

Università degli Studi di Padova

Facoltà di Scienze MM.FF.NN.

Dipartimento di Matematica Pura ed Applicata

Corso di Laurea in Matematica

Tesi di Laurea

Costruzione di mesh gerarchiche

Relatore: Prof. Fabio Marcuzzi

Laureando: Daniele Ferro

A.A. 2010-2011

Indice

1	Mesh	7
1.1	Introduzione	7
1.2	Elementi di base	7
1.3	Triangolazione e Mesh	9
1.4	Costruzione	11
1.5	Triangolazione vincolata	15
1.6	Triangolazione anisotropica	18
1.7	Creare una mesh	20
1.8	Mesh di una superficie parametrica	20
1.9	Ottimizzazioni	22
1.10	Mesh del contorno	22
2	Generazione automatica di una mesh gerarchica	25
2.1	Il problema	25
2.2	La libreria	26
2.2.1	Caratteristiche	27
2.2.2	Esempio di utilizzo	29
3	Risultati	31
3.1	Esempio 1	31
3.2	Esempio 2	34
4	Appendice A	41
4.1	Diffusione delle proprietà	41
4.2	Unificazione degli strati	43

Introduzione

Questa tesi si occuperà di analizzare vari aspetti di una mesh gerarchica, soprattutto dal punto di vista dell'efficienza computazionale. Il primo capitolo pone le basi per comprendere i vari aspetti di una mesh e della sua generazione. Successivamente espongo una possibile implementazione di una libreria che gestisce una mesh gerarchica, esaminandone vari aspetti e compromessi, in termini di efficienza computazionale e alcuni esempi del suo utilizzo, soprattutto soluzioni di problemi temporvarianti. L'ultima parte, invece, contiene dei frammenti di codice della libreria che evidenziano alcuni aspetti significativi.

Capitolo 1

Mesh

1.1 Introduzione

In questo capitolo intendo fornire una panoramica sui principali metodi per la creazione di una mesh, di cui darò una definizione in seguito; non intendo essere esaustivo, ma solo mostrare le strategie più usate. Inizierò con i singoli elementi finiti, qui considero triangoli e tetraedri, per poi esaminare triangolazioni e mesh.

Per non entrare troppo in dettagli tecnici, mi concentrerò principalmente sulle 2 dimensioni, nel piano, o su superfici, mentre accennerò solamente alle differenze in 3 o più dimensioni.

1.2 Elementi di base

Per creare una triangolazione, e quindi una mesh, è necessario definire quali saranno gli elementi che costituiranno i mattoni della mesh; nel caso di 2 e 3 dimensioni è consolidato usare triangoli e tetraedri, mentre nel caso generale multidimensionale semplici in d dimensioni.

Per ora, assumo il concetto classico di triangolo, definito da 3 punti non allineati in \mathbf{R}^2 , con area superficiale positiva.

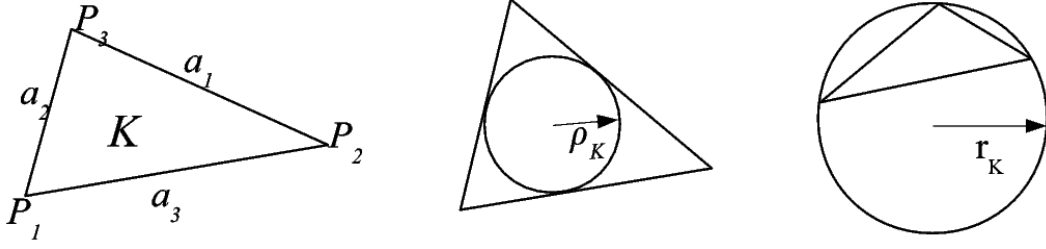


Figura 1.1: Triangoli con nomi convenzionali per vertici, lati, raggio del cerchio inscritto e circoscritto

Sono inoltre importanti i concetti di raggio circoscritto e inscritto, delle circonferenze rispettivamente circoscritta e inscritta; tali raggi si possono ricavare dalle seguenti:

$$r_K = \frac{L_1 \cdot L_2 \cdot L_3}{4 \cdot S_K} \quad (1.1)$$

$$\rho_K = \frac{S_K}{p_K} \quad (1.2)$$

Con L_i lunghezze dei lati a_i , S_k area e p_k perimetro del triangolo K.

Altra grandezza fondamentale è la qualità degli elementi di base, intesa come una misura della forma dei triangoli; più un triangolo è vicino ad essere equilatero, migliore sarà la sua qualità. Ciò è importante perché l'accuratezza del calcolo di un metodo agli elementi finiti è strettamente legata alla qualità dei triangoli. Alcune misure di qualità sono:

$$Q_K = \alpha \frac{h_{max}}{\rho_K} \quad (1.3)$$

$$Q_K = \beta \frac{h_s^2}{S_K} \quad (1.4)$$

Dove α e β sono costanti di normalizzazione affinché la qualità di un triangolo equilatero sia unitaria (in dettaglio $\alpha = \frac{\sqrt{3}}{6}$ e $\beta = \frac{\sqrt{3}}{12}$), h_{max} è la lunghezza

del lato maggiore e $h_s = \sqrt{\sum_{i=1}^3 L_i}$; queste misure forniscono un numero che va da 1 per un triangolo equilatero a ∞ per un triangolo degenere.

Tutte le definizioni date fino a questo punto si possono estendere facilmente ai tetraedri e, in generale, a semplici d-dimensionali, sostituendo al cerchio circoscritto la sfera circoscritta (o inscritta).

1.3 Triangolazione e Mesh

Triangolazione *Dati un insieme di punti in \mathbf{R}^2 (o \mathbf{R}^3), che tramite il loro involucro convesso definiscono un dominio Ω , si dice triangolazione di Ω , un ricoprimento del dominio tramite triangoli non degeneri, i cui vertici sono tutti e soli i punti dati.*

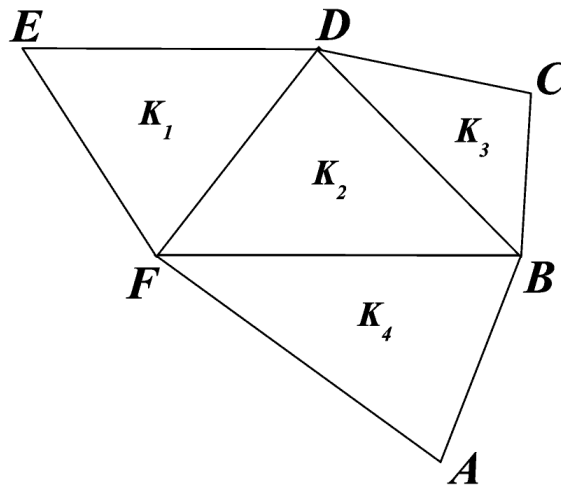


Figura 1.2: Una triangolazione di un dominio in \mathbf{R}^2 con 6 punti

Dalla definizione segue che la triangolazione di un dominio dato non è unica, come mostrato dalla figura 1.3.

Si considera pertanto una triangolazione più restrittiva, quella di Delaunay, che oltre alle ipotesi descritte sopra, deve soddisfare anche la seguente:

Ogni disco aperto (senza il bordo) circoscritto ai triangoli deve esser vuoto (non contenere vertici della triangolazione)

Tale definizione garantisce alcune proprietà, come la minimizzazione del massimo raggio circoscritto.

Nel paragrafo successivo mostrerò come sia possibile costruire una triangolazione di Delaunay, per ora voglio dare altre definizioni che torneranno utili.

Mesh *Sia Ω un dominio chiuso in \mathbf{R}^n . Una mesh è una triangolazione di Ω .*

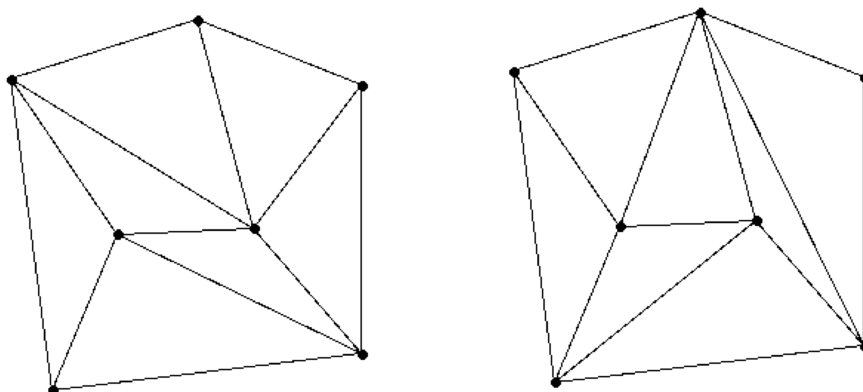


Figura 1.3: Con gli stessi punti si possono ottenere più triangolazioni

La differenza tra mesh e triangolazione sta nel fatto che in una triangolazione sono noti in partenza i vertici degli elementi, mentre per costruire una mesh bisogna inserire in modo opportuno i vertici all'interno di Ω ; inoltre una mesh potrebbe non avere un bordo convesso.

È comodo inoltre definire la qualità globale di una mesh, come la qualità massima dei suoi elementi:

$$Q_M = \max_{K \in M} Q_K \quad (1.5)$$

Introdurrò ora alcune definizioni relative a insiemi di elementi di una mesh, che torneranno utili in seguito.

Palla Sia P un vertice della triangolazione, la Palla associata a P è l'insieme di elementi che hanno P come vertice.

Base Sia P un generico punto nella triangolazione. La Base è l'insieme di elementi che contengono P .

Cavità Dato un generico punto P nella triangolazione, la Cavità è l'insieme di elementi i cui cerchi circoscritti includono P .

Tubo (Pipe) Sia s un segmento i cui estremi sono vertici della triangolazione. Il Tubo è l'insieme di elementi che hanno almeno un lato intersecato da s .

