UNIVERSITÀ DEGLI STUDI ROMA TRE
Sezione di Informatica e Automazione

Dipartimento di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi Di Laurea

Editor per grafi clusterizzati con supporto per operazioni di semplificazione e trasformazione

Relatore

Prof. Maurizio Patrignani

Candidato

Luca De Silvestris

Matricola 486652

Correlatore

Prof. Giuseppe Di Battista

Anno Accademico 2018/2019

Ai miei genitori

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Ringraziamenti

Lorem ipsum dolor sit amet, consectetur adipisci elit, sed do eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrum exercitationem ullamco laboriosam, nisi ut aliquid ex ea commodi consequatur. Duis aute irure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.Lorem ipsum dolor sit amet, consectetur adipisci elit, sed do eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrum exercitationem ullamco laboriosam, nisi ut aliquid ex ea commodi consequatur. Duis aute irure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.Lorem ipsum dolor sit amet, consectetur adipisci elit, sed do eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrum exercitationem ullamco laboriosam, nisi ut aliquid ex ea commodi consequatur. Duis aute irure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Luca De Silvestris

Premessa

Il lavoro che verrà descritto in questa tesi rappresenta la relazione finale di un progetto svolto dal candidato nell'ambito della visualizzazione delle informazioni sul tema dei grafi clusterizzati. Il progetto ha riguardato lo sviluppo del primo editor per clustered graphs e con supporto per operazioni di semplificazione e trasformazione e "flattizzazione" del grafo.

Le macro-aree di interesse sono quindi la visualizzazione delle informazioni, la teoria dei grafi, la programmazione web e Javascript. Tutti i concetti enunciati nella descrizione sintetica qui riportata verranno approfonditi, illustrati, documentati e motivati nel corso della lettura. Alla base dello studio vi è la necessità della creazione di un editor che faciliti lo sviluppo di applicazioni grafiche per la ricerca scientifica sui grafi.
La tesi si concentra in questi sei capitoli di seguito anticipati:

1. Stato dell' arte: breve panoramica sui concetti chiave del lavoro;

2. Strumenti e metodologie: un focus sugli strumenti e sulle tecnologie usate per risolvere i problemi e sulle metodologie;

3. Analisi dei requisiti e problemi: definizione degli obiettivi e raccolta dei requisiti progettuali e problemi da risolvere individuati durante le fasi di analisi;

4. Progettazione e implementazione: la descrizione dell' architettura del progetto finale, con approfondimento delle scelte progettuali;

5. Esempi di utilizzo e sviluppi futuri: considerazioni sulle possibili implementazioni future e esempi di applicazione;

6. Conclusioni: considerazioni sui risultati ottenuti.

Capitolo 1

Stato dell'arte

1.1 Definizioni preliminari

Di seguito andremo a definire le varie nozioni preliminari che dovranno essere ben note e delucidate prima di poter proseguire con la definizione di grafo clusterizzato. Le seguenti definizioni non hanno l'obiettivo di chiarire e delucidare a pieno il lettore riguardo gli argomenti trattati ma sono definizioni e spunti per poter definire almeno in minima parte l'ambito di studio su cui si andrà a sviluppare il lavoro svolto e anticipazioni per quanto concerne la struttura dati creata su cui l'editor andrà a lavorare.

1.1.1 Grafi

Un grafo G è definito da una coppia di insiemi $< V, E >$ in cui V è un insieme di nodi o vertici che possono essere connessi tra loro mediante l'insieme di archi E tale che i suoi elementi siano coppie di elementi dell'insieme V esprimibile come

$$E \subseteq V * V$$

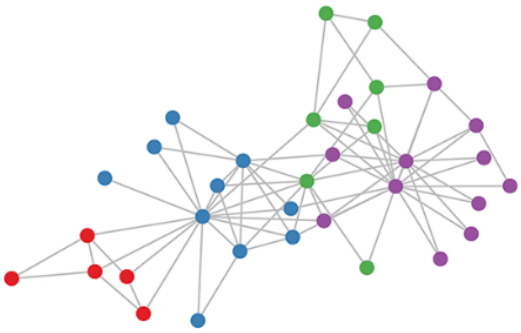

Figura 1.1: Esempio di grafo indiretto connesso

Si definisce poi grafo **completo** un grafo in cui ogni vertice è collegato con tutti gli altri vertici rimanenti di modo che l'insieme degli archi E è esplimibile come $E = V * V$. Dato un grafo è possibile avere un disegno $\tau(G)$ come mapping dei vertici su punti distinti del piano. Data la definizione di grafo è necessario ,ai fini di una introduzione al mondo della teoria dei grafi ed in particolare a quello dei grafi clusterizzati, enunciare la definizione di grafo **planare**.

Un grafo **planare** nella teoria dei grafi è definito come un grafo che può essere raffigurato in un piano in modo che non si abbiano archi che si intersecano. È facile da intuire che non tutti i grafi sono planari, e se ne riportano due famosi esempi in **Figura 1.2**

In particolare questi due grafi sono chiamati anche grafi di Kuratowski mediante i quali si può dare la definizione di grafo Planare anche come enunciato del ***Teorema di Kuratowski***:

Un grafo è planare se e solo se non contiene alcun sottografo che sia una espansione di K_5 o una espansione di $K_{3,3}$

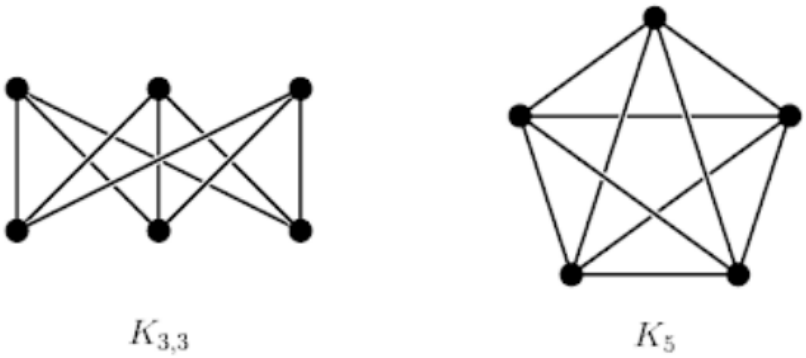

Figura 1.2: grafi di Kuratowski

Si ricorda inoltre che se un grafo è planare allora ammette un disegno planare $\tau(G)$. Dando un altra definizione si può evidenziare che un disegno $\tau(G)$ è un disegno planare ogni arco non interseca nulla eccetto i due vertici che connette. Si richiama inoltre all'algoritmo di Auslander e Parter del 1961. Mediante questo algoritmo del costo non lineare a livello di complessità computazionale ma uguale a $O(n^3)$ si è in grado di calcolare se un grafo in input è planare o meno ed è possibile formulare il seguente teorema

un grafo è un planare se il suo disegno è sempre colorabile con 4 colori diversi

Questo è un punto chiave nella teoria dei grafi quando poi si andrà a discutere dei grafi clusterizzati poco più avanti.

