

POLITECNICO DI TORINO

Corso di Laurea
in Matematica per l'Ingegneria

Documentazione progetto



Aurona Gashi
Francesco Matteazzi
Matteo Tonti

Anno Accademico 2022-2023

Sommario

È richiesta l'implementazione di un programma nel linguaggio ad oggetti C++, che esegua il metodo della bisezione del lato più lungo (versione complessa).

Indice

1	Analisi del progetto	1
1.1	Libreria "GeometryLibrary"	1
1.1.1	Classe "Vertex"	1
1.1.2	Classe "Edge"	2
1.1.3	Classe "Triangle"	3
1.1.4	Adiacenze	4
1.1.5	Operatori aggiuntivi	4
1.2	Librerie "ImportLibrary" e "OutputLibrary"	4
1.3	Libreria "SortingLibrary"	5
1.4	Algoritmo: main e "RefineLibrary"	5
2	UML	10
3	Unit test	11
4	Risultati	13

Capitolo 1

Analisi del progetto

1.1 Libreria "GeometryLibrary"

La prima libreria del nostro progetto è la libreria "GeometryLibrary".

Questa libreria contiene la definizione delle classi che abbiamo usato: abbiamo infatti scelto un approccio orientato alla programmazione a oggetti, in modo tale da poter operare su delle classi da noi definite che racchiudano le caratteristiche a noi utili degli elementi della mesh.

Abbiamo creato tre classi: "Vertex", che rappresenta i vertici della mesh (ovvero le celle 0-dimensionali), "Edge", che rappresenta i lati della mesh (cioè le celle 1-dimensionali), e "Triangle", che rappresenta i triangoli formati dai vertici e dai lati nella mesh (le celle 2-dimensionali).

1.1.1 Classe "Vertex"

La classe "Vertex" rappresenta i vertici della mesh e si compone di tre attributi pubblici.

- `_id`: è l'attributo che rappresenta l'indice del vertice, il suo tipo è "unsigned int" in quanto i vertici sono indicizzati decisionalmente e dunque, poiché è stata fatta la scelta di partire da 0 e assegnare gli indici in modo crescente, ha senso limitarci a tenere esclusivamente numeri non negativi.
- `_x`: questo attributo contiene la coordinata spaziale x del vertice. Il suo tipo è "double", in quanto lo spazio della mesh è idealmente continuo e dunque potremmo avere bisogno di più precisione rispetto a quella offerta dal tipo "float".
- `_y`: questo attributo contiene la coordinata spaziale y del vertice. Il suo tipo è "double", per ragioni analoghe a quelle dell'attributo `_x`.

Abbiamo poi definito un costruttore per la classe, che chiede 3 dati in input: un "unsigned int", che viene assegnato all'attributo `_id`, un "double", che viene assegnato all'attributo `_x`, e un altro "double", che viene assegnato all'attributo `_y`.

1.1.2 Classe "Edge"

La classe "Edge" rappresenta i lati della mesh e si compone di 5 attributi pubblici e un metodo pubblico, oltre al suo costruttore.

Gli attributi sono i seguenti:

- `_id`: è l'attributo che rappresenta l'indice del vertice e il suo tipo è "unsigned int", per ragioni analoghe all'attributo `_id` della classe "Vertex";
- `_length`: questo attributo contiene la lunghezza del lato, che viene calcolata all'interno del costruttore tramite il metodo `ComputeLength()`, che verrà spiegato più avanti. Il suo tipo è "double", in quanto si tratta di una lunghezza calcolata tramite le coordinate dei vertici del lato, che sono anch'essi "double".
- `_vertices`: questo attributo contiene gli indici dei vertici del lato: è di conseguenza un dato di tipo "vector<unsigned int>", in quanto conterrà due attributi `_id` dei vertici, che sono "unsigned int". Abbiamo optato per questo tipo di dato per facilitare la ricerca dei vertici dei lati nella lista dei vertici.
- `_edgeOfTriangles`: chiariremo meglio il significato di questo attributo nella sezione dedicata alle adiacenze. Il suo tipo è "vector<unsigned int>", e contiene l'insieme degli indici dei triangoli di cui il lato fa parte.
- `_status`: questo attributo indica se il lato è "spento" o "acceso", ovvero se è attivo (`true`) oppure no (`false`) a una certa iterazione. Raffinando progressivamente la mesh, infatti, alcuni lati vengono bisezionati e dunque non esistono più; al posto di eliminarli dalla lista dei lati, abbiamo deciso di disattivarli, per poter continuare a confondere l'`_id` con l'indice della posizione del lato nel contenitore `EdgesList` e agevolare così la ricerca. In quanto attributo binario, il suo tipo è "bool".