Guscio Sia a un lato di un triangolo nella triangolazione. Il guscio associato ad a è l'insieme di elementi che condividono tale lato

Con queste definizioni risulterà più semplice, nel prossimo capitolo, descrivere la costruzione della triangolazione.

1.4 Costruzione

La costruzione di una triangolazione di Delaunay si può effettuare considerando il duale di un diagramma di Voronoi, la cui definizione è la seguente.

Sia S un insieme di n punti P_i , in dimensione d ; il diagramma di Voronoi associato ad S è l'insieme di celle poligonali V_i , definite da

$$V_i = \{P : d(P, P_i) \leq d(P, P_j) \forall j \neq i\} \quad (1.6)$$

dove $d(\cdot, \cdot)$ è la distanza (euclidea) tra 2 punti.

Nel caso $d = 2$ la tassellazione che ne risulta si dice di Dirichlet.

In altre parole, ogni cella è l'insieme di punti più vicini al punto dato che ad ogni altro punto in S .

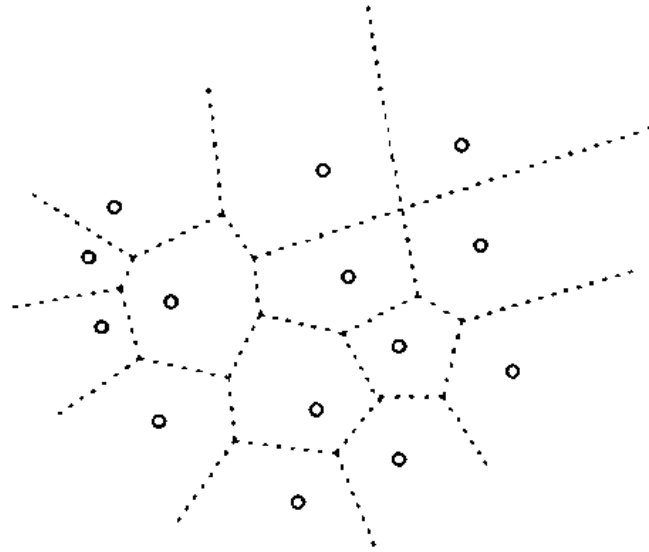


Figura 1.4: Diagramma di Voronoi in 2 dimensioni (Borouchaki [1])

Se si considera ora l'insieme dei triangoli i cui lati sono i segmenti che uniscono i punti di S , tra celle di Voronoi adiacenti, si ottiene una triangolazione Delaunay (la quale sarà univoca sotto certe condizioni).

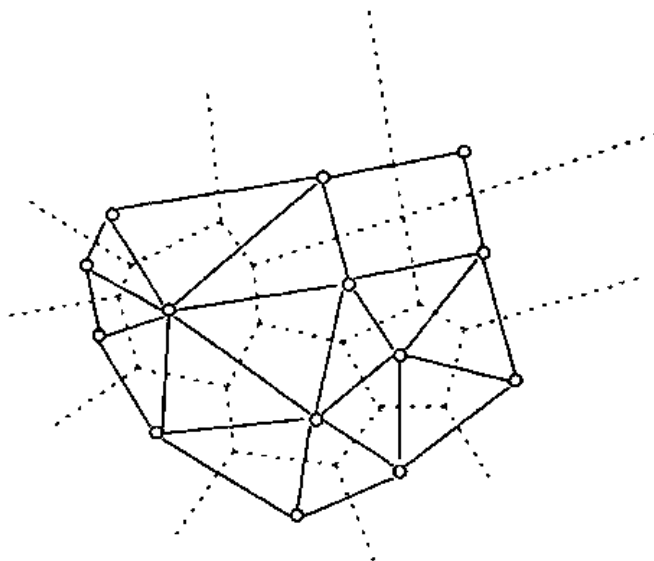


Figura 1.5: Triangolazione Delaunay ricavata dal diagramma di Voronoi (Borouchaki [1])

Per generare una triangolazione Delaunay, quindi, uno dei modi possibili è quello di costruire una tassellazione di Voronoi e poi unire i vertici dati secondo celle adiacenti; tale metodo però, è poco efficiente dal punto di vista computazionale.

Un altro metodo molto usato è il cosiddetto metodo incrementale; come dice il nome è un metodo iterativo che, a partire dalla triangolazione T_i (dell'involucro convesso di i punti), genera la triangolazione T_{i+1} in cui è stato aggiunto un nuovo punto.

Il fulcro di tale metodo è il Delaunay Kernel, una procedura che si schematizza in questo modo:

$$T_{i+1} = T_i - C_P + B_P$$

dove C_P è la cavità associata al punto P , mentre B_P è la palla associata a P ; non essendo P un vertice della triangolazione, al passo i -esimo, B_P è definita come l'insieme di elementi ottenuti unendo P con i lati esterni di C_P .

A seconda della posizione di P si hanno 2 casi: P è contenuto in T_i (figura 1.6) oppure P è esterno a T_i (figure 1.7 e 1.8). Nel secondo caso, la Cavità di P è ottenuta dalla Cavità secondo la definizione precedentemente data, con l'aggiunta degli elementi ottenuti unendo P con i lati in T_i visibili da P .

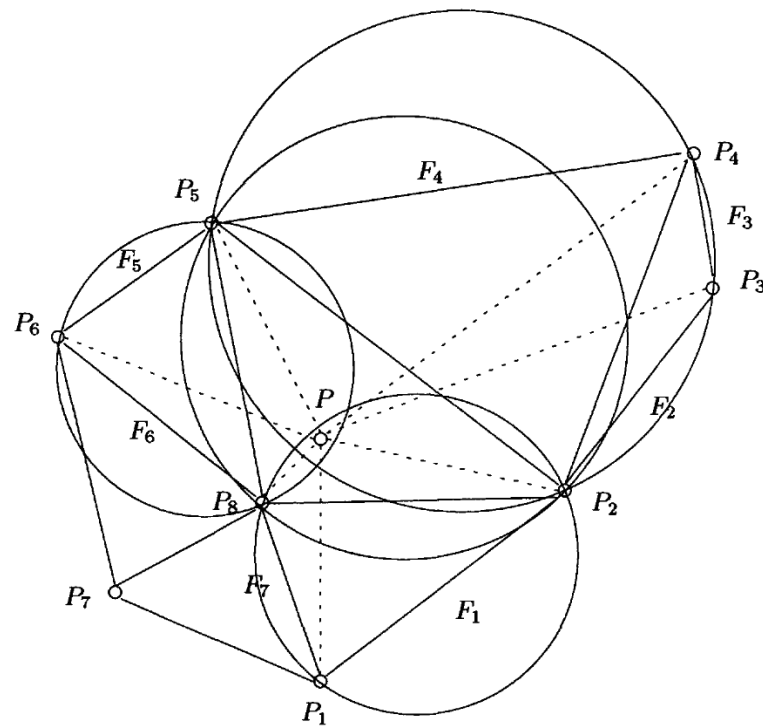


Figura 1.6: Costruzione della triangolazione con il metodo incrementale, P interno. P_i indicano i punti della triangolazione, P è il punto da inserire e F_i sono i lati visibili da P . (Borouchaki [1])

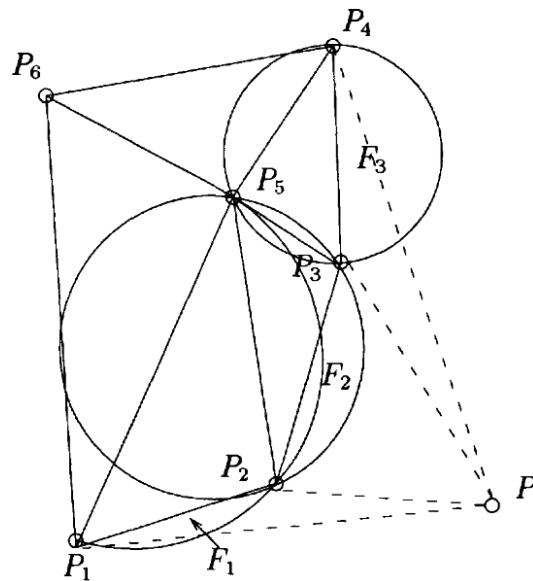


Figura 1.7: Costruzione della triangolazione con il metodo incrementale, P esterno e al di fuori di ogni disco circoscritto. La notazione è la stessa della figura precedente. (Borouchaki [1])

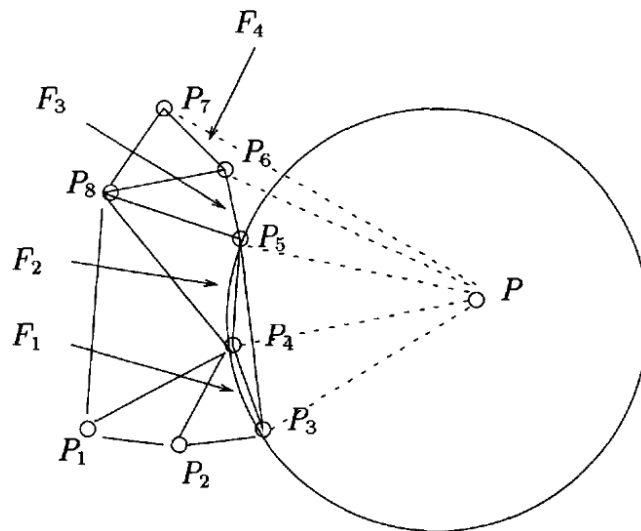


Figura 1.8: Costruzione della triangolazione con il metodo incrementale, P esterno ma incluso in uno dei dischi circoscritti. La notazione è la stessa della figura precedente. (Borouchaki [1])

Tale costruzione si basa sul seguente teorema

Sia T_i una triangolazione di Delaunay dell'involucro convesso dei primi i punti di S , allora T_{i+1} è una triangolazione di Delaunay di tale involucro e include P , il punto $i+1$ esimo come vertice.

