A.D. 1308
unipg
UNIVERSITÀ DEGLI STUDI
DI PERUGIA

UNIVERSITÀ DEGLI STUDI DI PERUGIA

DIPARTIMENTO DI INGEGNERIA

Corso di Laurea Triennale in

INGEGNERIA INFORMATICA ED ELETTRONICA

Progetto e sviluppo di un algoritmo per la visualizzazione di alberi in ambiente 3D immersivo

Relatore:

Prof. Emilio Di Giacomo

Candidato:

Riccardo Nicolini

ANNO ACCADEMICO 2022/2023

Indice

Introduzione

La visualizzazione dell'informazione è una disciplina che si occupa di rappresentare dati astratti in forma visuale, ricoprendo il ruolo di intermediario tra il complesso mondo dei dati astratti e la comprensione umana, soprattutto in un'epoca in cui la mole di dati generati e raccolti cresce in modo esponenziale. La visualizzazione aiuta a trasformare ingenti volumi di dati in informazioni facilmente leggibili. Tale processo, grazie all'avanzamento delle tecnologie di visualizzazione mediante realtà virtuale, conferisce la possibilità di interagire con i dati in modo sempre più immersivo.[1]

Nell'età dell'informazione attuale, la rappresentazione e l'interazione con dati complessi attraverso ambienti tridimensionali immersivi sta acquisendo un'importanza sempre maggiore. Tale tendenza è riscontrabile in numerosi ambiti applicativi, tra cui anche l'analisi dei dati. In questo ambito, la visualizzazione di dati modellabili come grafi o alberi riveste un ruolo di notevole importanza.

L'obiettivo della presente tesi è l'implementazione di un algoritmo per la visualizzazione di alberi in realtà immersiva tridimensionale. L'algoritmo implementato mira a produrre rappresentazioni di alberi in volume compatto ed ha complessità lineare nel numero di vertici dell'albero. L'algoritmo prende in input un albero e produce in output un layout dell'albero di input, cioè un'assegnazione di coordinate ai suoi vertici; dopodiché, mediante l'utilizzo di software ausiliari come Unity e Blender, si passa al rendering grafico dell'albero preso in input.

La prima parte della tesi volge ad analizzare i già esistenti algoritmi di visualizzazione bidimensionale di alberi, attuando una selezione accurata con lo scopo di scegliere gli algoritmi più funzionali per l'obiettivo della tesi.

Il corpo centrale dell'elaborato mira all'adattamento di tali algoritmi per l'applicazione in ambiente tridimensionale, utilizzando la complessità e l'occupazione spaziale come metodo di valutazione degli algoritmi ottenuti. Simultaneamente, si dettaglia l'approccio proposto, descrivendo il processo di progettazione e sviluppo degli algoritmi. La parte successiva volge all'utilizzo di tali algoritmi al fine di generare un rendering grafico visualizzabile con l'ausilio di software in realtà immersiva.

La parte finale della tesi analizza, a seguito di sperimentazioni, il comportamento degli algoritmi, in termini di tempo di esecuzione e volume occupato dal layout dell'albero.

La tesi è articolata in cinque capitoli:

- Capitolo 1: Nel primo capitolo viene introdotta la visualizzazione dei grafi, e nella fattispecie, degli alberi, andando ad analizzare algoritmi di visualizzazione bidimensionale esistenti.

- Capitolo 2: In questo capitolo vengono descritti, in termini di pseudo codice, gli algoritmi che permettono la generazione di un layout per la rappresentazione tridimensionale. Verrà, in seguito, analizzata la complessità e l'occupazione di volume di tali algoritmi.

- Capitolo 3: Nel terzo capitolo vengono forniti i dettagli dell'implementazione effettiva degli algoritmi, realizzata in Java, e il codice necessario per la realizzazione del modello 3D dell'albero.

- Capitolo 4: Il penultimo capitolo riguarda l'attività sperimentale, che mira ad analizzare le prestazioni degli algoritmi implementati.

- Capitolo 5: Nell'ultimo capitolo vengono riportate le conclusioni e possibili miglioramenti e sviluppi futuri.

Capitolo 1

Visualizzazione di Grafi e Alberi

In questo capitolo si parlerà della visualizzazione di alberi e grafi, due strutture dati ampiamente utilizzate e tra le più potenti nell'ambito della rappresentazione dell'informazione. Prima di addentrarci nelle tecniche di visualizzazione, è giusto comprendere brevemente cosa sono gli alberi e i grafi.[2]

I grafi sono strutture matematiche utilizzate per modellare sistemi reali, quali, le reti sociali, reti di trasporto, reti di calcolatori. Un grafo è composto da vertici ed archi che collegano coppie di nodi, che rappresentano le relazioni tra di essi. A seconda della natura applicativa, i grafi possono essere diretti o non diretti, e possono includere pesi sugli archi per indicare il costo della connessione.

Gli alberi sono grafi aciclici e connessi che rappresentano una relazione gerarchica tra elementi. Negli alberi radicati esiste un nodo chiamato radice ed ogni nodo è connesso mediante un percorso unico alla radice. Gli alberi radicati consentono di rappresentare intuitivamente una struttura gerarchica.

1.1 Visualizzazione di Grafi e Reti

La visualizzazione di grafi è una componente essenziale nell'ambito dell'analisi dei dati, essa mira a trasformare rappresentazioni astratte di dati in forme visivamente comprensibili.

Come già osservato nell'introduzione del capitolo, i grafi sono delle strutture matematiche composte da nodi ed archi, i quali a loro volta possono essere orientati, ovvero percorribili in una sola direzione, e pesati, includendo dei pesi per determinarne il costo di attraversamento.

Esistono diverse tecniche di visualizzazioni per i grafi, che si differenziano sia per le convenzioni grafiche adottate che per le tecniche algoritmiche utilizzate. Una convenzione

grafica stabilisce le regole per la rappresentazione dei vertici e degli archi del grafo. Una delle convenzioni grafiche più utilizzate è sicuramente la convenzione node-link, in base alla quale i vertici sono rappresentati da forme quali piccoli cerchi o quadrati e gli archi sono rappresentati come curve che connettono i vertici. Casi particolari della convenzione node-link sono la convenzione straight-line (in cui le curve che rappresentano gli archi sono segmenti), la convenzione polyline (in cui gli archi sono rappresentati come linee spezzate semplici), la convenzione orthogonal (in cui gli archi sono rappresentati come una sequenza di segmenti orizzontali e verticali). Altre convenzioni grafiche sono la rappresentazione mediante matrici di adiacenza, o le rappresentazioni mediante intersezioni e contatti, in cui le adiacenze tra i vertici sono rappresentate come intersezioni contatti tra oggetti geometrici.

Una delle tecniche più utilizzate per il calcolo di disegni straight-line di grafi è la tecnica force-directed che modella il grafo come un insieme di corpi soggetti a forze attrattive e repulsive che viene fatto evolvere fino ad una configurazione ad energia minima.

1.2 Visualizzazione di Alberi

Come illustrato ad inizio capitolo, gli alberi sono un caso particolare di grafi aciclici e connessi ed essi sono fondamentali per la rappresentazione di **dati strutturati gerarchicamente**. Una rappresentazione visiva efficace degli alberi può migliorare significativamente la comprensione e l'analisi di dati; va quindi evidenziata l'importanza di algoritmi che ottimizzino aspetti come l'utilizzo dello spazio, l'estetica e la leggibilità di un albero.

Esistono svariate tecniche per la generazione di layout di alberi, di seguito saranno elencati alcuni dei metodi possibili per adempiere a tale scopo, ponendo più importanza sulle tecniche di maggiore rilievo per la tesi.

LAYOUT BASATO SU LIVELLI

I layout basati su livelli organizzano l'albero in righe orizzontali (o verticali), ciascuna corrispondente a un livello diverso dell'albero, facilitandone l'identificazione. Gli algoritmi che sfruttano questo principio cercano di distribuire uniformemente i nodi all'interno di ciascun livello e di minimizzare la larghezza (o l'altezza) totale dell'albero, rendendo questa tecnica particolarmente adatta per visualizzare strutture dati gerarchiche in modo intuitivo e organizzato.

- I principali pregi di questo metodo sono la sua leggibilità e la rappresentazione gerarchica intuitiva. Essendo un metodo ampiamente utilizzato e documentato, esso favorisce una facile implementazione.

- I limiti del layout basato su livelli sorgono principalmente a causa della scalabilità, poiché per alberi molto grandi la visualizzazione può diventare ingombrante, e dall'uso inefficiente dello spazio, specialmente in casi di alberi irregolari, questa tecnica può portare ad un utilizzo inefficiente dello spazio.

DISEGNI H-V

La tecnica di visualizzazione di alberi HV (Horizontal-Vertical) è utilizzata per disegnare alberi binari (cioè alberi in cui ogni nodo ha al più due figli). In questo tipo di disegni gli archi sono rappresentati come segmenti orizzontali o verticali; pertanto un nodo si troverà allineato verticalmente o orizzontalmente con il nodo padre, cercando di ottimizzare l'uso dello spazio e migliorarne la leggibilità. Ogni livello dell'albero può essere orientato orizzontalmente o verticalmente, a seconda della strategia adottata per la visualizzazione.

- Il vantaggio di questa rappresentazione è sicuramente l'efficienza nell'utilizzo dello spazio poiché, alternando gli orientamenti, può sfruttare lo spazio disponibile in modo più efficace rispetto ad altri metodi.

- Il principale difetto di tale tecnica è il fatto che essa non mostra chiaramente la struttura a livelli dell'albero. Visto che i due figli di un nodo sono allineati uno orizzontalmente e uno verticalmente con il padre, i livelli dell'albero non sono mostrati su linee orizzontali distinte.

