POLITECNICO DI TORINO

Corso di Laurea
in Matematica per l'Ingegneria

Programmazione e Calcolo Scientifico

Raffinamento di una Mesh Triangolare

Vittoria Drago 283897, Anna Roma 285951, Francesca Coriale 283044

Anno Accademico 2022-2023

Indice

Elenco delle figure

Capitolo 1

Raffinamento

Il metodo della **bisezione del lato più lungo** è una tecnica utilizzata per dividere una mesh triangolare attraverso la suddivisione del lato più lungo di ogni triangolo.
Il procedimento di base per applicare questo metodo consiste nel considerare il triangolo T della mesh e effettuare le seguenti operazioni:

1. rintracciare il lato e^T più lungo del triangolo T;

2. calcolare il punto medio M_{e^T} del lato e^T e unirlo al vertice $V_{e^T}^T$ opposto al lato e^T.

In questo modo il triangolo T viene diviso in due sotto-triangoli T_1,T_2. Successivamente, è necessario rendere la triangolazione ammissibile cercando il triangolo S adiacente al lato e^T e raffinarlo unendo il nuovo punto creato M_e^T al vertice $V_{e^T}^S$ del triangolo S opposto al lato e^T.

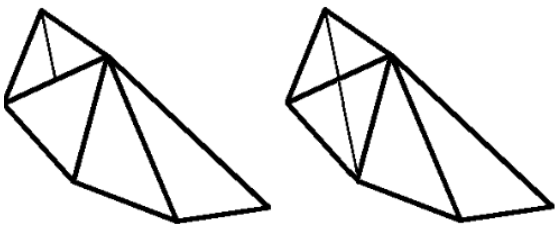

Figura 1.1.

1.1 Plant UML

Per l'implementazione di questo algoritmo è stata utilizzata la programmazione a oggetti. Di seguito si riporta la rappresentazione UML del codice.

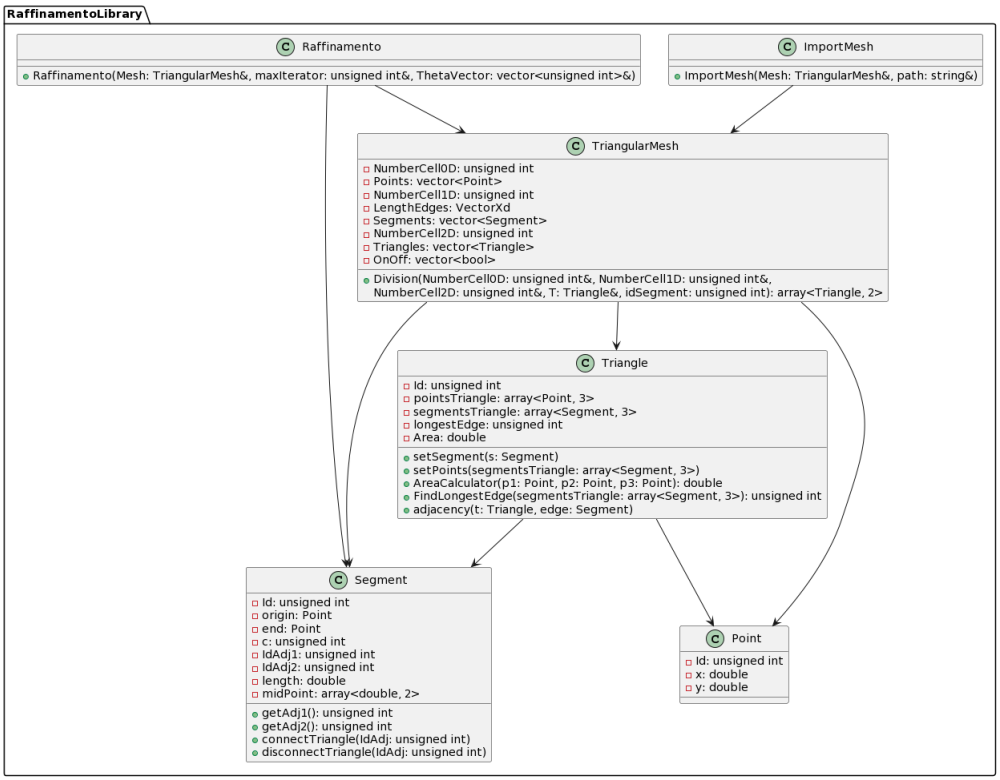

Figura 1.2.

Come si vede dalla figura sono state create 6 classi:

1. Point;
2. Segment;
3. Triangle;
4. TriangularMesh;
5. ImportMesh;
6. Raffinamento.

Capitolo 2

Descrizione classi e funzionamento del codice

2.1 Point

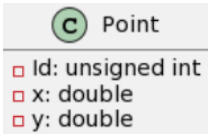

Figura 2.1. Classe Point

La classe Point, come suggerito dal nome stesso rappresenta un punto nello spazio. Contiene 3 attributi pubblici:

1. l'**id** del punto,dato di tipo *unsigned int*;
2. la sua ascissa **x**, dato di tipo *double*;
3. la sua ordinata **y**, dato di tipo *double*.