Dimostrazione: omessa

Si può modificare la procedura sopra descritta in modo da evitare il verificarsi del secondo caso; basta considerare, anziché l'involucro convesso dei primi i punti, il rettangolo (o un qualsiasi poligono comodo) che contiene S , in tal modo ogni nuovo punto inserito sarà sempre interno a tale rettangolo (o parallelepipedo se si lavora in tre dimensioni) e quindi si ricade sempre nel primo caso. È sufficiente poi rimuovere i triangoli esterni all'involucro convesso per ottenere la triangolazione voluta. Tale procedura semplificata si chiama Metodo Incrementale Ridotto.

Esistono inoltre altri metodi per generare una triangolazione, come quello tramite scambio di lati o l'algoritmo Sweeping, ma preferisco non appesantire la trattazione dato che non è l'obiettivo di questa tesi.

Infine, dato che gli algoritmi visti sono finalizzati alla costruzione di mesh con il calcolatore, bisogna ricordare che tutti i calcoli sono approssimati e che particolari accorgimenti sono necessari al fine di limitare propagazioni degli errori; esistono perciò degli algoritmi correttivi che rimuovono alcuni elementi malformati dovuti a errori di approssimazione.

1.5 Triangolazione vincolata

Dopo aver visto come costruire una triangolazione di un insieme di punti S , passo ora ad esaminare cosa cambia quando si aggiungono dei vincoli che la triangolazione deve rispettare; in particolare, si indicano alcuni segmenti che dovranno poi apparire come parti di elementi della triangolazione (cioè dovranno essere lati di triangoli).

Vediamo come procedere: innanzi tutto si aggiungono ad S gli estremi dei segmenti dati come vincoli, e si costruisce la triangolazione in uno dei modi visti in precedenza; successivamente, sfruttando opportuni algoritmi, si dovrà assicurare che i vincoli siano rispettati, tramite lo scambio di lati. Inoltre i vincoli posti devono essere ammissibili, e quindi soddisfare la seguente definizione:

Un segmento è Delaunay ammissibile se le celle di Voronoi associate con i suoi estremi condividono un lato.

Si può sviluppare un semplice algoritmo per verificare l'ammissibilità dei vincoli, basandosi sul seguente lemma:

Sia $a = AB$ il lato che si considera come vincolo e C_a il disco di diametro a passante per A e B . La retta che contiene a divide il piano in due semipiani, che denoto con H_a^+ e H_a^- . Il lato a si dice ammissibile se:

- C_a non contiene altri punti, oppure
- $C_a \cap H_a^+$ contiene alcuni punti, ma C_a^+ (il semidisco circoscritto al triangolo formato da a e dal punto in $C_a \cap H_a^+$) non contiene altri punti. Simmetricamente per H_a^- .

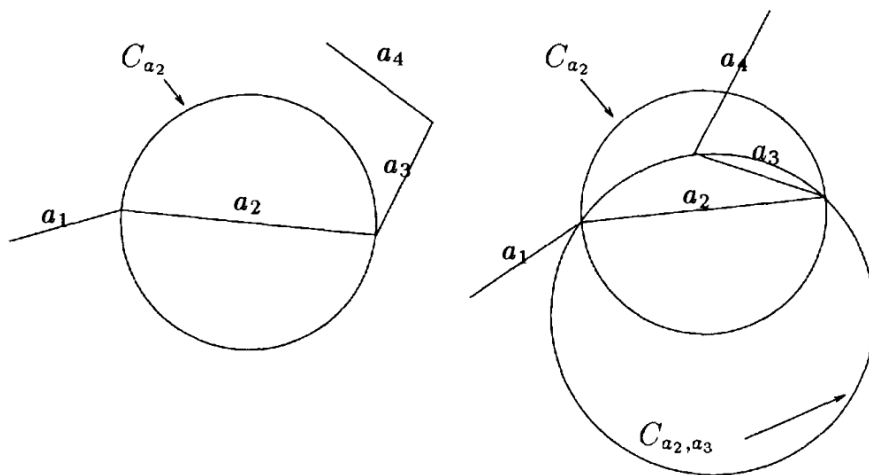


Figura 1.9: Un vincolo ammissibile (Borouchaki [1])

Per ottenere una triangolazione che soddisfa i vincoli, a partire da una generica triangolazione del dominio ci sono vari metodi:

Partizionamento dei vincoli

Ogni segmento che appartiene ai vincoli viene esaminato nel seguente modo: si trova il Tubo associato al segmento in questione; si trovano le intersezioni degli elementi del Tubo con il segmento a , (P_1, P_2, \dots) ; si sostituisce il segmento a con i segmenti $AP_1, P_1P_2, \dots, P_nB$ e si suddividono i triangoli del Tubo.

Scambio di lati

È anche possibile procedere nel seguente modo:
 si trova il Tubo associato al segmento a e si applica lo scambio di lati ad ogni lato dei triangoli intersecati da a , evitando cicli infiniti, fino a quando il segmento diventa un lato di un triangolo.

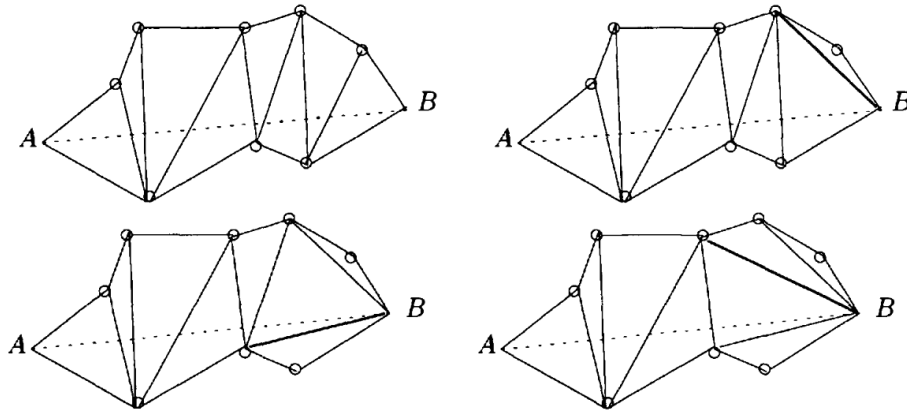


Figura 1.10: Metodo dello scambio dei lati con i primi 3 scambi (Borouchaki [1])

Forzare i vincoli

Si trova il Tubo associato ad a , dopodiché si nota che il segmento a divide il poligono che forma il tubo in 2 sotto-poligoni, per ognuno dei quali si cerca il vertice più vicino al segmento a si creano 2 triangoli che hanno il lato a in comune. Ora il poligono iniziale è stato suddiviso in 2 triangoli e 4 poligono più piccoli, a cui si può riapplicare lo stesso procedimento.

È da osservare che questi metodi non producono necessariamente una triangolazione Delaunay, perciò, dopo la loro applicazione, si procede scambiando i lati dei triangoli che non rispettano il criterio di Delaunay.

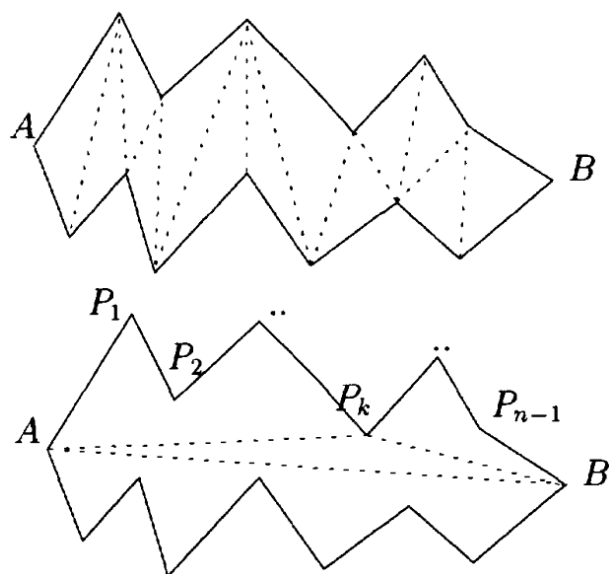


Figura 1.11: Metodo della forzatura dei vincoli (Borouchaki [1])

1.6 Triangolazione anisotropica

Fino a questo punto, la costruzione di una triangolazione avveniva in uno spazio euclideo, considerando il dominio omogeneo in ogni punto; molte applicazioni richiedono però una metrica non costante e con direzioni arbitrarie, a cui la triangolazione deve sottostare, per questo motivo si introduce una triangolazione anisotropica.

Si definisce perciò una metrica in tutto lo spazio, che fornirà sia la dimensione, sia le direzioni desiderate per gli elementi. In \mathbf{R}^2 una metrica è una matrice 2x2 definita positiva:

$$M(X) = \begin{pmatrix} a_X & b_X \\ b_X & c_X \end{pmatrix}$$

con $a_X > 0$, $b_X > 0$ e $a_X c_X - b_X^2 > 0$.

al variare del punto X nel dominio; la distanza tra 2 punti A e B sarà quindi:

$$d_M(A, B) = \int_0^1 \sqrt{{}^t y'(t) \cdot M(y) \cdot y'(t)} dt$$

dove $y(t)$ è una parametrizzazione della curva AB ($y(0) = A$, $y(1) = B$) e ${}^t y$ è l'operatore di trasposizione applicato a y .