Abbiamo poi definito il metodo `ComputeLength()`, che richiede in input un vettore di puntatori a "Vertex": esso prende le coordinate dei vertici puntati dai puntatori e calcola la distanza tra i due vertici usando il teorema di Pitagora, restituendo il valore ottenuto come "double".

Per costruire un Edge abbiamo definito un costruttore che chiede due dati in input: un "unsigned int", che sarà il nuovo indice del lato, e un vettore di puntatori a vertici.

La scelta del vettore di puntatori è motivata dal fatto che, per ottenere alcuni attributi del lato, abbiamo bisogno degli attributi dei vertici; abbiamo dunque deciso di passare al costruttore gli indirizzi di memoria della lista dei vertici e non i vertici stessi, in modo tale da risparmiare memoria e costruire correttamente le adiacenze.

L'attributo `_status` viene da noi impostato di default a "true"; l'attributo `_vertices` è inizializzato come vettore vuoto, a cui vengono inseriti in coda gli indici dei vertici puntati dai puntatori; l'attributo `_edgeOfTriangles` non è inizializzato nel costruttore e viene riempito in fase di costruzione della classe "Triangle"; infine, per il calcolo dell'attributo `_length` viene chiamato il metodo `ComputeLength()`, a cui diamo in input il vettore di vertici che è stato passato al costruttore.

1.1.3 Classe "Triangle"

L'ultima classe della libreria è la classe "Triangle", che rappresenta i triangoli della mesh. Questa classe è caratterizzata da 6 attributi e 2 metodi (tutti pubblici), oltre che da un costruttore.

Gli attributi sono i seguenti:

- `_id`: rappresenta l'indice del triangolo. Analogamente agli indici dei vertici e dei lati, è di tipo "unsigned int";
- `_vertices`: è l'insieme degli indici dei vertici del triangolo. Per facilitare la ricerca nella lista dei vertici, ne abbiamo salvato gli indici e non l'intero oggetto "Vertex", dunque il suo tipo è "vector<unsigned int>". I vertici sono salvati in senso antiorario;
- `_edges`: è l'insieme degli indici dei lati del triangolo. Per ragioni analoghe all'attributo `_vertices`, il suo tipo è "vector<unsigned int>";
- `_area`: rappresenta l'area del triangolo, calcolata all'interno del costruttore chiamando il metodo `Area()`, che verrà spiegato più avanti. Il suo tipo è "double", in quanto viene calcolata a partire dalle coordinate dei vertici del triangolo;
- `_longestEdge`: contiene l'indice del lato di lunghezza massima del triangolo, dunque è un "unsigned int". Viene calcolato tramite il metodo `LongestEdge()`, che verrà spiegato più avanti;
- `_status`: indica se il triangolo è attivo oppure no, in quanto durante il raffinamento alcuni triangoli vengono sostituiti da triangoli più piccoli ottenuti dividendo quello di partenza. Il suo tipo è dunque "bool".

Abbiamo poi i due metodi della classe. Il metodo `LongestEdge()` richiede in input un vettore di puntatori a lati e restituisce l'id del lato indicato dai puntatori di lunghezza massima (dunque il suo output è "unsigned int"). Questo indice è ottenuto confrontando gli attributi `_length` dei tre lati.

Il metodo `Area()` richiede in input un vettore di puntatori a vertici e, a partire dalle loro coordinate, e dunque dai loro attributi `_x` e `_y`, calcola l'area del triangolo da loro formato (dunque l'output è un "double") tramite la formula di Gauss, che per il triangolo formato dai vertici (x_1, y_1) , (x_2, y_2) , (x_3, y_3) è la seguente:

$$A = \frac{1}{2} (x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2))$$

L'area viene in questo modo calcolata con il segno, ed è positiva solo se i vertici sono ordinati in senso antiorario; se l'area risulta negativa, vengono scambiati gli elementi in posizione 1 e 2 del vettore dei vertici del triangolo e viene restituito l'opposto dell'area.

Infine, il costruttore della classe "Triangle" richiede in input 3 dati: un "unsigned int", che viene assegnato all'attributo `_id`; un vettore di puntatori a vertici, che viene usato in primo luogo per inserire in coda a `_vertices` uno alla volta gli indici dei vertici che compongono ai triangoli, e poi per calcolare l'area del triangolo chiamando il metodo

Area() a cui viene dato in input lo stesso vettore di puntatori; in ultimo luogo viene richiesto un vettore di puntatori a Edge, che viene usato per inserire in coda all'attributo `_edges` l'indice dei lati che compongono i triangoli, per poi andare a modificare l'attributo `_edgeOfTriangles` degli Edge puntati, inserendo in coda ad esso l'id del triangolo (questo passaggio sarà spiegato meglio nella prossima sezione), e infine vengono passati al metodo `LongestEdge()` per calcolare l'indice del lato più lungo dei triangoli. L'attributo `_status` è inizializzato a "true".