1.1.2 Alberi

Data la definizione di grafo si può dunque definire un albero **tree** come un grafo connesso non orientato e senza cicli come mostrato in **Figura 1.3**. Per essere

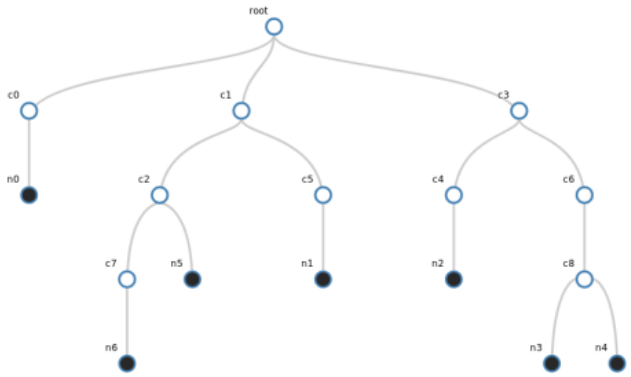

Figura 1.3: esempio di albero n-ario

tale quindi il grafo in questione o deve possedere un solo cammino per ogni coppia di vertici oppure essere aciclico massimale. Ogni nodo appartenente ad un albero possiede un livello dato alla somma $1 + L(padre)$ intendendo che la radice dell'albero ha livello 0.

La radice è definita come l'unico nodo dell'albero che non possiede archi entranti ma solo archi uscenti e a cui solitamente viene data rappresentazione del livello più basso dell'albero in altre parole è l'unico nodo che non presenta nodi con livello inferiore(i nodi padri) ma solo nodi con livello superiore(i nodi figli). I nodi dell'albero diversi dalla radice possono inoltre essere partizionati in due categorie:

nodi interni definiti come nodi con almeno un figlio e suddivisibili ulteriormente in nodi bassi in cui tutti i figli sono foglie e nodi alti che possiedono almeno un figlio nodo interno;

foglie che non possiedono figli e per questo non hanno archi uscenti.

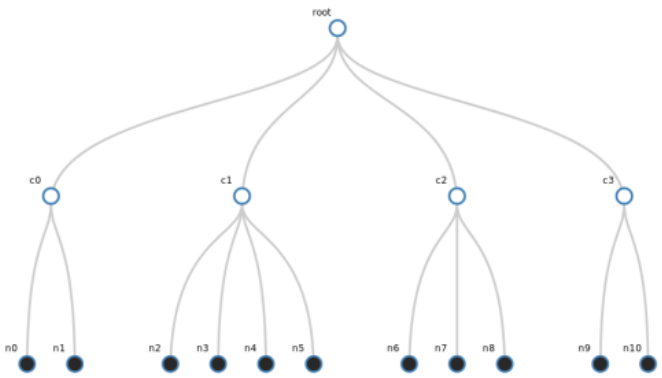

Figura 1.4: esempio di flat Tree

Un nodo è inoltre definibile **omogeneo** se tutti i figli di quel nodo sono foglie o sono nodi interni. Data la definizione di nodo omogeneo, viene detto che un albero è omogeneo se e solo se tutti i suoi nodi sono nodi omogenei. Date queste definizioni introduttive relative agli alberi è possibile definire un albero flat (come mostrato in **Figura 1.4**) come un albero in cui tutte le sue foglie hanno profondità pari a due.

Definendo in questo modo un albero flat è facile notare come ogni albero flat è anche omogeneo avendo tutti i nodi interni che hanno come figli esclusivamente foglie ed il nodo radice che possiede solo nodi interni come figli.

Per completezza si voglio dare anche le definizioni di profondità di un nodo e di altezza di un albero rispettivamente come la lunghezza del cammino dalla radice al nodo e la profondità massima dei suoi nodi. Una volta terminate le definizioni iniziali si può proseguire con quella di grafo clusterizzato.

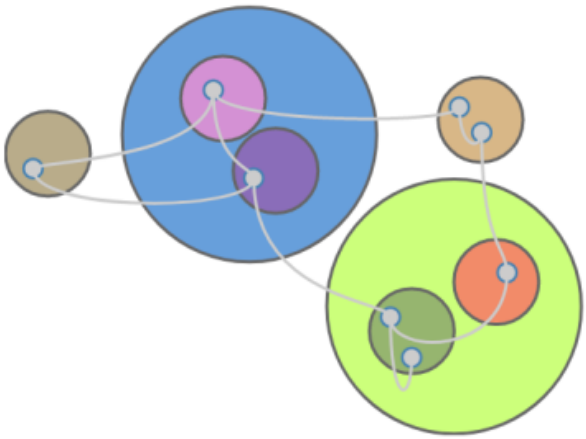

Figura 1.5: esempio di grafo clusterizzato

1.2 C-Graph

Un grafo clusterizzato (**c-graph**),di cui ne è un esempio la **Figura 1.5**, è definito come un grafo **planare** con una gerarchia ricorsiva definita sui suoi vertici. In altre parole un *c-graph* C è definito come una coppia $< \mathbf{G}, \mathbf{T} >$ dove:
$G = (V, E)$ è un grafo planare definito come *underlying graph* di C
T è un albero radicato in cui l'insieme delle foglie di T coincide con l'insieme dei nodi V di G ed ogni nodo interno è rappresentato da un **cluster**. Un cluster potrà quindi contenere altri cluster al suo interno e/o nodi dell'insieme V. Per quanto riguarda la struttura dati quella sarà analizzata con più consapevolezza nei capitoli successivi in cui verranno elencate e motivate scelte di progetto partendo dalle definizioni date ora.

1.2.1 clustered planarity

Avendo definito la struttura c-graph C si può andare ad analizzare come si comporta per quanto concerne il disegno $\tau(C)$. Questo disegno sarà un disegno planare **c-planar** se:

(i) ogni cluster è rappresentato da una singola regione del piano che contiene solo i suoi vertici;

(ii) nessun bordo di un cluster del disegno τ(c-graph) si interseca con altri bordi di altri cluster;

(iii) ogni arco si interseca con un cluster al più una volta;

Per ciò che concerne la teoria dell'informatica e dei grafi, i problemi relativi alla planarità e al disegno planare di grafi clusterizzati risultano essere ancora di grande importanza e oggetto di molte ricerche in quanto ancora non si è in grado di definire la complessità del decidere quando un c-graph ammette un disegno planare clusterizzato. In altre parole risulta essere un problema aperto quello di capire a quale classe di complessità appartiene il problema sopra definito, se in P o NP. La classe di complessità P consiste di tutti quei problemi di decisione che possono essere risolti con una macchina di Turing deterministica in un tempo che è polinomiale rispetto alla dimensione dei dati di ingresso mentre la classe **NP** consiste di tutti quei problemi di decisione le cui soluzioni positive possono essere verificate in tempo polinomiale avendo le giuste informazioni, o, equivalentemente, la cui soluzione può essere trovata in tempo polinomiale con una macchina di Turing non deterministica. Tutto questo poi richiama anche il problema del millennio *P contro NP* che risulta ancora non risolto e che consiste

nel capire se esistono problemi computazionali per cui è possibile "verificare" una soluzione in tempo polinomiale ma non è possibile "decidere", sempre in tempo polinomiale, se questa soluzione esiste. Si vuol precisare che le definizioni di macchina di Turing e di grammatiche di Chomsky non saranno date in questo lavoro poichè non necessarie ai fini dell'argomento trattato.