È chiaro che questa forma generale è computazionalmente inutilizzabile, però se M è indipendente dal punto X , si ottiene:

$$\begin{aligned}d_M(A, B) &= \|AB\|_M \\ \|x\| &= \sqrt{x \cdot {}^t x} \\ x \cdot {}^t x &= {}^t x \cdot M \cdot x\end{aligned}$$

In tal modo la metrica dello spazio euclideo è il caso particolare $M = \mathbf{I}$, mentre nel caso anisotropico $M = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$

Rimane il caso in cui la metrica varia a seconda della posizione; si può superare tale ostacolo, e ricondursi al caso precedente operando una di queste 3 approssimazioni:

1. Si sostituisce localmente lo spazio di Riemann con lo spazio euclideo definito dalla metrica nel punto P considerato. Il sistema

$$\begin{cases} d_{M(P)}(O_k, P_1) = d_{M(P)}(O_k, P_2) \\ d_{M(P)}(O_k, P_1) = d_{M(P)}(O_k, P_3) \end{cases}$$

consente di trovare O_k , centro del triangolo considerato e usando $r_k = d(O_k, P_1)$ si può applicare il kernel di Delaunay

$$\frac{d_{M(P)}(O_k, P)}{r_k} < 1$$

2. Si introduce una misura relativa al punto P e al triangolo K sotto esame. Si ottiene il criterio di Delaunay seguente:

$$\alpha_{M(P)}(P, K) + \alpha_{M(P_1)}(P, K) < 2$$

Tale approssimazione è migliore della precedente, anche se più dispendiosa da punto di vista computazionale.

3. Usando la stessa idea della seconda approssimazione, si può includere non solo un punto del triangolo in esame, ma tutti e 3 i vertici, portando alla relazione:

$$\alpha_{M(P)}(P, K) + \sum_{i=1}^3 \alpha_{M(P_i)}(P, K) < 4$$

Si può dimostrare che tutte le approssimazioni portano ad un kernel Delaunay valido.

1.7 Creare una mesh

Vediamo ora come costruire una mesh, sfruttando le varie tipologie di triangolazioni fino ad ora viste.

Ricordo che una mesh è una triangolazione di un dominio definito solamente dalla discretizzazione del suo contorno.

La prima operazione nel generare una mesh, è quella di creare la mesh vuota, cioè una mesh del dominio i cui unici vertici sono solo quelli del contorno. Tale costruzione si può fare ad esempio, sfruttando il metodo incrementale (ridotto) e poi applicare delle modifiche locali (scambio di lati) per ricostruire eventuali porzioni del contorno mancanti. Successivamente si rimuovono gli elementi esterni al dominio e si ottiene così la mesh vuota. Il successivo passo sarà quello di aggiungere punti interni, detti anche punti di campo. L'inserimento di tali punti può avvenire in diversi modi e seguendo svariati criteri, ora spiegherò solamente uno dei più usati.

Creazione lungo i lati

Si esaminano i lati dei triangoli che formano la mesh vuota e se sono troppo lunghi (in relazione alla misura definita) si inseriscono punti al loro interno in modo da suddividere il lato in segmenti che si avvicinano il più possibile ad avere lunghezza unitaria (secondo la metrica). Tale procedimento potrebbe generare punti troppo vicini tra loro su lati adiacenti, per cui si filtra l'insieme dei punti che sono stati aggiunti per eliminare quelli eventualmente troppo vicini tra loro. Infine si creano i triangoli sfruttando i nuovi punti inseriti (figure 1.12 e 1.13).

1.8 Mesh di una superficie parametrica

Vorrei ora descrivere un caso particolare di mesh, dato che utilizzerò proprio questa tipologia nei test da me effettuati.

Una superficie parametrica è definita come una funzione da \mathbf{R}^2 in \mathbf{R}^3 e una mesh di tale superficie è una approssimazione della stessa con elementi semplici (triangoli). Il metodo più semplice per ottenerla è generare la mesh del dominio (in \mathbf{R}^2) e poi mappare i vertici dei triangoli con la funzione; il problema è che i triangoli generati potrebbero approssimare male la superficie, pertanto si deve ricorrere ad un controllo basato sulla seguente stima:

$$\delta_k = \sup \|\sigma(u, v) - K(u, v)\|$$

dove $K(u, v)$ è l'immagine del punto di coordinate (u, v) nel triangolo K , mentre σ è la funzione che definisce la superficie.

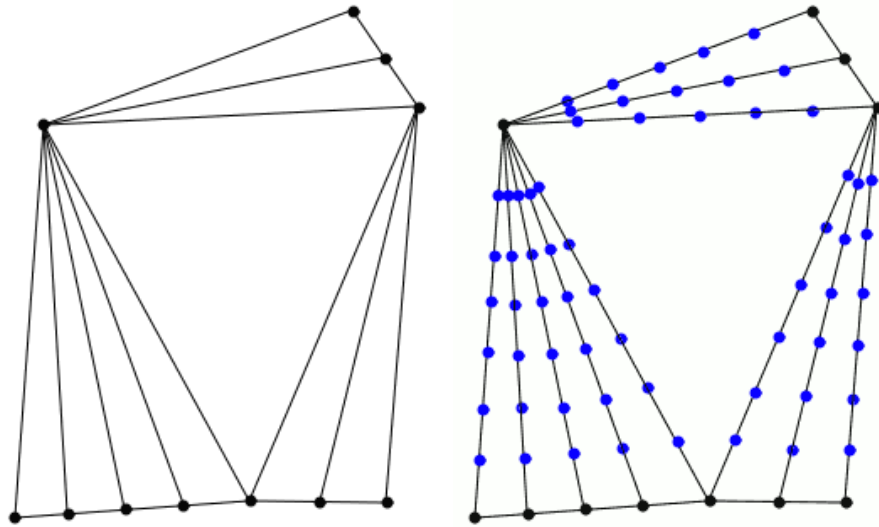


Figura 1.12: Mesh vuota e mesh subito dopo l’inserimento dei punti di campo (in blu)

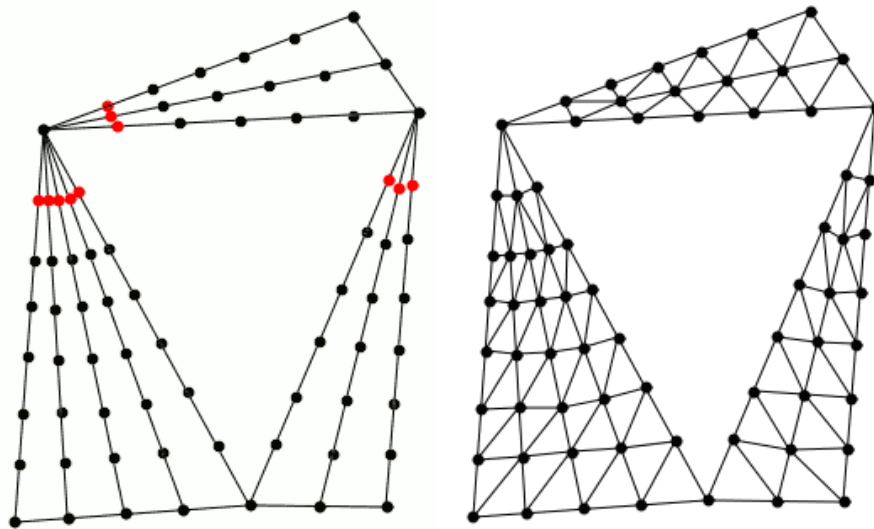


Figura 1.13: In rosso sono evidenziati i punti di campo troppo vicini tra loro. A destra la mesh dopo la loro rimozione.

Si tratta di una “distanza” del triangolo dalla superficie ed è tale quantità che si cerca di minimizzare. Si possono generare altre stime, ma sono sempre legate alla curvatura della superficie considerata.

1.9 Ottimizzazioni

In molte applicazioni, come il calcolo agli elementi finiti, è importante che la mesh soddisfi certi requisiti di qualità, che potrebbero non essere soddisfatti subito dopo la costruzione della stessa; per questo motivo si usa un algoritmo di ottimizzazione, il quale cerca di massimizzare una delle misure di qualità viste nel primo paragrafo. È altresì possibile includere direttamente la procedura di ottimizzazione nell'algoritmo di generazione della mesh. L'ottimizzazione fonda su alcune operazioni basilari, che si possono suddividere in topologiche e geometriche: le prime modificano le connessioni dei vertici dei triangoli, le seconde modificano la loro posizione.

1.10 Mesh del contorno

L'ultima parte di questo capitolo è dedicata alla costruzione del contorno discreto per la creazione di una mesh. Nel caso bidimensionale, si vuole approssimare una curva, che delimita il dominio della mesh, con una poligonale il più possibile vicino alla curva di partenza.

Il modo più banale di ottenere questo risultato è di partizionare uniformemente la curva con un gran numero di punti, per poi unirli e generare la linea poligonale; tale metodo porta però alla generazione di un gran numero di segmenti, spesso inutili se la curvatura è molto bassa, oppure insufficienti a curvature elevate.

Un modo semplice per approssimare meglio la curva è quello di fornire un valore di soglia, ε , a cui dovranno sottostare i segmenti secondo la relazione $\frac{h}{d} \leq \varepsilon$ dove h è la distanza massima tra il segmento e la curva, e d è la lunghezza del segmento (figura 1.14).

Lo stesso concetto si ripresenta nell'approssimazione di una superficie tridimensionale con elementi triangolari o quadrangolari. Si estende la definizione precedente in modo da approssimare il pezzetto di superficie interessato con un prisma a base triangolare, in cui il rapporto $\frac{h}{Area} \leq \varepsilon$. Altre stime si possono basare sulla distanza di alcuni punti del triangolo (baricentro, vertici) con la superficie.

Ho utilizzato questo tipo di stime negli esempi di utilizzo della libreria Mesh (pag 31).