SPACE FILLING

Le tecniche di visualizzazione Space Filling rappresentano un albero in modo che ogni parte dello spazio disponibile venga utilizzato. A differenza dei metodi basati su nodi e collegamenti, dove i nodi sono rappresentati da simboli ed i collegamenti da linee, i metodi Space Filling sfruttano l'intera area di visualizzazione allocata per mostrare parti della struttura dati; in questo modo viene massimizzato l'utilizzo dello spazio disponibile per la rappresentazione.

L'esempio più caratteristico ed utilizzato di questo tipo di tecniche è il **TreeMap**, nel quale lo spazio viene diviso in rettangoli che rappresentano i nodi dell'albero e ciascun rettangolo è proporzionale ad un valore quantitativo specifico, come per esempio la dimensione di un documento di un File System.

- I vantaggi nell'utilizzo di questa rappresentazione sono sicuramente l'efficienza nell'utilizzo dello spazio, dal momento che ogni parte dello spazio disponibile è sfruttata, e la rappresentazione quantitativa, essendo particolarmente efficaci nel rappresentare dimensioni quantitative dei dati, permettendo confronti chiari e veloci.

- Lo svantaggio principale sta nella comprensione dei dati: mentre sono efficaci nel mostrare le dimensioni relative dei dati, possono rendere più difficile l'interpretazione di relazioni specifiche tra i nodi.

ALTRI METODI

Esistono diversi altri metodi utilizzati nell'ambito della visualizzazione di alberi, come ad esempio:

- Rappresentazione **radiale**, nel quale l'albero è disposto in un layout circolare, con la radice al centro e i nodi figli disposti attorno ad essa in cerchi concentrici.

- Visualizzazione **Force-Directed** applica algoritmi force-directed per posizionare i nodi in modo che tutti gli archi abbiano circa la stessa lunghezza.

In conclusione, ogni tecnica possiede punti di forza e punti di debolezza, essi possono essere scelti in base alla natura dei dati e agli obbiettivi della visualizzazione.

1.2.1 Algoritmi per la visualizzazione di Alberi

In questa sezione verranno analizzati in dettaglio tre algoritmi basati sulle tecniche discusse precedentemente, poiché centrali nello sviluppo della tesi, nello specifico **Layered Tree Drawing**, l'**HV-Tree Drawing** ed il **TreeMap**.

1.2.1.1 Layered Tree Drawing

Il layered Tree Draw è un algoritmo basato sulla visualizzazione a livelli. L'albero è organizzato in linee orizzontali, ciascuna corrispondente ad un livello diverso; esso prende in input un albero e restituisce come output un layout layered di disegno.

L'algoritmo presenta dei criteri estetici da rispettare, quali:

- Downwarderss: Ogni figlio deve essere più in basso rispetto al padre;

- Planarità: Gli archi non si devono incrociare tra di loro;

- Simmetria: Sottoalberi uguali devono essere disegnati in maniera congruente.

Il layer Tree Draw è un algoritmo ricorsivo che sfrutta il paradigma **divide et impera**, ovvero ogni problema viene suddiviso in più sottoproblemi indipendenti di dimensione inferiore fino ad arrivare ad un caso base, successivamente viene effettuata una ricostruzione della soluzione finale ricombinando le soluzioni dei sottoproblemi.

Nell'algoritmo il *caso base* corrisponde ad un sottoalbero costituito da un solo nodo; se il sottoalbero possiede più di un nodo, l'algoritmo viene chiamato ricorsivamente sulla radice del sottoalbero.

Nella fase di combinazione, i sottoalberi sono disegnati in modo che la distanza tra di essi sia almeno di 2, la radice viene posta verticalmente di una unità più in alto e orizzontalmente centrata rispetto alle radici dei sottoalberi. (Figura 1.1)

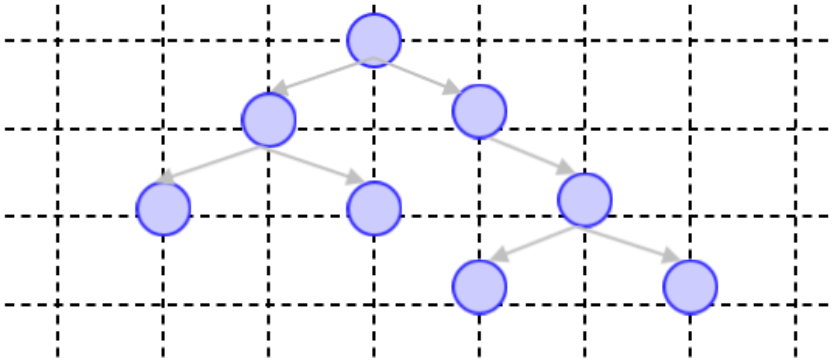

Figura 1.1: Esempio di `Layer-Tree-Draw`.

Di seguito lo pseudocodice dell'algoritmo.

Algorithm 1: Layered-Tree-Draw(T)

```
2 Siano T_1, T_2, ..., T_m i sottoalberi di T
3 for i ← 1 to m do
4     if T_i ha solo un vertice then
5         Disegna T_i //caso base;
6     else
7         Layered-Tree-Draw(T_i) //Chiamata ricorsiva per il sottoalbero T_i;
8         Posiziona T_i a destra di T_{i-1} ad una distanza orizzonatle di 2, posiziona la
            radice di T a metà tra la radice del sottoalbero T_1 e la radice del
            sottoalbero T_m
```

E' facile vedere che l'algoritmo ha una complessità $O(n)$, dove n rappresenta il numero di nodi che costituiscono l'albero. Questo perché l'algoritmo visita ogni nodo una sola volta.

Per quanto riguarda l'**area** utilizzata, l'algoritmo occupa al massimo $O(n^2)$

In conclusione, Layer-Tree-Draw è un algoritmo che calcola un disegno Γ di un albero generico tale che:

- Γ è downward, planare e simmetrico;
- Tutti i vertici sono distanziati almeno di 1;
- La complessità è $O(n)$;
- L'area occupata è $O(n^2)$.

1.2.1.2 HV-Tree Drawing

L'algoritmo HV-Drawing è usato per rappresentare alberi binari; in questo tipo di rappresentazioni gli archi sono segmenti orizzontali o verticali. L'algoritmo può essere adattato per operare anche su alberi con un numero qualsiasi di figli, ma in tal caso, ovviamente, gli archi devono essere rappresentati con segmenti obliqui.

L'obiettivo di tale algoritmo sta nel limitare l'occupazione dell'area totale, ed anche esso sfrutta un approccio ricorsivo mediante la tecnica del divide et impera.

Il *caso base* dell'algoritmo corrisponde ad un sottoalbero costituito da un solo nodo; se il sottoalbero è costituito da più nodi l'algoritmo viene chiamato ricorsivamente su ogni nodo.

La combinazione dei sottoalberi sfrutta logiche diverse, un esempio è Right-Heavy-Tree, nel quale il sottoalbero più grande viene posto all'estrema destra rispetto a tutti i sottoalberi figli dello stesso nodo.

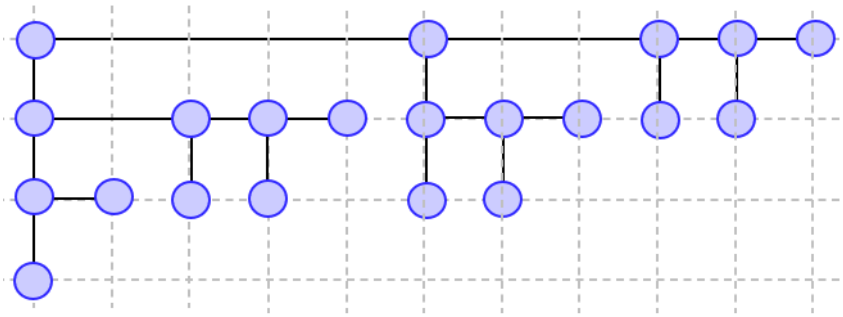

Figura 1.2: Esempio di `HV-Tree-Draw`.

Di seguito lo pseudocodice dell'algoritmo.

Algorithm 2: HV-Tree-Draw(T)

```
2 Sia T un albero binario
3 if T ha solo un vertice then
4 |   Disegna T //caso base;
5 else
6 |   Γ_s ← HV-Tree-Draw(T_s) //Chiamata ricorsiva sottoalbero sinistro;
7 |   Γ_d ← HV-Tree-Draw(T_d) //Chiamata ricorsiva sottoalbero destro;
8 |   combina Γ_s e Γ_d;
```

Anche in questo caso, la **complessità** dell'algoritmo è lineare ma, a differenza del metodo precedente, si può dimostrare che l'**area** massima occupata è $O(n\,log n)$, questo perché la larghezza del layout sarà al massimo $n-1$ e l'altezza al massimo $log n$.

In conclusione, HV-Tree-Draw è un algoritmo che calcola un disegno Γ di un albero binario tale che:

- Γ è downward, planare e simmetrica;
- La complessità è $O(n)$;
- L'area occupata è $O(n\,log n)$.

1.2.1.3 TreeMap

I treemap sono delle tecncihe di visualizzazione basate sullo space-filling. il quale mira a sfruttare l'intera area di visualizzazione disponibile.

Nello specifico, i treemap rappresentano i nodi mediante rettangoli annidati, raffiguranti la gerarchia, di dimensioni proporzionali rispetto al valore quantitativo del nodo stesso.

L'algoritmo che implementa i treemap è di natura ricorsiva. Partendo da un'area allocata, lo spazio disponibile viene suddiviso tra i nodi figli della radice secondo tecniche trattate in seguito; successivamente, nel caso in cui un nodo abbia figli, l'algoritmo viene chiamato ricorsivamente sullo spazio allocato precedentemente ai figli.

Qui di seguito riportato lo pseudocodice nel caso generale.

