrido: per approfondimenti su tali strutture rifarsi a [Sam06]. La politica di suddivisione in cluster utilizzata é quella secondo la quale un cluster puó contenere un solo nodo dell'indice, mentre i cluster possono essere memorizzati

in un database locale, utilizzando il sistema software *Oracle Berkeley DB*, un esempio molto noto di *DBMS* di tipo embedded, introdotto in [Bdb06d]. Questa versione preliminare é un prototipo: data la sua natura modulare ed espandibile, il framework *OMSM* favorisce future espansioni. É naturale proporre l'implementazione di nuove tecniche nei vari livelli in maniera indipendente, ampliando le possibilitá di utilizzo di questa architettura di memorizzazione a seconda delle varie esigenze. Ad esempio é possibile introdurre nuovi indici spaziali o diverse politiche di clustering: un'estensione importante puó essere la gestione di dati remoti o distribuiti, introducendo tecniche di compressione nella rappresentazione dei dati per minimizzare la quantitá di banda utilizzata. Il funzionamento di questo framework é stato ampiamente discusso nel paragrafo 6.5.

8.2 Una tecnica utile a molti scopi

Il framework *OMSM*, realizzato in questa tesi, supporta la creazione di tecniche di semplificazione di complessi simpliciali mantenuti in memoria secondaria: queste tecniche possono essere utilizzate in diversi ambiti della modellazione geometrica.

L'utilizzo dei complessi simpliciali riscuote grande interesse in svariati domini applicativi, ad esempio nella gestione di terreni per sistemi informativi geometrici o nella visualizzazione di dati scientifici: attualmente stanno assumendo grande importanza le griglie di tetraedri, le quali vengono utilizzate per modellare campi scalari tridimensionali o volumi. Sfruttando le funzionalitá offerte dal framework *OMSM*, é possibile decomporre questi modelli indipendentemente dalla dimensione dello spazio metrico euclideo nel quale sono immersi visto che il processo di indicizzazione si basa sul concetto di *punto rappresentativo* di una certo simplesso, il quale ne descrive le caratteristiche piú importanti. Con questa tecnica ci si riduce a gestire solamente punti multi–dimensionali, utilizzando un qualsiasi indice spaziale. Una volta decomposto il complesso simpliciale sotto esame, é possibile generalizzare l'algoritmo descritto in [CMRS03] applicando un qualsiasi algoritmo di semplificazione iterativa a seconda del tipo di mesh memorizzata: ad esempio possiamo utilizzare le tecniche descritte in [Gar99a], [Pri00] e [CMRS03] se stiamo utilizzando triangolazioni oppure quelle descritte in [DdF02] e [dFD06] se dobbiamo gestire tetraedralizzazioni.

Con questo approccio é possibile ridurre il livello di dettaglio di un certo complesso simpliciale, la cui dimensione potrebbe eccedere la quantitá di memoria *RAM* disponibile in un calcolatore. Una descrizione accurata di un certo oggetto richiede un numero elevato di simplessi e comporta alti costi di elaborazione: riducendo la risoluzione del complesso simpliciale in input, possiamo ridurne l'occupazione spaziale e quindi é possibile mantenerlo direttamente in *RAM*.

L'operazione di semplificazione é importante anche perché permette la creazione di un modello multi–risoluzione di un certo oggetto $\mathcal{O}$: questa tecnica consente di ottenere una serie di rappresentazioni di $\mathcal{O}$ a diverso livello di dettaglio, variando la risoluzione del modello a seconda dei requisiti posti dall'utente. Questo argomento é di grande interesse nella ricerca attuale: in [Gar99b] e [DdFM+06] ne vengono delineate le principali proprietá e le prospettive di ricerca. Vediamo ora in che modo sia possibile ottenere un modello con tali caratteristiche attraverso l'operazione di semplificazione.

La maggior parte dei modelli multi–risoluzione si differenzia per l'operazione di modifica da applicare al complesso simpliciale: per questo motivo é stato sviluppato il modello *Multi–Tesselazione*, introdotto in [Pup96] e [Pup98]. In letteratura é noto come modello *MT*, dall'inglese *Multi–Tesselation*: é sviluppato dal *Gruppo di Ricerca in Modellazione Geometrica e Computer–Grafica* del Dipartimento di Informatica e Scienze dell'Informazione (noto come *D.I.S.I.*), afferente all'Universitá degli Studi di Genova. Il modello *MT* nasce per la gestione delle triangolazioni: in seguito é stato esteso per poter gestire i complessi simpliciali in [dFMP99a] e [dFMP99b], quelli cellulari in [Mag99], le griglie di tetraedri in [DdF02] ed i dominii non manifold in [dFMPS04]. Questo modello é indipendente dall'operatore di modifica pertanto gli altri modelli multi–risoluzione possono essere visti come casi particolari di quello *MT*, come descritto in [Mag99].

Secondo quanto descritto in [DdFM+06], per costruire un modello multi–risoluzione di un oggetto $\mathcal{O}$ sono necessari i seguenti elementi:

- una *mesh di base*, la quale contiene il complesso simpliciale che approssima inizialmente l'oggetto $\mathcal{O}$;
- un'insieme di *modifiche*, le quali aggiornano localmente la mesh;
- una *relazione di dipendenza* fra le modifiche apportate alla mesh, codificata attraverso un *grafo diretto aciclico*, noto in letteratura come *DAG* (dall'inglese *Direct Acyclic Graph*). Le diverse rappresentazioni di $\mathcal{O}$ verranno ottenute visitando il *DAG*.

Quindi l'operatore di semplificazione scelto avrá banalmente il compito di aggiornare la mesh: la soluzione che proponiamo é particolarmente indicata per codificare in maniera efficiente la relazione di dipendenza fra le modifiche apportate. Nella semplificazione iterativa le varie rappresentazioni dipendono da quelle precedenti e quindi é possibile ottenere in maniera automatica una relazione d'ordine fra le modifiche applicate ad ogni passo, come dimostrato in [DdFPS06]. Questa tecnica di costruzione assume che il complesso simpliciale in input possa essere completamente contenuto in memoria primaria e quindi non é adatta alla gestione dei modelli trattati in questa tesi: per ovviare a questo problema sono state sviluppate alcune tecniche per creare un modello multi–risoluzione in memoria secondaria, come quelle descritte

in [SG05] ed in [DdFPS06]. Queste tecniche gestiscono solamente griglie di triangoli o modelli di terreni: allo stato attuale non esistono algoritmi per generare modelli multi–risoluzione a partire da griglie di tetraedri mantenute in memoria secondaria. Questa mancanza é un problema importante visto che questo tipo di rappresentazioni stanno divenendo molto importanti nell'ambito della visualizzazione scientifica in quanto permettono la gestione di campi scalari multi–dimensionali.

Il nostro approccio permette l'utilizzo di diverse tecniche di semplificazione e quindi é possibile implementare una piattaforma capace di supportare diversi modelli multi–risoluzione, i quali si differenziano a seconda dell'operatore di modifica utilizzato. In letteratura ne sono stati sviluppati molti: ad esempio in [EDD+95], [Hop96], [PH97], [RCHQ98], [RL00] e [HG01] ne vengono descritti alcuni. La maggior parte di questi modelli vengono costruiti a partire da griglie di triangoli: inoltre in [dFD06] vengono passate in rassegna alcune fra le tecniche piú importanti nella modellazione multi–resoluzione delle griglie di tetraedri. Visto che il nostro framework é capace di memorizzare complessi simpliciali a prescindere dalla dimensione dello spazio metrico euclideo nel quale sono immersi, siamo in grado di gestire modelli multi–risoluzione a partire sia da triangolazioni immerse negli spazi metrici euclidei $\mathbb{E}^2$ ed $\mathbb{E}^3$ e sia da tetraedralizzazioni.

Dato che il modello *MT* é indipendente dall'operatore di modifica utilizzato, possiamo utilizzare il framework *OMSM* per costruire un modello *Multi–Tesselazione* in memoria secondaria: il raggiungimento di questo obiettivo é molto importante visto che questo modello non supporta ancora la costruzione della relazione di dipendenza fra le modifiche apportate ad un complesso simpliciale mantenuto in memoria secondaria, a prescindere dalla dimensione dello spazio metrico euclideo nel quale é immerso. In realtá sono stati ottenuti dei risultati intermedi che potrebbero essere utilizzati per il raggiungimento di questo obiettivo.

In [MB98] e [MB00] viene descritta una versione modulare di un modello *MT*, adatta alla gestione di terreni: viene mantenuta in memoria primaria una rappresentazione del terreno a bassa risoluzione mentre quelle a risoluzione piú accurata sono caricate dinamicamente dal supporto di memorizzazione. In [Ser06] e [SMdF06] viene descritta una versione client/server di un modello *MT*, il quale viene suddiviso in moduli secondo un qualche principio, a prescindere dalla dimensione dello spazio metrico euclideo nel quale é immerso il complesso simpliciale descritto dal modello *MT*. I moduli verranno memorizzati su un server remoto, dal quale il client puó scaricare le porzioni del modello di cui ha bisogno, utilizzando un'organizzazione simile a quella introdotta nel framework *OMSM*. Questa versione client/server della *Multi–Tesselation* sará utilizzata nell'archivio digitale di modelli multi–risoluzione della Rete Europea di Eccellenza *AIM@SHAPE*: per maggiori approfondimenti rifarsi a [Aim04]. In [Dan05] ed in [Sob06] vengono esposti una serie di risultati intermedi che possono essere di grande aiuto nella nostra analisi,

riguardanti la memorizzazione di un modello MT in maniera efficiente.