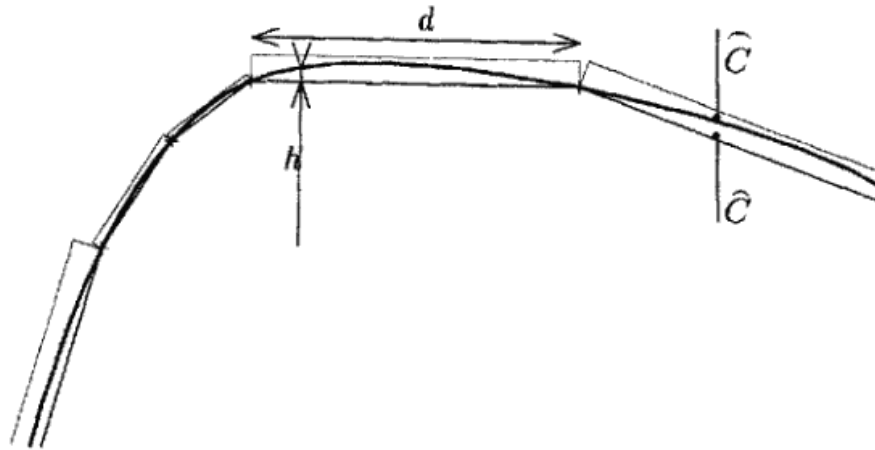


Figura 1.14: Approssimazione di porzioni di curva con un segmento (Borouchaki [1])

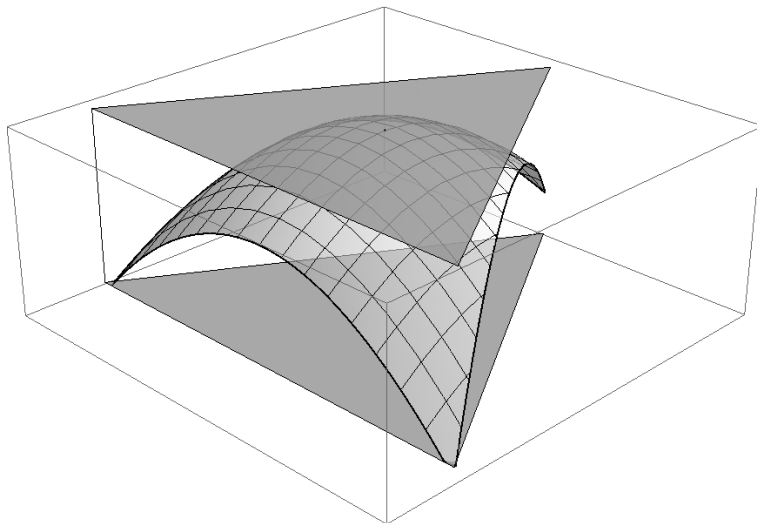


Figura 1.15: Prisma triangolare che contiene una porzione di superficie da approssimare

Capitolo 2

Generazione automatica di una mesh gerarchica

2.1 Il problema

Il problema che mi sono proposto di risolvere con questo progetto è quello di trovare un modo efficiente di sfruttare una mesh, al fine di risolvere problemi evolutivi, cioè problemi la cui soluzione varia nel tempo. Un classico problema di questo tipo è la risoluzione dell'equazione del calore, per la quale si vogliono trovare le superfici isoterme per un dato dominio bidimensionale, che risulteranno essere superfici variabili nel tempo; tali superfici vengono poi analizzate per ulteriori calcoli, che vengono appunto eseguiti sulla mesh che approssima la superficie.

L'obiettivo è quello di ottenere una mesh sufficientemente fitta, per rispettare la tolleranza sull'errore di approssimazione del problema definito nel continuo, ma allo stesso tempo è necessario che la generazione di tale mesh sia effettuata in modo veloce (possibilmente in tempo reale).

Ci sono due approcci semplici al problema: il primo è quello di generare una mesh statica che sia precisa ovunque nel suo dominio, in modo da garantire la precisione desiderata per tutta la durata del processo. Come conseguenza, però, si genera un'enorme quantità di elementi (punti e triangoli) che non sono necessari ad ogni singolo passo dell'iterazione, con enorme spreco di memoria e tempo computazionale.

Un'alternativa è quella di generare la mesh ottimale ad ogni passo e questo risolve lo spreco di memoria, ma solleva un ulteriore problema: la generazione di una mesh è un'operazione molto costosa partendo da zero, e utilizzerebbe troppo tempo, anche perché l'evoluzione della soluzione rende tipicamente necessario solo un cambiamento parziale della mesh.

Una buona soluzione, che nasce dal compromesso delle due strategie precedenti, è l'utilizzo di una mesh gerarchica, ovvero una mesh con una struttura a strati, in cui ogni strato contiene elementi più raffinati di quelli sottostanti. L'efficacia di questo tipo di struttura risiede nel fatto che quando si vuole modificare la mesh non serve generarla nuovamente, ma è sufficiente rimuovere gli strati più alti e poi procedere con il raffinamento aggiungendo nuovi strati dove serve (Marcuzzi [3]).

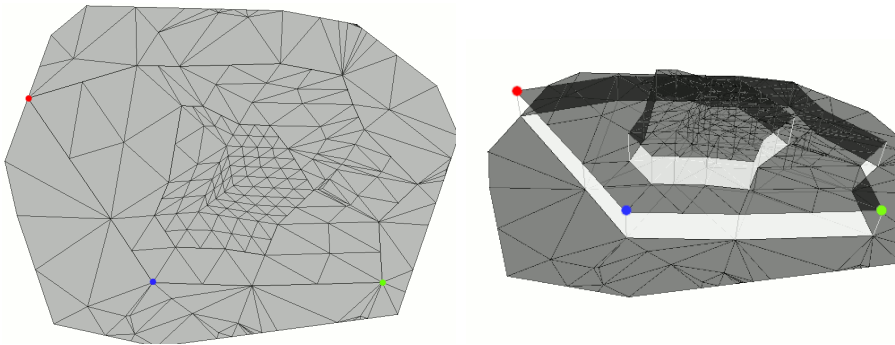


Figura 2.1: Rappresentazione di una mesh gerarchica. A sinistra la vista “appiattita” dall’alto, a destra la vista a strati. I puntini colorati indicano i punti corrispondenti, per maggior chiarezza.

Il lavoro che ho svolto è stato quello di scrivere una libreria che gestisce appunto una mesh di questo tipo; nel prossimo paragrafo ne descrivo il funzionamento.

2.2 La libreria

La libreria che ho sviluppato si occupa della gestione della mesh gerarchica sopra descritta. La libreria non è in grado di generare una mesh autonomamente e per far ciò utilizza la libreria Triangle [4], la quale produce la mesh grezza iniziale, sulla quale poi vengono eseguiti i raffinamenti. Uno schema dei moduli che la compongono è il seguente:

Modulo	Descrizione
libTriangle	libreria di supporto, per la generazione della mesh iniziale
libMesh	la libreria che gestisce la mesh gerarchica
classi Punto, Triangolo, Strato	classi usate internamente per rappresentare i vari elementi
showmesh	Visualizzatore della mesh

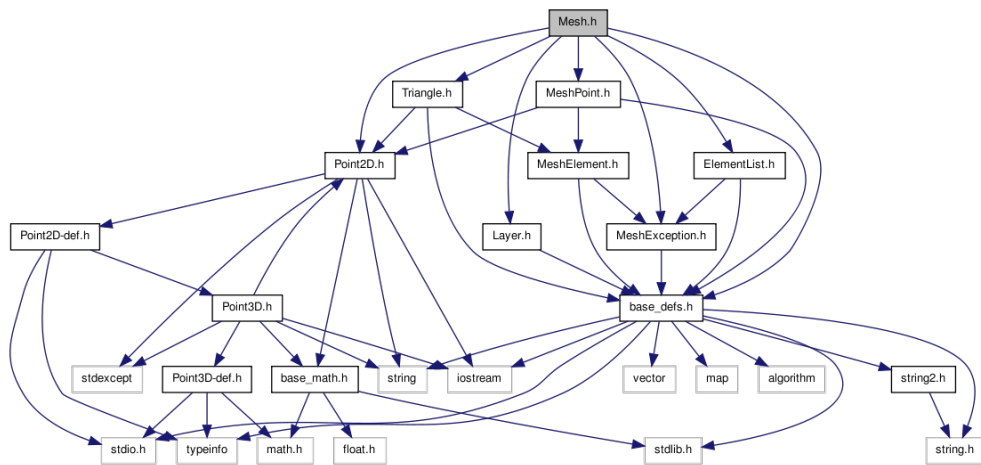


Figura 2.2: Grafo di inclusione dei vari componenti della libreria. La documentazione completa è disponibile in formato HTML.

2.2.1 Caratteristiche

Le mesh che è possibile gestire sono bidimensionali e tutti i vertici degli elementi sono definiti da due coordinate. Ciononostante è possibile associare ad ogni elemento delle quantità numeriche che possono rappresentare altre qualità fisiche o geometriche. Gli elementi costituenti sono solamente triangolari e sono descritti tramite i 3 vertici. Inoltre ogni elemento, punto o triangolo, appartiene ad uno o più strati (questo verrà spiegato in dettaglio nei prossimi paragrafi), e contiene informazioni su quali elementi stanno sotto e sopra di esso, affinché sia poi possibile la rimozione degli strati.

Ora vorrei descrivere alcuni aspetti che sono stati cruciali nello sviluppo di tale libreria.

Soppressione dei lati

Nonostante sembrerebbe fondamentale tenere in memoria tutta la gerarchia di elementi (punti, lati, triangoli), ciò aumenta considerevolmente la complessità della libreria quando si fanno delle operazioni complesse sulla mesh, come la rimozione o l'appiattimento di più strati, perchè bisogna aggiornare ogni elemento per evitare inconsistenze. Rimuovendo del tutto le strutture per i lati, si semplifica molto il codice e si riduce il tempo di esecuzione per la maggior parte delle operazioni. Proprio per questo motivo non ho incluso nessuna struttura apposita per rappresentare i lati.

Memoria condivisa

La struttura della mesh necessita di tenere in memoria una serie di liste di elementi (punti, triangoli), per ogni strato e anche nel caso si lavori con più mesh contemporaneamente. Per ridurre al massimo l'utilizzo di memoria, queste liste non memorizzano gli elementi ogni volta, ma solamente un puntatore ad essi, in modo da non avere elementi duplicati in memoria. Gli elementi, a loro volta, contengono un puntatore alle liste che li contengono e possono essere rimossi dalla memoria solo quando nessuna lista li utilizza.