Esistono diverse tecniche per assegnare lo spazio occupato ad ogni singolo nodo, le più note sono:

Algorithm 3: Treemap(n, *space*)

```
1 //n è il nodo corrente e space è lo spazio disponibile;
2 for each m figlio di n do
3     Calcola il bound di m secondo space //Varia in base all'algoritmo scelto;
4     Assegna il bound ad m;
5     Rimuovi il bound da space //Diminuisce lo spazio disponibile per i nodi
        fratelli;
6     if m ha figli then
7         Treemap(m, m.bound);
```

- Slice and Dice
- Squarified
- Strip
- Pivot
- Split

Verranno analizzati in dettaglio gli algoritmi di **Slice and Dice** e **Split**, poiché utilizzati nello specifico nella presente tesi.

SLICE AND DICE

Nello Slice and Dice, ogni rettangolo associato ad un nodo viene diviso tramite linee parallele in rettangoli più piccoli rappresentanti i figli del nodo, basando la divisione sulla dimensione dei figli. Ad ogni livello della gerarchia, l'orientazione delle linee viene alternata tra orizzontale e verticale.

Prendendo come esempio l'albero in figura 1.3 è possibile applicare l'algoritmo Slice and Dice ed ottenere un risultato visibile in figura 1.4

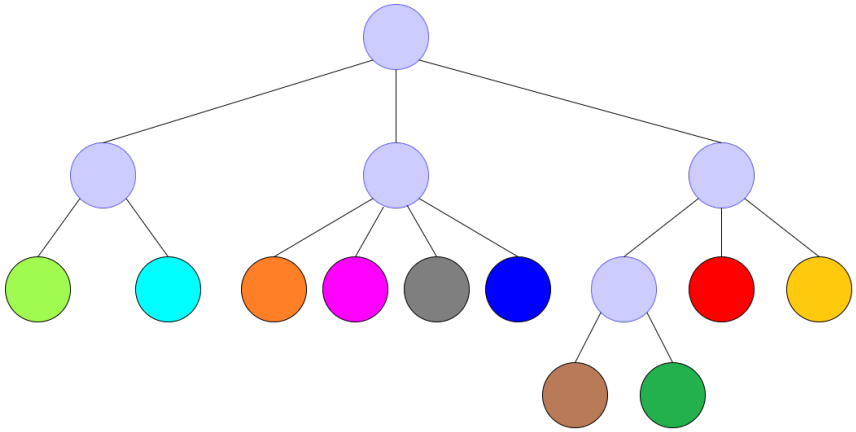

Figura 1.3: Esempio di `albero generico`.

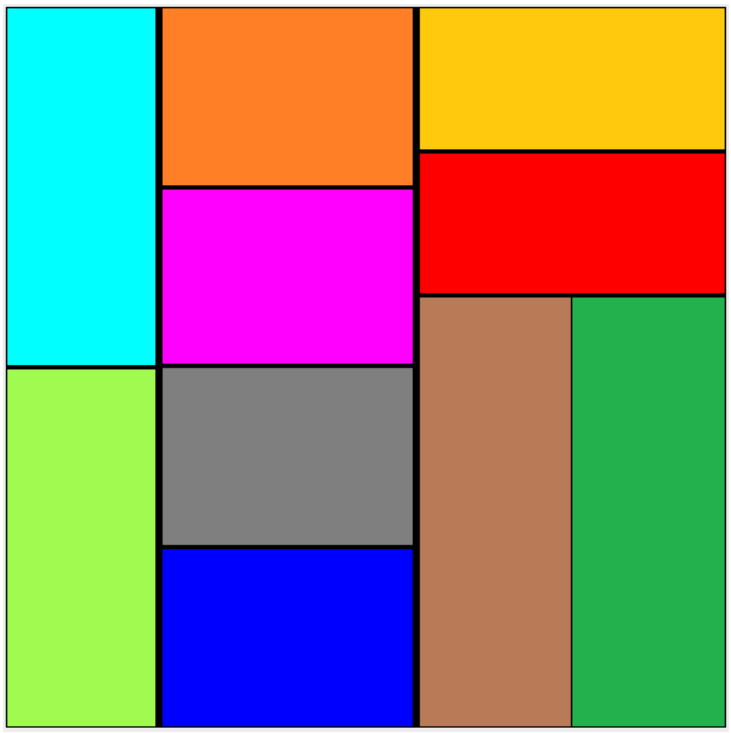

Figura 1.4: Esempio di `TreeMap` utilizzando Slice and Dice.

Lo Slice and Dice è l'algoritmo più semplice da implementare e possiede una complessità lineare. Di contro, specialmente per dataset molto grandi, possiede un rapporto di forma dei rettangoli pessimo.

In conclusione, l'algoritmo che sfrutta la logica Slice and Dice disegna un treemap tale che:

- L'ordinamento dei nodi è preservato;
- Ha una buona leggibilità per dataset piccoli;
- La complessità è $O(n)$;
- L'area occupata è $\Theta(n)$;
- L'aspect Ratio dei rettangoli è pessimo.

SPLIT

Lo split è un algoritmo che ha l'obiettivo di migliorare il rapporto di forma pessimo dello Slice and Dice.

La regione di spazio disponibile R viene divisa da una linea, orizzontale o verticale, in due regioni R_1 e R_2 in funzione della larghezza e dell'altezza dello spazio disponibile. Quindi se la larghezza disponibile è maggiore dell'altezza, lo spazio viene diviso tramite una linea verticale, avviene il contrario se l'altezza è maggiore della larghezza.

L'insieme dei nodi figli S è poi diviso in due sottoinsiemi S_1 e S_2 in modo che, basandosi sulla dimensione dei figli, l'Area totale(S_1) sia simile all'area totale(S_2).

Riprendendo l'albero esemplificativo in figura 1.3, applicando l'algoritmo Split si ottiene un layout di questo tipo:

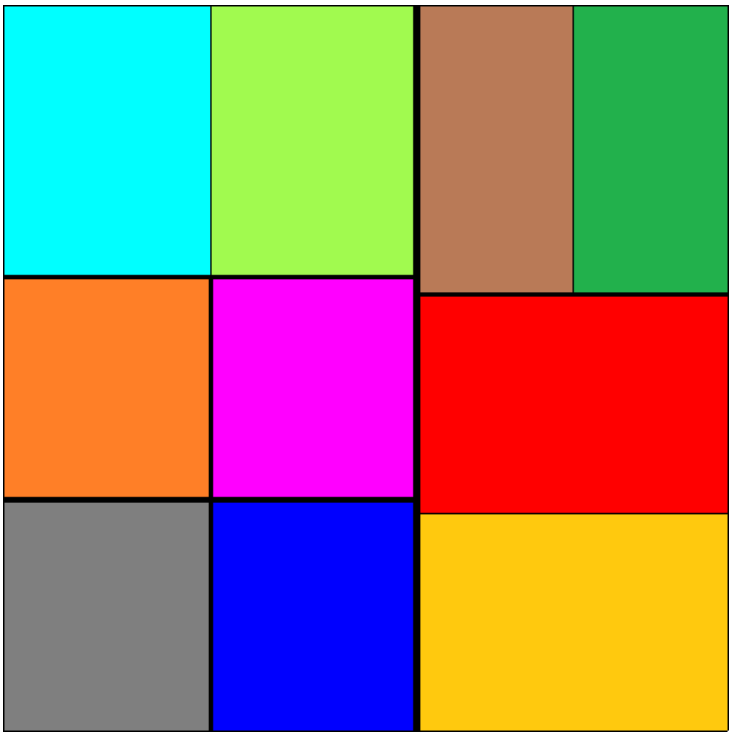

Figura 1.5: Esempio di `TreeMap` utilizzando Split.

Lo Split è un metodo che può essere implementato con complessità lineare, possiede un rapporto di forma buono (circa 2.5). D'altra parte, l'ordinamento viene parzialmente preservato.

Riassumendo l'algoritmo che sfrutta la logica Split disegna un treemap tale che:

- L'ordinamento dei nodi è parzialmente preservato;
- La complessità è $O(n)$;
- L'area occupata è $\Theta(n)$;
- L'aspect Ratio dei rettangoli è buono (≈ 2.5).

1.3 Visualizzazione in Ambiente Immersivo

L'obiettivo principale della visualizzazione in ambiente immersivo è quello di fornire agli utenti un modo più naturale e intuitivo di esplorare e comprendere grandi volumi di dati. Questo poiché attraverso l'utilizzo di dispositivi immersivi, come i visori, gli utenti possono letteralmente "entrare" nel dataset, ciò non solo facilita la comprensione ma apre nuove prospettive per l'analisi e l'interpretazione dei dati.

L'ambiente immersivo rappresenta un'avanguardia tecnologica nella rappresentazione dell'informazione. Essa sfrutta le potenzialità della realtà virtuale (VR), realtà aumentata (AR) e della realtà mista (MR), permettendo esperienze visive più intuitive e coinvolgenti.

Occorre far chiarezza sulle differenze fra queste tre tipologie:

- **Realtà Virtuale (VR)**: La VR crea un ambiente digitale completamente immersivo sostituendo il mondo reale; l'utente è quindi totalmente immerso in un'esperienza virtuale.

  Questa peculiarità la rende particolarmente adatta ad applicazioni come simulazioni di formazione (ad esempio per medici o piloti) e nelle esperienze di intrattenimento (come le visite ai musei).

- **Realtà Aumentata (AR)**: L'AR sovrappone elementi digitali al mondo reale, consentendo l'interazione di informazioni virtuali posizionate nell'ambiente fisico.

  Il vantaggio principale è la capacità di integrare oggetti virtuali nel contesto del mondo reale senza isolare l'utente dall'ambiente esterno.

- **Realtà Mista (MR)**: La MR combina la logica AR e VR in modo che oggetti reali e virtuali possano coesistere ed interagire fra di loro.