In base a queste considerazioni, l'approccio risolutivo per la costruzione di un modello multi–risoluzione in memoria secondaria potrebbe consistere nell'utilizzo di una tecnica di semplificazione iterativa del modello in input, avendo cura di mantenere su un supporto di memorizzazione le dipendenze fra gli aggiornamenti applicati al complesso simpliciale in modo da poterle riorganizzare in maniera efficiente alla fine del processo di modifica. L'utilizzo del framework *OMSM* garantisce una soluzione assolutamente generale: come abbiamo giá visto, possiamo indicizzare un qualsiasi complesso simpliciale (sia una triangolazione e sia una tetraedralizzazione) e questa proprietá rende la nostra soluzione adatta alla gestione di svariati modelli geometrici. Inoltre é possibile utilizzare un qualsiasi algoritmo di semplificazione iterativa, a seconda del tipo di complesso simpliciale che stiamo gestendo: in questo modo é possibile ottenere diversi tipi di modelli. In realtá vogliamo costruire un modello multi–risoluzione quindi siamo interessati alla codifica della relazione di dipendenza fra modifiche: una possibile soluzione sarebbe quella di memorizzare le dipendenze fra gli aggiornamenti secondo il formato definito in [DM07a] mentre le modifiche vengono applicate. L'ultimo passo da compiere sarebbe quello di costruire il grafo delle dipendenze in memoria secondaria, a partire da quelle memorizzate come risultato dei passi precedenti. In questo modo si potrá costruire una versione di un modello MT in memoria secondaria senza perdere in generalitá vista la sua indipendenza dall'operatore di semplificazione: di conseguenza sará possibile ottenere automaticamente diversi modelli multi–risoluzione sia di griglie di triangoli immersi in $\mathbb{E}^2$ o in $\mathbb{E}^3$ e sia griglie di tetraedri.

Bibliografia

[AAPV01] P. Agarwal, L. Arge, O. Procopiuc, and J. Vitter. A Framework for Index Bulk Loading and Dynamization. *Lecture Notes in Computer Science*, Volume 2076, 2001.

[ABA06] *The Allen Brain Atlas Project*, 2006. Sviluppato da *The Allen Institute for Brain Science* e disponibile sul Web all'indirizzo *http://www.brain–map.org*.

[ABH+97] L. Arge, R. Barve, D. Hutchinson, O. Procopiuc, L. Toma, E. Vengroff, and R. Wickeremesinghe. *TPIE: a Transparent and Parallel I/O Environment*, 1997. Sviluppato presso il *Computer Science Department, Duke University* e disponibile sul Web all'indirizzo *http://www.cs.duke.edu/TPIE*.

[ABH+02] L. Arge, R. Barve, D. Hutchinson, O. Procopiuc, L. Toma, E. Vengroff, and R. Wickeremesinghe. *TPIE: the User Manual and Reference*, 2002.

[Ada90] J. Adameck. *Abstract and concrete categories.* John Wiley & Sons Editor, 1990.

[AGR01] N. Amato, M. Goodrich, and E. Ramos. A Randomized Algorithm for Triangulating a Simple Polygon in Linear Time. *Discrete and Computational Geometry*, Volume 26, 2001.

[AGS97] R. Agrawal, A. Gupta, and S. Sarawagi. *Modeling the Multidimensional Databases.* In *Proceedings of the* 13th *International Conference on Data Engineering*, 1997.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison–Wesley Editor, 1995.

[AI01a] W. Aref and I. Ilyas. SP–GiST: An Extensible Database Index for Supporting Space Partitioning Trees. *Journal of Intelligent Information Systems (JIIS)*, 2001.

[AI01b] W. Aref and I. Ilyas. *A Framework for Supporting the Class of Space Partitioning Trees.* Technical Report TR–01–002, Department of Computer Science, Purdue University, 2001.

[AI01c] W. Aref and I. Ilyas. *An extensible Index for the Spatial Databases.* In *Proceedings of the* 13th *International Conference on Scientific and Statistical Database Management*, 2001.

[Aim04] The *AIM@SHAPE Project: Advanced and Innovative Models And Tools for the development of Semantic–based systems for the Handling, Acquiring and Processing knowledge Embedded in multidimensional digital objects*, 2004. Disponibile sul Web all'indirizzo *http://www.aimatshape.net.*

[And01] G. Andrews. *The Concurrent Programming Principles and Practice.* Benjamin Cummings Publisher, 2001.

[AS83] D. Abel and J. Smith. A Data Structure and Algorithm based on a Linear Key for a Rectangle Retrieval Problem. *Computer Vision, Graphics and Image Processing*, Volume 24, 1983.

[AS94] P. Agarwal and S. Suri. *Surface Approximation and Geometric Partitions.* In *Proceedings of the* 5th *ACM–SIAM Symposium about Discrete Algorithms*, 1994.

[ASI05] H. Ali, A. Saad, and A. Ismail. The PN–tree: a parallel and distributed multidimensional index. *Distributed Parallel Databases*, Volume 17, 2005.

[Att06] M. Attene. *JMeshLib, a C++ API to manage manifold triangle meshes*, 2006. Disponibile sul Web all'indirizzo *http://jmeshlib.sourceforge.net.*

[AV88] A. Agrawal and J. Vitter. The I/O Complexity of Sorting and Related Problems. *Communications of ACM*, Volume 31, 1988.

[AV05] W. Aref and J. Vitter. *The SP–GiST Library: a General Index Framework for Space Partitioning Trees*, 2005. Department of Computer Science, Purdue Uuniversity. Disponibile sul Web all'indirizzo *http://www.cs.purdue.edu/spgist.*

[AVL62] G. Adelson-Velskii and E. Landis. An Algorithm for the Organization of Information. *Doklag Akademi Nauk SSSR*, Volume 146, 1962.

[AWC05] S. Anastasiadis, R. Wickremesinghe, and J. Chase. *LERNA: an Active Storage Framework for Flexible Data Access and Management.* In *Proceedings of the* 14th *IEEE International Symposium on High Performance Distributed Computing*, 2005.

[AYCD98] S. Amer-Yahia, S. Cluet, and C. Delobel. *Bulk Loading Techniques for Object Databases and an Application to Relational Data.* In *Proceedings of the* 24th *International Conference on Very Large Databases*, 1998.

[BA82] M. Ben-Ari. *Principles of Concurrent Programming.* Prentice Hall International, 1982.

[Bab88] H. Babbage. *The Analytical Engine*, 1888. Disponibile sul Web all'indirizzo *http://www.fourmilab.ch/babbage/.*

[Bar04] A. Barabási. *Link: la scienza delle reti.* Einaudi Editore, 2004.

[Bas96] W. Basil. *Developing Products and Their Rhetoric from a Single Hierarchical Model.* In *Proceedings of the Annual Conference of the Society for Technical Communication*, 1996.

[Bay72] R. Bayer. Symmetric Binary B–Trees: Data Structures and Manteinance Algorithms. *Acta Informatica*, Volume 1, 1972.

[BBC97] A. Belussi, E. Bertino, and B. Catania. *Manipulating Spatial Data in Constraint Databases.* In *Proceedings of the* 5th *International Symposium on Spatial Databases*, 1997.

[Bdb06a] *Oracle Berkeley DB – C++ API Version 4.5*, 2006. Sviluppato da *Oracle Corporation* e disponibile sul Web all'indirizzo *http://www.oracle.com/technology/documentation/berkeley–db/db/api_ cxx/frame.html.*

[Bdb06b] *Getting Started with the Oracle Berkeley DB Replicated Applications for C++*, 2006. Sviluppato da *Oracle Corporation* e disponibile sul Web all'indirizzo *http://www.oracle.com/technology/-documentation/berkeley–db/db/index.html.*

[Bdb06c] *Getting Started with the Oracle Berkeley DB Transactions Processing for C++*, 2006. Sviluppato da *Oracle Corporation* e disponibile sul Web all'indirizzo *http://www.oracle.com/technology/documentation/berkeley–db/db/index.html.*

[Bdb06d] *Berkeley DB: the Oracle Embedded Database*, Release 4.5, 2006. Sviluppato da *Oracle Corporation* e disponibile sul Web all'indirizzo *http://www.oracle.com/database/berkeley–db/index.html.*

[BDE98] G. Barequet, M. Dickerson, and D. Eppstein. On Triangulating the 3–Dimensional Polygons. *Computational Geometry Theory and Applications*, Volume 10, 1998.

[BDF+99] E. Boman, D. Devine, A. Fisk, R. Heaphy, B. Hendrickson, C. Vaughan, U. Catalyurek, and D. Bozdag. *The Zoltan Library – Data Management Services for Parallel Applications*, 1999. Sviluppato da *Sandia National Laboratories* e disponibile sul Web all'indirizzo *http://www.cs.sandia.gov/Zoltan*.