1.2.2 flat clustered planarity

Prima di poter analizzare i progressi svolti nell'ambito del disegno di grafi clusterizzti è necessario dare una definizione di riduzione polinomiale che sarà poi impiegata in questa sezione. La Riduzione polinomiale tra problemi è definibile come segue:

*Un problema **A** si riduce polinomialmente ad un problema **B** se, data un'istanza a di A, è possibile costruire in tempo polinomiale un'istanza b del problema B tale che a è affermativa se e solo se b è affermativa.*

Si noti inoltre che se un problema A si riduce ad un problema B allora risolvendo efficientemente B siamo in grado di risolvere efficientemente anche A. Mediante il lavoro svolto dal professor Maurizio Patrignani nel 2018 si visto che mediante una riduzione polinomiale definibile come:

$$C\_planarity <=_p Flat\ C\_planarity$$

può essere ridotto il *c-graph* C definito come C(G,T) in una istanza **equivalente** $C_f(G_f, T_F)$ in cui T_f è flat come espresso nel lemma che segue:

Una istanza $C(G,T)$ di un c-graph con n vertici e c cluster può essere ridotta in tempo $O(n+c)$ in una istanza equivalente $C_f(G_f, T_F)$ in cui T è omogeneo, $lar(T)$ definita come la radice dell'albero ha almeno due figli e $h(T) <= n-1$

Mediante questa riduzione si può trasformare qualunque c-graph in un grafo in cui tutti i cluster sono nodi interni di un albero flat e e che essendo flat risulta essere anche omogeneo. Come visto risolvendo questo problema definito Flat Clustered planarity ed essendo Clustered Planarity riducibile polinomialmente a questo allora si risolverebbe anche esso Questa riduzione è stata poi implementata nel lavoro svolto e di possibile impiego per l'utente che andrà a lavorare sul sistema quindi sarà vista con uno sguardo più attento nel capitoli successivi.

Inoltre si vuol precisare che molte delle immagini esplicative utilizzate in questo scritto sono state prese direttamente dall'editor durante istanze di lavoro. Nel capitolo successivo verranno descritti gli strumenti utilizzati per la realizzazione per passare poi alle tecnologie e alle metodologie utilizzate con relativi esempi di impiego e motivazioni.

Capitolo 2

Tecnologie e metodologie

Di seguito sono riportate tutte le tecnologie utilizzate, ognuna accompagnata da una descrizione, che a volte comprenderà anche un esempio di utilizzo e la motivazione che ha spinto ad utilizzare tale tecnologia. Oltre questo sono anche riportate tutte le metodologie applicate per poter realizzare il progetto.

2.1 Visualizzazione delle infomazioni

L'intero lavoro svolto si basa su due importanti settori informatici quali la teoria dei grafi, con basi di teoria della complessità, e la visualizzazione delle informazioni. Per quanto concerne quest'ultimo, può essere definito come l'utilizzo o comunque l'impiego di rappresentazioni visuali ed interattive di dati astratti con il compito di amplificarne la conoscenza degli stessi o anche come la comunicazione di dati attraverso l'utilizzo di interfacce visuali. Questo particolare settore è di grande rilievo sia per l'analisi dei dati che per la presentazione e quindi per la comunicazione degli stessi. Importante però è non assegnare alla visualizzazione delle informazioni il semplice scopo di creazione di grafici per scopi primi di informazione in quanto essa è la base su cui si andrà a lavorare.

Per la rappresentazione dei dati che rappresentano la struttura del c-graph sono stati di notevole importanza i principi di questo ambito informatico. Di seguito vedremo le strategie adottate dal sistema per la rappresentazione del c-graph ricordando che esso è una coppia $< G, T >$ con inclusion tree T e underlying graph G.

2.1.1 rappresentazione inclusion-Tree

Per quanto concerne la rappresentazione dell'albero di inclusione , essendo un albero radicato essendoci un rapporto padre-figlio non binario ci sono due possibili strategie: node-link e space-filling. Nella strategia node-link i nodi solo rappresentati come dei punti mentre i link ovvero i collegamenti,gli archi sono raffigurati come linee. A seconda poi della rappresentazione Node-link scelta si possono avere diverse organizzazione ne è un esempio della rappresentazione HV drawing la stessa della **Figura 2.1** in cui gli archi possono essere solamente orizzontali e verticali e che risulta essere molto utile per la rappresentazione di circuiti elettronici.

Figura 2.1: rappresentazione node-link HW drawing di un albero

Nella strategia Space-filling si tenta invece di superare i limiti del Node-link che riguardano la gestione non ottimale dello spazio orizzontale in quanto la parte alta della rappresentazione solitamente risulta essere poco densa di informazioni

al contrario di quella bassa. Nella strategia Space-filling forme geometriche, solitamente rettangolari, rappresentano i nodi e i figli sono rappresentati da altre forme geometriche inserite all'interno del padre anche se anche questa rappresentazione presenta svariati problemi tra cui la distinzione difficile di tagli orizzontali da quelli verticali o la gestione non ottimale di alberi di grande profondità. Di seguito sono mostrate alcune rappresentazioni diverse che seguono la strategia Space-filling.

Figura 2.2: rappresentazioni Space-filling

Essendo la strategia Space-filling di difficile lettura e soprattutto non ottimale per la rappresentazione di un albero collegato ad un grafo si è optato per la strategia Node-link scegliendo la rappresentazione "layered drawing" in cui i nodi sono organizzati in livelli e per ogni nodo la coordinata y dipende dal livello mentre la coordinata x deve esser trovata e dipende dal numero di figli del livello e dello spazio orizzontale a disposizione per la rappresentazione.

2.1.2 rappresentazione underlying Graph

Avendo discusso della rappresentazione dell'inclusion Tree è necessaria la stessa attenzione per la rappresentazione dell'Underlying graph. Il modello utilizzato è il **Force directed**, modello principale utilizzato per la visualizzazione di grafi che utilizza una rappresentazione Node-link, che fa in modo di emulare il sistema

fisico formato da cariche elettriche rappresentate dai nodi e da molle rappresentati dagli archi. L'idea alla base del modello è quella che il raggiungimento di una condizione di equilibro corrisponda ad una rappresentazione grafica gradevole. ogni rappresentazione di un grafo è dunque formata da due componenti: il **modello** che descrive le forze in gioco e l'**algoritmo** ovvero la tecnica che permette di stabilire il punto di equilibrio accettabile. Ne esistono diverse varianti del modello Force directed e la scelta per quanto concerne il progetto in questone è ricaduta sul modello Spring Embedding, basato sulla combinazione di forze elettriche e forze meccaniche che andrà a formare una rappresentazione simile a quella mostrata nella **Figura 2.3**

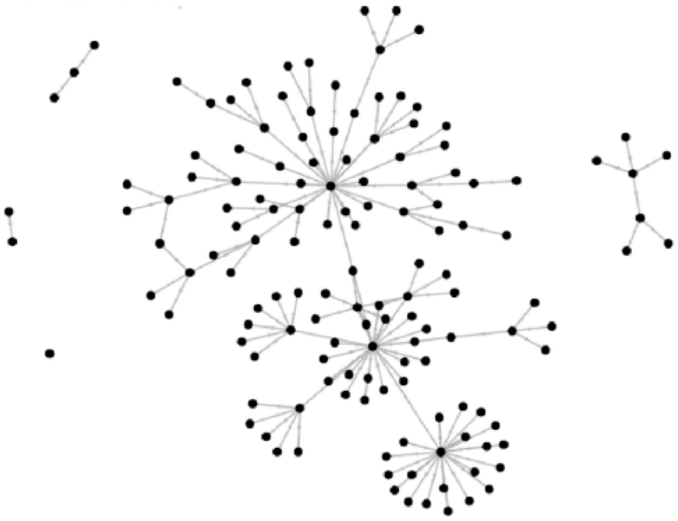

Figura 2.3: rappresentazione Spring Embedding

2.2 Scelte di linguaggi e librerie

Avendo parlato della visualizzazione delle informazioni, è ancora da discutere su quale delle possibili tecnologie è stata utilizzata allo scopo della rappresentazione dinamica e user-friendly per poter creare l'editor. Si è optato quindi per l'impiego di Javascript e della libreria D3.js che saranno viste nel dettaglio per motivazioni e punti di forza.