  La Mixed Reality trova la principale applicazione nella progettazione industriale, consentendo, ad esempio, di visualizzare come una parte meccanica virtuale si adatti ad un contesto reale.

Riassumendo, mentre la realtà virtuale offre una completa immersione in un mondo virtuale, la realtà aumentata arricchisce il mondo reale con elementi virtuali e la realtà mista crea un'esperienza ibrida che permette ad oggetti fisici e digitali di interagire.

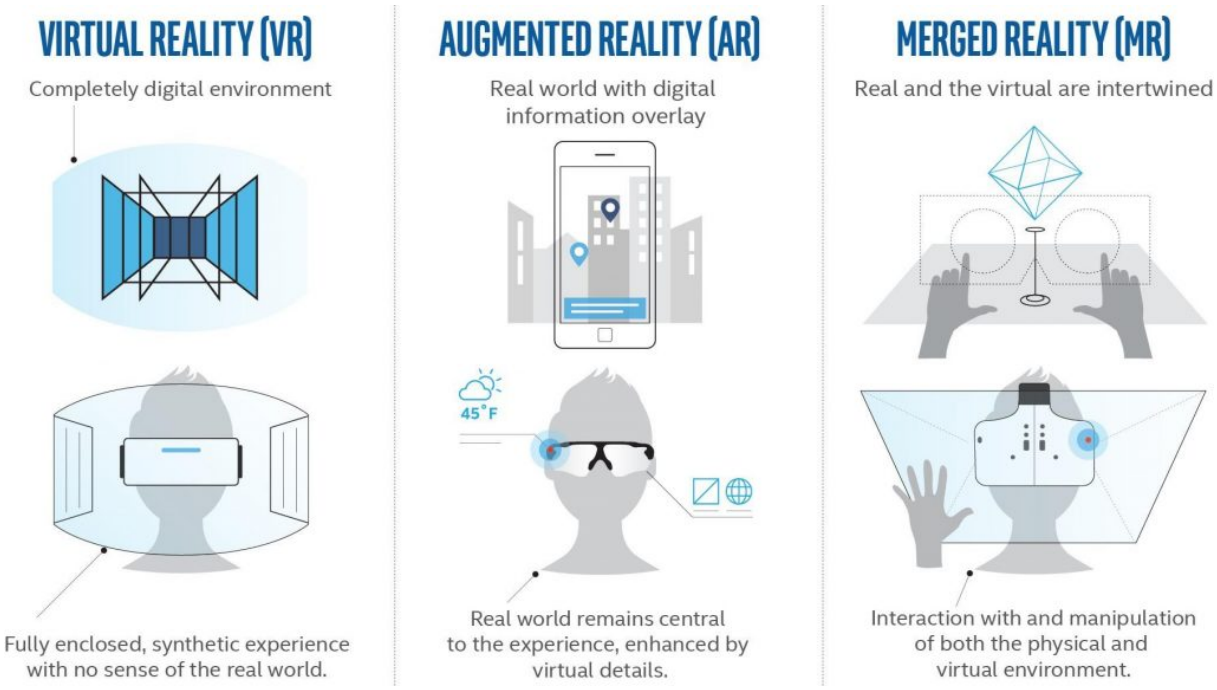

Figura 1.6: Differenza tra VR, AR, MR

Capitolo 2

Algoritmi per la visualizzazione di alberi in 3D

In questo capitolo, si analizzeranno gli algoritmi utilizzati per la produzione di un layout tridimensionale rappresentante un albero. Nella fattispecie, verranno sfruttati gli algoritmi di visualizzazione bidimensionale di alberi generici, discussi nel capitolo precedente, adattandoli ad una versione tridimensionale.

Nelle seguenti sezioni verranno analizzati gli algoritmi necessari per il raggiungimento dell'obiettivo, l'analisi della complessità computazionale e l'occupazione di volume generata da ogni metodo.

2.1 Algoritmi implementati

Gli algoritmi sfruttano tutti lo stesso principio di funzionamento generale; ad ogni nodo viene associato un confine bidimensionale, secondo tecniche algoritmiche differenti, il quale viene rappresentato da un rettangolo.

Dopodiché, grazie ai bound (confini bidimensionali) assegnati ad ogni nodo, vengono calcolate le coordinate x e y per il posizionamento del vertice nello spazio. La coordinata z viene calcolata tramite la profondità del nodo all'interno dell'albero, assegnando ad ognuno di essi un'altezza nello spazio.

Tale metodo permette di generare una struttura visiva gerarchica, facilitando l'interpretazione delle relazioni genitoriali tra i nodi.

Di seguito viene fornito lo pseudocodice per l'assegnazione delle coordinate z.

Algorithm 4: Z-Coordinates(T)

```
1 M = 0;
2 for each n di T do
3 |   if n.depth > M then
4 |   |   n.depth = M;
5 for each n di T do
6 |   n.z = (M - n.depth) //Imposta la coordinata z in base alla profondità;
```

L'algoritmo generico prende in input un albero qualunque e genera in output un layout tridimensionale rappresentante la posizione dei nodi nello spazio.

Algorithm 5: 3D-Algorithm(T)

```
1 treeBounds(T) //calcola i bound secondo una delle tecniche utilizzate;
2 Z-Coordinates(T) //calcola la coordinata z di ogni nodo
```

Ogni nodo dell'albero avrà associato un confine bidimensionale (o bound) rappresentante la sua occupazione nel piano xy in base alla dimensione. La differenziazione principale tra gli algoritmi è determinata dal metodo utilizzato per calcolare i bound occupati dai nodi.

2.1.1 Layer Tree Draw 3D

Dal momento che il layer Tree Draw è un algoritmo di visualizzazione prettamente bidimensionale, va ideato un metodo per conferire al layout finale profondità e poter sfruttare al meglio la dimensione spaziale aggiuntiva.

L'idea più semplice ed intuitiva, che può essere sfruttata, è cambiare orientamento di disposizione dei nodi per ogni livello dell'albero. In questo modo si conferisce tridimensionalità mantenendo una logica semplice ed il più possibile affine ai metodi di disegno layered.

L'algoritmo sfrutta quindi l'idea classica del layer tree a cui però va aggiunta una logica per definire i bound di ogni nodo.

L'algoritmo opera in due fasi. Nella prima fase si esegue una visita in post-ordine dell'albero e si calcolano le coordinate di ogni nodo relativamente alla posizione del padre.

Le coordinate relative dei nodi vengono calcolate in base all'occupazione di volume dei sottoalberi.

Il caso base si presenta quando un nodo non ha figli, assegnando ad esso un bound di dimensioni fissate.

Se il nodo ha figli, la funzione calcola i confini (o bound) di questi figli ricorsivamente. L'orientamento e la disposizione dei figli dipendono dalla profondità del nodo nell'albero: a profondità pari, i figli sono allineati lungo la direzione x; a profondità dispari, sono allineati lungo la direzione y. Ad ogni nodo vengono assegnate delle coordinate in funzione del genitore.

Va fatto notare che in questa prima fase vengono assegnati dei **bound relativi**, ovvero calcolati in funzione del nodo genitore, ma che non rappresentano il posizionamento finale del modello.

Dopo aver calcolato i bound relativi, l'algoritmo assegna ad ogni nodo un **bound assoluto** che definisce la sua posizione e dimensione globalmente.

La funzione procede in modo ricorsivo, ma invece di calcolare i confini (o bound) in base al genitore, aggiorna i limiti di ogni nodo utilizzando la posizione assoluta calcolata a partire dalla radice dell'albero. Questo assicura che ogni nodo abbia una posizione definita nel contesto globale, indipendentemente dalla struttura dell'albero.

Di seguito è presentato un esempio di pseudocodice delle due fasi.

Algorithm 6: relativeBounds(n, $xPosition$, $yPosition$)

```
1  Sia n un nodo dell'albero
2  Siano xPosition e yPosition la posizione iniziale
3  if n non ha figli then
4      Assegna il bound base ad n in posizione (xPosition, yPosition)
5  else
6      for each m figlio di n do
7          if n è a profondità pari then
8              relativeBounds(m, xPosition + offset, yPosition);
9              Incrementa offset in base alla larghezza di m;
10             Imposta maxSize come nodo con altezza più grande;
11         else
12             relativeBounds(m, xPosition, yPosition + offset);
13             Incrementa offset in base all' altezza di m;
14             Imposta maxSize come nodo con larghezza più grande;
15     if n è a profondità pari then
16         Assegna il bound ad n ponendolo in mezzo ai figli (secondo X) e di
           dimensioni appena calcoalte
17     else
18         Assegna il bound ad n ponendolo in mezzo ai figli (secondo Y) e di
           dimensioni appena calcoalte
```

Algorithm 7: absoluteBounds(n, $xPosition$, $yPosition$)

```
1  Sia n un nodo dell'albero
2  Siano xPosition e yPosition le coordinate in cui posizionare la radice
3  Assegna alla radice le coordinate xPosition e yPosition
4  if n ha figli then
5      for each m figlio di n do
6          if n è a profondità pari then
7              absoluteBounds(m, xPosition + offset, yPosition);
8              Incrementa offset in base alla larghezza di m;
9          else
10             absoluteBounds(m, xPosition, yPosition + offset);
11             Incrementa offset in base all'altezza di m;
```

2.1.2 TreeMap 3D - Slice And Dice

Lo Slice and Dice è un metodo intuitivo per costruire treemap, che sfrutta un approccio sistematico per assegnare spazio a ciascun nodo di un albero in base alla sua importanza relativa. L'algoritmo inizia dalla radice dell'albero e procede verso il basso, assegnando ad ogni nodo uno spazio proporzionale al suo valore o peso all'interno dello spazio disponibile (Algoritmo 3). Ecco una descrizione dettagliata dell'algoritmo:

- L'algoritmo riceve come input la radice dell'albero e lo spazio totale disponibile per l'allocazione dei confini dei nodi. Lo spazio disponibile può essere rappresentato come un rettangolo all'interno del quale ogni nodo deve essere inserito.