[BDF+06a] E. Boman, D. Devine, A. Fisk, R. Heaphy, B. Hendrickson, C. Vaughan, U. Catalyurek, D . Bozdag, and W. Mitchell. *Zoltan 2.0 - Data Management Services for Parallel Applications - The Developer's Guide*. Technical Report SAND2006–2955, Sandia National Laboratories, 2006.

[BDF+06b] E. Boman, D. Devine, A. Fisk, R. Heaphy, B. Hendrickson, C. Vaughan, U. Catalyurek, D . Bozdag, and W. Mitchell. *Zoltan 2.0 - Data Management Services for Parallel Applications - The User's Guide*. Technical Report SAND2006–2958, Sandia National Laboratories, 2006.

[BdPLS90] C. Batini, G. de Petra, M. Lenzerini, and G. Santucci. *La Progettazione Concettuale dei Dati*. Franco Angeli Editore, 1990.

[Ben75] J. Bentley. Multidimensional Binary Search Trees used for Associative Searching. *Communications of the ACM*, 1975.

[BG03] J. Belleson and E. Grochowski. *The Era of Giant Magnetoresistive heads*: a Technical report of Hitachi Global Storage Technologies and IBM OEM, 2003. Disponibile sul Web all'indirizzo *http://www.hitachigst.com/hdd/technolo/gmr/gmr.htm*.

[BGK04] E. Beinot, A. Godfrind, and R. Kothuri. *Pro Oracle Spatial*. Apress Editor, 2004.

[BHS02] B. Buchmann, H. Höpfner, and K. Sattler. *An Extensible Storage Manager for Mobile DBMS*. In *Proceedings of the* 5th *International Baltic Conference on DBs and ISs*, 2002.

[BKSS90] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. *The* R^**–tree: An Efficient and Robust Access Method for Points and Rectangles*. In *Proceedings of the ACM–SIGMOD International Conference on Management of Data*, 1990.

[BM72] R. Bayer and E. McCreight. Organization and Mantenance of Large Ordered Indices. *Acta Informatica*, Volume 1, 1972.

[BM76] J. Bondy and S. Murty. *Graph Theory with Applications*. North–Holland Editor, 1976.

[BNM03] P. Bozanis, A. Nanopoulos, and Y. Manolopoulos. LR-tree: a Logarithmic Decomposable Spatial Index Method. *Computer Journal*, Volume 46, 2003.

[BOH+92] A. Buchmann, M. Ozsu, M. Hornick, D. Georgakopoulos, and F. Manola. A Transaction Model for the Active Distributed Object Systems. In *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publisher, 1992.

[Boo07] G. Booch. *The Object–oriented Analysis and Design with Applications*. Addison–Wesley Editor, 2007.

[Bow01] I. Bowman. *Hybrid Shipping Architectures: a Survey*. Technical report, University of Waterloo, 2001.

[BP95] A. Biliris and E. Panagos. *An High Performance Configurable Storage Manager*. In *Proceedings of the* 11th *International Conference on Data Engineering*, 1995.

[BS06] F. Brabec and H. Samet. *Spatial Index Demos Project*, 2006. Disponibile sul Web all'indirizzo *http://donar.umiacs.umd.edu/-quadtree/index.html*.

[BSW77] J. Bentley, D. Stanat, and E. Williams. The Complexity of Finding Fixed–radius Nearest Neighbours. *Information Processing Letters*, Volume 6, 1977.

[BW02] M. Berr and C. Wells. *The Toposes, Triples and Theories*. Springer Verlag Publisher, 2002.

[Can07a] D. Canino. *The OMSM Library Reference Manual*, 2007.

[Can07b] D. Canino. *The OMSM Library Tutorial*, 2007. In preparazione.

[Can07c] D. Canino. *The OMSM Programs Reference Manual*, 2007.

[CBD+07] U. Catalyurek, G. Boman, D. Devine, D. Bozdag, R. Heaphy, and L. Riesen. *Hypergraph–based Dynamic Load Balancing for Adaptive Scientific Computations*. In *Proceedings of the* 21st *International Parallel and Distributed Processing Symposium*, 2007.

[CES71] E. Coffman, E. Elphick, and A. Shoshani. System Deadlocks. *Computer Surveys*, Volume 3, 1971.

[CFB00] S. Ceri, P. Fraternali, and A. Bangio. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks*, Volume 33, 2000.

[CFB+03] S. Ceri, P. Fraternali, A. Bangio, M. Brambilla, S. Comai, and M. Matera. *Designing Data–Intensive Web Applications.* Morgan Kaufmann Publisher, 2003.

[CG91] S. Chen and D. Gordon. Front–to–Back Display of BSP–Trees. *IEEE Computer Graphics and Algorithms*, 1991.

[CGG+04] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. *Adaptive TetraPuzzles: Efficient Out–of–Core Construction and Visualization of Gigantic Multiresolution Polygonal Models.* In *Proceedings of the ACM SIGGRAPH Conference*, 2004.

[CGK+88] T. Chen, G. Gibson, R. Katz, D. Patterson, and M. Schultz. *Two papers on RAIDs.* Technical Report UCB–CSD 88–479, Computer Science Department, University of California, 1988.

[CGP03] S. Chandran, A. Gupta, and A. Patgawkar. *A Fast Algorithm to Display Octrees.* Technical report, Indian Institute of Technology, 2003.

[CH96] T. Cormen and M. Hirschl. *Early Experiences in Evaluating the Parallel Disk Model with the ViC Implementation.* Technical Report PCS–TR96–293, Computer Science Department, Darmouth College, 1996.

[Cha91] B. Chazelle. Triangulating a Simple Polygon in Linear Time. *Discrete and Computational Geometry*, Volume 6, 1991.

[Che76] P. Chen. The Entity–Relationship Model: toward a Unified View of Data. *ACM Transactions on Database Systems*, Volume 1, 1976.

[CK95] P. Callahan and R. Kosaraju. A Decomposition of Multidimensional Point Sets with Applications to K–Nearest–Neighbors and n–Body Potential Fields. *Journal of the ACM*, Volume 42, 1995.

[CKS02] W. Correa, J. Klosowski, and C. Silva. *iWALK: Interactive Out–of–Core Rendering of Large Models.* Technical Report TR–653–02, Princeton University, 2002.

[CLC03] Y. Chang, C. Liao, and H. Chen. NA–trees: a Dynamic Index for Spatial Data. *Journal of Information Science and Engineering*, Volume 19, 2003.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* Mc Graw–Hill Editor, 1990.

[CMRS03] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External Memory Management and Simplification of Huge Meshes. *IEEE Transactions on Visualization and Computer Graphics*, Volume 9, 2003.

[CMS98] P. Cignoni, C. Montani, and R. Scopigno. A comparison of the mesh simplification algorithms. *Computer and Graphics*, Volume 22, 1998.

[CMZ04a] E. Cecchet, J. Marguerite, and W. Zwoenepoel. *C–JDBC: Flexible Database Clustering Middleware.* In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[CMZ04b] E. Cecchet, J. Marguerite, and W. Zwoenepoel. *C-JDBC: Flexible Clustered JDBC*, 2004. Disponibile sul Web all'indirizzo *http://c–jdbc.objectweb.org.*

[Cod85] E. Codd. Is your DBMS Really Relational? *Computer World*, 1985.

[Coh81] D. Cohen. On Holy Wars and a Plea for Peace. *IEEE Computer*, Volume 14, 1981.

[Col06] M. Colton. Is there a Database in your Future? *The Embedded Technology Journal*, 2006.

[Com79] D. Comer. The Ubiquitous B–Tree. *ACM Computing Surveys*, Volume 11, 1979.

[Cor91] *CORBA – the Common Object Request Broker Architecture*, 1991. Disponibile sul Web all'indirizzo *http://www.corba.org.*

[CPK+05] J. Chuugani, B. Purnomo, S. Krishnan, J. Cohen, S. Venkatasubranian, D. Johnson, and S. Kumar. vLOD: High–Fidelity Walkthrough of Large Virtual Environment. *IEEE Transactions on Visualization and Computer Graphics*, Volume 11, 2005.

[CPP00] *The C++ Resources Network*, 2000. Disponibile sul Web all'indirizzo *http://www.cplusplus.com.*

[CPZ97] P. Ciaccia, M. Patella, and P. Zezula. *M–tree: an Efficient Access Method for Similarity Search in Metric Spaces.* In *Proceedings of the* 23rd *International Conference on Very Large Data Bases*, 1997.

[CR71] R. Courant and H. Robbins. *Che Cos'é la Matematica?* Universale Bollati–Boringhieri Editore, 1971.

[CRZP97] P. Ciaccia, F. Rabitti, P. Zezula, and M. Patella. *The M–tree Project: an Access Method for Similarity Search*, 1997. Disponibile sul Web all'indirizzo *http://www–db.deis.unibo.it/Mtree*.

[CW00] S. Chandhuri and G. Weikum. Rethinking Database System Architecture: towards a Self–timing RISC–style Database System. *The VLDB Journal*, 2000.

[Dan05] E. Danovaro. *Multi–resolution Modeling of Discrete Scalar Fields*. PhD thesis, DISI, Universitá degli Studi di Genova, 2005.