2.2.1 Javascript

JavaScript è un linguaggio di scripting orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione Web lato client. Essendo un sistema web con molti input da parte dell'utente a cui far fronte, Javascript consente un'ottima gestione di questi eventi. Le caratteristiche principali del linguaggio sono la debole tipizzazione, l'essere un linguaggio interpretato e non compilato con una sintassi relativamente simile a quella Java. Un altro punto a favore è la notevole facilità con cui si può lavorare con file con estensione .Json utilizzata proprio per l'import o l'export dei dati da rappresentare o della struttura dati della rappresentazione. Un altro punto a favore è il supporto alla programmazione asincrona, cioè alla possibilità di eseguire attività in background che non interferiscono con il flusso di elaborazione principale, che per questo linguaggio risulta essere quasi una necessità essendo un linguaggio Single-threaded I due principali elementi che consentono di sfruttare il modello di programmazione asincrono in JavaScript sono gli eventi e le callback. Per completezza si vuol dare anche la definizione di programmazione asincrona come una forma di programmazione parallela che permette ad un'unità di lavoro di funzionare separatamente dal thread principale ed "avvertire" quando avrà finito l'esecuzione di quel task.

2.2.2 D3.js

A supporto della scelta dell'utilizzo del linguaggio di scripting javascript vi è l'utilizzo della libreria utilizzata per la visualizzazione delle informazioni D3. D3.js (o solo D3 per Data-Driven Documents) è una libreria JavaScript per creare visualizzazioni dinamiche ed interattive partendo da dati organizzati, visibili attraverso un comune browser. Per fare ciò si serve largamente degli standard web: SVG, HTML5, e CSS.La libreria JavaScript D3, incorporata in una pagina web HTML, utilizza funzioni JavaScript per selezionare elementi del DOM, creare elementi SVG, aggiungergli uno stile grafico, oppure transizioni, effetti di movimento e/o tooltip. Questi oggetti possono essere largamente personalizzati utilizzando lo standard web CSS. In questo modo grandi collezioni di dati possono essere facilmente convertiti in oggetti SVG usando semplici funzioni di D3 e così generare ricche rappresentazioni grafiche di numeri, testi, mappe e diagrammi. Per completezza si ricorda che Scalable Vector Graphics abbreviato in SVG, indica una tecnologia in grado di visualizzare oggetti di grafica vettoriale e, pertanto, di gestire immagini scalabili dimensionalmente. Le figure espresse mediante SVG possono essere dinamiche e interattive. Il Document Object Model (DOM) per SVG, che include il completo XML DOM, consente una animazione in grafica vettoriale diretta ed efficiente attraverso il Javascript. D3 consente dunque di associare dati arbitrari a un Document Object Model (DOM) e quindi applicare trasformazioni basate sui dati al documento per avere una manipolazione efficiente dei documenti basata sui dati.

Per completezza si specifica inoltre che il Document Object Model (DOM) è un'interfaccia multipiattaforma e indipendente dalla lingua che tratta un documento XML o HTML come una struttura ad albero come mostrato nella **Figura 2.4**

in cui ciascun nodo è un oggetto che rappresenta una parte del documento. Il DOM rappresenta un documento con un albero logico. Ogni ramo dell'albero termina in un nodo e ogni nodo contiene oggetti. I metodi DOM consentono l'accesso programmatico all'albero; con loro si può cambiare la struttura, lo stile o il contenuto di un documento. Ai nodi possono essere associati gestori di eventi. Una volta attivato un evento, i gestori degli eventi vengono eseguiti. Un altro punto a favore dell'impiego di d3 e quindi di javascript riguarda le prestazioni. D3 è estremamente veloce, supporta set di dati di grandi dimensioni e comportamenti dinamici per l'interazione e l'animazione.

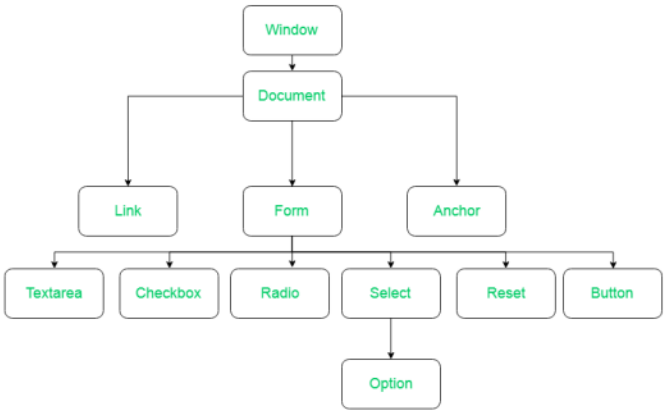

Figura 2.4: rappresentazioni Space-filling

2.3 Scelta della struttura dati

Inizialmente la struttura dati del progetto si basava su una unica struttura dati, in particolare un Array, contenente tutte le informazioni relative al c-Graph. La scelta è stata subito abbandonata a causa dei problemi riscontrabili con l'avere una struttura definibile quasi monolitica. Si è dunque passati ad una struttura

ad oggetti ed in particolare si è scelto di utilizzare le strutture dello standard ECMAScript 6 quali map e set risultando essere una scelta migliore rispetto all'utilizzo di array per ragioni prestazionali, sicurezza e miglior accesso ai dati da rappresentare spesso senza bisogno di iterare completamente su tutti i valori indicizzati al loro interno. Inoltre risultano essere oggetti specializzati: il set garantisce l'unicità dei valori, la map conserva l'ordine di inserimento e non ha alcun legame con la prototype chain.Per completezza si ricorda inoltre che un oggetto Map è un oggetto che contiene coppie $< KEY, VALUE >$ in cui qualunque valore, sia esso oggetto o tipo primitivo, può esser usato come chiave e come valore. Ad avvalorare la scelta si nota che entrambi sono oggetti *iterable* facili da iterare e i criteri di uguaglianza sulle chiavi della map sono più stretti di un normale $===$ e hanno funzionamenti e prestazioni migliori in caso di frequenti aggiunte o rimozioni di valori. In particolare l'impiego degli oggetti sopra definiti e motivati nel caso in questione sarà discussa nel capitolo 4 in maniera molto approfondita.

Capitolo 3

Analisi e problemi

In questo capitolo, dopo aver preso nota delle tecnologie e metodologie utilizzate e descritte nel *Capitolo 2* ,si vedrà più in dettaglio lo studio dei requisiti e dei problemi risolti nel lavoro svolto,prima di passare alla descrizione della progettazione dell'applicativo.

3.1 Obiettivi formativi

Gli obiettivi formativi inizialmente pensati per essere realizzati durante lo svolgimento del tirocinio,erano quelli di poter realizzare una libreria in grado di poter utilizzare due tipi di algoritmi: algoritmo con fase di streaming e fase di collisione e un algoritmo in cui si sarebbe eliminata la fase di streaming e di vedere le differenze a livello di tempo di esecuzione tra i due. Date le tempistiche ridotte del tirocinio questa idea è stata però abbandonata per dare maggior attenzione allo studio delle tecniche per aumentare le prestazioni computazionali arrivando a due obiettivi da raggiungere: lo sviluppo di un software veloce che potesse essere di facile impiego per l'utente utilizzatore, il tutto completato dall'implementazione di classi automatizzate di test che accompagnano lo sviluppo. L'attività di

definizione dei requisiti a breve termine è stata, piú che un punto di partenza,una fase ricorrente del lavoro alternata allo sviluppo vero e proprio della libreria. Il paragrafo che segue raccoglie tutti i requisiti che sono stati individuati e le problematiche da risolvere prese in considerazione durante il lavoro.