- Per ogni nodo figlio della radice, l'algoritmo calcola il confine (o bound) basandosi sullo spazio disponibile in quel momento. Questo passaggio determina quanto spazio sarà allocato a ciascun nodo, proporzionalmente alla sua dimensione.

- Dopo aver assegnato il confine a un nodo, lo spazio che esso occupa viene rimosso dallo spazio disponibile totale. In questo modo, il nodo successivo calcolerà il proprio confine basandosi sullo spazio rimasto, assicurando che l'allocazione dello spazio sia gestita in modo efficiente e che non ci siano sovrapposizioni tra i nodi.

- Se il nodo corrente ha dei figli, l'algoritmo viene applicato ricorsivamente a ciascuno di essi. Lo spazio disponibile per i figli di un nodo è definito dal confine che è stato assegnato al nodo padre. Questo assicura che l'intero spazio assegnato a un nodo sia utilizzato per ospitare i suoi figli, mantenendo la struttura gerarchica dell'albero.

L'algoritmo che rende possibile la divisione secondo il metodo Slice and Dice è il seguente:

Algorithm 8: sliceAndDice(n, *availableSpace*, *size*)

```
1 Sia n il nodo in cui è invocato il metodo;
2 Sia availableSpace l'area disponibile;
3 Sia size la dimensione totale dei nodi fratelli da allocare;
4 if n è a profondità dispari then
5 |   Calcola il bound tramite la dimensione del nodo e size, dividendo lo spazio
  |     disponibile con una linea verticale
6 else
7 |   Calcola il bound tramite la dimensione del nodo e size, dividendo lo spazio
  |     disponibile con una linea orizzontale
8 return bound
```

Una volta calcolati i confini (o bound) di ogni nodo, può essere invocato il metodo per calcolare la coordinata z del layout tridimensionale finale. (Algoritmo 4)

Bisogna evidenziare che, nonostante la sua semplicità implementativa, lo Slice and Dice presenta delle limitazioni in termini di estetica visiva e leggibilità man mano che le dimensioni dell'albero aumentano. Questo metodo, infatti, può generare confini (o bound) con proporzioni meno ottimali per i nodi, specialmente in alberi con molti livelli o nodi. Tale situazione deriva dalla divisione sequenziale e proporzionale dello spazio, che può portare a rettangoli molto stretti e lunghi per alcuni nodi, rendendo il layout finale più complesso da interpretare.

2.1.3 TreeMap 3D - Split

Il metodo Split rappresenta un'alternativa avanzata per la crezione di treemap, sviluppata per rispondere ai problemi sorti dal metodo Slice and Dice, in particolare per quanto riguarda il deterioramento del rapporto di forma dei rettangoli con l'aumentare delle dimensioni dell'albero. Questa metodologia adotta un approccio differente, progettato per preservare un aspect ratio più uniforme e bilanciato dei confini dei nodi, indipendentemente dalla complessità o dalla grandezza dell'albero analizzato.

Questo algoritmo si distacca significativamente dall'approccio utilizzato da Slice and Dice, soprattutto per quanto concerne la modalità di assegnazione dello spazio ai nodi.

Nel caso base dell'algoritmo Split, ovvero quando l'input è costituito da un singolo nodo, l'intero spazio disponibile viene assegnato direttamente a quel nodo, senza ulteriori

suddivisioni. Questo garantisce che, in assenza di ulteriori nodi fratelli con cui dividere lo spazio, il nodo in questione occupi l'intera area a lui destinata.

Quando, invece, l'input comprende più di un nodo, l'algoritmo procede dividendo l'insieme di nodi in due gruppi bilanciati per dimensione. A questo punto, viene calcolato e assegnato uno spazio adeguato (o bound) al primo gruppo, basandosi sulle dimensioni relative dei gruppi e le dimensioni dello spazio disponibile per determinare la porzione di spazio da allocare. Successivamente, lo spazio rimanente, risultante dalla sottrazione dello spazio allocato al primo gruppo, viene assegnato al secondo gruppo di nodi. L'algoritmo viene quindi applicato ricorsivamente a questi due gruppi, seguendo questa logica di divisione fino a raggiungere il caso base per ogni nodo.

Dopo aver definito i bound per tutti i nodi dello stesso livello, l'algoritmo Split viene invocato ricorsivamente anche per i nodi che hanno figli. In questa fase, il bound assegnato a ciascun nodo padre diventa lo spazio disponibile (available space) per i suoi figli, permettendo così di allocare lo spazio interno in maniera coerente con la struttura gerarchica dell'albero.

Di seguito lo pseudocodice dell'algoritmo split.

Algorithm 9: split(*nodes*, *availableSpace*)

```
1  Sia nodes i nodi su cui è invocato il metodo;
2  Sia availableSpace l'area disponibile;
3  if nodes ha un nodo n then
4      Assegna come bound di n tutto l'availableSpace
5      if n ha figli then
6          split(n, bound)
7  else
8      Dividi in due gruppi bilanciati nodes;
9      Assegna un bound al primo gruppo di nodi in funzione della dimensione del
         gruppo e alle dimensioni dello spazio disponibile;
10     Rimuovi il bound dall'availableSpace;
11     split(group1, bound);
12     split(group2, availableSpace);
13 return bound
```

Una volta calcolati i confini (o bound) di ogni nodo, può essere invocato il metodo per calcolare la coordinata z del layout tridimensionale finale. (Algoritmo 4).

2.2 Analisi delle Complessità

In questa sezione verranno analizzate le complessità degli algoritmi descritti precedentemente. La complessità è un argomento fondamentale, poiché fornisce una misura teorica dell'efficienza di un algoritmo in termini di tempo di esecuzione.

Nell'ambito dell'indagine sugli algoritmi implementati, emerge un tema ricorrente che caratterizza l'efficienza dei metodi: tutti presentano complessità **lineare**.

Se l'algoritmo è di complessità lineare, indicata con la notazione $O(n)$, significa che il tempo di esecuzione necessario per eseguire l'algoritmo aumenta proporzionalmente alla dimensione dell'input. Questa proprietà è particolarmente desiderabile in molti contesti applicativi, in quanto assicura che l'algoritmo possa scalare in modo efficiente con l'aumentare delle dimensioni del problema.

L'algoritmo **Layer Tree Draw 3D** è composto da due funzioni: `relativeBounds` (algoritmo 6) e `absoluteBounds` (Algoritmo 7)

- `relativeBounds` è chiamata ricorsivamente per ogni nodo dell'albero. Ogni nodo itera su tutti i suoi figli una volta; questo significa che ogni nodo dell'albero viene visitato esattamente una volta. Dal momento che le operazioni di assegnazione dei bound hanno complessità costante, $O(1)$, la complessità di questa parte di algoritmo è $O(n)$.

- `absoluteBounds` segue un approccio ricorsivo, visitando ogni nodo dell'albero esattamente una volta ed iterando poi su tutti i suoi figli. Pertanto, la complessità di questa parte è anch'essa $O(n)$

Dato che la struttura ricorsiva non ha sovrapposizioni, ovvero ogni nodo viene elaborato una volta per ogni funzione, la complessità totale dell'algoritmo sarà la somma delle due funzioni: $O(n) + O(n) = O(n)$.

Infine, considerando le operazioni di assegnazione dei bound e di calcolo del massimo costanti $O(1)$, si può concludere che la complessità totale dell'algoritmo è $O(n)$.

L'algoritmo **Slice and Dice** è composto anche esso da due parti `TreeMap` (Algoritmo 3) e `sliceAndDice` (Algoritmo 8).

- `sliceAndDice` assegna il bound al nodo preso in input, ciò implica che la complessità di questa funzione è costante, $O(1)$.

- `TreeMap` è una funzione che procede ricorsivamente attraverso tutti i nodi dell'albero. Ogni nodo viene visitato esattamente una volta, ciò implica che la funzione

`sliceAndDice` viene chiamata una volta per ogni nodo e il suo lavoro è proporzionale al numero di nodi fratelli. Nel caso peggiore, se tutti i nodi sono sullo stesso livello, la complessità è ancora lineare. La funzione TreeMap viene chiamata ricorsivamente per ogni insieme di figli di un nodo, questo significa che l'algoritmo esplora l'intero albero passando una volta per ogni nodo. La complessità di questa operazione è quindi lineare rispetto al numero dei nodi n.

In conclusione, dal momento che le operazioni chiave come l'assegnazione di un bound o la rimozione dello spazio disponibile sono costanti, l'algoritmo possiede una complessità lineare $O(n)$.

L'algoritmo **Split** visita ricorsivamente tutti i nodi dell'albero. Il caso base avviene quando la lista di nodi presi in input è costituita da un solo nodo, in questo caso l'assegnazione del bound al nodo viene eseguita in un tempo costante $O(1)$.

Nel caso in cui la lista contenga più di un nodo, viene divisa in due gruppi mediante una funzione ausiliaria.

L'algoritmo calcola poi il bound per il primo gruppo di nodi e aggiorna lo spazio disponibile rimouovendo l'area assegnata a questo gruppo. L'algoritmo viene chiamato ricorsivamente sui due gruppi di nodi. La complessità della ricorsione dipende dal numero di nodi nei gruppi e dalla profondità dell'albero. Dal punto di vista teorico, se l'albero è ben bilanciato e i nodi sono divisi equamente tra i gruppi, ogni chiamata ricorsiva gestisce metà dei nodi della chiamata precedente portando ad una complessità logaritmica rispetto al numero di divisioni. Tuttavia, poiché ogni nodo è visitato una volta nel processo di divisione, la complessità totale dell'algoritmo, in termini di operazioni sui nodi, rimane $O(n)$.