1.1.4 Adiacenze

Nell'algoritmo del raffinamento, è necessario conoscere le adiacenze dei triangoli. Dato un lato di un triangolo e volendo ricercare il triangolo della lista complessiva che condivide il lato, per evitare di dovere scorrere tutti i triangoli e per ogni triangolo guardare ogni indice degli `_edges` e fare confronti finché non si trova quello corretto, il che risulterebbe computazionalmente oneroso, abbiamo deciso di memorizzare all'interno di ogni lato gli indici dei triangoli di cui esso fa parte. Questi indici vengono memorizzati all'interno dell'attributo `_edgeOfTriangles` del lato; in questo modo, conoscendo il lato, dobbiamo cercare il triangolo adiacente tra quelli in tale attributo. Ogni volta che viene creato un triangolo, nel suo costruttore in automatico viene aggiunto ai lati che compongono il triangolo il suo indice nel vettore `_edgeOfTriangles`: in questo modo le adiacenze vengono automaticamente aggiornate ogni volta che viene creato un triangolo. Risulta fondamentale in questo senso dare in input al costruttore di triangoli i puntatori agli Edge, in quanto se passassimo un Edge questa modifica verrebbe fatta solo nella "copia" su cui lavora il costruttore e non nel lato effettivo della lista dei lati: ciò produrrebbe errori nell'algoritmo. Notiamo che non eliminiamo mai i triangoli che non sono più attivi da `_edgeOfTriangles`, in quanto ci basta richiedere, quando cerchiamo un triangolo adiacente, che il suo attributo `_status` sia vero: dunque `_edgeOfTriangles` può avere anche più di due elementi, l'importante è che non ci siano mai più di 2 elementi che sono indici di triangoli attivi in quanto ciò violerebbe l'ammissibilità della mesh.

1.1.5 Operatori aggiuntivi

Per l'esecuzione di successive operazioni di confronto tra oggetti della classe Triangle, abbiamo definito un operatore booleano di disuguaglianza specifico per tale classe. Esso è definito tramite la disuguaglianza tra le aree dei triangoli e, per gestire triangoli le cui aree sono quasi uguali, abbiamo impostato una tolleranza di pari al massimo tra $(10^{-12})^2$ e il più piccolo numero di macchina.

1.2 Librerie "ImportLibrary" e "OutputLibrary"

Per l'importazione e l'esportazione delle mesh, abbiamo creato due librerie.

La libreria "ImportLibrary" contiene 3 funzioni, "ImportVertices", "ImportEdges" e "ImportTriangles": esse leggono dei file .csv e ne estraggono le informazioni utili per la creazione della mesh.

La funzione "ImportVertices" prende come input il percorso del primo file e un vettore di vertici; legge dal file e crea dei vertici che aggiunge a questo vettore.

La funzione "ImportEdges" richiede come input il secondo file, il vettore dei vertici creato da "ImportVertices", che gli serve per trovare i vertici da passare al costruttore di lati, e un vettore di lati, a cui aggiunge in coda quelli che crea.

Gli input della funzione "ImportTriangles" sono il vettore dei vertici e quello dei lati da cui trovare i vertici e i lati da passare al costruttore dei triangoli, e un vettore di triangoli a cui aggiungere in coda i nuovi triangoli che crea.

La libreria "OutputLibrary" ha la funzione di stampare su dei nuovi file i risultati dell'algoritmo. La libreria si compone di 3 funzioni, "ExportVertices", "ExportEdges" e "ExportTriangles": ognuna richiede in input il vettore di ciò che esporta, inoltre ExportEdges richiede la lista dei vertici in quanto è necessario conoscere le coordinate degli estremi dei lati. Le funzioni stampano su tre file differenti alcuni dei dati della mesh raffinata, in un formato tale da poter essere letto dal programma ParaView (oppure Matlab per quanto riguarda i triangoli) per la creazione delle immagini di tale mesh.

1.3 Libreria "SortingLibrary"

In questa libreria abbiamo scritto gli algoritmi di sorting che abbiamo utilizzato per ordinare la lista dei triangoli per area.

Abbiamo scelto due diversi algoritmi, HeapSort e InsertionSort: l'HeapSort ci serve all'inizio, in quanto abbiamo un vettore di triangoli disordinato, e il costo computazionale è quello ottimale, cioè $O(n \log(n))$ (dove n è la lunghezza del vettore).

Ad ogni iterazione dell'algoritmo vogliamo riordinare il vettore con i nuovi triangoli, in modo tale da raffinare sempre quello con area massima; conviene in questo caso usare l'InsertionSort, il cui costo computazionale nel caso peggiore è $O(n^2)$, ma quando il vettore iniziale è già ordinato è solo $O(num \cdot n)$, dove num è il numero di nuovi triangoli che abbiamo inserito nel vettore all'ultima iterazione.