3.2 Analisi dei requisiti e scelte progettuali

Gran parte dello studio rivolto all'analisi dei requisiti è stato svolto, come si addice ad un software sviluppato con **metodologia agile**, nelle prime iterazioni e riguarda tutte quelle esigenze descritte nel *Capitolo 1* e che poi hanno trovato un buon riscontro nelle tecnologie analizzate nel *Capitolo 2.* Di seguito in **Figura ??** è descritto in maniera semplificata il diagramma degli oggetti di dominio, risultato dall' analisi del progetto nella prima iterazione.

Come si nota la classe templetizzata descritta nel *paragrafo 2.3* è stata fin da subito pensata come l'oggetto base da cui partire e su cui basare le iterazioni. Un altro fattore su cui si è scelto di lavorare per facilitare l'utilizzo come si conviene ad una buona libreria. Per rendere la suddetta accessibile e utilizzabile in pieno si è scelto di dare poche funzionalità all'utente mostrando dunque solo le operazioni di interesse lasciando a lui la possibilità di inserire alcune condizioni di utilizzo che saranno mostrate nel capitolo successivo.I problemi che sono stati affrontati con maggior interesse riguardano due importanti fattori: la facilità di impiego e la velocità computazionale.

Per quanto riguarda la velocità computazionale la soluzione ottima o comunque quella che si è visto essere la migliore è stata l'introduzione della programmazione parallela e concorrente e la templetizzazione delle classi. La programmazione generica in C++ infatti, nella creazione degli oggetti, dà un grande aiuto non solo

per quanto riguarda la flessibilità del progetto ma anche alla velocità computazionale, in quanto i template vengono elaborati e creati a tempo di compilazione lasciando quindi meno lavoro da svolgere a tempo di esecuzione dell'applicativo. Per poter essere di sostegno a qualunque tipologia di progetto e impiego si è preferito lasciare all'utente la possibilità di inserire le condizioni di bordo che egli ritiene più utili per il proprio scopo. Per **condizioni di bordo** si intendono le condizioni che rispondono alla domanda di come si comporta una particella con una certa velocità quando "incontra" il bordo fittizio del reticolo scelto.

Durante una successiva analisi si è poi optato per alcune modifiche progettuali che sono mostrate nel diagramma ad oggetti di seguito in Figura **??** seguendo i principi espressi dal libro "UML distilled. Guida rapida al linguaggio di modellazione standard"[Fow02] ed utilizzando **draw.io**, software di diagrammi online gratuito per la creazione di diagrammi di flusso, diagrammi di processo, organigrammi, ER e diagrammi di rete.

Nell'ingegneria del software, UML, acronimo di unified modeling language ovvero di "linguaggio di modellizzazione unificato" è un linguaggio di modellazione e specifica basato sul paradigma orientato agli oggetti. Il nucleo del linguaggio fu definito nel 1996 da Grady Booch, Jim Rumbaugh e Ivar Jacobson. Lo stantard è tuttora gestito dall'Object Management Group. UML svolge un'importantissima funzione di "lingua franca" nella comunità della progettazione e programmazione ad oggetti utilizzato anche dalla gran parte della letteratura del settore informatico per descrivere soluzioni analitiche e progettuali in modo sintetico e comprensibile ad un vasto pubblico.

Tutto ciò che é stato descritto fino ad ora in questo capitolo rientra in tutta quella branca dell'informatica definita ingegneria del software che divide la creazione del progetto in tre parti distinte ma sviluppate quasi in parallelo: analisi, progettazione ed implementazione. La progettazione è infatti una fase del ciclo di vita del software. Sulla base della specifica dei requisiti prodotta dall'analisi, la progettazione ne definirà i modi in cui tali requisiti saranno soddisfatti, entrando nel merito della struttura che dovrà essere data al sistema software che deve essere implementato. In particolare durante il lavoro svolto è stata utilizzata una metodologia definita come **Agile**.

Nell'ingegneria del software, l'espressione "metodologia agile", o sviluppo agile del software, si riferisce a un insieme di metodi di sviluppo del software emersi a partire dai primi anni 2000. Di grande spunto è stato anche il libro "Agile Software Development: Principles, Patterns, and Practices"[Mar02] I metodi agili si contrappongono al modello a cascata e altri processi software tradizionali, proponendo un approccio meno strutturato e focalizzato sull'obiettivo di consegnare al cliente, in tempi brevi e frequentemente, software funzionante e di qualità. Questi principi sono definiti nel "Manifesto per lo sviluppo agile del software" [KB01],

pubblicato nel 2001 da Kent Beck, Robert C. Martin e Martin Fowler in cui si specificano le seguenti considerazioni:

Gli individui e le interazioni pù che i processi e gli strumenti

Il software funzionante più che la documentazione esaustiva

La collaborazione col cliente più che la negoziazione dei contratti

Rispondere al cambiamento più che seguire un piano

Fra le pratiche promosse dai metodi agili si trovano: la formazione di team di sviluppo piccoli, poli-funzionali e auto-organizzati, lo sviluppo iterativo e incrementale, la pianificazione adattiva, e il coinvolgimento diretto e continuo del cliente nel processo di sviluppo. Si passa ora ad analizzare la libreria con particolare interesse all' inizializzazione e alla fase di spostamento delle particelle (detto *Stream*).

Capitolo 4

Progettazione e implementazione

In questo capitolo saraà illustrato il risultato del lavoro di progettazione e implementazione svolto fino al termine del tirocinio; i problemi individuati nel capitolo 3 sono stati qui risolti utilizzando le tecnologie e le metodologie illustrate nel capitolo 2 partendo dalla costruzione del reticolo discreto (Lattice). Si procederà a descrivere la libreria partendo dalle prime operazioni svolte dall'utente.

4.1 Implementazione del reticolo discreto

Al fine di realizzare un software utilizzabile in più situazioni l'utente utilizzatore ha la possibilità di indicare il numero di dimensioni su cui si andrà a lavorare, ovvero se utilizzare la libreria su uno, due oppure su tre assi e di specificare il numero di direzioni o velocità da assegnare ad ogni particella. Questa creazione del reticolo è possibile utilizzando la classe delle celle templetizzata
template <std::size_t dim, std::size_t dis, class Index, Index const (*v)[dis][dim], class T = double> **class lb_cell**
in cui *dim* specifica la dimensione, *dis* il numero delle velocità e *const (*v)[dis][dim]* specifica la loro distribuzione nello spazio così come visto nel paragrafo 1.2.2 in

cui si sono trattate le tipologie di reticoli discreti più utilizzati e la classificazione $DnQm$.

Considerato poi il grande utilizzo delle classificazioni D2Q9 e D3Q27 si è provveduto ad implementare questi due specifici reticoli discreti distribuendo le velocità nel caso del D2Q9 nel modo illustrato di seguito:

static const int **D2Q9_lattice[9][2]**{
{ +0, +0 }, // center d0
{ +0, +1 }, // north d1
{ +1, +1 }, // north-east d2
{ +1, +0 }, // east d3
{ +1, -1 }, // south-east d4
{ 0, -1 }, // south d5
{ -1, -1 }, // south-west d6
{ -1, 0 }, // west d7
{ -1, +1 } // south-east d8

Distribuzione delle velocità per reticolo discreto D2Q9

Utilizzando la funzione **Using** del linguaggio, che consente la dichiarazione di un tipo e l'utilizzo di un alias che prende le specifiche indicate:

using **D2Q9** = lb_cell<2, 9, int, **&D2Q9_lattice**, double>;

si è poi facilitata l'assegnazione dei valori della classe generica nel caso delle classificazioni sopra citate per una assegnazione quanto più immediata possibile. Da notare infatti che utilizzando l'alias D2Q9 si costruisce il reticolo utilizzando la distribuzione mostrata sopra. Si ricorda inoltre che nel linguaggio di programmazione C++ la & restituisce l' indirizzo di memoria dell' operando cui è applicato.