2.3 Occupazione di Volume

Come affermato ad inizio capitolo, l'occupazione di volume costituisce un parametro cardine per il confronto degli algoritmi implementati.

Dato che sia il LayerTreeDraw 3D che il TreeMap 3D generano un modello tridimensionale basato sulla profondità dell'albero, l'altezza del layout di uscita non dipende dagli algoritmi ma esclusivamente dall'albero.

Si può quindi affermare che l'altezza del modello tridimensionale sarà $O(h)$, dove h rappresenta l'altezza dell'albero.

La differenza volumetrica dei layout di uscita è quindi caratterizzata dall'area occupata per l'assegnazione dei bound ai nodi.

Nel caso del **Layer Tree Draw** bidimensionale, l'area massima occupabile è $O(n^2)$, a causa della logica di costruzione dell'algoritmo.

Allo stesso modo, per il layerTreeDraw 3D, l'area massima occupabile sul piano xy è di ordine $O(n^2)$.

Un esempio per dimostrare questa osservazione potrebbe essere un albero costituito da tre livelli; nel primo livello è presente la radice, i restanti $n-1$ nodi sono posizionati per metà nel secondo livello ed i restanti $\frac{n'}{2}$ sono tutti figli di un nodo scelto casualmente dal secondo livello.

Ottenendo quindi un albero di questo tipo:

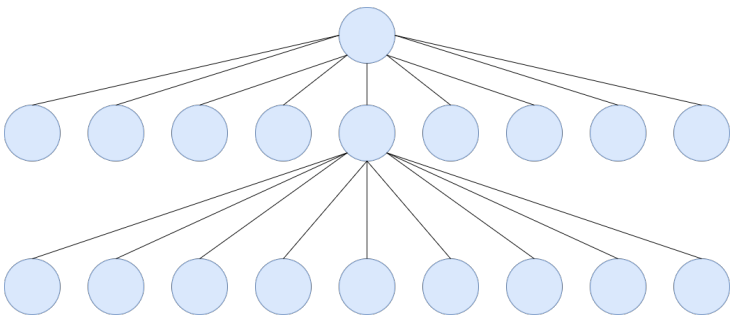

Figura 2.1: Esempio albero sbilanciato

Dal momento che l'algoritmo Layer Tree Draw 3D cambia l'orientamento dei vertici ad ogni livello, il terzo livello conterrà $\frac{n'}{2}$ nodi lungo una direzione, ed il secondo livello $\frac{n'}{2}$ nella direzione ortogonale rispetto al precedente.

Di seguito è proposta una visualizzazione tridimensionale dell'albero sbilanciato realizzato con l'algoritmo LayerTreeDraw3D (figura 2.2).

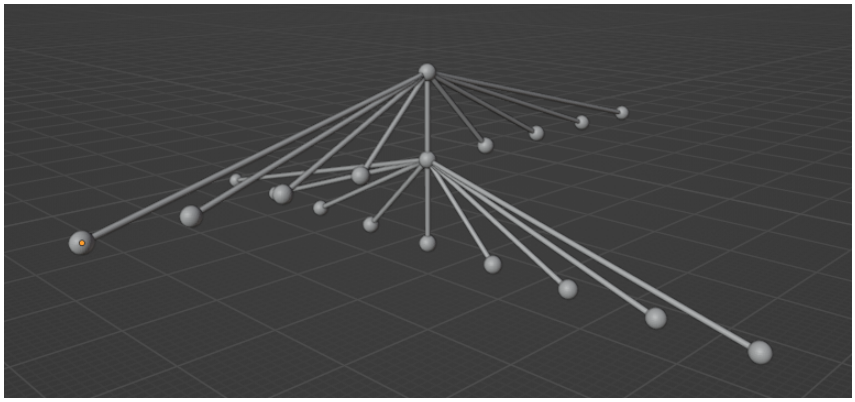

Figura 2.2: Esempio albero sbilanciato tridimensionale

In questo caso specifico, l'area occupata sul piano diventa $\frac{n^2}{4}$, dove n rappresenta il numero dei nodi dell'albero.

In conclusione, nel caso peggiore, il layer Tree Draw 3D genera un layout di un volume al massimo $O(n^2 \cdot h)$, dove h rappresenta l'altezza dell'albero e n il numero dei nodi.

Per quanto riguarda il **TreeMap**, per definizione, gli algoritmi che seguono questa logica cercano di occupare l'intera area disponibile. Se considerassimo un'area disponibile con dimensioni di $\sqrt{n} \cdot \sqrt{n}$, il massimo spazio utilizzato per assegnare i limiti (bound) agli elementi dell'albero sarebbe di n.

Ne consegue che l'area occupata sul piano xy, utilizzando un algoritmo di treeMap 3D, è di ordine $O(n)$. Dal momento che l'altezza del layout di uscita dipende esclusivamente dalla profondità dell'albero, si può concludere che il volume occupato dal TreeMap 3D è $O(n \cdot h)$, dove h rappresenta l'altezza dell'albero ed n il numero dei nodi.

In sintesi, i due algoritmi proposti generano un'occupazione volumetrica riassumibile in:

- Layer Tree Draw 3D: $O(n^2 \cdot h)$
- TreeMap 3D: $O(n \cdot h)$

Un'osservazione importante riguardo alle rappresentazioni bidimensionali è che il Layer Tree, sebbene occupi un'area di $O(n^2)$, offre una visualizzazione chiara delle gerarchie all'interno dell'albero. D'altra parte, il TreeMap, a causa dell'uso di bound annidati, può comportare difficoltà nella comprensione dei rapporti tra i nodi genitori e figli nell'albero.

Tuttavia, questo problema è superato mediante l'adozione di una rappresentazione tridimensionale. L'introduzione di una dimensione aggiuntiva, la "z", consente di organizzare l'albero in diversi livelli visivi. Questo approccio riduce significativamente il volume complessivo della rappresentazione rispetto al Layer Tree Draw 3D e migliora notevolmente la facilità di lettura dell'albero, rendendo i rapporti gerarchici più chiari ed intuitivi.

Capitolo 3

Implementazione

Questo capitolo costituisce il nucleo centrale dell'elaborato, in quanto si concentra sull'implementazione dettagliata degli algoritmi oggetto di studio della tesi.

L'implementazione degli algoritmi è una fase cruciale del processo di ricerca, poiché consente di trasformare concetti astratti in soluzioni concrete e operative.

L'obiettivo principale di questo capitolo è quello di offrire una visione chiara e dettagliata del processo di implementazione. Verranno esplorate le strutture dati, gli algoritmi ausiliari, le scelte di progettazione e le logiche di funzionamento che sottendono agli algoritmi stessi.

3.1 Dettagli sull'Implementazione

Il linguaggio di programmazione scelto per l'implementazione è Java. Prima di addentrarci in dettaglio sugli algoritmi vale la pena introdurre le classi ausiliarie utilizzate per lo sviluppo.

RECTANGLE

La classe `Rectangle.java` rappresenta un rettangolo nello spazio, fondamentale per una facilitazione nel calcolo dei bound di ogni nodo. L'oggetto rettangolo è rappresentato da un punto, raffigurante il vertice in alto a sinistra del rettangolo, dalla larghezza e dall'altezza. I metodi della classe implementano le operazioni di:

- Calcolo del centro (`center()`);
- Sottrazione tra due rettangoli (`remove(Rectangle r)`);
- Calcolo dell'Aspect Ratio (`aspectRatio()`).

NODE

La classe `Node.java` rappresenta un nodo dell'albero al quale sono assegnati determinati attributi fondamentali: `data`, rappresentante la dimensione del sottoalbero che ha come radice il nodo analizzato; `childern`, una lista contenente tutti i figli del nodo, `bound`, rappresentante lo spazio occupato da ogni nodo; `coordinates`, rappresentante le coordinate del nodo nello spazio; `depth`, rappresenta la profondità del nodo.

TREE

La classe `Tree.java` implementa l'albero. Come attributi presenta esclusivamente `root`, il nodo radice dell'albero, e `treeNodes`, una lista contenente tutti i nodi appartenenti a quell'albero.

Il metodo `calculateSubtreeSize(Node n)` calcola ricorsivamente la dimensione di ogni sottoalbero, logica fondamentale per la realizzazione dell'algoritmo.

Il metodo `calculateDepth(Node n)` calcola ricorsivamente la profondità di ogni nodo.

Tutti gli algoritmi che verranno analizzati funzionano prendendo in input un albero, rappresentato in un file di testo contenente gli id e le relazioni tra i nodi, e restituisce, come output, un file testuale contenente i dati del file di input e le coordinate del layout finale.

3.1.1 Implementazione LayerTreeDraw3D

L'implementazione dell'algoritmo Layer Tree Draw 3D è sviluppata all'interno della classe `LayerTreeDraw3D.java`.

Il metodo `layoutTree(Node root, int xPosition, int yPosition)` prende in input la radice dell'albero e le coordinate in cui posizionarla. Dopodiché, sono chiamati i due metodi che permettono di costruire le coordiante relative ed infine quelle assolute.

`relativeBounds(Node n, int x, int y)` prende in ingresso un nodo e la posizione calcolata in funzione del padre. Il caso base si presenta quando il nodo non ha figli ed il metodo assegna al nodo un bound di dimensioni fissate.

Se il nodo ha dei figli, il metodo calcola i bound di questi figli ricorsivamente. L'orientamento di disposizione dipende dalla profondità del nodo nell'albero. Dopo aver calcolato i limiti dei figli, il nodo corrente riceve un bound che racchiude tutti i suoi figli.