L'unico costruttore della classe è denominato **"Point"**, è inizializzato con valori di default e costruisce un oggetto di tipo Point prendendo in input tutti gli attributi della classe.
All'interno della classe viene definito l'operatore di confronto **"=="**. Questo operatore viene utilizzato per confrontare due oggetti Point e determinare se sono uguali confrontando solo il membro Id degli oggetti. Restituisce *true* se gli oggetti hanno lo stesso valore per il membro Id, altrimenti restituisce *false*.

2.2 Segment

Un'istanza della classe segment rappresenta un segmento nello spazio.
Vengono definiti i seguenti attributi pubblici:

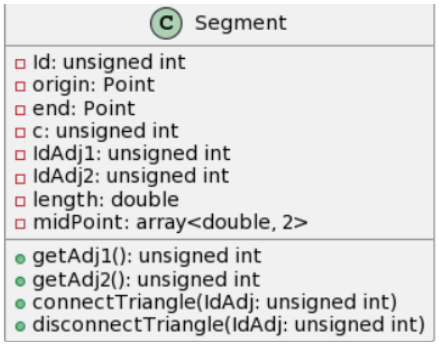

Figura 2.2. Classe Segment

1. l'**Id** del segmento, dato di tipo *unsigned int*;
2. l'**origin** del segmento, dato di tipo Point;
3. l'**end** del segmento, dato di tipo Point.
4. **c**, dato di tipo unsigned int che verrà utilizzato nelle funzioni disconnectTriangle e connectTriangle;
5. **IdAdj1,IdAdj2**, dati di tipo *unsigned int* che memorizzeranno l'Id dei triangoli adiacenti;
6. **midPoint**, array di dati di tipo double di dimensione 2 pari alle coordinate del punto medio del segmento.

All'interno di questa classe viene definito un costruttore denominato **Segment** che, dopo essere stato inizializzato a un valore di default, costruisce un segmento prendendo in input variabili di tipo *const* corrispondenti agli attributi pubblici definiti nella classe. All'interno della classe vengono costruiti i seguenti metodi:

1. **LenghtEdge** che, prendendo in input dati di tipo Point (origin, end) calcola la lunghezza del lato sfruttando la seguente formula:
lenghtEdge=(end.x-origin.x, end.y-origin.y).norm
2. **getAdj1** e **getAdj2** che restituiscono dati di tipo *unsigned int* pari agli Id dei due triangoli a cui appartiene il segmento.

Inoltre, vengono create anche due funzioni di tipo void:

1. **connectTriangle**;
2. **disconnectTriangle**.

Entrambe consentono rispettivamente di connettere e disconnettere i segmenti ai triangoli a cui appartengono tramite l'utilizzo di una variabile **c** di tipo *unsigned int* che aumenta o diminuisce il suo valore per memorizzare o eliminare i triangoli adiacenti.

2.3 Triangle

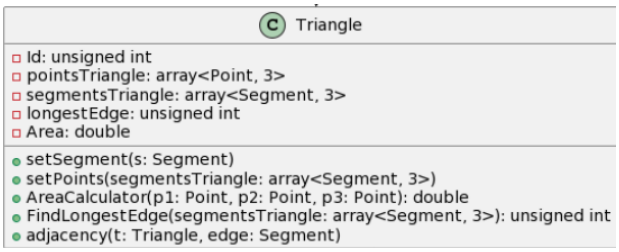

Figura 2.3. Classe Triangle

La classe Triangle è il modello che descrive le proprietà e le operazioni comuni a tutti i triangoli. É formata da attributi pubblici:

1. **id**, dato di tipo *unsigned int*;
2. **pointsTriangle**, array di tre dimensioni formato da dati di tipo Point contenenti le informazioni relative ai vertici;
3. **segmentsTriangle**, array di tre dimensioni costituito da dati di tipo Segment contenenti le informazioni relative ai lati;
4. **longestEdge**, dato di tipo unsigned int che contiene l'Id del lato più lungo del triangolo;
5. **Area**, dato di tipo double relativo all'area del triangolo.

All'interno della classe vengono implementati i seguenti metodi:

1. **setSegments** utilizzato per assegnare correttamente i segmenti al triangolo, in base alla corrispondenza delle origini e delle estremità dei segmenti con i punti del triangolo;
2. **setPoints** utilizzato per ordinare correttamente i punti del triangolo, in modo che i segmenti siano collegati correttamente tra di loro;
3. **AreaCalculator** che riceve in input tre dati di tipo Point corrispondenti ai vertici del triangolo, calcola e restituisce l'area utilizzando la formula di Gauss per il calcolo delle aree: Dati un poligono con n lati e con vertici (x_i, y_i) $i = 0, ..., n$;l'area A viene calcolata come segue
$$\frac{1}{2}\left|\sum_{i=1}^{n} x_i y_{i+1} - x_{i+1} y_i\right|$$
4. **findLongestEdge** che prende in input l'array di tre dati di tipo segment contenente i lati del triangolo e restituisce l'id del lato di lunghezza maggiore;

5. **adjacency** memorizza l'Id del triangolo per chiamare la funzione 'connectTriangle' che, in base al lato che ha in imput 'adjacency', prende nota dei due triangoli ad esso adiacenti.