[DBH+02] D. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. The Zoltan Data Management Services for Parallel Dynamic Applications. *Computing in Science and Engineering*, 2002.

[DBH+06] D. Devine, G. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek. *The Parallel Hypergraph Partitioning for Scientific Computing*. In *Proceedings of the* 20th *International Parallel and Distributed Processing Symposium*, 2006.

[DdF02] E. Danovaro and L. de Floriani. Half–Edge Multi–Tesselation. In *Proceedings of the* 1st *International Symposium on 3D Data Processing, Visualization and Transmission*, 2002.

[DdFM+06] E. Danovaro, L. de Floriani, P. Magillo, E. Puppo, and D. Sobrero. Level–of–Detail for Data Analysis and Exploration: a Historical Overview and some new Perspectives. *Computer Graphics*, Volume 30, 2006.

[DdFPS05] E. Danovaro, L. de Floriani, E. Puppo, and H. Samet. *Multi–resolution Out–Of–Core Modeling of Terrain and Geological Data*. In *Proceedings of the* 13th *International ACM Symposium on Advances in Geographic Information Systems*, 2005.

[DdFPS06] E. Danovaro, L. de Floriani, E. Puppo, and H. Samet. *Out–of–Core Multi–resolution Terrain Modeling*. In *Modelling and Management of Geographical Data over Distributed Architectures*. Springer–Verlag Publisher, 2006.

[dFD06] L. de Floriani and E. Danovaro. *Generating, Representing and Querying Level–of–Detail Tetrahedral Meshes*. In *Proceedings of Dagsthul Seminar on Scientific Visualization: Extracting Information and Knowledge from Scientific Data Sets*. Springer–Verlag Publisher, 2006.

[dFGH04] L. de Floriani, D. Greenfieldboyce, and A. Hui. *A data structure for non–manifold simplicial d–complexes.* In *Proceedings of the* 2nd *Eurographics Symposium on Geometry Processing*, 2004.

[dFH03] L. de Floriani and A. Hui. *A Scalable Data Structure for the 3–dimensional Non–Manifold Objects.* In *Proceedings of the* 1st *Eurographics Symposium on Geometry Processing*, 2003.

[dFH05a] L. de Floriani and A. Hui. *Data Structures for Simplicial Complexes: an Analysis and Comparison.* In *Proceedings of the* 3th *Eurographics Symposium on Geometry Processing*, 2005.

[dFH05b] L. de Floriani and A. Hui. *Representing Non–Manifold Shapes in Arbitrary Dimensions.* In *Proceedings of the Israel–Korea Binational Conference on New Technologies and Visualization Methods for Product Development on Design and Reverse Engineering*, 2005.

[dFMP99a] L. de Floriani, P. Magillo, and E. Puppo. *Data Structures for Simplicial Multi–Complexes. Advances in Spatial Databases*, 1651, 1999.

[dFMP99b] L. de Floriani, P. Magillo, and E. Puppo. *Multiresolution Representation of Shapes based on Cell Complexes. Discrete Geometry for Computer Imagery*, Volume 1568, 1999.

[dFMP00] L. de Floriani, P. Magillo, and E. Puppo. VARIANT: A System for Terrain Modeling at Variable Resolution. *Geoinformatica*, 2000.

[dFMPS04] L. de Floriani, P. Magillo, E. Puppo, and D. Sobrero. A Multi–Resolution Topological Representation for Non–Manifold Meshes. *Computer–Aided Design Journal*, Volume 36, 2004.

[dFPM99] L. de Floriani, E. Puppo, and P. Magillo. *Applications of Computational Geometry to Geographical Iinformation Systems.* In *Handbook of Computational Geometry.* Elsevier Science, 1999.

[Die05] R. Diestel. *Graph Theory.* Springer Verlag Publisher, 2005.

[Dis97] B. Discoe. *The Virtual Terrain Project*, 1997. Disponibile sul Web all'indirizzo *http://www.vterrain.org.*

[DK03] M. Dogson and S. Kristensen. Hausdorff Dimension and Diophantine Approximation. *Fractals Geometry and Applications: a Jubilee of Benoit Mandelbrot*, 2003.

[DM07a] E. Danovaro and P. Magillo. *Modular MT – from Simplification to Computation of Dependencies*, 2007.

[dM07b] A. di Marco. *The Geometry of Commodity Hard–Disks.* Technical Report DISI–TR–07–05, DISI, Universitá degli Studi di Genova, 2007.

[DN65] R. Doley and G. Neumann. *The General–Purpose Filesystem for a Secondary Storage*, 1965. Disponibile sul Web all'indirizzo *http://www.multicians.org/papers.html.*

[DRSS96] A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan. *Clustering Techniques for minimizing External Path Length.* In *Proceedings of the* 22nd *International Conference on Very Large Databases*, 1996.

[DWS+97] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, and M. Mineev-Weinstein. *ROAMing Terrain: Real–Time Optimally Adapting Meshes.* In *Proceedings of the IEEE Visualization Conference*, 1997.

[EA03] B. Eckel and C. Allison. *Thinking in C++ – Practical Programming*, 2003. Disponibile sul Web all'indirizzo *http://mindview.-net/Books/TICPP/ThinkingInCPP2e.html.*

[Eck00] B. Eckel. *Thinking in C++ – Introduction to Standard*, 2000. Disponibile sul Web all'indirizzo *http://mindview.net/Books/-TICPP/ThinkingInCPP2e.html.*

[ECLT76] K. Eswaran, J. Cray, J. Lorie, and I. Traigan. The Notions of Consistency and Predicate Locks in a Database System. *Communications of ACM*, Volume 19, 1976.

[EDD+95] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution Analysis of Arbitrary Meshes. *Computer Graphics*, Volume 29, 1995.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer–Verlag Publisher, 1987.

[EEA05] M. Eltabakh, R. Eltarras, and W. Aref. *To Trie or Not to Trie? Realizing Space–partitioning Trees inside PostgreSQL: Challenges, Experiences and Performance.* Technical Report CSD–TR–05–008, Department of Computer Science, Purdue Uuniversity, 2005.

[EMN89] R. El-Masri and S. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings Publisher, 1989.

[ES97] C. Esperanca and H. Samet. An overview of the SAND spatial database system. In *Communications of ACM.* ACM Press, 1997.

[Esu04] *The SUSE Linux Enterprise Distribution*, 2004. Disponibile sul Web all'indirizzo *http://www.novell.com/linux/*.

[FB74] R. Finkel and J. Bentley. Quadtrees: the Data Structure for Retrieval on Composite Keys. *Acta Informatica*, Volume 4, 1974.

[FBF77] J. Friedman, J. Bentley, and R. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software*, Volume 3, 1977.

[FKN80] H. Fuchs, Z. Kedem, and F. Naylor. On Visible Surface Generation by a priori Tree Structures. *ACM Computer Graphics*, 1980.

[FM84] A. Fournier and D. Montuno. Triangulating Simple Polygons and Equivalent Problems. *ACM Transactions on Graphics*, Volume 7, 1984.

[FP03] A. Fox and D. Patterson. Self–Repairing Computers. *Scientific American*, 2003.

[FR89] C. Faloutsos and S. Roseman. *Fractals for Secondary Key Retrieval.* In *Proceedings of the* 8th *ACM Symposium on Principles of Database Systems*, 1989.

[Fra97] M. Franklin. *Concurrency Control and Recovery*, 1997. Contenuto in *The Computer Science and Engineering Handbook*, UMIACS, University of Maryland.

[FS01] R. Farias and C. Silva. Out–of–Core Rendering of Large Unstructured Grids. *IEEE Computer Graphics and Applications*, Volume 21, 2001.

[FS05] T. Foley and J. Singerman. *Kd-tree Acceleration Structures for a GPU raytracer.* In *Proceedings of the Graphics Hardware Conference*, 2005.

[Gar82] I. Gargantini. Linear Octree for Fast Processing of 3–dimensional objects. *Computer Graphics and Image Processing*, 1982.

[Gar84] M. Gardner. *Klein bottles and other surfaces.* In *The* 6th *book of mathematical games from Scientific American.* University of Chicago Press, 1984.

[Gar99a] M. Garland. *Quadric–Based Polygonal Surface Simplification.* PhD thesis, Carnegie Mellon University, 1999.

[Gar99b] M. Garland. *Multiresolution Modeling: Survey & Future Opportunities.* In *Proceedings of the Eurographics Symposium on Geometry Processing*, 1999.

[GCC87] *GCC, the GNU Compiler Collection*, 1987. Disponibile sul Web all'indirizzo *http://gcc.gnu.org.*

[GD87] R. Geist and S. Daniel. A Continuum of Disk Scheduling Algorithms. *ACM Transactions on Computer Systems*, Volume 5, 1987.

[GG98] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, Volume 30, 1998.

[GH97] M. Garland and P. Heckbert. Surface Simplification Using Quadric Error Metrics. *Computer Graphics*, 1997.

[GH98] M. Garland and P. Heckbert. *Simplifying Surfaces with Color and Texture using Quadric Error Metrics.* In *Proceedings of the IEEE Visualization Conference*, 1998.

[GHKO81] J. Gray, P. Homan, H. Karth, and R. Obermarck. *A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System.* Technical Report RJ–3006, IBM Research Laboratory, 1981.