4.2 Creazione dello spazio discreto

Una volta deciso il tipo di reticolo discreto da utilizzare si passa alla costruzione e alla creazione e all'inizializzazione dei vari contenitori o *storage*(nome utilizzato nella libreria).

La creazione avviene mediante un costruttore a cui vengono passati in input la dimensione degli assi su cui verrà creato il reticolo. A puro titolo di esempio se si stesse utilizzando un sistema a due dimensioni, inserendo in input i valori {3,3} in cui il primo valore rappresenta la dimensione desiderata dell'asse X e il secondo valore l'asse delle Y il reticolo discreto verrebbe costruito come in **Figura 4.1**

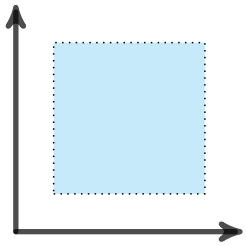

Figura 4.1: spazio discreto

È stato inoltre implementato un sistema che attua la strategia del controllo incrociato, riportando un messaggio di errore nel momento in cui si prova ad inserire valori per più o meno assi cartesiani di quelli su cui si è scelto di lavorare, in modo tale da evitare un utilizzo non desiderato. Utilizzato il costruttore il risultato sarà la creazione dello "spazio di lavoro" che risulterà però vuoto; l'utente può ora passare alla fase di inizializzazione.

4.3 Inizializzazione degli Storage

L'inizializzazione avviene invocando l'apposito metodo in cui si ha la possibilità di inserire:le **direzioni**, ovvero le velocità e che costituiranno le nostre particelle; lo **stato** della cella, in cui si specifica lo stato fisico della materia di cui è composta la cella; la massa dei punti.

Per l'implementazione come si è detto nel **Capitolo 2**, in cui si sono trattati gli strumenti e le metodologie applicate, si è optato per l'utilizzo degli *STD::VECTOR<>* generalizzando il tipo di dati. Utilizzando la funzione **typedef** del linguaggio i vari vettori sono definiti come

typedef std::vector<value_type> **storage_type**

avendo dunque la generalizzazione sul tipo di dato utilizzato di modo da poter avere una sola definizione di "storage" e render il codice più pulito. Nella libreria i vari vector sopracitati sono denominati rispettivamente: *Storage*, *state_points* e *mass_of_points*, tutti del tipo *storage_type* visto prima.

La struttura più "complessa" è quella dello **storage** in cui sono inserite le particelle e le rispettive velocità in un unico *vector* secondo lo schema riportato in **Figura ??**

Come si vede in figura è stata utilizzata una rappresentazione del D2Q9 e si nota che ogni particella può ed è espressa in relazione alle sue velocità e dalla posizione nel vettore. Questo ci ha permesso di ottimizzare lo spazio di indirizzamento della distribuzione cosa che non sarebbe potuta accadere se si fosse optato per l'utilizzo degli Array in quanto quella scelta particolare sarebbe andato a incidere sull'eccessivo utilizzo dello stack.

Per poter inizializzare lo spazio di lavoro creato bisogna dunque specificare i valori della distribuzione inserendo in input: un vector con le velocità che verrà inizializzato come detto sopra, un vector per specificare le celle di stato solido e uno per le masse delle varie parcelle. Il vettore con le informazioni sullo stato è fondamentale in quanto il metodo Lattice Boltzmann e tutta fluidodinamica si comporta diversamente al variare dello stato fisico della cella. Incontrando una cella solida ad esempio si potrebbe infatti avere un urto che porterebbe ad una variazione del verso della velocità. Dal vettore degli stati passato in input durante l'inizializzazione, si costruisce anche il vettore

storage_type ***boundaries_points***

che, mediante l'utilizzo di valori booleani, dà conoscenza delle celle di bordo del nostro spazio di riferimento etichettando con valore uno (*true*) la cella che fa parte del bordo. In **Figura ??** è mostrato un esempio in cui, sempre per un caso a due dimensioni, vengono inizializzate anche le celle di bordo. Si nota come le particelle siano inserite nello Storage a partire da quella con Y maggiore e X minore fino ad arrivare a quella con Y minore e X maggiore.

Questa inizializzazione del vettore boundaries_points è operata mediante l'utilizzo di due metodi: *check_solid_bound* e *check_boundaries_point*.

check_solid_bound prendendo in input un intero, ovvero l'indice dello storage da analizzare, restituisce in output un booleano true nel caso in cui la cella con indice in ingresso è bordo di una cella solida.

check_boundaries_point utilizzato per restituire il bordo perimetrale del reticolo discreto.

Di seguito, in Figura **??** è riportato un esempio concreto di inizializzazione di un reticolo discreto in cui sono evidenziate: le celle di stato fisico solido, le celle

di fluido inserito e quelle di bordo. Infine viene inizializzato anche un vettore *storage_ type axes* di interi, di supporto per l'utilizzo di metodi che da un indice ritornano le coordinate del punto e viceversa, che ha dimensione pari al numero di assi su cui si estende il reticolo discreto e i cui valori ne specificano la grandezza degli stessi.

4.4 Iterazione

Una volta inizializzato il reticolo discreto l'utente può eseguire le iterazioni. Queste come visto sono strutturate in due step, o fasi compiute in intervalli di tempo separati: Stream e Collision, che adesso saranno mostrate più in dettaglio.

4.4.1 Stream

$$f_i(\vec{x} + \vec{e_i}\delta_t, t + \delta_t) = f_i^t(\vec{x}, t + \delta_t)$$

Nella prima fase si avrà lo **Stream** in cui ci sarà la propagazione delle particelle, che si muoveranno nelle varie celle del reticolo discreto secondo le distribuzioni di velocità definite in precedenza, come mostrato in Figura **??** per una distribuzione D2Q9

Ricordando la disposizione delle velocità e delle particelle del vettore Storage, per poter realizzare l'effetto mostrato in figura, si passa quindi a una scansionare ogni particella e per ognuna di esse le proprie velocità, vedendo quali di esse si muoveranno ed in quale direzione. Dato che le particelle saranno anche collegate alla loro massa ed avendo visto che utilizzando LBM si conserva la quantità di moto, ad ogni iterazione la massa totale sarà conservata ma la distribuzione risulterà variata.

4.4.2 Condizioni di bordo e di angolo

Come solo accennato in precedenza parlando di scelte e metodologie applicate al progetto svolto, nel Capitolo 2, si sono lasciate le condizioni di bordo a scelta dell'utente. Sono state infatti implementate semplici condizioni di default, come necessità per poter passare alla seconda fase, che simulassero **l'uscita della particella** da reticolo andando quindi a sostituire quella particolare velocità con un valore nullo come mostrato in Figura **??**. L'utilizzatore ha la possibilità di sostituire questa condizione riscrivendo il metodo *Boundaries_ conditions* e implementando quella più consona al proprio modello. Esistono vari tipi di condizioni di bordo tra le quali spiccano:inflow, outflow, no-slip, free-slip e la bounce-back. È riportato un esempio di condizioni al contorno testate in precedenza in cui il reticolo possiede un bordo fisso e solido che a contatto le particelle mosse nella fase di Stream provocherà loro un urto e un conseguente cambio della direzione della velocità (**bounce-back**)come mostrato in Figura **??**.

Inoltre, come con le condizioni da attuare ai bordi del reticolo discreto, si è lasciata all'utente la possibilità di avere delle condizioni sugli angoli della griglia diverse da quelle di bordo, creando un metodo *"corners_ condition"* in cui sono state inserite come condizioni di default le stesse implementate per quelle di bordo.