2.4 TriangularMesh

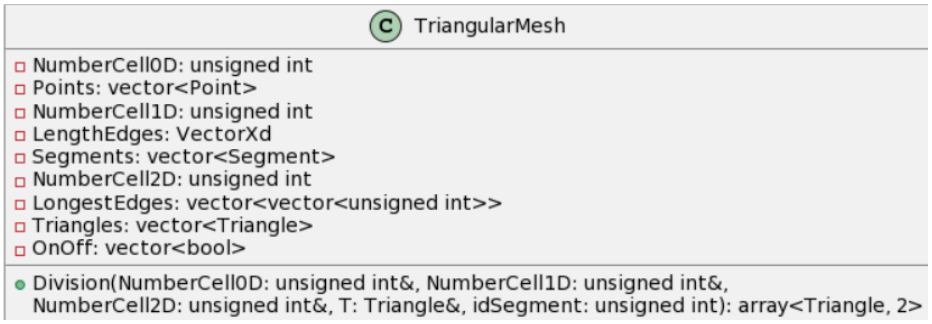

Figura 2.4. Classe TriangularMesh

QUesta classe è responsabile del raffinamento dei triangoli e utilizza gli oggetti definiti nelle classi prima. Sono presenti i seguenti attriabuti pubblici:

1. **NumberCell0D**, dato di tipo unsigned int che definisce la quantità di dati presenti nel file Cell0D ovvero la quantità di punti all'interno della Mesh;
2. **Points** vettore di elementi di tipo Point corrispondenti ai punti della mesh;
3. **NumberCell1D**, dato di tipo *unsigned int* che definisce la quantità di dati presenti nel file Cell1D ovvero la quantità di segmenti all'interno della Mesh;
4. **Segments** vettore di elementi di tipo Segment corrispondenti ai segmenti della mesh;
5. **NumberCell2D**, dato di tipo *unsigned int* che definisce la quantità di dati presenti nel file Cell2D ovvero la quantità di triangoli all'interno della Mesh;
6. **Triangles** vettore di elementi di tipo Triangle corrispondenti ai triangoli della mesh;
7. **OnOff** vettore contenente valori booleani che permette di capire se l'elemento alla posizione i-esima è già stato diviso ('Off') oppure no ('On').

All'interno della classe sono presenti i seguenti metodi:

1. **CreateMidpoint** che, ricevendo in input l'Id di un segmento, trova il suo punto medio e lo restituisce come oggetto di tipo 'Point';
2. **GetUtilities** contiene una serie di cicli for nei quali si cercano i lati opportuni per creare i nuovi triangoli. Restituisce un array di tre elementi contenente in ordine l'Id del vertice opposto al lato più lungo, l'Id del lato a sinistra e l'Id del lato a destra del lato più lungo.

3. **Division** che riceve in input i valori di tipo unsigned int corrispondenti a NumberCell1D e NumberCell2D, un dato di tipo Triangle T, un unsigned int $idSegment$ e un dato di tipo Point $Midpoint$. Esso ha il compito di effettuare la bisezione rispetto al punto medio del lato maggiore su T, successivamente si occupa di assegnare in modo sequenziale nuovi Id ai triangoli creati T_1 e T_2 e al segmento che unisce il punto medio del lato maggiore al vertice opposto. L'output viene restituito attraverso un array di dimensione 2 contenente due dati di tipo Triangle T_1 e T_2 .

4. **DivisionAdjacent** che funziona esattamente come 'Division', ma con la differenza che quest'ultimo metodo si occupa di dividere il triangolo adiacente a quello di area più grande, già diviso in 'Division'.

2.5 ImportMesh

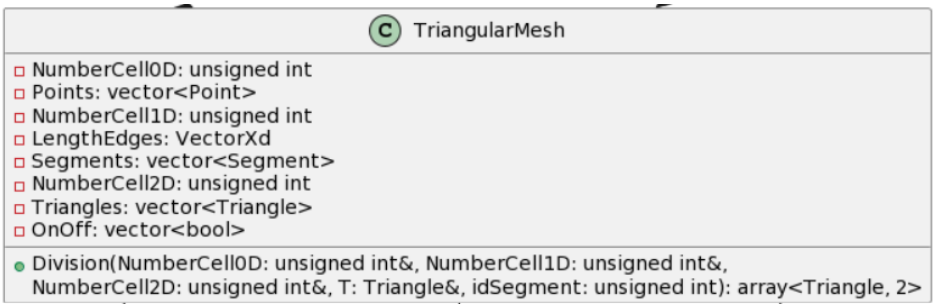

Figura 2.5. Classe TriangularMesh

La classe in questione ha il compito di importare tutti i dati presenti nei file "Cell0D", "Cell1D" e Cell2D in modo da permettere di affettuare il raffinamento sulla Mesh costruita grazie a questi ultimi.