`absoluteBounds(Node n, int x, int y)` ripercorre ricorsivamente l'albero in modo da assegnare le coordinate assolute ai bound calcolati con il metodo precedente.

```
    void relativeBounds(Node node, int x, int y) {

        if (node.getChildren().isEmpty()) {
            node.setBound(new Rectangle(new Point(x, y),
               LEAF_WIDTH, LEAF_HEIGHT));

        } else {
            int maxSize = 0;
            int offset = 0;

            for (Node child : node.getChildren()) {
                if(node.getDepth()%2 == 0){
                    relativeBounds(child, x + offset, y);
                    offset += child.getBound().getWidth();
                    maxSize = Math.max(maxSize, (int)child.
                       getBound().getHeight());
                }
                else{
                    relativeBounds(child, x, y + offset);
                    offset += child.getBound().getHeight();
                    maxSize = Math.max(maxSize, (int)child.
                       getBound().getWidth());
                }
            }

            if(node.getDepth()%2 == 0){
                double parentX = (offset-LEAF_WIDTH)/2;
                node.setBound(new Rectangle(new Point(parentX, y)
                   , offset, maxSize));
            }
            else{
                double parentY = (offset-LEAF_HEIGHT)/2;
                node.setBound(new Rectangle(new Point(x, parentY)
                   , maxSize, offset));
            }

        }
    }
```

```
    void absoluteBounds(Node node, int x, int y){
        node.setBound(new Rectangle(new Point(x, y), node.
           getBound().getWidth(), node.getBound().getHeight()));

        if(!node.getChildren().isEmpty()){
            int offset = 0;
            for(Node child : node.getChildren()){

                if(node.getDepth()%2 == 0){
                    absoluteBounds(child, x + offset, y + (int)(
                       node.getBound().getHeight() - child.
                       getBound().getHeight()) / 2);
                    offset += child.getBound().getWidth();
                } else {
                    absoluteBounds(child, x + (int)(node.getBound
                       ().getWidth() - child.getBound().getWidth()
                       ) / 2, y + offset);
                    offset += child.getBound().getHeight();
                }
            }
        }
    }
```

3.1.2 Implementazione TreeMap

L'implementazione degli algoritmi che sfruttano la logica del TreeMap è sviluppata nella classe statica `TreeMap.java`.

Gli algoritmi sono progettati in modo tale che quando uno dei metodi viene invocato su un albero specifico T, i bound (o confini) dei nodi all'interno dell'albero vengono aggiornati in base alla logica specifica implementata nel metodo stesso.

3.1.2.1 Slice and Dice

Il metodo `treeMapBoundsSlicenDice(Tree T, Rectangle availableSpace)` prende in input l'albero T, su cui invocare il metodo, e lo spazio disponibile, su cui calcolare i bound. Successivamente, in funzione dello spazio disponibile, assegna il bound ad ogni nodo sfruttando tutto lo spazio presente.

Il corpo centrale dell'algoritmo è determinato dal metodo `layoutNodes`, descritto nel codice seguente:

```
void layoutNodes(List nodes, Rectangle avSpace){
int size = 0;
for(Node n : nodes){
        Rectangle bound = sliceAndDice(n, avSpace, size);
        n.setBound(bound);
        avSpace.remove(bound);
        size+= n.getData();

        if(!n.getChildren().isEmpty())
            layoutNodes(n.getChildren(), n.getBound());
    }
}
```

Il metodo prende in ingresso una lista di nodi (rappresentanti i figli della radice alla prima chiamata) e lo spazio disponibile, rappresentato da un rettangolo.

Per ogni nodo, viene calcolato il bound occupato tramite la funzione `sliceAndDice(node n, Rectangle availableSpace, int size)` che restituisce un rettangolo rappresentante il bound del nodo. Successivamente, il bound viene assegnato come attributo al nodo e rimosso dallo spazio disponibile. L'integer size è utilizzato dalla funzione `sliceAndDice` per calcolare le proporzioni del bound in base alla grandezza del nodo.

Infine, se il nodo per il quale si è appena calcolato il bound ha figli, viene invocato ricorsivamente il metodo.

La funzione `sliceAndDice (Node n , Rectangle availableSpace , int size)` calcola l'effettivo bound del nodo in base alla tecnica Slice And Dice e lo restituisce come rettangolo.

```
Rectangle sliceAndDice (Node n , Rectangle s , int size){
int tot = n.getParent().getData() - 1 - size;

if(n.getDepth()%2 != 0){
        Rectangle bound = new Rectangle(s.getVertex(), s.
           getWidth() * (n.getData()/tot), space.getHeight());
        return bound;
    }
    else{
        Rectangle bound = new Rectangle(s.getVertex(), s.
           getWidth(), s.getHeight() * (n.getData()/tot));
        return bound;
    }
}
```

3.1.2.2 Split

Il metodo split è implementato all'interno della stessa classe `TreeMap.java`, a differenza però dello Slice And Dice, esso è implementato in modo differente.

Il metodo `treeMapBoundsSplit(Tree T, Rectangle availableSpace)` prende in input l'albero, lo spazio disponibile e assegna i bound ai nodi secondo la logica dello Split.

`split(List nodes, Rectangle availableSpace)` è un metodo ricorsivo che prende in input una lista di nodi e lo spazio disponibile per allocarli.

Il caso base avviene quando la lista nodes in input contiene un solo nodo, in questo modo l'intera area disponibile è assegnata al nodo stesso.

Se il nodo ha dei figli, viene invocato ricorsivamente il metodo, passandogli come valori i nodi figli ed il bound appena calcolato.

Nel caso in cui la lista nodes contenga più di un nodo, viene invocata una funzione `groups(List nodes)` che suddivide i nodi in due gruppi, il più possibile simili, utilizzando come parametro la grandezza dei nodi, in modo da generare un treemap il più possibile bilanciato.

In questo passaggio, si calcola prima la dimensione complessiva del primo gruppo e la somma totale dei nodi. Successivamente, per definire il limite (bound), si procede dividendo lo spazio disponibile in due sezioni. Questa divisione verrà effettuata in modo orizzontale o verticale, basandosi sulla dimensione predominante (altezza o larghezza) dello spazio a disposizione.

Il bound calcolato (rappresentante il primo gruppo di nodi) sarà rimosso dallo spazio disponibile, e il metodo sarà invocato ricorsivamente sul primo gruppo di nodi e sul secondo gruppo. Il primo avrà come spazio disponibile il bound calcolato precedentemente ed il secondo lo spazio disponibile a seguito della rimozione.

```
    void split (List nodes, Rectangle space){

        if(nodes.size()==1){
            nodes.getFirst().setBound(space);

            if(!n.getChildren().isEmpty()){
                split(nodes.getFirst().getChildren(), space);
            }
        }
        else{
            List<List<Node>> groups = groups(nodes);
            List firstGroup = groups.getFirst();
            List secondoGroup = groups.getLast();

            for(Node n : nodes){
                tot += n.getData();
                if(firstGroup.contains(n)){
                    data += n.getData();
                }
            }

            Rectangle bound;
            if(space.width >= space.height){
                bound = new Rectangle(space.vertex, space.width *
                    (first/tot), space.height);
            }
            else{
                bound = new Rectangle(space.vertex, space.width,
                   space.height * (first/tot));
            }
            space.remove(bound);
            split(firstGroup, bound)
            split(secondGroup, space)
        }
    }
```

In conclusione, l'implementazione di questi tre algoritmi permette di assegnare, secondo tecniche diverse, un bound ad ogni nodo. Successivamente, grazie al metodo `calculateZCoordinates()` le coordinate di ogni nodo sono assegnate ed il layout grafico può essere disegnato mediante un programma di rendering grafico.

3.2 Interfaccia grafica e Visualizzazione

Al fine di verificare il corretto comportamento di ogni algoritmo, è stata implementata una semplice interfaccia grafica che permette di visualizzare gli effettivi bound di ogni nodo.

A seguito del calcolo dei bound con uno dei tre metodi, l'interfaccia grafica disegna a schermo l'albero risultante.

Di seguito è proposto un esempio visivo di un treemap calcolato su un albero di 1000 nodi.

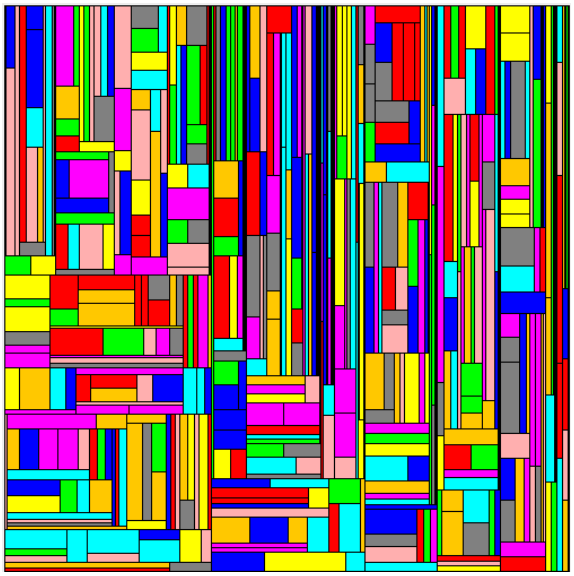

Figura 3.1: Slice and Dice

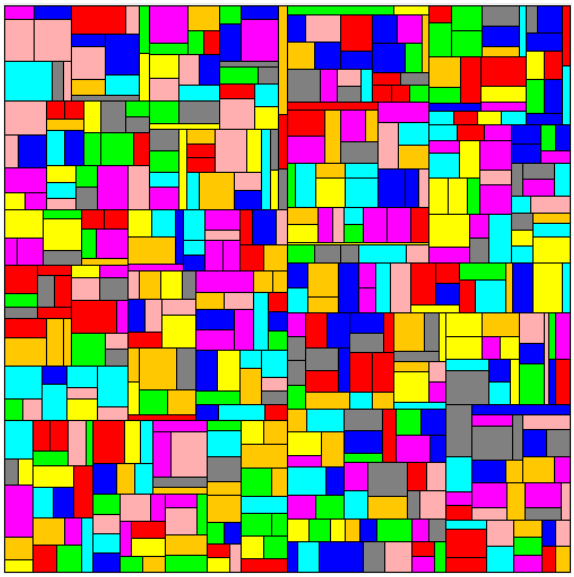

Figura 3.2: Split

3.3 Realizzazione modello 3D

Nella presente sezione, ci occuperemo di delineare il processo attraverso il quale il modello tridimensionale di un albero viene portato a compimento.