4.4.3 Collisione

$$f_i^t(\vec{x}, t + \delta_t) = f_i(\vec{x}, t) + \frac{1}{\tau_f}(f_i^{eq} - f_i)$$

Analizzata la fase di Stream e visti alcuni esempi di condizioni al contorno si passa ora al secondo step dell'iterazione andando quindi ad analizzare la fase di **Collisione** in cui, nello stesso intervallo di tempo, ogni particella si sposterà sul nodo vicino, a seconda della propria direzione andando a concludere l'iterazione

con il ricalcolo delle distribuzioni. Durante questa fase viene inoltre ricalcolata la funzione di distribuzione aggiornata in modo tale che l'algoritmo possa procedere con un'altra iterazione e quindi con una nuova alternanza delle due fasi, conservando localmente massa e momento. Si è notato che in questo step le collisioni dipendono da quanto le distribuzioni si discostano da quelle di equilibrio, indicando un rapporto stretto tra le due fasi. Considerata dunque la vicinanza tra le due fasi, sia a livello temporale che di relazione si è dunque utilizzata la fase di collisione quasi in concomitanza con la fase di Stream.

Di seguito, in Figura **??** è mostrato un esempio di iterazione completa, con condizioni al contorno che comprendono la risposta al bordo solido denominato "wall". L'iterazione di esempio si compone di quattro fasi: pre-stream; post-stream; collision and reversing; bounce-back.

4.5 Caricare solidi

Per poter essere completamente aperta a qualunque tipo di utilizzo l'utente utilizzatore ha la possibilità di inserire, all'interno del reticolo discreto, forme e figure solide. Per dare questa possibilità è stata implementata la funzione *void* ***insert_ stl****(string fname, vector<triangle> &v* con la quale è possibile leggere file con estensione .stl.

STL (Acronimo di "Standard Triangulation Language") è un formato di file, binario o ASCII, nato per i software di stereolitografia CAD. È utilizzato nella prototipazione rapida (rapid prototyping) attraverso software CAD. Un file .stl rappresenta un solido la cui superficie è stata discretizzata in triangoli. Esso consiste delle coordinate X, Y e Z ripetute per ciascuno dei tre vertici di ciascun triangolo, con un vettore per descrivere l'orientazione della normale alla superficie.

In Figura ?? è mostrato un esempio di un solido sferico costruito come file con estenzione .stl. Data la definizione di file con estenzione .stl per poter utilizzare la funzione *insert_stl* è stata creata una classe triangle a cui in input vengono assegnati tre valori, uno per ogni asse. Questi tre valori sono di tipo V3, classe con un costruttore sovraccarico che può avere in input un puntatore ad un array, con i valori sugli assi, oppure direttamente le coordinate X, Y, Z che specificheranno dove creare l'oggetto. Una volta caricato il file l'utente ha il compito di settare il vettore *state_points*, che durante l'inizializzazione è necessario anche per costruire il vettore **check_boundaries_point**.

4.6 Bitmap

È stata implementata la possibilità di creare immagini e salvarle direttamente nella memoria interna del calcolatore utilizzato. In particolare si è optato per l'utilizzo di immagini con estensione .bmp. Un file BMP è un file bitmap, cioè un file di immagine grafica che immagazzina i pixel sotto forma di tabella di punti e che gestisce i colori sia in true color che attraverso una paletta indicizzata. La struttura di un file bitmap è la seguente: intestazione del file (in inglese file header); intestazione del bitmap (in inglese bitmap information header); paletta (opzionale); corpo dell'immagine. Le immagini bitmap, anche dette immagini **raster**, si contrappongono alle immagini vettoriali ed a tutta la grafica vettoriale, nella quale gli elementi vengono geometricamente ubicati nell'immagine mediante l'indicazione delle coordinate dei punti di applicazione, anzichè descrivendoli utilizzando una griglia di pixel. Un esempio che riporta la differenza tra le due tipologie di immagini è riportata in figura Figura ??.

Il formato file BMP è in grado di memorizzare immagini digitali bidimensionali

sia monocromatiche che a colori, in varie profondità di colore e, opzionalmente, con: compressione dati, canali alfa e profili colore. Nel codice è stata importata una libreria esterna *bitmap_image* con cui è stato implementato il metodo *void bitmap_image()* il quale: crea un oggetto bitmap la cui grandezza tiene conto delle dimensioni del reticolo; ne disegna le distribuzioni delle velocità con una palette di colori che passa da quelli meno accesi per una densità di particelle e di velocità minori, fino ad arrivare a colori più accesi dove è presente un numero elevato di particelle e velocità; salva l'immagine creata e disegnata in memoria una volta terminato il programma.

L'utente utilizzatore ha anche la possibilità di analizzare, facendo molteplici iterazioni, il cambiamento nel pre e post per ogni iterazione attraverso un numero di immagini BMP quante ne sono state stampate durante l'intera esecuzione del processo.

4.7 Testing

Come detto nel paragrafo 3.1 lo sviluppo è stato accompagnato e guidato da test. In particolare è stato inserito il file *Catch.hpp* distribuito sotto la licenza software Boost.

Boost è un insieme di librerie per il linguaggio di programmazione C++ che fornisce supporto per attività e strutture quali algebra lineare, generazione di numeri pseudocasuali, multithreading, elaborazione di immagini, espressioni regolari e test di unità. Molte di queste librerie sono rilasciate sotto Boost Software License, licenza progettata per consentire a Boost di essere utilizzato con progetti software gratuiti e proprietari.

Lo sviluppo guidato dal testing dell'applicativo è una parte del ciclo di vita

del software di fondamentale importanza, non solo per individuare le carenze di correttezza, completezza e affidabilità delle componenti software in corso di sviluppo ma anche per valutare se il comportamento del software, prima della distribuzione, rispetta i requisiti. Inoltre si parla di **sviluppo guidato** in quanto i test devono accompagnare tutte le fasi dello sviluppo i modo da vedere se dopo un determinato cambiamento sia venuta meno la stabilit'a del codice. L'obiettivo dell'impiego di test è quello di mantenere malfunzionamenti del codice e bug, che hanno costi di debugging proporzionali al tempo di "creazione" e alla quantità di rige di codice, in zone a spiccata località di modo da essere facilmente identificabili ed eliminare la **regressione**.

Capitolo 5

Esempi di utilizzo e possibili sviluppi

Di seguito saranno viste le varie possibilitá prese in considerazione e non ancora implementate. L'idea di base è quindi quella di estendere non le funzionalità ma le tipologie di implementazioni, tramite l'utilizzo di strutture dati diverse, in modo da avere come obiettivo finale quello di poter dare all'utente utilizzatore la possibilità di scegliere l'implementazione desiderata.

5.1 Quadtree e Octree

5.1.1 Definizione delle strutture

Un **Quadtree**, anche detto albero quadramentale, è una struttura dati ad albero in cui ogni nodo interno ha esattamente quattro figli, come mostrato in Figura **??**.Questa implementazione in particolare è molto utilizzata in computer grafica poichè in uno spazio a due dimensioni, presa una determinata area, un nodo ne rappresenta un riquadro di delimitazione che copre una parte dello spazio che viene indicizzato.