Il costruttore della classe 'ImportMesh' prende come argomenti un oggetto 'TriangularMesh' e una stringa contenente il path dei file in cui si trovano i dati della mesh da importare.
Il processo di import dei file viene suddiviso in tre fasi per i tre diversi tipi di celle: 0D, 1D e 2D.

1. Import dei file 0D:

   - viene aperto il file 'Cell0Ds.csv' all'interno della directory specificata;
   - se il file non viene aperto correttamente, viene mostrato un messaggio di errore;
   - vengono lette le righe del file e memorizzate in una lista di stringhe;
   - viene rimosso il primo elemento della lista, che corrisponde all'intestazione del file;

- viene calcolato il numero di celle 0D e assegnato alla variabile ‘Mesh.NumberCell0D‘;
- viene effettuato un ciclo sulla lista delle righe.
- per ogni riga, vengono estratti gli elementi separati da spazi usando ‘istringstream‘ per convertire i valori nella rappresentazione corretta.
- vengono creati oggetti ‘Point‘ con i valori estratti e aggiunti al vettore ‘Mesh.Points‘;

2. Import dei file 1D:

   - Il processo è simile a quello del file 0D ma in questo caso viene apero il file Cell1D;
   - viene calcolato il numero di celle 1D e assegnato alla variabile ‘Mesh.NumberCell1D‘.
   - vengono creati oggetti ‘Segment‘ con i valori estratti e aggiunti al vettore ‘Mesh.Segments‘.

3. Import dei file 2D:

   - il processo è simile a quello del file 1D ma in questo caso viene aperto il file ‘Cell2Ds.csv‘;
   - viene calcolato il numero di celle 2D e assegnato alla variabile ‘Mesh.NumberCell2D‘.
   - vengono creati oggetti ‘Triangle‘ con i valori estratti e aggiunti al vettore ‘Mesh.Triangles‘.
   - si memorizza l’adiacenza chiamando il metodo ’adjacency’ su il triangolo appena creato una volta per ogni lato.

Una volta completato il processo di import dei file, l’oggetto Mesh conterrà i punti, i segmenti e i triangoli corrispondenti alla mesh importata.

2.6 Raffinamento

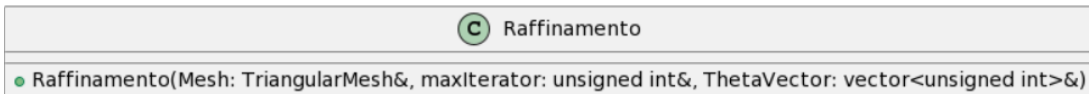

Figura 2.6. Classe Raffinamento

La classe contiene solamente il costruttore **"Raffinamento"** che prende in input un dato di tipo Mesh , una variabile di tipo int pari al numero di triangoli da raffinare e un vettore di variabili unsigned int contenente gli id dei triangoli che si vuole raffinare.

- Si itera un numero di volte pari a "maxIterator" utilizzando un ciclo "for";
- si accede all’elemento "i"-esimo del vettore "ThetaVector" per definire il punto medio del suo lato più lungo;

- utilizzando Division l'i-esimo triangolo viene diviso in due nuovi triangoli che vengono inseriti in un vettore chiamato "NewTriangles";
- aggiorna il vettore "OnOff" dell'oggetto "Mesh" per indicare che il triangolo originale non è più attivo;
- accede ai triangoli adiacenti al lato più lungo;
- verifica le condizioni per determinare su quale triangolo adiacente effettuare la divisione("adj1" o "adj2");
- infine, aggiunge i nuovi triangoli al vettore "Triangles" e aggiorna il vettore "OnOff" di conseguenza.

Tramite l'utilizzo di **push_back** si aggiungono elementi ai vettori Triangles e OnOff dell'oggetto Mesh.

Capitolo 3

Sorting

All'interno del codice è stato necessario implmentare un algoritmo di sorting in quanto l'obbiettivo di tale codice è quello di raffinare solo i primi theta triangoli con Area maggiore.
É stato applicato l'algoritmo di fusione (**MergeSort**) sulle Aree. Questo algoritmo opera secondo il paradigma **divide et impera**:

- viene suddiviso il problema principale in due o più sottoproblemi che vengono risolti applicando nuovamente lo stesso principio fino a giungere a problemi elementari che possono essere risolti in maniera diretta;
- vengono combinate le soluzioni dei sotto-problemi in modo opportuno così da ottenere una soluzione del problema di partenza.