Diffusione delle proprietà

Un concetto chiave, utile in numerose operazioni che si possono eseguire sulla mesh, è quello di diffusione delle proprietà. Si tratta di propagare una certa proprietà a triangoli adiacenti, in modo ricorsivo. Questo meccanismo viene usato ad esempio per raffinare una mesh partendo da un certo triangolo, restando dentro un dominio assegnato. Il codice che realizza questo algoritmo è riportato nell'appendice A (pagina 41).

Unificazione degli strati

Una delle operazioni che è possibile svolgere sulla mesh gerarchica è la fusione di più strati adiacenti in uno solo. Questo pone il problema di scorrere la mesh verticalmente, cioè tra uno strato e l'altro. Per ovviare a ciò ogni triangolo contiene l'indice dell'elemento che nasconde (che sarà uno solo, dato che gli elementi sovrastanti sono più piccoli) e ogni punto contiene la lista di triangoli di cui è vertice (i punti non vengono nascosti). Il codice che realizza questo algoritmo è riportato nell'appendice A (pagina 43).

2.2.2 Esempio di utilizzo

Riporto ora un semplice esempio di come utilizzare la libreria.

```
#include "Mesh.h"
#include "Point2D.h"

// Funzione che controlla la qualità di un triangolo
// Vedere sotto per i dettagli
int if_func(const Triangle &tr, void *p);

int main() {

    Mesh m;                // oggetto Mesh

    Point2Dd p1(0,0);      //
    Point2Dd p2(0,1);      //
    Point2Dd p3(1,0);      // 4 punti nel piano
    Point2Dd p4(1,1);      //

    m.add_point(p1, 0);    //
    m.add_point(p2, 0);    // aggiungo i punti alla mesh
    m.add_point(p3, 0);    // nel livello 0
    m.add_point(p4, 0);    //

    m.generate("QDB");     // genero la mesh, vedere la
                           // documentazione di Triangle per
                           // i parametri

    m.refine();            // rifinisce la mesh con l'algoritmo
                           // standard creando un nuovo strato

    m.save_mesh("file.m"); // salva la mesh su file

    m.refine_if(if_func);  // rifinisce la mesh solo per i
                           // triangoli per cui if_func
                           // restituisce non-zero
                           // vedere sotto per una funzione di
                           // questo tipo

    m.remove_last_layer(); // rimuove lo strato più alto
}
```

30CAPITOLO 2. GENERAZIONE AUTOMATICA DI UNA MESH GERARCHICA

```
    return 0;
}

// controlla se il triangolo tr necessita raffinazione
int if_func(const Triangle &tr, void *p) {

    // calcola l'area del triangolo

    double area = tr.get_area_const();

    // se l'area è maggiore di 0.1, restituisce un valore che
    // indica come rifinire il triangolo

    if (area > 0.1) return MESH_REFINE_1;

    // altrimenti non è necessaria la raffinazione
    return 0;
}
```

Il programma sopra riportato non ha alcuna effettiva utilità, vuole solo essere un esempio su come utilizzare la libreria concretamente. Nella directory contenente i sorgenti sono disponibili altri due esempi, che verranno discussi in dettaglio nel prossimo capitolo.

Capitolo 3

Risultati

In questo capitolo riporto alcune simulazioni fatte con la libreria Mesh, che ne mostrano la migliore efficienza nella risoluzione di alcuni problemi di test, come il calcolo dell'area di una superficie.

3.1 Esempio 1

Questo esempio mira a verificare la convergenza della procedura di raffinamento per il calcolo dell'area di una superficie all'aumentare del numero degli elementi. La superficie scelta è la seguente:

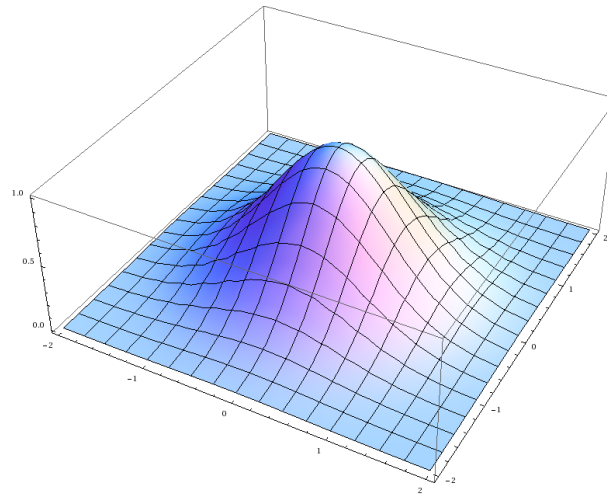
$$\Sigma : f(x, y) = e^{-(x^2+y^2)}$$

$$\text{nel dominio } \Omega = [-2, 2] \times [-2, 2]$$

La cui area approssimata è 17.4148141.

Si vuole calcolare l'area della superficie Σ tramite una mesh del suo dominio. Il test viene prima eseguito con una mesh che si infittisce ovunque e successivamente con una mesh che viene raffinata solamente dove l'errore supera una certa soglia. Tale approccio è possibile perchè si conosce l'area esatta da calcolare, ma in problemi reali la soluzione non è nota a priori e pertanto si deve ricorrere a stime di errore.

Qui non sto sfruttando le peculiarità della mesh gerarchica, il cui vantaggio sta proprio nel poter essere ripulita rimuovendo strati, ma sto mostrando come sia possibile utilizzare una mesh di questo tipo anche per applicazioni tradizionali.

Figura 3.1: La superficie Σ

Lo pseudocodice di questo test è il seguente¹.

```
// Prima parte - mesh omogenea

imposto numero di punti iniziali = 4
genero la mesh iniziale
per i da 1 a Iterazioni_massime fai:
    calcolo errore sull'area
    se errore sotto la soglia esci dal ciclo
    altrimenti rifinisco la mesh ovunque
fineciclo

// Seconda parte - mesh adattiva

imposto numero di punti iniziali = 4
genero la mesh iniziale
per i da 1 a Iterazioni_massime fai:
    calcolo errore sull'area
    se errore sotto la soglia esci dal ciclo
    altrimenti rifinisco la mesh dove i triangoli
        hanno una bassa qualità
fineciclo
```

¹Questo esempio è contenuto nel file test1_area.cpp

Si vuole raggiungere un affinamento tale che l'errore percentuale non superi lo 0.01%.

I risultati ottenuti sono i seguenti:

	Mesh omogenea	Mesh adattiva
Iterazioni	6	6
Punti	8281	2859
Triangoli	16200	5504
Memoria	3316.24 Kb	1390.92 Kb
Area	17.4142	17.4134
Errore area	-0.003579%	-0.007924%
Tempo	94ms	47ms

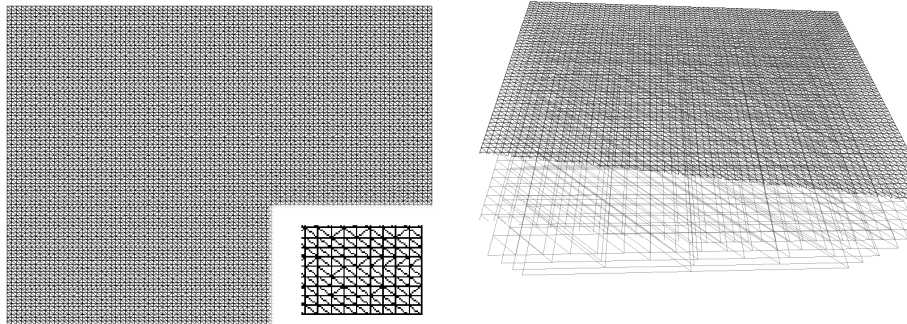


Figura 3.2: Rappresentazione della mesh omogenea. A sinistra visualizzazione dall'alto, con ingrandimento di un particolare, a destra è messa in evidenza la struttura a strati

Come si nota il numero di elementi è nettamente inferiore nel secondo caso, come si può vedere anche dal confronto visivo tra le 2 mesh. Ciò riduce di conseguenza il tempo di calcolo e l'utilizzo di memoria, mantenendo comunque la soglia di errore sotto al valore richiesto.

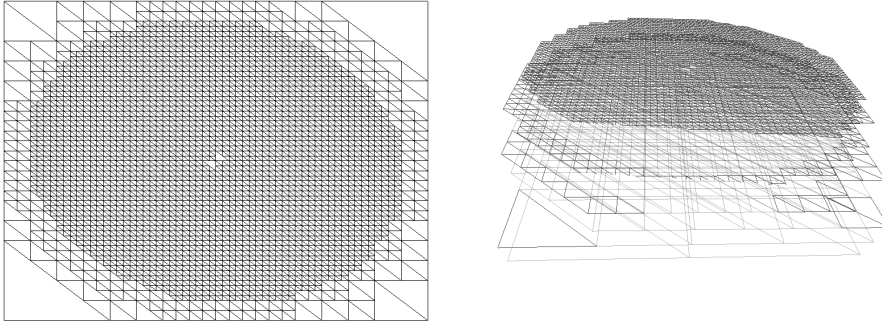


Figura 3.3: Rappresentazione della mesh adattiva. A sinistra visualizzazione dall'alto, a destra si vede la struttura a strati.