1.4 Algoritmo: main e "RefineLibrary"

In questa sezione analizziamo l'algoritmo, dunque anche il main e la libreria che abbiamo utilizzato per la funzione che fa la gran parte del lavoro.

Il main del nostro programma contiene le istruzioni che fanno da guida al codice. Al programma passiamo da linea di comando 4 argomenti. Il primo è la percentuale di raffinamento che vogliamo usare, che può essere grande a piacere e corrisponde al numero di iterazioni calcolate come percentuale sulla grandezza iniziale della lista dei triangoli; i rimanenti 3 sono i percorsi dei file da cui importiamo rispettivamente vertici, lati e triangoli.

Il main inizia costruendo un vettore di vertici ("verticesList"), uno di lati ("edgesList") e uno di triangoli ("trianglesList") ottenuti chiamando le funzioni della libreria ImportLibrary e passando i percorsi dei file da linea di comando come input; questi vettori

contengono gli elementi ordinati per `_id`. I triangoli in `trianglesList` vengono successivamente ordinati per area decrescente con un `HeapSort` in un altro vettore chiamato `"sortedTriangles"`. Poi viene chiamata la funzione `"Refine"` di `RefineLibrary`; questa funzione chiede in input le tre liste dei vertici, lati e triangoli, la lista di triangoli ordinata e la percentuale di raffinamento. All'interno di questa funzione, in primo luogo viene calcolato il numero di iterazioni necessarie calcolando il valore della percentuale calcolata sulla lista dei triangoli, poi vengono creati alcuni parametri che servono nella funzione `Bisect`, che è chiamata al suo interno. In seguito, viene eseguito il seguente ciclo:

```
for (i = 0:numero di iterazioni):
    Stampa che stai raffinando l'i-esimo triangolo
    while (lo status del primo elemento di sortedTriangles è "false"):
        Eliminalo da sortedTriangles
        Bisect(primo elemento di sortedTriangles)
        sortedTriangles = InsertionSort(sortedList che
            include i nuovi triangoli)
```

La funzione `Bisect` prende in input: il vettore ordinato di triangoli `"sortedTriangles"`, il vettore `"trianglesList"`, il triangolo da raffinare `"triangle"`, il vettore di vertici `"verticesList"`, il vettore di lati `"edgesList"`, il contatore `"counter"` (se $\neq 0$ il triangolo, poichè è adiacente a un lato precedentemente dimezzato, viene ulteriormente raffinato per mantenere la mesh ammissibile), il vettore degli `_id` dei nuovi vertici creati `"newVertices"` e quello degli `_id` dei nuovi lati `"newEdges"` (necessari nelle chiamate di funzione quando il counter è diverso da 0).

L'outline della funzione è il seguente:

STEP 1.a.

Crea un nuovo vertice nel punto medio del lato più lungo del triangolo
da bisezionare

Crea due nuovi lati da quello che è stato dimezzato, e un altro che
congiunge il nuovo vertice con il vertice opposto al lato dimezzato

Crea due nuovi triangoli da quello di partenza

if (il triangolo raffinato non è il primo):

STEP 1.b.

Connetti il nuovo vertice al vertice creato all'iterazione precedente
con un nuovo lato, affinché la mesh sia ancora ammissibile

Crea due nuovi triangoli, bisezionando quello giusto dei due nuovi

endif.

STEP 2.

if (il lato bisezionato è al bordo)

end.

else if (il lato dimezzato è il lato più lungo del triangolo adiacente):

Dividi il triangolo adiacente in due nuovi triangoli

```
end.  
else if (il lato dimezzato non è il più lungo del triangolo adiacente):  
    Bisect(triangolo adiacente)  
end if.
```

Nello step 1.a i nuovi vertici, i nuovi lati e i nuovi triangoli frutto della bisezione vengono inseriti rispettivamente in `verticesList` e `newVertices`, in `edgesList` e `newEdges`, in `trianglesList` e `sortedTriangles`. Inoltre, il lato bisezionato e il triangolo raffinato vengono spenti.

Lo step 1.b viene eseguito quando stiamo effettuando il raffinamento di un triangolo dovuto ad una chiamata ricorsiva della funzione `Bisect`. Per mantenere la mesh ammissibile è necessario connettere il punto medio (`prec`) del lato bisezionato in precedenza al nuovo vertice creato (`newVertex`) raffinando il triangolo corrente.