Ciò che contraddistingue il paradigma del *divide et impera* è il fatto che i sotto problemi sono istanze dello stesso problema originale ma di dimensioni ridotte. Tale paradigma può essere strutturato nelle seguenti tre fasi:

1. Decomposizione: identifica un numero piccolo di sotto-problemi dello stesso tipo, ciascuno definito su un insieme di dati di dimensione inferiore a quello di partenza.
2. Ricorsione: risoluzione ricorsiva di ciascun sotto-problema fino a ottenere insiemi di dati di dimensioni tali che i sotto-problemi possano essere risolti direttamente.
3. Ricombinazione: combina le soluzioni dei sotto-problemi per fornire una soluzione al problema di partenza.

Nel caso del MergeSort, il paradigma può essere descritto nel seguente modo:

1. Decomposizione: se la sequenza da ordinare ha più di due elementi, viene divisa in due sotto-sequenze uguali (o quasi) in lunghezza.
2. Ricorsione: ordina ricorsivamente le due sotto-sequenze.
3. Ricombinazione: fusione delle due sotto-sequenze in un'unica sottosequenza ordinata.

Per implementare l'algoritmo, è necessario specificare come le due sotto-sequenze ordinate possano essere fuse. Per rendere l'algoritmo più semplice, di solito si utilizza un array aggiuntivo.

```
MergeSort( a, sinistra, destra ):
                                    ⟨pre: 0 ≤ sinistra ≤ destra ≤ n – 1⟩
  IF (sinistra < destra) {
    centro = (sinistra+destra)/2;
    MergeSort( a, sinistra, centro );
    MergeSort( a, centro+1, destra );
    Fusione( a, sinistra, centro, destra );
  }
```

Figura 3.1. MergeSort **?**

Le prime tre istruzioni della struttura "if" nel codice in figura 3.1 corrispondono alla fase di decomposizione del problema e alla soluzione ricorsiva dei due sotto-problemi. L'invocazione della funzione "Fusione" permette di realizzare la fase di ricombinazione, fondendo in un unico segmento ordinato i due segmenti '[sinistra, centro]' e '[centro+1, destro]' ordinati prodotti dalla ricorsione.

Capitolo 4

Main

All'interno del **main** viene importato l'oggetto mesh, vengono ordinati e raffinati i triangoli in base a un valore "theta", e infine vengono esportati i risultati in diversi file di output. Di seguito le varie operazioni implementate:

- Per prima cosa viene effettuato un controllo del numero di argomenti passati alla riga di comando tramite "argc": se "argc" è uguale a 1 significa che non è stato aggiunto niente a linea di comando e viene stampato un messaggio di errore;
- viene creato un oggetto 'TriangularMesh' chiamato 'Mesh';
- viene estratto il primo argomento dalla linea di comando ("argv[1]") e viene convertito in un valore intero "theta" utilizzando la funzione **"stoi()"**;
- viene estratto il secondo argomento dalla linea di comando ("argv[2]") e viene assegnato alla variabile 'directory';
- viene creato un oggetto "ImportMesh" passando l'oggetto "Mesh" e la directory contenente i percorsi dei file "cell0D", "cell1D", "cell2D",come argomenti. Questo oggetto si occupa dell'importazione del file mesh;
- Per evitare di modificare il vettore originale *Triangles* viene creato un vettore "v" e riempito con gli ID dei triangoli presenti in "Mesh".
- viene chiamata la funzione "MergeSort" passando "Mesh", il vettore "v", l'indice di partenza e l'indice di fine. Questa funzione esegue l'ordinamento del vettore "v" utilizzando l'algoritmo di ordinamento Merge Sort e restituisce un nuovo vettore "SortedA";
- viene creato un vettore "ThetaVector" e viene riservata la memoria per 'theta' elementi tramite l'utilizzo di **.reserve**.
- si effettua un controllo su "theta": se è maggiore o uguale alla dimensione di "SortedA", viene stampato un messaggio di errore. Altrimenti, vengono inseriti gli ultimi "theta" elementi di "SortedA" in "ThetaVector".

- viene calcolato il valore massimo del numero di iterazioni "maxIterator" come la dimensione di "ThetaVector";
- viene effettuato il "Raffinamento" dei triangoli passando in input al costruttore 'Raffinamento' "Mesh", "maxIterator" e "ThetaVector";
- Vengono aperti tre file di output per memorizzare i risulati del raffinamento.
  - I vertici vengono salvati in "outputFile0D";
  - i lati dei nuovi triangoli vengono memorizzati in "outputFile1D";
  - i nuovi triangoli vengono salvati in "outputFile2D".
- Infine, vengono chiusi i file di output.

Capitolo 5

Test effettuati

Effettuare test è un'attività fondamentale nello sviluppo di un codice perchè consentono di verificare la correttezza delle sue funzioni, identificare errori e problemi di prestazioni e assicurare che le modifiche apportate non abbiano effetti indesiderati sul suo funzionamento.

Nel nostro caso i test sono stati implementati utilizzando un framework di test chiamato Google Test (gtest). Offre un set di strumenti per scrivere e gestire i test unitari in C++ e fornisce un ambiente di esecuzione per i test, assert per le asserzioni e strumenti per la scrittura dei test e l'organizzazione dei risultati.