[GHSS95] B. Goyal, J. Haritsa, S. Seshadri, and V. Sruivasan. *Index Concurrency Control in Firm Real–Time DBMS.* In *Proceedings of the* 21st *International Conference on Very Large Databases*, 1995.

[Gib92] G. Gibson. *Redundant Disk Arrays: Reliable and Parallel Secondary Storage.* Technical Report UCB–CSD 92–613, Computer Science Department, University of California, 1992.

[GKS98] E. Gettys, F. Keller, and M. Skove. *Fisica classica e moderna: Elettromagnetismo, Ottica e Fisica Nucleare.* Mc–Graw Hill Editore, 1998.

[Gra78] J. Gray. Notes on Database Operating Systems. In *Operating Systems: an Advanced Course.* Springer–Verlag Publisher, 1978.

[GS78] L. Guibas and R. Sedgewick. *A Dichromatic Framework for the Balanced B–Trees.* In *Proceedings of the* 19th *Annual IEEE Symposium on Foundations of Computer Science*, 1978.

[GS95] R. Guting and M. Schneider. Real–based Spatial Data Types: the ROSE algebra. *The VLDB Journal*, Volume 4, 1995.

[Gut84] A. Guttman. *R–Trees: a Dynamic Index Structure for Spatial Searching.* In *Proceedings of the International Conference on Management of Data*, 1984.

[GYGM04] P. Ganesan, B. Yang, and H. Garcia-Molina. *One Torus to Rule Them All: Multidimensional Queries in P2P Systems.* In *Proceedings of the 7th International Workshop on the Web and Database*, 2004.

[Hal72] C. Halt. Some Deadlock Properties of the Computer Systems. *Computer Surveys*, Volume 4, 1972.

[Har96] G. Hart. *The Virtual Polyhedra Project*, 1996. Disponibile sul Web all'indirizzo *http://www.georgehart.com/virtual-polyhedra.*

[Hat02] A. Hatcher. *Algebraic Topology.* Cambridge University Press, 2002.

[Hav68] J. Havender. Avoiding Deadlock in Multitasking Systems. *The IBM Systems Journal*, Volume 7, 1968.

[Hel01] M. Held. FIST: Fast Industrial–Strength Triangulation of Polygons. *Algorithmica*, Volume 30, 2001.

[HG01] A. Hubeli and M. Gross. Multiresolution Methods for Non–Manifolds Models. *IEEE Transactions on Visualization and Computer Graphics*, Volume 7, 2001.

[Hil94] F. Hill. *The pleasures of Perp Dot Products.* In *Graphics Gems II.* Academic Press, 1994.

[Hip00] R. Hipp. *SQLite: an embedded database*, 2000. Disponibile sul Web all'indirizzo *http://www.sqlite.org.*

[HKS$^+$00] J. Hellerstein, M. Kornacker, M. Shah, M.Thomas, and C. Papadimitriou. *The GiST Indexing Project*, 2000. Disponibile sul Web all'indirizzo *http://gist.cs.berkeley.edu.*

[HNP95] J. Hellerstein, J. Naughton, and A. Pfeffer. *Generalized Search Trees for Database Systems.* In *Proceedings of the 21st International Conference on Very Large Databases*, 1995.

[Hof89] C. Hoffmann. *The Geometric and Solid Modeling: an Introduction.* Morgan Kaufmann Publisher, 1989.

[Hop96] H. Hoppe. *The Progressive Meshes.* In *Proceedings of the SIGGRAPH Conference*, 1996.

[Hop98] H. Hoppe. *Smooth View–Dependent Level–of–Detail Control and its Application to Terrain Rendering*. In *Proceedings of the IEEE Visualization Conference*, 1998.

[HP94] J. Hellerstein and A. Pfeffer. *The RD–Tree: an Index Structure for Sets*. Technical Report CS–TR–1252, University of Wisconsin, 1994.

[HSHH07a] D. Horn, J. Singerman, M. Houston, and P. Hanrahn. *Interactive Kd–tree GPU Raytracing*, 2007. Disponibile sul Web all'indirizzo *http://graphics.stanford.edu/papers/i3dkdtree*.

[HSHH07b] D. Horn, J. Singerman, M. Houston, and P. Hanrahn. *Interactive Kd–tree GPU Raytracing*. In *Proceedings of the* 21th *Symposium on Interactive 3D Graphics and Games*, 2007.

[HSW89] A. Heinrich, H. Six, and P. Widmayer. *The LSD–tree: Spatial Access to Multidimensional Points and non Points Objects*. In *Proceedings of the* 15th *International Conference on Very Large Databases*, 1989.

[Hut81] J. Hutchinson. Fractals and Semi–Similarity. *The Indiana University Mathematics Journal*, Volume 30, 1981.

[HVdF06] A. Hui, L. Vaczlavik, and L. de Floriani. *A Decomposition–based Representation for 3–dimensional Simplicial Complexes*. In *Proceedings of the* 4th *Eurographics Symposium on Geometry Processing*, 2006.

[IA95] D. Ibaroudene and R. Acharya. Parallel Display of Objects Represented by Linear Octrees. *IEEE Transactions on Parallel and Distributed Systems*, Volume 6, 1995.

[IBM01] *IBM DB2 Spatial Extender*, 2001. Sviluppato da *IBM Corporation* e disponibile sul Web all'indirizzo *http://www-306.ibm.-com/software/data/spatial/db2spatial*.

[IEE85] *The IEEE POSIX Standard*, 1985. Disponibile sul Web all'indirizzo *http://standards.ieee.org/regauth/posix/*.

[IG03] M. Isenburg and S. Gumhold. Out–of–core Compression for Gigantic Polygon Meshes. *ACM Transactions on Graphics*, Volume 22, 2003.

[IL05] M. Isenburg and P. Lindstrom. *Streaming Meshes*. In *Proceedings of the IEEE Visualization Conference*, 2005.

[ILGS03] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. *Large Mesh Simplification using Processing Sequences.* In *Proceedings of the IEEE Visualization Conference*, 2003.

[Ily02] I. Ilyas. *Issues in Spatial Databases Indexing*, 2002. In *The Fall CS 641 Midterm Course Notes*, Department of Computer Science, Purdue University.

[IM80] S. Isloor and T. Marsland. The Deadlock Problem: an Overview. *IEEE Computer*, Volume 13, 1980.

[Jaf03] A. Jaffer. *Multidimensional Space–Filling Curves*, 2003. Disponibile sul Web all'indirizzo *http://swiss.csail.mit.edu/~jaffer/-Geometry/MDSFC.*

[Jag90] H. Jagadish. *Linear Clustering of the Objects with Multiple Attributes.* In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1990.

[Joh66] N. Johnson. The Convex Solids with Regular Faces. *The Canadian Journal of Mathematics*, 1966.

[Jul22] G. Julia. Mémoire sur la permutabilité des fractions rationnelles. *Annales Scientifiques de l'École Normale Supérieure*, 1922.

[Kan90] P. Kanellakis. *Elements of Relational Database Theory.* In *Handbook of Theoretical Computer Science.* Elsevier Science, 1990.

[Kno06] A. Knoll. *A Survey of Octree Volume Rendering Methods.* In *Proceedings of the* 1st *Annual IRTG Workshop*, 2006.

[Knu73] D. Knuth. *Sorting and Searching.* In *The Art of Computer Programming.* Addison–Wesley Editor, 1973.

[Kor99] M. Kornacker. *High–Performance Extensible Indexing.* In *Proceedings of the* 25th *International Conference on Very Large Databases*, 1999.

[Kor00] M. Kornacker. *Access Methods for Next Generation Database Systems.* PhD thesis, University of California, Berkeley, 2000.

[KR81] H. Kung and J. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transaction on Database Systems*, Volume 6, 1981.

[KS91] H. Korth and A. Silberschatz. *Database Systems Concepts.* Mc–Graw Hill Editor, 1991.

[KS97a] N. Katayama and S. Satoh. *The SR–tree: An Index Structure for High–Dimensional Nearest Neighbor Queries.* In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1997.

[KS97b] N. Katayama and S. Satoh. The SR–tree: An Index Structure for High Dimensional Nearest Neighbor Queries. *Transactions of the Institute of Electronics, Information and Communication Engineers*, Volume J80–D–I, 1997.

[KWPH06] A. Knoll, I. Wald, S. Parker, and C. Hansen. *Interactive Isosurface Ray Tracing of Large Octree Volumes.* In *Proceedings of the 1st IEEE Symposium on Interactive Ray Tracing*, 2006.

[Lew87] J. Lewis. A Generalized Stochastic Subdivision. *ACM Transactions on Graphics*, Volume 6, 1987.

[Lew88] J. Lewis. *Is the Fractal Model Appropriate for Terrain?*, 1988. Disponibile sul Web all'indirizzo *http://www.idiom.com/~zilla/*.

[Lin00] P. Lindstrom. *The Out–of–Core Simplification of Large Polygonal Models.* In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics*, 2000.

[Lin03] P. Lindstrom. *Out–of–Core Construction and Visualization of Multiresolution Surfaces.* In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics*, 2003.

[LLLV05] M. Lage, T. Lewiner, H. Lopes, and L. Velho. *CHF: A scalable topological data structure for tetrahedral meshes.* In *Proceedings of the 18th Brazilian Symposium on Computer Graphics and Image Processing*, 2005.