Per capire qual è il vertice giusto, abbiamo pensato a due metodi. Il primo consiste nell'usare il prodotto vettoriale tra i vettori di \mathbb{R}^2 (`opposite`, `tb.v0`) e (`prec`, `tB.v0`), dove `tB.v0` (abbreviazione di `edgesList[toBisect]._vertices[0]`) è un vertice del lato bisezionato preso casualmente tra i due estremi. Se il prodotto vettoriale è nullo, o meglio se è minore di una tolleranza piccola, allora i due vettori sono collineari e `prec` appartiene a un lato di `newT1`, e quindi si procede a dividere `newT1` in due nuovi triangoli; se invece il prodotto vettoriale è diverso da 0, il triangolo da dividere è `newT2`.

Nel secondo metodo invece calcoliamo le aree con segno dei triangoli formati, in ordine, dai punti (`newVertex`, `opposite`, `tB.v0`) e (`newVertex`, `opposite`, `prec`) (vedi fig. 1.3), per capire se i punti sono disposti in senso antiorario (area positiva) oppure orario (area negativa). Se il prodotto delle aree con segno di `newT1` e `newT3` è positivo, allora le aree sono concordi e i due set di punti sono orientati nello stesso verso; delle semplici considerazioni geometriche ci permettono di concludere che `prec` appartiene a `newT1`. Se le aree sono discordi, invece, `prec` appartiene a `newT2`.

Nello step 2, i primi due casi sono le condizioni di terminazione: il lato bisezionato si trova al bordo della mesh oppure il triangolo adiacente ha come lato più lungo il lato precedentemente bisezionato (in questo caso viene creato un nuovo lato per unire il punto medio del lato bisezionato a quello opposto); in entrambi i casi la mesh risulterà ammissibile. Altrimenti, viene richiamata la funzione `Bisect` sul triangolo adiacente.