Si è deciso di effettuare test su ogni classe che è stata creata.

5.1 Test Point

All'interno della classe **Point** è stato implementato un solo test che verifica che l'oggetto point abbia l'ID, la coordinata x e la coordinata y corrispondenti ai valori attesi. Se tutti gli assert (EXPECT_EQ) passano, il test viene considerato corretto.

5.2 Test Segment

Come nella classe Point, anche nella classe **Segment** è stato verificato che il costruttore abbia lavorato nella maniera corretta. Si verifica che l'oggetto segment abbia l'ID, l'origin e l'end corrispondenti ai valori attesi. Se tutti gli assert (EXPECT_EQ) passano, il test viene considerato corretto. Un altro test presente in Segment è il test MidPoint che verifica che l'array midPoint abbia le coordinate corrispondenti ai valori attesi.

5.3 Triangle

All'interno della classe **Triangle** è stato implementato un metodo che si assicura che il costruttore svolga correttamente il suo compito verificando che l'oggetto Triangle abbia l'Id, l'array dei vertici e l'array dei lati attesi.

É stato effettuato anche un controllo sul calcolo dell'area in cui si verifica che il calcolo venga effettuato correttamente.
L'ultimo test che è stato implementato riguarda quello che controlla la corretta esecuzione di FindLongestEdge che verifica che prendendo in esame un Triangolo, FindLongestEdge ritorni il valore esatto del suo lato di lunghezza maggiore.

5.4 Test raffinamento

All'interno della classe **Raffinamento** viene verificata la correttezza delle funzioni che sono responsabile del raffinamento della mesh.
Nel test CreateMidPoint si verifica che l'associazione del nuovo Id al punto medio del lato di lunghezza maggiore del triangolo in questione sia avvenuta correttamente.
Successivamente, viene implementato il test che verifica che la funzione GetUtilities restituisca il lato a destra, il lato a sinistra e il vertice opposto rispetto al lato di lunghezza maggiore del triangolo.
Infine, sono stati implementati i test che verificano che la divisione dei triangoli avvenga in modo corretto controllando che Division restituisca correttamente i due triangoli costruiti dividendo il triangolo in questione e che DivisionAdjacency effettui la suddivisione attesa sul triangolo adiacente.

Capitolo 6

Complessità computazionale

Uno degli obiettivi principali di questo progetto è individuare i migliori algoritmi in termini di complessità computazionale, sia in spazio che in tempo, cercando di minimizzare il tempo di esecuzione e l'utilizzo di memoria. Consideriamo il caso pessimo, che corrisponde al costo massimo su tutte le possibili istanze di dimensione n. Esamineremo le principali funzioni e algoritmi per comprendere la loro complessità computazionale.

La costruzione della Mesh avviene attraverso la funzione di importazione dei file, la quale si occupa esclusivamente di leggere i file e memorizzare il loro contenuto, con un costo complessivo di $O(1)$. Quando costruiamo oggetti di tipo Segment e Triangle la funzione *ImportMesh* si occupa anche di calcolare la lunghezza del lato, il punto medio, nel primo caso e l'area nel secondo. Tuttavia anche in questo caso si ottiene un costo complessivo di $O(1)$, che è ottimo.

Per quanto riguarda gli algoritmi implementati nelle funzioni che eseguono il raffinamento effettivo (ad esempio il costruttore 'Raffinamento' e il metodo 'GetUtilities'), spesso vengono utilizzati cicli **for** e costrutti condizionali **if**. Possiamo fornire una stima approssimativa del costo computazionale, tenendo presente che ogni blocco di istruzioni e costrutti annidati contribuisce alla somma dei costi delle singole istruzioni e costrutti:

- I cicli **for** hanno un costo proporzionale a $m+\sum_{i=0}^{m-1} t_i$, dove t_i è il costo del corpo del ciclo alla i-esima iterazione, mentre m è il valore massimo + 1 che può raggiungere l'iteratore i.

- Per i costrutti condizionali **if**, bisogna considerare che solo uno dei due blocchi viene eseguito. Il costo dipende anche dalla complessità dell'espressione condizionale. Se la condizione è una semplice espressione booleana, il costo è dell'ordine di $O(1)$. Tuttavia, in generale, il costo di valutazione di un costrutto **if** può essere approssimato dalla formula: costo(condizione) + max{costo(blocco1), costo(blocco2)}.

Inoltre, vogliamo dimostrare che l'algoritmo di ordinamento MergeSort, che segue il paradigma "Divide et Impera", ha una complessità computazionale nel tempo di $O(n \log n)$. Considerando una chiamata su n elementi, l'esecuzione della funzione 'Merge' richiede $O(n)$, mentre le altre operazioni richiedono $O(1)$. Considerando anche le chiamate

ricorsive della funzione 'MergeSort', per ottenere il valore asintotico del costo computazionale, dobbiamo utilizzare una relazione di ricorrenza, indicando con $T(n)T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 2T(n/2) + cn & \text{altrimenti} \end{cases}$
dove $2T(n/2)$ è il costo di ordinamento dei due vettori, mentre cn è il costo per la fusione dei due vettori ordinati.