[Lop93] F. Lopez. A complete minimal Klein bottle in $\mathbb{R}^3$. *Duke Math. Journal*, Volume 71, 1993.

[LP81] F. Luccio and L. Pagli. *Le Reti logiche ed i Calcolatori.* Bollati Boringhieri Editore, 1981.

[LP02] P. Lindstrom and V. Pascucci. *Terrain Simplification Simplified: A General Framework for View–Dependent Out–of–Core Visualization.* In *IEEE Transactions on Visualization and Computer Graphics*. IEEE Press, 2002.

[LPC$^+$00] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Gintzon, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. *The Digital Michelangelo Project: 3D Scanning of*

Large Statues. In *Proceedings of the SIGGRAPH Conference*, 2000.

[LRC+02] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and H. Huebner. *The Level of Detail for 3D Graphics.* Morgan Kaufmann Publisher, 2002.

[LS01] P. Lindstrom and T. Silva. *A Memory Insensitive Technique for Large Model Simplification.* In *Proceedings of the IEEE Visualization Conference*, 2001.

[LSB07] *LSB – the Linux Standard Base*, 2007. Disponibile sul Web all'indirizzo *http://www.linux–foundation.org/en/LSB.*

[LW77] D. Lee and C. Wong. Worst–case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quadtrees. *Acta Informatica*, Volume 9, 1977.

[LZ05] W. Lee and B. Zheng. *DSI: a Fully Distributed Spatial Index for Location–based Wireless Broadcast Services.* In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, 2005.

[MA97] D. Mount and S. Arya. *The ANN Library for Approximate Nearest–Neighbor Searching.* In *Proceedings of the* 2nd *Annual Center for Geometric Computing on Computational Geometry*, 1997.

[MAA+98] C. Mohan, D. Agrawal, G. Alonso, A. El-Abhadi, M. Kannath, and B. Reinwald. *The IBM Exotica Project*, 1998. Disponibile sul Web all'indirizzo *http://www.almaden.ibm.com/cs/exotica.*

[Mag99] P. Magillo. *Spatial Operations on Multiresolution Cell Complexes.* PhD thesis, DISI, Universitá degli Studi di Genova, 1999.

[Man77] B. Mandelbrot. *The Fractal Geometry of Nature.* W. H. Freeman & Company, 1977.

[Man88] M. Mantyla. *An Introduction to the Solid Modeling.* Computer Science Press, 1988.

[Man01] S. Maneewongvatana. *The Multi–dimensional Nearest–Neighbor Searching with Low–dimensional Data.* PhD thesis, Computer Science Department, University of Maryland, 2001.

[Man05] R. Mansfield. OOP is much better in Theory than in Practice. *DevX.com*, 2005. Disponibile sul Web all'indirizzo *http://www.-devx.com/opinion/Article/26776/.*

[Mar99] V. Markl. *MISTRAL: Processing Relational Queries using the Multidimensional Access Technique*. PhD thesis, Institut für Informatik der Technischen Universtät München, 1999.

[MB98] P. Magillo and V. Bertocci. *A modular approach to the construction of Multi–Triangulations*. Technical Report DISI–TR–98–19, DISI, Universitá degli Studi di Genova, 1998.

[MB00] P. Magillo and V. Bertocci. *Managing Large Terrain Data-Sets with a Multiresolution Structure*. In *Proceedings of the 11th International Workshop Database and Expert Systems Applications*, 2000.

[Men42] F. Menabrea. *Sketch of the Analytical Engine*, 1842. Disponibile sul Web all'indirizzo *http://www.fourmilab.ch/babbage/*.

[MJLF84] M. McKusick, B. Joy, S. Leffler, and R. Fabry. A Fast Filesystem for UNIX. *ACM Transactions on Computer Systems*, Volume 2, 1984.

[MM91] F. Musgrave and B. Mandelbrot. The Art of Fractal Landscapes. *IBM Journal of Research and Development*, 1991.

[MM99a] S. Maneewongvatana and D. Mount. *It's okay to be Shinny, if your friends are Fat*. In *Proceedings of the 4th Annual Center for Geometric Computing on Computational Geometry*, 1999.

[MM99b] S. Maneewongvatana and D. Mount. *The Analysis of Approximate Nearest–Neighbor Searching with Clustered Point Sets*. In *Proceedings of the 6th DIMACS Implementation Challenge Workshop*, 1999.

[Moh94] C. Mohan. *Advanced Transaction Models: Survey and Critique*. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[Mon04] *The MONO Project*, 2004. Sviluppato da *Novell Corporation* e disponibile sul Web all'indirizzo *http://www.mono–project.com*.

[Moo95] D. Moore. *The cost of Balancing the Generalized Quadtrees*. In *Proceedings of the 3rd Symposium on Solid Modeling and Applications*, 1995.

[Mor66] G. Morton. A Computer–oriented Geodetic Database and a New Technique in File Sequencing. Technical report, IBM Ltd., 1966.

[MS93] J. Melton and A. Simon. *Understanding the New SQL: a Complete Guide*. Morgan Kaufmann Publisher, 1993.

[MSD07] *Microsoft Developer Network – the Windows API*, 2007. Disponibile sul Web all'indirizzo *http://msdn2.microsoft.com/en–us/library/aa383750.aspx.*

[Msh05] *The MeshLab Project*, 2005. Disponibile sul Web all'indirizzo *http://meshlab.sourceforge.net.*

[MT84] S. Mullender and A. Tanenbaum. Immediate files. *Software Practice and Experience*, Volume 14, 1984.

[Mur05] A. Murugapan. *Comparison of R-tree, MVR–tree and TPR–tree.* Technical report, College of Computing, Georgia Institute of Technology, 2005.

[Mus93] F. Musgrave. *Methods for Realistic Landscape Imaging.* PhD thesis, Yale University, 1993.

[MV96] M. Moughdill and S. Vassiliadis. *Precise Interrupts.* In *Proceedings of IEEE Micro Conference*, 1996.

[MVS03] *Microsoft Visual Studio .NET*, 2003. Sviluppato da *Microsoft Corporation* e disponibile sul Web all'indirizzo *http://www.microsoft.com/italy/msdn/vstudio.*

[Naf83] *The National Agency for Finite Element Methods and Standards*, 1983. Disponibile sul Web all'indirizzo *http://www.nafems.org/.*

[Nel90] V. Nelson. Fault–Tolerant Computing: Fundamental Concepts. *IEEE Computer*, Volume 23, 1990.

[Net02] *The Microsoft .NET Framework*, 2002. Sviluppato da *Microsoft Corporation* e disponibile sul Web all'indirizzo *http://www.microsoft.com/italy/msdn/prodotti/netframework/default.mspx.*

[New79] G. Newton. Deadlock Prevention, Detection and Resolution: an Annotated Bibliography. *Operating Systems Review*, Volume 13, 1979.

[NGHS97] S. Narayanan, B. Goyal, J. Haritsa, and S. Seshadri. *Robust Real–Time* Index Concurrency Control. Technical Report TR–97–103, Indian Institute of Technology, 1997.

[NM94] A. Narkhede and D. Manocha. *Fast polygon triangulation algorithm based on Seidel algorithm*, 1994. Department of Computer Science, University of North Carolina, Capel Hill. Disponibile sul Web all'indirizzo *http://www.cs.unc.edu/~dm.*

[NS86a] R. Nelson and H. Samet. A Consistent Hierarchical Representation for Vector Data. *Computer Graphics*, Volume 20, 1986.

[NS86b] R. Nelson and H. Samet. *A Population Analysis of Quadtrees with Variable Node Size.* Technical Report TR–1740, UMIACS, University of Maryland, 1986.

[NS87] R. Nelson and H. Samet. *A Population Analysis for Hierarchical Data Structures.* In *Proceedings of the ACM–SIGMOD Conference*, 1987.

[NS04] B. Nam and A. Sussman. *A Comparative Study of Spatial Indexing Techniques for the Multidimensional Scientific Datasets.* Technical Report 2004–03, UMIACS, University of Maryland, 2004.

[NS06] B. Nam and A. Sussman. *Indexing Cached Multidimensional Objects in Large Main Memory Systems.* Technical Report 2006–16, UMIACS, University of Maryland, 2006.

[OG04] *IEEE Portable Applications Standards Commitee*, 2004. Disponibile sul Web all'indirizzo *http://www.pasc.org/plato*.

[Oli05] J. Oliveira. *The MAC, PC and UNIX PLY Reading/Writing*, 2005. Sviluppato presso il *Department of Computer Science, University College of London* e disponibile sul Web all'indirizzo *http://www.cs.ucl.ac.uk/staff/joao.oliveira/*.

[Oll78] W. Olle. *The CODASYL Approach to the Database Management.* John Wiley & Sons Editor, 1978.

[O'R94] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[Ora05] *Oracle 10g Release 2*, 2005. Sviluppato da *Oracle Corporation* e disponibile sul Web all'indirizzo *http://www.oracle.com*.

[Ore82] J. Orenstein. Multidimensional Tries used for Associative Searching. *Information Processing Letters*, Volume 14, 1982.