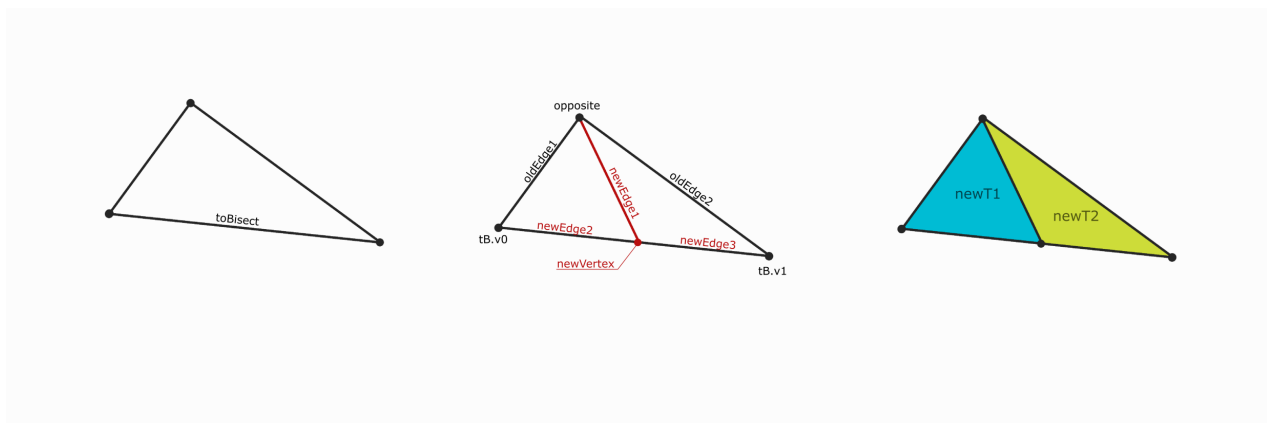


Figura 1.1: Step 1.a

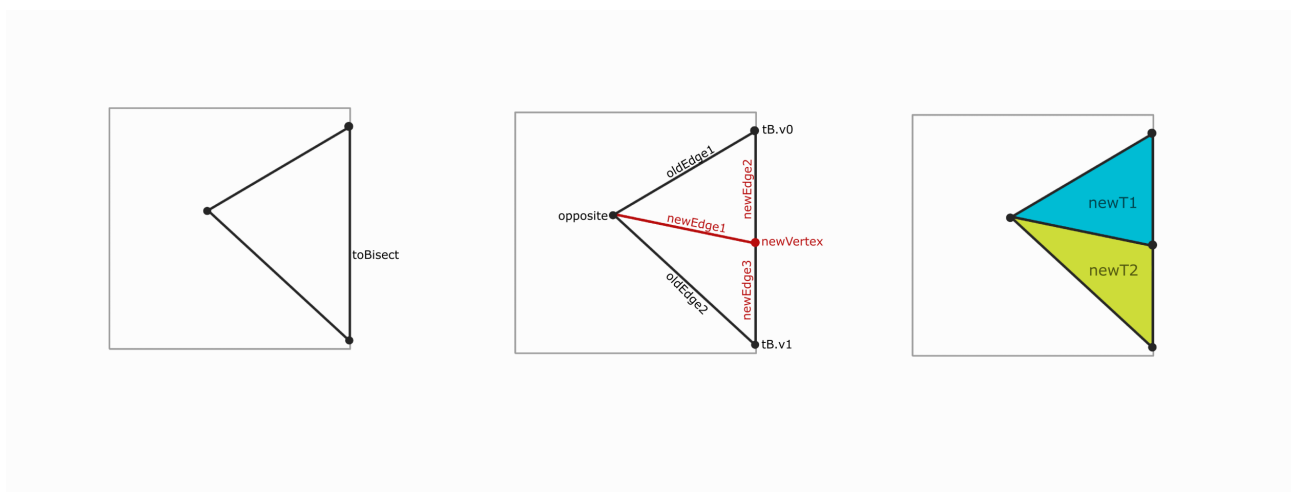


Figura 1.2: Step 1.a, caso al bordo (una condizione di terminazione)

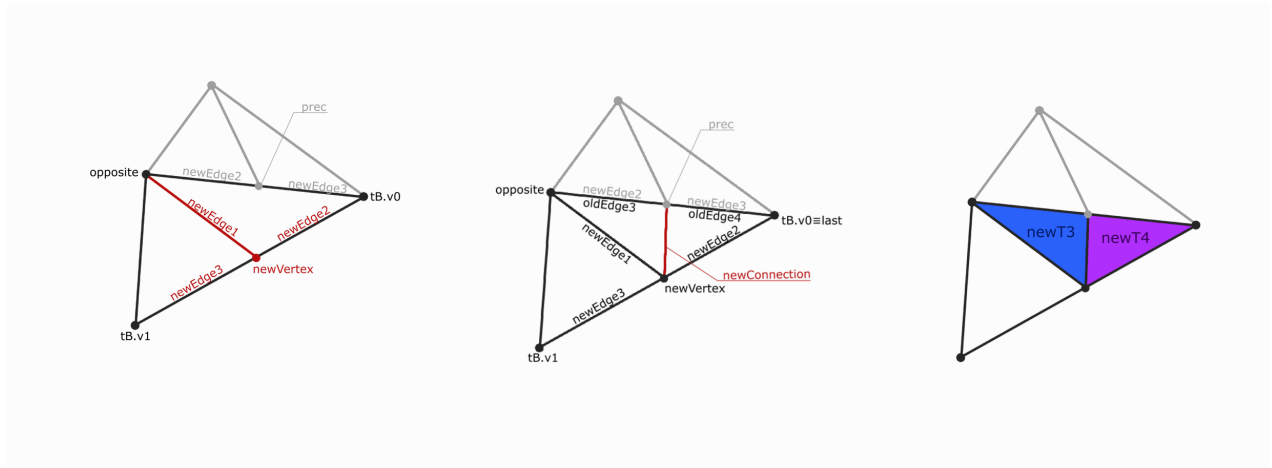


Figura 1.3: Step 1.b

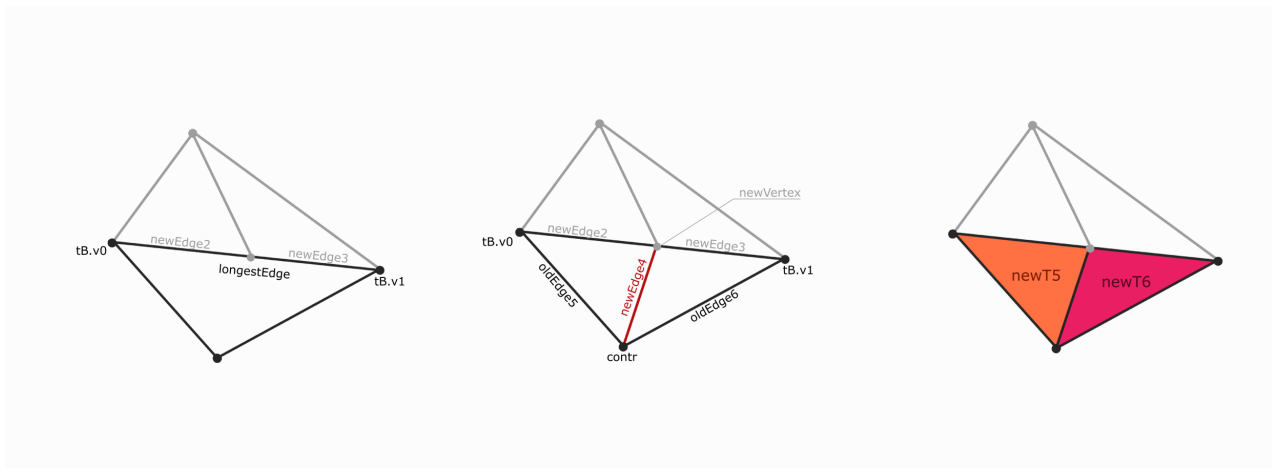


Figura 1.4: Step 2, caso lato più lungo coincidente con lato più lungo del triangolo adiacente (una condizione di terminazione)

Capitolo 2

UML

Di seguito riportiamo lo schema che abbiamo creato su PlantUML della struttura del codice per quanto riguarda la programmazione a oggetti.