La radice dell'albero ovvero quel nodo privo di arco entrante può esser visto come la rappresentazione dell'intera area. Identificando uno spazio a due dimensioni con l'albero quadramentale di esempio mostrato in figura Figura **??** si otterrà la suddivisione mostrata in Figura **??**. Il fondamento alla base dell'assegnazione dell'intera area alla radice dell'albero trova spiegazione nel fatto che un albero senza radice non può essere acceduto risultando quindi inutilizzabile. Quando si lavora invece su spazi a tre dimensioni vengono utilizzati gli **Octree**, anche chiamati alberi ottali, su cui valgono gli stessi principi detti prima per gli alberi quadramentali con la differenza che ogni nodo ha esattamente otto figli, dividendo così lo spazio sui tre assi (x, y, z). Implementando anche questa struttura dati l'utilizzatore può dunque creare il reticolo discreto intero come la radice dell'albero (che sia Quadtree oppure Octree) andando poi a vedere ogni cella come una foglia dell'albero.

5.1.2 Teoremi stipulati sulle strutture

Sarà ora descritta la funzione $O(f(x))$ utilizzata per i successivi due teoremi. La notazione matematica O-grande è utilizzata per descrivere il comportamento asintotico delle funzioni. Il suo obiettivo è quello di caratterizzare il comportamento di una funzione per argomenti elevati in modo semplice ma rigoroso, al fine di poter confrontare il comportamento di più funzioni fra loro. Più precisamente, è usata per descrivere un limite asintotico superiore per la magnitudine di una funzione rispetto ad un'altra, che solitamente ha una forma più semplice. Ha due aree principali di applicazione: in matematica, dove è solitamente usata per caratterizzare il resto del troncamento di una serie infinita, in particolare di una serie asintotica ed in informatica dove risulta utile all'analisi della complessità degli algoritmi. Importanti i teoremi stipulati sugli alberi quadramentali secondo

cui: la profondità di un Quadtree per un set P di punti nel piano è al massimo uguale a

$$log(s/c) + 3/2$$

dove c è la distanza minima tra due punti qualsiasi in P e s è la lunghezza laterale del quadrato iniziale che contiene P. Inoltre un quadtree di profondità d che memorizza un insieme di n punti ha un numero di nodi e un tempo di costruzione entrambi pari a

$$O((d+1)n)$$

Data poi la definizione di albero bilanciato come albero in cui due nodi vicini differiscono al massimo di uno in profondità questa può essere applicata per bilanciare l'albero dell'esempio in Figura **??** così come mostrato in Figura 5.1.2 nto di un Quadtree

5.1.3 Inserimento particelle

Dopo la creazione si passa all'inzializzazione e in questa fase si nota il punto di forza di questa struttura dati in quanto si possono assegnare con precisione le particelle nel punto dello spazio desiderato. Il rilevamento delle collisioni è una parte essenziale della computer grafica applicata alla fluidodinamica e rappresenta una delle operazioni più dispendiose. Per evitare queste operazioni, all'aumentare degli oggetti, per non farli collidere in un unico nodo, questo si dividerà per

assegnare ad ogni particella la propria foglia come mostrato in **Figura ??**. Da questo si notano anche che le collisioni possibili sono molto ridotte in quanto una particella in uno specifico nodo non entrerà in collisione con un'altra appartenente ad un nodo di un'altro semipiano di profondià minore.

5.2 Griglie a infittimento e autoadattive

Il metodo Lattice Boltzmann utilizza come visto griglie cartesiane che nell'attuale lavoro svolto si potrebbero definire statiche. Implementando il reticolo con la struttura dati a quadtree della sezione precedente si risolverebbe questo problema potendo infatti **infittire** le zone di interesse per poter analizzare la posizione delle particelle o del solido inserito in maniera più accurata. Volendo infittire invece la griglia cartesiana di riferimento si andrebbe però ad influire negativamente sulle prestazioni ma una griglia più veloce a livello computazionale sarebbè per'o inaccurata. La soluzione ottima si ha nell'introduzione di una griglia a infittimento che aumenta l'accuratezza solo ed esclusivamente nelle zone di interessa come mostrato in Figura **??**.

Esistono due tipologie di griglie a infittimento: a priori ed autoadattive. Nelle seconde si nota che, oltre ad avere un infittimento nelle aree del reticolo discreto più soggette alla presenza di particelle, adattano automaticamente l'infittimento nelle zone più soggette ad avere densità maggiore.

5.3 Esempi di utilizzo

In questo paragrafo vedremo un esempio di utilizzo incentrato sul vettore di *storage*, sul suo stato pre iterazione e la sua variazione post iterazione. Come già detto infatti utilizzando l'algoritmo LBM si conserva, così come la quantità

di moto, anche la massa ma, dopo l'alternanza delle fasi di stream e collision, risulterà cambiata la sua distribuzione e quella delle velocità. Per l'esempio che mostreremo è stato creato un reticolo discreto a due dimensioni in cui gran parte delle particelle si trovano nella parte in alto a sinistra della griglia. Per la rappresentazione verrà utilizzata la funzione descritta nel paragrafo 4.6 per salvare un BMP dello stato pre e post varie iterazioni. In **Figura ??** sono mostrate le immabini bitmap memorizzate in memoria che identificano le situazioni antecedenti e successive all'iterazione.

Conclusioni

Il metodo *Lattice Boltzmann* come visto è tuttora utilizzato in tutti gli ambiti inerenti la fluidodinamica proprio per l'algoritmo che, a contrario delle equazioni di Navier-Stokes viste nel capitolo 1, è facilmente realizzabile a livello di computazione e ciò ha spinto l'attività di tirocinio formativo a trovare ed analizzare altri aspetti, che potessero affiancare l'utilizzo di questo metodo, da implementare e realizzare.

All'inizio dell'attività di tirocinio una volta preso nota l'importanza nell'ambito della computer grafica e delle simulazioni dell'utilizzo del metodo Lattice Bolzmann sono stati quindi individuati due obiettivi chiave su cui porre un'attenzione particolare. Primo dei quali, non per importanza, è l'aumento della velocità di esecuzione che potesse comunque adattarsi ed essere utilizzato su qualunque tipologia di macchina. Il secondo si concentra invece sulla facilità di utilizzo da parte dell'utente finale, obiettivo messo in primo piano da molti sviluppatori di librerie.

Al termine del tirocinio formativo e con il risultato della progettazione della libreria sono stati raggiunti entrambi gli obiettivi che furono posti tre mesi prima. Il primo, la realizzazione di un software veloce è stato raggiunto mediante l'introduzione di due paradigmi di programmazione che puntano proprio alla velocità di esecuzione come visto nei paragrafi 2.2 e 2.3: la programmazione concorrente e parallela, che fa uso di più thread di esecuzione per suddividere un problema in più sottoproblemi

da svolgere in parallelo introducendo quindi il concetto di multi-tasking e la programmazione generica, che mediante l'utilizzo delle classi e dei metodi templetizzati o "modelli" risolti a tempo di compilazione anzichè a tempo di esecuzione migliora anche il riutilizzo del codice e il refactoring.

I risultati ottenuti in questa tesi rappresentano un primo passo esplorativo nella direzione dello studio della velocità computazionale e del confronto di essa a parità di prestazioni della macchina. La velocità computazione infatti tiene conto: delle strutture dati utilizzate, per gli storage e per il reticolo discreto e della quantità di celle, particelle e distribuzioni di velocità di cui è composta la griglia, ovvero dalla tipologia di reticolo scelto dall'utente.

Bibliografia

[Fow02] Martin Fowler. *UML distilled. Guida rapida al linguaggio di modellazione standard.* Pearson; 4 edizione, 2002.

[KB01] Robert C. Martin e Martin Fowler Kent Beck. Manifesto for agile software development. 2001.

[Mar02] Rober C. Martin. *Agile Software Development: Principles, Patterns, and Practices.* Pearson; 1st edition, 2002.