[OS83a] Y. Ohsawa and M. Sakauchi. *The BD–Tree: a new n–dimensional Data Structure with Highly Efficient Dynamic Characteristics.* In *Proceedings of the Information Processing Conference*, 1983.

[OS83b] Y. Ohsawa and M. Sakauchi. The Multidimensional Data Management Structure with Efficient Dynamic Characteristics. *Systems, Computers and Controls*, Volume 14, 1983.

[OSP05] *Oracle Spatial & Oracle Locator: Location Features for Oracle Database 10g Release 2*, 2005. Sviluppato da *Oracle Corporation* e disponibile sul Web all'indirizzo *http://www.oracle.com/technology/products/spatial/index.html.*

[Osu05] *The OpenSUSE Linux distribution*, 2005. Disponibile sul Web all'indirizzo *http://www.opensuse.org.*

[OvL82] M. Overmars and J. von Leeuwen. Dynamic Multidimensional Data Structures based on Quadtree and Kd–trees. *Acta Informatica*, Volume 17, 1982.

[PAAV03] O. Procopiuc, P. Agarwal, L. Arge, and J. Vitter. *Bkd–tree: a Dynamic Scalable kd–tree.* In *Proceedings of the* 8th *International Symposium on Spatial and Temporal Databases*, 2003.

[PAGL06] J. Pombo, M. Aldegunde, and A. Garcia-Louriero. *Optimization of an Octree–based 3D Parallel Meshing Algorithm for the Simulation of Small–feature Semiconductor Devices.* In *Parallel Computing: Current and Futures Issues of High–End Computing.* NIC Press, 2006.

[Pap86] C. Papadimitriou. *The Theory of The Concurrency Control.* Computer Science Press, 1986.

[Par94] J. Paredoens. *Spatial Databases: the Final Frontier.* In *Proceedings of the* 5th *International Conference on the Database Theory*, 1994.

[PBB+02] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Saastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. *The Berkeley/Stanford ROC Project*, 2002. Disponibile sul Web all'indirizzo *http://roc.cs.berkeley.edu.*

[PBFO02] D. Patterson, A. Brown, A. Fox, and D. Oppenheimer. *Recovery – Oriented Computing (ROC): Motivation, Definition, Techniques and Case Studies.* Technical Report UCB/CSD–02–1175, Computer Science Department, UC Berkeley, 2002.

[Per02] G. Perelman. *The Entropy Formula for the Ricci Flow and its geometric applications*, 2002. Disponibile sul Web all'indirizzo *http://arxiv.org/abs/math/0211159.*

[Per03a] G. Perelman. *Finite Extinction Time for the Solutions to the Ricci flow on certain 3–manifolds*, 2003. Disponibile sul Web all'indirizzo *http://arxiv.org/abs/math/0307245*.

[Per03b] G. Perelman. *The Ricci Flow with Surgery on the 3–manifolds*, 2003. Disponibile sul Web all'indirizzo *http://arxiv.org/abs/-math/0303109*.

[PGK88] D. Patterson, G. Gibson, and R. Katz. *A Case for Redundant Arrays of Independent Drives*. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1988.

[PH97] J. Popovic and H. Hoppe. *Progressive Simplicial Complexes*. In *Proceedings of the SIGGRAPH Conference*, 1997.

[Pic02] S. Piccardi. *GaPiL: Guida alla Programmazione in Linux*, 2002. Disponibile sul Web all'indirizzo *htpp://gapil.truelite.it*.

[Pol03] K. Polthier. Inside the Klein Bottle. *Plus Magazine*, 2003.

[Pri00] C. Prince. *Progressive Meshes for Large Models of Arbitrary Topology*. Master's thesis, University of Washington, 2000.

[Pro97a] C. Procopiuc. *Applications of Clustering Algorithms*, 1997. In *The Spring CS–234 Course Notes*, *Computer Science Department*, *Duke University* e disponibile sul Web all'indirizzo *http://www.cs.duke.edu/∼pankaj/spring97/cps234.html*.

[Pro97b] O. Procopiuc. *Data Structures for Spatial Database Systems*, 1997. In *The Spring CS–234 Course Notes*, *Computer Science Department*, *Duke University* e disponibile sul Web all'indirizzo *http://www.cs.duke.edu/∼pankaj/spring97/cps234.html*.

[PS85] F. Preparata and M. Shamos. *Computational Geometry – an Introduction*. Springer–Verlag Publisher, 1985.

[PS97] E. Puppo and R. Scopigno. *Simplification, LOD and Multiresolution – Principles and Applications*. In *Tutorial Notes of Eurographics Conference*, 1997.

[Pup96] E. Puppo. Variable Resolution Triangulations. *Computational Geometry*, Volume 11, 1996.

[Pup98] E. Puppo. *Variable Resolution Terrain Surfaces*. In *Proceedings of the 8th Canadian Conference on Computational Geometry*, 1998.

[QS05] J. Qi and V. Shapiro. *ε–Regular Sets and Intervals*. In *Proceedings of IEEE International Conference on Shape Modeling and Applications*, 2005.

[QS06] J. Qi and V. Shapiro. ε–Topological Formulation of Tolerant Solid Modeling. *Computer–Aided Design*, Volume 38, 2006.

[RB92] J. Rossignac and P. Borrel. *The Multi–Resolution 3D Approximations for Rendering Complex Scenes.* Technical Report RC–17697, IBM Corporation, 1992.

[RCHQ98] J. Ribelles, M. Chover, J. Huerta, and R. Quirós. *Multiresolution Ordered Meshes.* In *Proceedings of the* 4th *IEEE Conference on Information Visualization*, 1998.

[Req96] A. Requicha. *The Geometric Modeling: A First Course*, 1996. Laboratory for Molecular Robotics, Computer Science Department, University of Southern California. Disponibile sul Web all'indirizzo *http://www–pal.usc.edu/~requicha/book.html.*

[Rid01] J. Riddel. *Umbrello UML Modeler*, 2001. Disponibile sul Web all'indirizzo *http://uml.sourceforge.net.*

[RL00] S. Rusinkiewicz and M. Levoy. *QSplat: A Multiresolution Point Rendering System for Large Meshes.* In *Proceedings of the SIGGRAPH Conference on Computer Graphics*, 2000.

[RMF+00] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elkardt, and R. Bayer. Integrating the UB–tree into a Database System Kernel. *The VLDB Journal*, 2000.

[Rob81] J. Robinson. *The K–D–B–tree: a Search Structure for Large Multidimensional Dynamic Indexes.* In *Proceedings of the ACM–SIGMOD International Conference on Management of Data*, 1981.

[RT74] D. Ritchie and K. Thompson. The UNIX Timesharing System. *Communications of ACM*, Volume 17, 1974.

[RTBS05] P. Rhodes, X. Tuang, R. Bergeron, and T. Sparr. *Out–of-core Visualization using Iterator aware Multidimensional Prefetching.* In *Proceedings of the Visualization and Data Analysis*, 2005.

[SA94] H. Samet and W. Aref. The Spatial Data Models and Query Processing. In *Modern Database Systems: the Object Model, Interoperability and beyond.* ACM Press, 1994.

[Sag94] H. Sagan. *Space–filling Curves.* Springer Verlag Publisher, 1994.

[Sam80] H. Samet. Deletion in 2–dimensional Quadtrees. *Communications of ACM*, Volume 23, 1980.

[Sam84] H. Samet. Quadtree and Related Hierarchical Data Structures. *Computing Surveys*, Volume 16, 1984.

[Sam90a] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*. Addison Wesley Editor, 1990.

[Sam90b] H. Samet. *The design and Analysis of Spatial Data Structures*. Addison–Wesley Editor, 1990.

[Sam95] H. Samet. *Spatial Data Structures*. In *Modern Database Systems: the Object Model, Interoperability and beyond*. ACM Press, 1995.

[Sam03] H. Samet. *The Object–based and Image–based Objects Representations*. Technical Report 2003–94, UMIACS, University of Maryland, 2003.

[Sam06] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publisher, 2006.

[SCC+02] C. Silva, Y. Chiang, W. Correa, J. El-Sana, and P. Lindstrom. *The Out–Of–Core Algorithms for Scientific Visualization and Computer Graphics*. In *Proceedings of the IEEE Visualization Conference*, 2002.

[Sei91] R. Seidel. A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons. *Computational Geometry Theory and Applications*, Volume 1, 1991.

[Sel05] M. Seltzer. Beyond Relational Databases: there is more to Data Access than SQL. *ACM Queue*, Volume 3, 2005.

[Ser06] M. Servetto. *Scomposizione in Moduli e Gestione di Modelli Multirisoluzione Modulari*. Master's thesis, DISI, Universitá degli Studi di Genova, 2006.

[SES98a] M. Sweet, C. Earls, and B. Spitzak. *FLTK: Fast Light Toolkit*, 1998. Disponibile sul Web all'indirizzo *http://www.fltk.org*.

[SES98b] M. Sweet, C. Earls, and B. Spitzak. *The Fast Light Toolkit 1.1.7 Programming Manual*, 1998. Disponibile sul Web all'indirizzo *http://www.fltk.org*.

[SG05] E. Shaffer and M. Garland. A Multiresolution Representation for Massive Meshes. *IEEE Transactions on Visualization and Computer Graphics*, Volume 11, 2005.