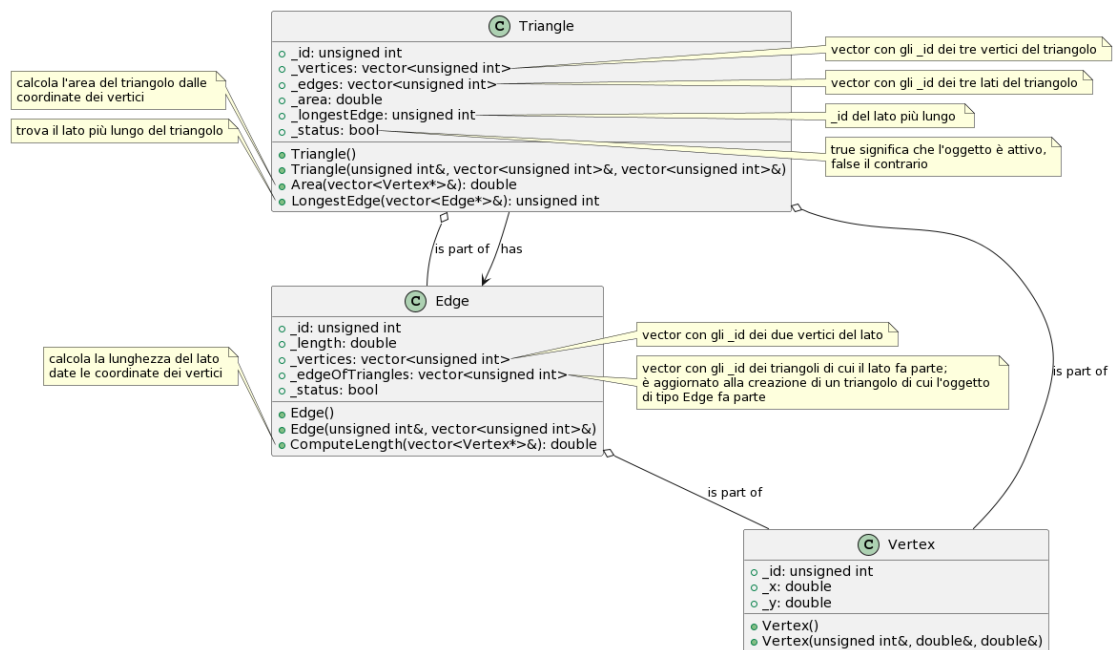


Figura 2.1: Schema in PlantUML della struttura del codice. Le note forniscono brevi spiegazioni degli attributi e dei metodi definiti all'interno delle classi.

In particolare, abbiamo riportato anche le relazioni tra le classi: le relazioni di aggregazione "is part of" indicano, intuitivamente, le relazioni di appartenenza necessarie per costruire nuovi oggetti di tipo Edge e Triangle, come per esempio un vertice che fa parte di un lato. La relazione "Triangle has Edge" indica la relazione di adiacenza descritta dall'attributo `_edgeOfTriangles`, spiegata nella sez. 1.1.4.

Capitolo 3

Unit test

In questa sezione descriviamo i test unitari che abbiamo implementato.

Abbiamo deciso di eseguire in tutto 9 test suddivisi per “categoria”: tre test sui costruttori, due test sugli algoritmi di sorting, due test sull’import e un test sull’algoritmo di raffinamento.

Test sui costruttori (TestConstructors):

1. TestVertex: verifica il corretto funzionamento del costruttore dei vertici,
2. TestEdge: verifica il corretto funzionamento del costruttore dei lati;
3. TestTriangle: verifica il corretto funzionamento del costruttore dei triangoli.

Test sugli algoritmi di sorting (TestSorting): Sono stati eseguiti su vettori di int (vector<int>):

1. TestHeapSort: verifica la corretta implementazione dell’algoritmo dell’Heapsort;
2. TestInsertion: verifica la corretta implementazione dell’algoritmo dell’Insertion Sort.

Test sull’import (TestImport):

Abbiamo creato tre file fittizi contenenti vertici, lati e triangoli, inventati per i test:

- 0dtry.csv, contenente i vertici e avente la stessa struttura del file Cell0Ds.csv;
- 1dtry.csv, contenente i lati e avente la stessa struttura del file Cell1Ds.csv;
- 2dtry.csv, contenente i triangoli e avente la stessa struttura del file Cell2Ds.csv;

I test eseguiti sono i seguenti:

1. TestVertices: verifica la corretta importazione dei vertici;
2. TestEdges: verifica la corretta importazione dei lati;
3. TestTriangles: verifica la corretta importazione dei triangoli;

Test sull'algoritmo di raffinamento (TestRefine):

1. TestBisect: verifica il corretto funzionamento della funzione Bisect.

Nonostante siano test unitari, abbiamo dovuto utilizzare i costruttori anche nei test sull'import e nel test sull'algoritmo di raffinamento perché le funzioni testate sono state definite per lavorare con oggetti da noi creati e quindi non per oggetti generici.