Infine, applicando il **Teorema Fondamentale delle Ricorrenze**, con $\alpha = 2$, $\beta = 2$ e $f(n) = n$, otteniamo $\gamma = 1$. La relazione di ricorrenza possiede il limite superiore:

$$T(n) = O(f(n) \log_\beta n) = O(n \log n)$$

dove $\log n$ indica il logaritmo in base 2 di n.

Per quanto riguarda la complessità in spazio dell'algoritmo di ordinamento, bisogna considerare l'utilizzo del vettore ausiliario w di n elementi che può essere utilizzato per tutte le operazioni di fusione. Pertanto, l'algoritmo richiede una complessità di $O(n)$ in spazio.

Capitolo 7

Pseudocodice Raffinamento complesso

Per raffinare un triangolo T, è possibile prendere in considerazione un metodo diverso rispetto a quello implementato. Tale metodo prevede di implementare le seguenti operazioni:

1. rintracciare il lato e^T più lungo del triangolo T;
2. calcolare il punto medio M_{e^T} del lato e^T e unirlo al vertice $V_{e^T}^T$ opposto al lato e^T generando in questo metodo i triangoli T_1 e T_2;
3. successivamente, per rendere la triangolazione si cerca il triangolo e_T e si raffina applicando il metodo di **bisezione del lato più lungo** al triangolo adiacente S;
4. si procede unendo il nuovo punto creato M_{e^s} con M_{e^T} creando due nuovi sottotriangoli;
5. si ripetere tale procedura finchè la mesh non risulterà ammissibile.

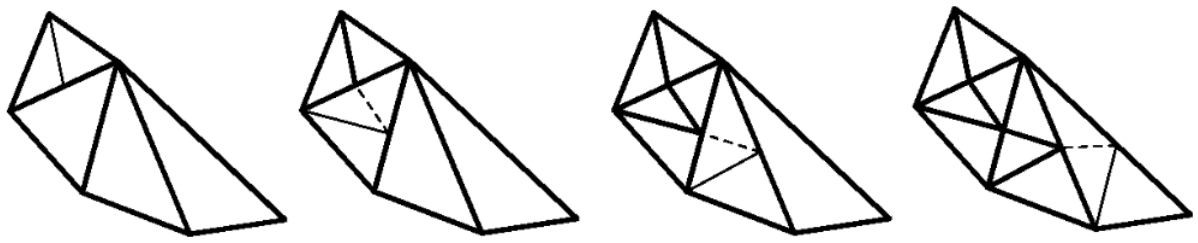

Figura 7.1. Raffinamento complesso

Di seguito viene riportato lo pseudocodice di tale procedura:

Division(T)
OnOff[T] := Off
OnOff[T_1] := **On**
OnOff[T_2] := **On**
if longestEdges[T] ==**longestEdges**[S]
DivisionAdjacent(S)
OnOff[S] := Off
OnOff[S_1] := **On**
OnOff[S_2] := **On**
else if longestEdges[T]! =**longestEdges**[S]
Division(S)
OnOff[S] := Off
per ogni newTriangle in [S1, S2]
OnOff[newTriangle] := On
if longestEdges[T] **in** S_1**.Edges**
DivisionAdjacent(S_1)
OnOff[S_1] := **Off**
per ogni newTriangle in S1
OnOff[newTriangle] := On
else DivisionAdjacent(S_2)**.**
OnOff[S_2] := **Off**
per ogni newTriangle in S2
OnOff[newTriangle] := On

- Per quanto riguarda la prima fase, si procede applicando la funzione **Division** al triangolo T in modo da creare i due triangoli T_1 e T_2;
- viene fatto successivamente un controllo sul lato in comune tra S e T:
  - se il lato in comune è il lato di lunghezza maggiore sia di S che di T allora è necessario unire con un segmento M_{e^T} con W_{e^S} dove W_{e^S} è il punto medio del lato maggiore di S (funzione **DivisionAdjacent**).
  - se W_{e^S} non è uguale a W_{e^T} allora si effettua la divisione tramite Division sul triangolo S creando due nuovi triangoli S_1, e S_2 e successivamente è utile aggiungere in coda gli id dei rispettivi lati di lunghezza maggiore. Si effettua un controllo sui lati di questi ultimi e si effettua la divisione del triangolo che ha uno dei lati uguali a un lato del triangolo di partenza T tramite DivisionAdjent.
- Alla fine di entrambi i controlli sarà necessario aggiornare con il valore "Off" il vettore "OnOff" alla posizione T_1 e T_2 e aggiungere tanti nuovi valori "On" in coda al vettore per segnalare la presenza di nuovi triangoli che potrebbero essere raffinati;
- ultimo passaggio fondamentale per la buona riuscita dell'algoritmo è quello di definire le adiacenze dei nuovi triangoli.