[SGV+05] A. Singh, S. Gopisetty, K. Voruganti, D. Pease, and L. Liu. *An Hybrid Access Model for Storage Area Networks*. In *Proceedings of the* 13th *NASA Goddard Conference on Mass Storage Systems and Technologies*, 2005.

[Sha01] *The OFF Format Specifications*, 2001. Sviluppato dal *Princeton Shape Retrieval and Analysis Group* presso il Department of Computer Science, Princeton University e disponibile sul Web all'indirizzo *http://shape.cs.princeton.edu.*

[SMdF06] M. Servetto, P. Magillo, and L. de Floriani. *Modular Multi–Tesselation for Distributed Applications.* Technical Report DISI–TR–06–16, DISI, Universitá degli Studi di Genova, 2006.

[Smi78] A. Smith. Bibliography on Paging and Relating Topics. *Operating Systems Review*, Volume 12, 1978.

[Smi81] A. Smith. Bibliography on File and I/O System Optimization and Relating Topics. *Operating Systems Review*, Volume 15, 1981.

[Sob06] D. Sobrero. *Efficient Representations for Multi–resolution Modeling.* PhD thesis, DISI, Universitá degli Studi di Genova, 2006.

[Sol04] S. Solin. Il futuro del vostro Hard Disk. *Le Scienze*, 2004.

[SP00] M. Salas and A. Polo. *The mQ–tree: a Multidimensional Access Method based on a Non–Binary Tree.* In *Proceedings of the* 7th *Conference on Extending Database Technology*, 2000.

[Sql96] *PostgreSQL – The world's most advanced open source database*, 1996. Sviluppato dal *PostgreSQL Global Development Group* e disponibile sul Web all'indirizzo *http://www.postgresql.org.*

[Sql03] *The SQL Standards Page*, 2003. Disponibile sul Web all'indirizzo *http://www.jcc.com/sql.htm.*

[Sta85] R. Stalmann. *Free Software Foundation*, 1985. Disponibile sul Web all'indirizzo *http://www.fsf.org.*

[Sta94] *The Stanford Computer Graphics Laboratory*, 1994. Disponibile sul Web all'indirizzo *http://graphics.stanford.edu.*

[STL94] *The Standard Template Library Programmer's Guide*, 1994. Disponibile sul Web all'indirizzo *http://www.sgi.com/tech/stl/index.html.*

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison–Wesley Editor, 1986.

[SW06] K. Sang-Wook. Synchronization in an Embedded DBMS Environment. *The International Journal of Computer Science and Network Security*, Volume 6, 2006.

[SWND07] D. Shreiner, M. Woo, J. Neider, and T. Davis. *The OpenGL Programming Guide.* Addison–Wesley Editor, 2007.

[SZL92] W. Schroeder, J. Zarge, and W. Lorensen. *Decimation of Triangles Meshes.* In *Proceedings of the SIGGRAPH Conference*, 1992.

[Tan91] A. Tanenbaum. *Architettura del Computer: un Approccio Strutturale.* Prentice Hall International, 1991.

[Tan92] A. Tanenbaum. *I Moderni Sistemi Operativi.* Prentice Hall International, 1992.

[TAW+02] L. Torvalds, H. Anvin, C. Wright, K. Cook, and J. Uphoff. *The Linux Kernel Archives*, 2002. Disponibile sul Web all'indirizzo *http://www.kernel.org.*

[TC01] E. Tonti and F. Cosmi. *The Discrete Physics Project*, 2001. Disponibile sul Web all'indirizzo *http://discretephysics.dic.units.it.*

[TD01] L. Torvalds and D. Diamond. *Rivoluzionario per caso: come ho creato Linux solo per divertirmi.* Garzanti Editore, 2001.

[TH81] H. Tropf and H. Herzog. Multidimensional Range Search in Dynamically Balanced Trees. *Applied Iinformatics*, 1981.

[THB06] A. Tanenbaum, J. Herder, and H. Bosh. File Size Distribution in UNIX Systems: Then and Now. *Operating Systems Review*, Volume 40, 2006.

[Thu97] W. Thurston. *3–Dimensional Geometry and Topology.* Princeton University Press, 1997.

[TO04] T. Tu and D. O'Hallaron. *Balance Refinement of Massive Linear Octree Datasets.* Technical Report CMU–CS–04–129, School of Computer Science, Carnegie Mellon University, 2004.

[TOL02] T. Tu, D. O'Hallaron, and J. López. *ETREE: a Database–Oriented Method for Generating Large Octree–Meshes.* In *Proceedings of the* 11th *International Meshing Roundtable*, 2002.

[TOL03] T. Tu, D. O'Hallaron, and J. Lopez. *The ETREE Library: A System for Manipulating Large Octrees on Disk.* Technical Report CMU–CS–03–174, School of Computer Science, Carnegie Mellon University, 2003.

[Tou91] G. Toussaint. Efficient Triangulation of Simple Polygons. *The Visual Computer*, Volume 7, 1991.

[TR91] A. Thomasian and I. Ryu. The Performance Analysis of Two–Phase Locking. *IEEE Transactions on Software Engineering*, Volume 17, 1991.

[Tri05] H. Trickey. *APE – The ANSI/POSIX Environment*, 2005. Disponibile sul Web all'indirizzo *http://cm.bell–labs.com/plan9/.*

[Ull89] J. Ullman. *Database and Knowledge–based Systems.* Computer Science Press, 1989.

[VCL+07] H. Vo, S. Callahan, P. Lindstrom, V. Pascucci, and C. Silva. Streaming Simplification of Tetrahedral Meshes. *IEEE Transactions on Visualization and Computer Graphics*, Volume 13, 2007.

[vdBS01] J. van den Berckern and B. Seeger. *An Evaluation of Generic Bulk–Loading Techniques.* In *Proceedings of the* 27th *International Conference on Very Large Databases*, 2001.

[vH97] D. van Heesch. *Doxygen: the Source Code Documentation Automatic Generator Tool*, 1997. Disponibile sul Web all'indirizzo *http://www.doxygen.org.*

[Vit01] J. Vitter. External Memory Algorithms and Data Structures dealing with Massive Data. *ACM Computing Surveys*, Volume 33, 2001.

[vN45] J. von Neumann. *A First Draft of a Report on the EDVAC.* Technical report, Moore School of Electrical Engineering, University of Pennsylvania, 1945.

[Vos88] R. Voss. *The Fractals in nature: from characterization to simulation.* In *The Science of Fractal Images.* Springer–Verlag Publisher, 1988.

[VOU04] K. Voruganti, M. Ozsu, and R. Unran. An Adaptive Data–Shipping Architecture for Client Caching Data Management Systems. *Distributed and Parallel Databases*, Volume 15, 2004.

[VS94a] J. Vitter and E. Shriver. The Algorithms for Parallel Memory: Hierarchical Multilevel Memories. *Algorithmica*, Volume 12, 1994.

[VS94b] J. Vitter and E. Shriver. The Algorithms for Parallel Memory: Two–Level Memories. *Algorithmica*, Volume 12, 1994.

[VV96] E. Vengroff and J. Vitter. *I/O–Efficient Scientific Computation using TPIE*. In *Proceedings of the Goddard Conference on Mass Storage Systems and Technologies*, 1996.

[Wal01] *The PLY Format Specifications*, 2001. Sviluppato dal *Walkthru Group* presso il Department of Computer Science, UNC e disponibile sul Web all'indirizzo *http://cs.unc.edu/∼geom/Powerplant/ply.doc*.

[Web00] *WebML – The Web Modeling Language*, 2000. Sviluppato dal *WebML Research Group* presso il Dipartimento di Elettronica ed Informazione, Politecnico di Milano e disponibile sul Web all'indirizzo *http://www.webml.org*.

[Weg87] P. Wegner. *Dimensions of Object–based Language Design*. In *Proceedings of the ACM Symposium on Object–Oriented Programming: Systems, Languages and Applications*, 1987.

[WYM98] W. Wang, J. Yang, and R. Muntz. *PK–tree: A Spatial Index Structure for High Dimensional Point Data*. In *Proceedings of the* 5th *International FODO Conference*, 1998.

[XLT04] J. Xu, W. Lee, and X. Tang. *Exponential Index: A parametrized distributed indexing scheme for data on air*. In *Proceedings of the* 2nd *ACM/USENIX International Conference on Mobile Systems, Applications and Services*, 2004.

[YWM97] J. Yang, W. Wang, and R. Muntz. *Yet Another Spatial Indexing Structure*. Technical Report UCLA 97–0039, Computer Science Department, University of California, 1997.

[Zal69] V. Zalgaller. The Convex Polyhedra with Regular Faces. *Consultants Bureau*, 1969.

[Zha00] X. Zhao. *Trie Methods for Structured Data on Secondary Storage*. PhD thesis, School of Computer Science, Mc–Gill University, Montreal, 2000.

[Zob83] D. Zobel. The Deadlock Problem: a Classifying Bibliography. *Operating Systems Review*, Volume 17, 1983.

[ZSAR98] P. Zezula, P. Savino, G. Amato, and F. Rabitti. Approssimate Similarity Retrieval with M–Trees. *The VLDB Journal*, 1998.