Capitolo 4

Risultati

Di seguito riportiamo i risultati dell'algoritmo. Per ciascuno dei dataset forniti, abbiamo creato tre diverse immagini tramite il software ParaView, in modo tale da visualizzare come la mesh varia con percentuale diverse di raffinamento. Per ciascun test dunque abbiamo stampato la mesh iniziale, la mesh con percentuale di raffinamento pari a 50 e la mesh con percentuale di raffinamento pari a 100. Ricordiamo che queste percentuali vengono definite sulla dimensione iniziale della lista dei triangoli, e dunque non corrispondono al numero di iterazioni.

Possiamo notare come la mesh del dataset "Test2", essendo composta da triangoli molto "regolari"(ogni triangolo ha almeno un lato parallelo ad un asse e le aree sono proporzionali), tende a rendere i triangoli congruenti e uniformare la mesh. La mesh del dataset "Test1" invece è composta da triangoli più randomici ed è difficile individuare un pattern di evoluzione.

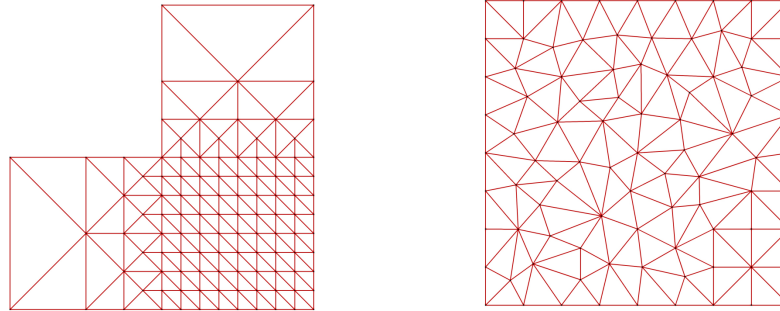


Figura 4.1: Mesh iniziali dei dataset: a sinistra quella fornita da "Test2" e a destra quella fornita da "Test1"

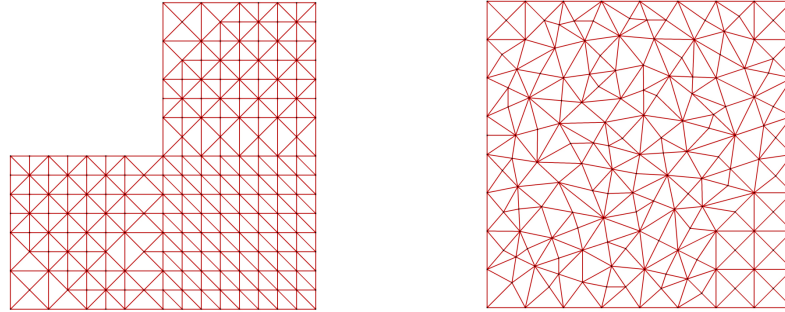


Figura 4.2: Mesh con il raffinamento al 50%: per l'immagine a sinistra, ottenuta da "Test2", ciò corrisponde a 93 iterazioni, mentre per l'immagine a destra, ottenuta da "Test1", corrisponde a 72 iterazioni.

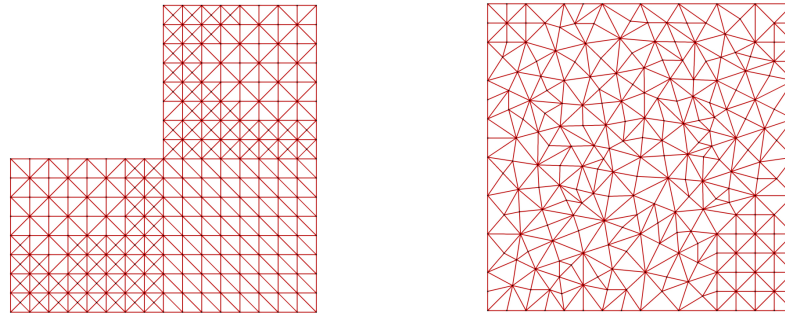


Figura 4.3: Mesh con il raffinamento al 100%: per l'immagine a sinistra, ottenuta da "Test2", ciò corrisponde a 186 iterazioni, mentre per l'immagine a destra, ottenuta da "Test1", corrisponde a 144 iterazioni.