3.2 Esempio 2

In questo esempio applico la mesh ad una superficie appositamente costruita che è dipendente dal tempo. Rappresenta la soluzione di un problema temporale.

$$\Sigma : f(x, y) = \sqrt[3]{\arctan \sqrt[3]{x - t}}$$

nel dominio $\Omega = [-2, 2] \times [-2, 2]$
e con $t \in [-1, 1]$

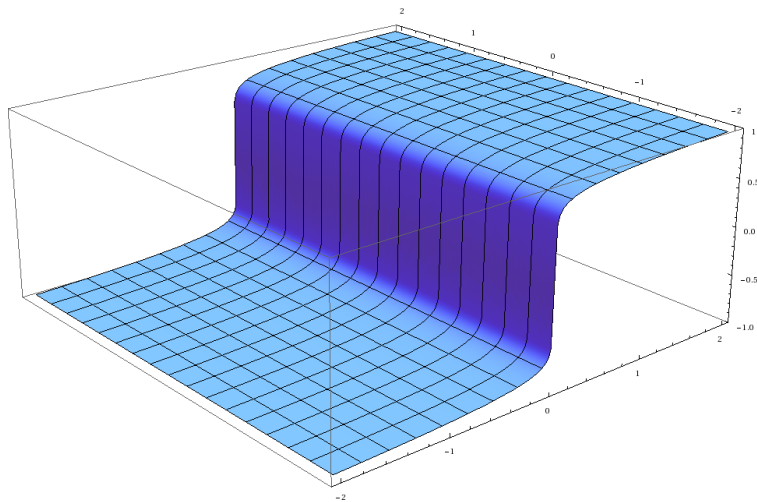


Figura 3.4: Superficie Σ , per $t = 0.1$

L'obiettivo di questo esempio è quello di generare una mesh per la quale l'errore di approssimazione è sotto una certa soglia. Tale errore è definito come:

$$Err = \max_{i=1\dots N} \|B(f(T_i)) - f(B(T_i))\| \quad (3.1)$$

Dove:

N = numero di triangoli visibili della mesh.

$B()$ = operatore che calcola il baricentro di un triangolo.

$f()$ = la funzione definita sopra, che mappa un punto/triangolo nel suo corrispondente.

T_i = triangolo i -esimo.

Anche in questo caso il raffinamento avviene usando tre metodi diversi. Il primo è semplicemente un raffinamento della mesh uniforme in tutto il dominio, fino a raggiungere la soglia richiesta (in questo caso 0.4).

Il secondo metodo cerca di risparmiare memoria raffinando la mesh solo dove è necessario, ma senza mai rimuovere triangoli. Ciò viene ottenuto raffinando solo i triangoli per i quali l'errore è maggiore della soglia richiesta.

Nell'ultimo caso sfrutto proprio la caratteristica della stratificazione della mesh, rimuovendo gli ultimi strati prima di passare al successivo raffinamento.

Lo pseudocodice dei tre casi è il seguente².

```
// Prima parte - mesh omogenea

imposto numero di punti iniziali = 4
genero la mesh iniziale
per tempo da -1 a +1 fai:
  calcolo errore della mesh
  se errore sopra la soglia:
    per i da 1 a Iterazioni_massime fai:
      rifinisco la mesh
      calcolo errore della mesh
      se errore sotto la soglia esci dal ciclo
    fineciclo
  fineciclo
```

²Questo esempio è contenuto nel file test2.quality.cpp

```
// Seconda parte - mesh incrementale

imposto numero di punti iniziali = 4
genero la mesh iniziale
per tempo da -1 a +1 fai:
  calcolo errore della mesh
  se errore sopra la soglia:
    per i da 1 a Iterazioni_massime fai:
      rifinisco la mesh dove necessario
      calcolo errore della mesh
      se errore sotto la soglia esci dal ciclo
    fineciclo
fineciclo

// Terza parte - mesh gerarchica

imposto numero di punti iniziali = 4
genero la mesh iniziale
per tempo da -1 a +1 fai:
  calcolo errore della mesh
  se errore sopra la soglia:
    per i da 1 a Iterazioni_massime fai:
      rifinisco la mesh dove necessario (creando nuovi strati)
      calcolo errore della mesh
      se errore sotto la soglia esci dal ciclo
    fineciclo
  rimuovo gli strati aggiunti
fineciclo
```

I risultati ottenuti sono i seguenti:

	Mesh omogenea	Mesh incrementale	Mesh gerarchica
Cicli	6	6	6
Iterazioni	5	17	34
Punti			
media	8281	2488	687
picco	8281	5343	896
Triangoli			
media	16200	4136	1122
picco	16200	8672	1508
Memoria:			
media	3316.24 Kb	1029.43 Kb	416.18 Kb
picco	3316.24 Kb	2044.83 Kb	510.56 Kb
Errore medio	0.369	0.377	0.377
Tempo	251ms	317ms	41ms

Le mesh generate nei tre casi si presentano così

Primo caso

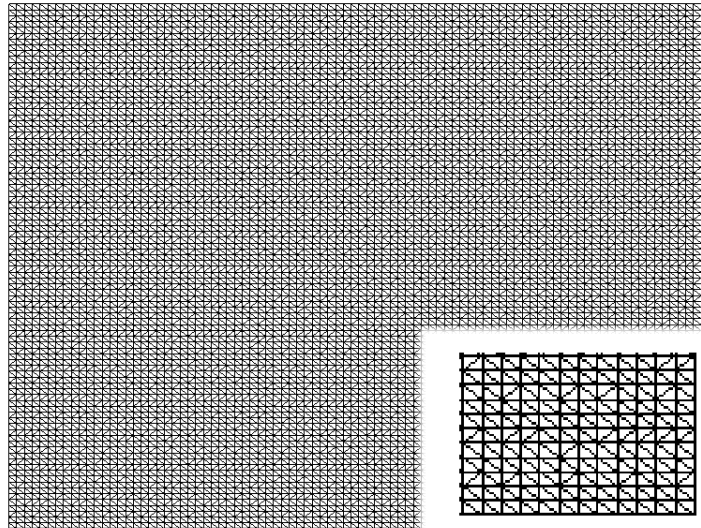
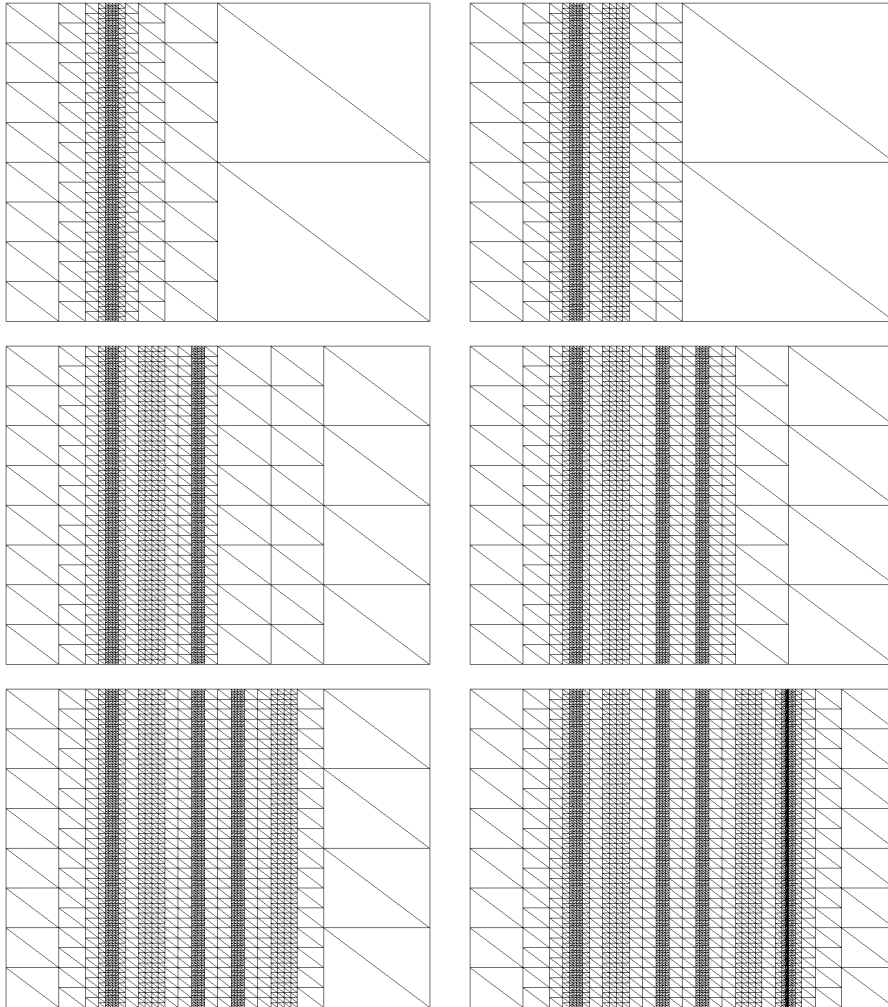


Figura 3.5: La mesh generata, una volta raggiunta la precisione richiesta, non viene più modificata