Si procede allo stesso modo sull'adiacente del triangolo S. Si nota che in questo caso sarà necessario definire all'interno della classe Segment la definizione dell'operatore "==" tra segmenti ovvero due segmenti vengono definiti ugali se hanno stesso Id.

Capitolo 8

Conclusioni

In conclusione, l'algoritmo implementato permette di raffinare la mesh importata dividendo i primi Theta triangoli in ordine decrescente di aree, con Theta numero intero variabile. Il codice implementato è in grado di importare ed esportare su file di tipo 'csv' i dati fondamentali della mesh e permette di dividere i triangoli e i loro adiacenti come richiesto dalla consegna.
La scelta di dividere un numero finito di triangoli limita il raffinamento globale della struttura, dal momento che i triangoli di area minore non vengono alterati dalle modifiche dell'algoritmo. Tuttavia una buona scelta del parametro Theta permette comunque di ottenere una buona condizione di raffinamento. La scelta dell'uso di vettori (dove possibile) e dell'algoritmo di sorting MergeSort ha permesso di ridurre al minimo i costi computazionali, che nel caso peggiore sono dell'ordine di $O(n \log n)$.
Infine, il codice passa correttamente tutti i test implementati, dunque si può concludere che anche fuori dall'ambiente di creazione, l'algoritmo permette di raggiungere l'obbiettivo della consegna. Di seguito si riportano i risultati ottenuti sul software Paraview con diversi valori di θ.

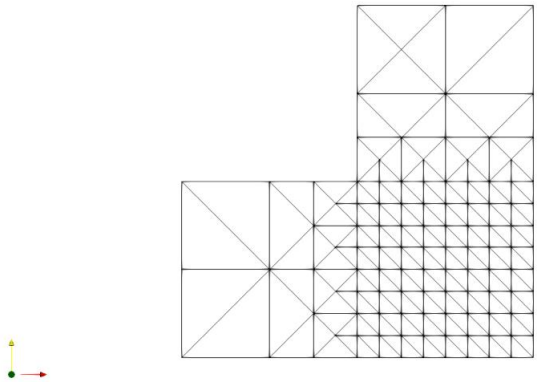

Figura 8.1. theta=3

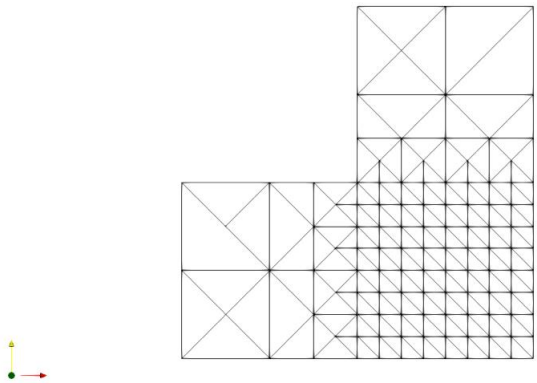

Figura 8.2. theta=5

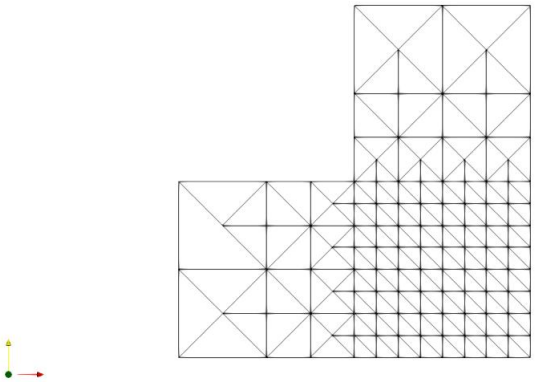

Figura 8.3. theta=10

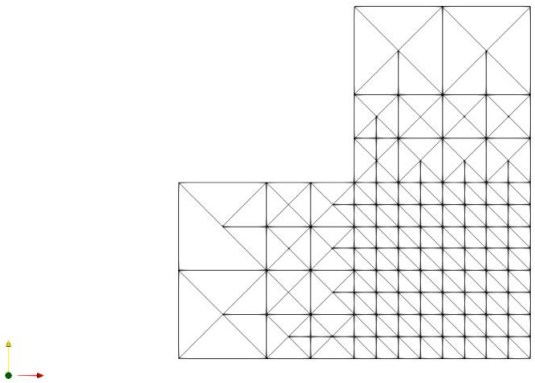

Figura 8.4. theta=20

Bibliografia

- Stanley B. Lippman, Barbara E. Moo, Josée Lajoie "C++ Primer", Addision Wesley, Fifth Edition, August 2012
- Roberto Grossi, Pierluigi Crescenzi, Giorgio Gambosi, "Strutture di Dati e Algoritmi", Pearson, Seconda Edizione, Settembre 2012
- Aditya Y. Bhargava, "Grokking Algorithms", Manning Shelter Island, 2016
- Gayle Laakmann Mcdowell, "Cracking the Coding Interview", CareerCup, Sixth Edition, 2015