Secondo caso

Figura 3.6: La mesh generata evolve nel tempo, seguendo la curva Σ

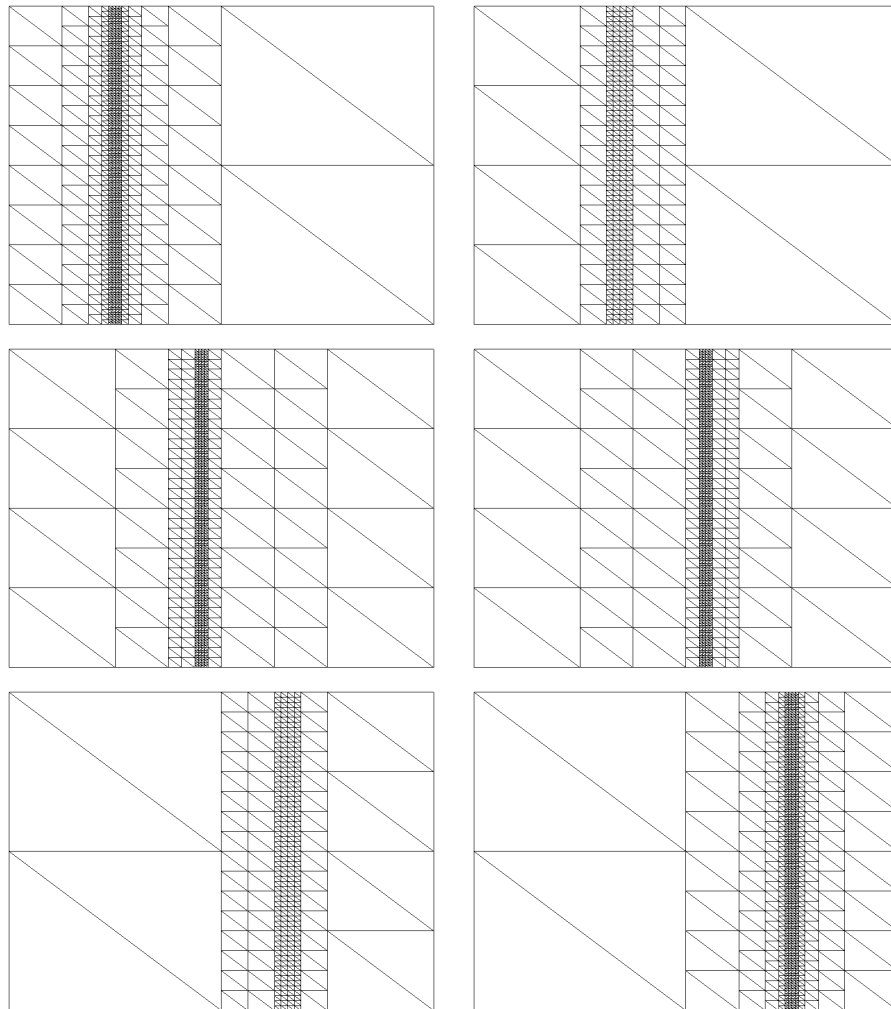
Terzo caso

Figura 3.7: Anche in questo caso la mesh evolve nel tempo, ma le parti in cui non è necessario un gran numero di elementi vengono deraffinate.

Come si nota, nell'ultimo caso, il numero massimo di punti utilizzati è solamente il 16% del caso precedente e l'11% del primo caso. Inoltre anche il tempo di calcolo è drasticamente inferiore; ad ogni passo temporale, infatti, vengono rimossi gli strati precedentemente aggiunti, che contengono triangoli piccoli in zone dove non sono necessari.

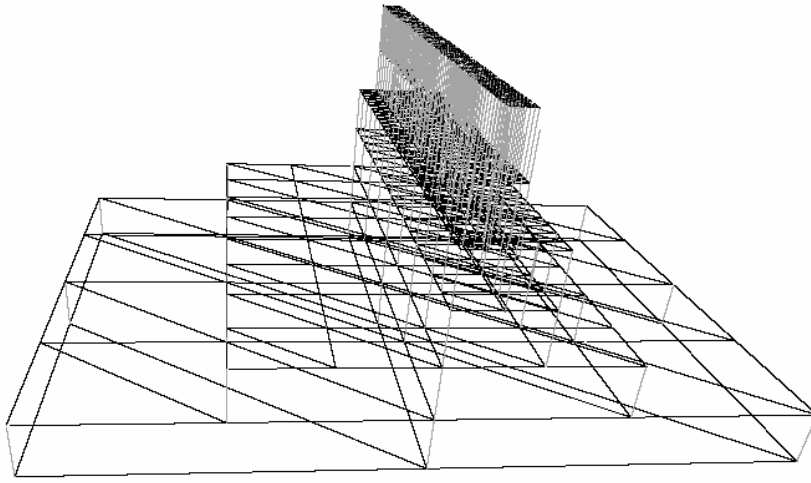


Figura 3.8: Rappresentazione 3D della mesh gerarchica nel terzo caso

***Nota:** le figure delle mesh presenti in questo capitolo sono state generate con il visualizzatore di mesh distribuito con la libreria.*

Capitolo 4

Appendice A

Riporto qui alcune porzioni di codice della libreria che sono di particolare interesse. Si vedano i riferimenti nel testo per una loro spiegazione.

4.1 Diffusione delle proprietà

La seguente porzione di codice rappresenta l'algoritmo di diffusione delle proprietà, che in questo caso ho chiamato *virus*, in relazione a questo tipo di propagazione “virale”.

```
int Mesh::start_virus(  
    // indice del triangolo da cui partire  
    int startidx,  
  
    // funzione che decide se un triangolo deve essere infettato  
    bool (*validate)(const Triangle &, const Triangle &, void *),  
  
    // puntatore generico da passare a validate  
    void *p,  
  
    // vettore in cui mettere i triangoli infettati  
    vector<int> *list)  
{  
  
    // verifico il valore iniziale del virus  
    if (virus_value == 0) {  
        // azzero il valore
```

```

    virus_value = 1;
    // applico a tutti i triangoli
    LIST_ITERATE(triangles, i)
        get_triangle(i).set_prop(0);
    LIST_ITERATE_END
}

int ret = 0;

// verifico se il triangolo può essere infettato
// usando validate() come funzione di controllo
if (validate != NULL &&
    validate(get_triangle(startidx), get_triangle(startidx), p))
{

    // diffonde il virus
    ret = spread_virus(startidx, validate, p, list);

    // incremento il valore del virus per non avere interferenze
    // con successive infezioni
    ++virus_value;
}

return ret;
}

int Mesh::spread_virus(
    int startidx,
    bool (*validate)(const Triangle &, const Triangle &, void *),
    void *p,
    vector<int> *list) {

    // controllo se il triangolo è già stato infettato
    Triangle &t = get_triangle(startidx);
    if (t.get_prop() == virus_value) return 0;

    int count = 0;

    // infetto il triangolo esaminato
    t.set_prop(virus_value);

```

```

// e lo aggiungo alla lista degli infetti se è possibile
if (list != NULL) list->push_back(startidx);
++count;

// calcolo i triangoli adiacenti
vector<int> n;
get_triangle_neighbours(startidx, n);

// infetto gli adiacenti
for (int i=0; i<n.size(); ++i) {
    if (validate != NULL &&
        validate(t, get_triangle(n[i]), p))
    {
        count += spread_virus(n[i], validate, p, list);
    }
}

return count;
}

```

4.2 Unificazione degli strati

Espongo di seguito l'algoritmo di unificazione degli strati, che fonde più strati in uno solo.

```

void Mesh::flatten(

    // livello di partenza
    int startl,

    // livello finale
    int endl) throw(MeshException) {

    // controllo sui dati iniziali
    if (endl == -1) endl = get_n_layers()-1;
    if (startl < 0 || startl >= get_n_layers() || endl < startl)
        throw MeshException("[Mesh::flatten] "MESH_EXCEPTION_LAYER);
    if (endl == startl) return;

```

```

// ciclo su tutti i triangoli dello strato più basso
for (int i=layers[startl].get_n_triangles()-1; i>=0; --i) {

    // se è un triangolo nascosto lo ignoro
    int tid = layers[startl].get_triangle(i);
    Triangle &t = get_triangle(tid);
    if ( !t.is_hidden()) continue;

    // calcolo i triangoli sopra di esso, fino allo strato endl
    vector<int> triabove;
    get_triangles_above(tid, triabove, endl);

    // ciclo sui triangoli soprastanti
    for (int j=0; j<triableve.size(); ++j) {
        int aid = triabove[j];
        Triangle &at = get_triangle(aid);

        // è sull'ultimo strato da fondere
        // e quindi lo "schiaccio" sullo strato startl
        if (at.get_layer() == endl) {
            at.set_layer(startl);
            at.set_birth_layer(startl);
            at.set_hidden_element(t.get_hidden_element());

            // è in uno strato intermedio
        } else {

            // se sopra di esso non ci sono triangoli
            if (at.get_n_above_triangles() == 0) {
                at.set_layer(startl);
                at.set_birth_layer(startl);
                at.set_hidden_element(t.get_hidden_element());

                // altrimenti lo cancello
            } else {
                del_triangle(aid);
            }
        }
    }
}

// se il triangolo ha degli elementi sopra di lui

```

```

    // lo cancello
    if (t.get_n_above_triangles() > 0) del_triangle(tid);
}

// ricalcolo le appartenenze agli strati
int nlayers = get_n_layers();
int layersafter = nlayers-endl-1;
for (int i=nlayers-1; i>=startl; --i) layers.pop_back();
for (int i=0; i<=layersafter; ++i) layers.push_back(Layer());

// aggiusto gli strati per i punti sopra lo strato endl
int deltalayer = endl-startl;
LIST_ITERATE(points, i)
    MeshPoint &p = get_point(i);
    if (p.get_layer() < startl) continue;
    if (p.get_birth_layer() >= startl) {
        if (p.get_birth_layer() <= endl)
            p.set_birth_layer(startl);
        else
            p.set_birth_layer(p.get_birth_layer()-deltalayer);
    }
    if (p.get_layer() <= endl)
        p.set_layer(startl);
    else
        p.set_layer(p.get_layer()-deltalayer);
    int nl = p.get_birth_layer();
    layers[nl].increase_points();
LIST_ITERATE_END

// aggiusto gli strati per i triangoli sopra lo strato endl
LIST_ITERATE(triangles, i)
    Triangle &t = get_triangle(i);
    if (t.get_layer() < startl) continue;
    if (t.get_birth_layer() >= startl) {
        if (t.get_birth_layer() <= endl)
            t.set_birth_layer(startl);
        else
            t.set_birth_layer(t.get_birth_layer()-deltalayer);
    }
    if (t.get_layer() <= endl)
        t.set_layer(startl);

```

```
    else
        t.set_layer(t.get_layer()-deltalayer);
    int nl = t.get_birth_layer();
    layers[nl].add_triangle(i);
    if (t.is_hidden()) layers[nl].increase_hidden_triangles();
LIST_ITERATE_END
}
```

Bibliografia

- [1] P. L. George, H. Borouchaki *Delaunay Triangulation and Meshing*, 1998
- [2] G. F. Carey *Computational Grids*, 1997 articolo su mesh gerarchiche
- [3] F. Marcuzzi, M. Morandi Cecchi, M. Venturin *An anisotropic unstructured triangular adaptive mesh algorithm based on error and error gradient information*, 2008
- [4] J. R. Shewchuk, *Triangle - A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator* (source code)
<http://www.cs.cmu.edu/~quake/triangle.html>