Una volta che le coordinate sono state stabilite, il passo successivo è quello di tradurre questi dati numerici in un'entità visiva palpabile. Il processo di 'disegno' dei nodi, o meglio, la loro renderizzazione grafica, trasforma l'astrazione matematica in una rappresentazione concreta che può essere visualizzata e analizzata.

3.3.1 Blender

Blender è un software di grafica tridimensionale open source, potente e versatile, utilizzato per creare una vasta gamma di contenuti visivi in 3D, esso è uno strumento completo che supporta l'intero pipeline di produzione in 3D.

Blender viene fornito con un interprete Python incorporato, il che significa che è possibile scrivere script Python direttamente all'interno di Blender per automatizzare compiti ripetitivi, come la modellazione, il texturing o il rendering.

In questa sottosezione verrà analizzato il codice python che permette di generare un modello tridimensionale dell'albero a partire dalle coordinate dei nodi calcolate mediante gli algoritmi precedentemente discussi.

Una volta preso in input il file contenente le relazioni e le coordinate appartenenti ad ogni nodo, lo script sfrutta due funzioni:

- `createSphere(location)`: sfrutta la funzione nativa di blender per generare una sfera di raggio unitario passandogli delle coordinate.

- `createCylinder(startLocation, endLocation))`: è una funzione che genera un cilindro partendo da un punto iniziale ed un punto finale.

  La funzione calcola, mediante le posizioni prese in input, la distanza euclidea tra i due punti, la direzione nello spazio e la posizione del cilindro nello spazio.

  Per calcolare la rotazione di un cilindro vengono utilizzati, i quaternion di rotazione. In Blender, si utilizzano gli angoli di Eulero per descrivere l'orientamento di un oggetto nello spazio. L'utilizzo di questi angoli, se non adoperati in modo opportuno, potrebbe portare a fenomeni noti come gimbal lock, ovvero quando due assi di rotazione si allineano, causando la perdita di un grado di libertà nel sistema di rotazione. Per evitare questa problematica sono utilizzati i quaternioni.

Di seguito vengono forniti i codici implementati:

```
def createSphere(location, radius=1.0):
bpy.ops.mesh.primitive_uv_sphere_add(radius=radius, location=
   location)
```

```
def createCylinder(startLocation, endLocation, radius=0.3):
startLocation = Vector(startLocation)
endLocation = Vector(endLocation)

length = math.dist(startLocation, endLocation)
direction = endLocation - startLocation
rot_quat = direction.to_track_quat('Z', 'Y')

bpy.ops.mesh.primitive_cylinder_add(
    radius=radius,
    depth=length,
    location=(startLocation + endLocation) / 2,
    rotation=rot_quat.to_euler()
)
```

Definite queste due funzioni, viene utilizzato un metodo ricorsivo che permette di renderizzare l'albero preso in input:

```
def draw_tree(node, parentLocation):
currentLocation = node_coordinate[node]
createSphere(currentLocation)

if node in node_relationships:
    for child in node_relationships[node]:
        childLocation = node_coordinates[child]
        create_cylinder(currentLocation, childLocation)
        draw_tree(child, node_coordinates[child])
```

Di seguito è raffigurato un esempio di modello 3D generato tramite blender.

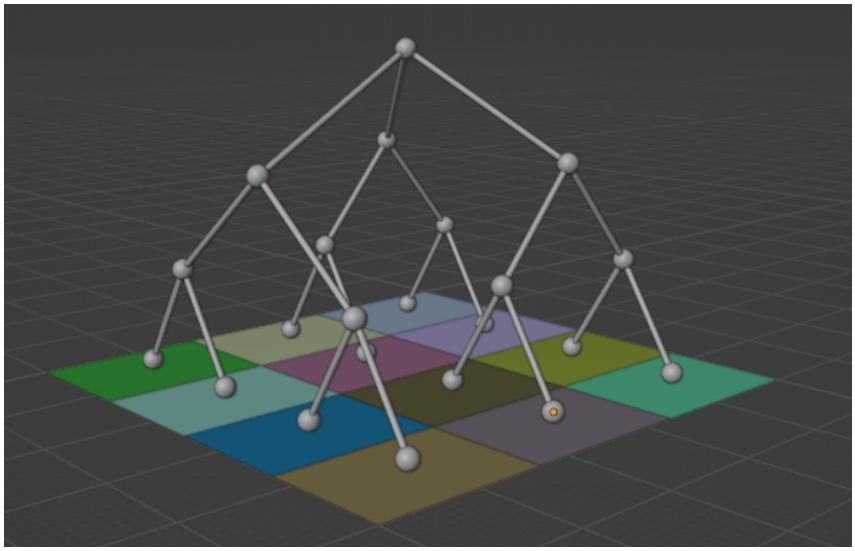

Figura 3.3: Esempio modello 3D in blender

3.3.2 Unity

Unity è un motore di gioco in tempo reale (game engine) che è utilizzato per sviluppare giochi, simulazioni in tempo reale e per la visualizzazione di contenuti in realtà virtuale ed aumentata.

Mentre Blender è utilizzato per la creazione di assets, nel caso dell'elaborato di tesi un modello 3D, Unity è progettato per portare questi contenuti in ambienti interattivi e simulativi. Il modello prodotto da blender è in formato FBX (filmbox), il quale è facilmente importabile in Unity.

Utilizzando le specifiche librerie presenti in Unity, il modello importato può essere visualizzato in realtà virtuale o aumentata, semplificando notevolmente la comprensione della struttura dati.

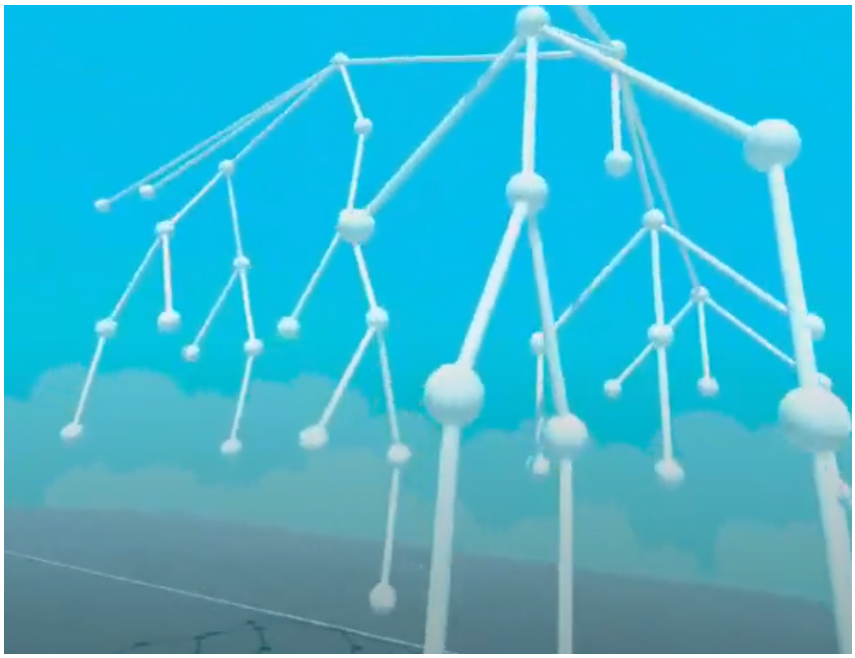

Figura 3.4: Esempio visualizzazioine in Unity

PIPELINE PROGETTO

Riassumendo, si può analizzare il percorso che porta dal file testuale, rappresentante l'albero, alla visualizzazione in Unity:

Figura 3.5: Pipeline dell'algoritmo

1. Java prende in input un file .txt rappresentante un albero;
2. I tre algoritmi implementati calcolano le coordinate;
3. Viene prodotto un file di output;
4. Blender utilizza queste coordinate per costruire un modello 3D dell'albero;
5. Viene generato un file FBX;
6. Unity prende in input questo asset e lo rappresenta in realtà virtuale.

Capitolo 4

Attività Sperimentale e Risultati

In questo capitolo, si procederà con l'analisi sperimentale degli algoritmi sviluppati, adottando come principali indicatori di valutazione i **tempi di esecuzione**, il **volume occupato** da ciascun layout prodotto e, specificatamente per gli algoritmi basati sulla logica dei Treemap, l'**Aspect Ratio** medio dei rettangoli generati.

L'obiettivo della sperimentazione è duplice: da un lato, quantificare la performance temporale degli algoritmi al variare della dimensione dell'albero in input, dall'altra, valutare l'efficienza nell'utilizzo dello spazio di visualizzazione e l'ottimalità dei layout prodotti, con un focus particolare sull'equilibrio dimensionale dei rettangoli nei Treemap, misurato attraverso l'Aspect Ratio medio.

Per condurre questa analisi, verranno generati alberi casuali di varie dimensioni.

4.1 Impostazione dell'Esperimento

Al fine di testare l'efficienza dell'algoritmo sviluppato a fronte di diverse configurazioni di input, è stata implementata la classe di prova `randomTreeGenerator.java`

La classe sfrutta due liste: `availableNodes` e `connectedNodes`; la prima contiene i nodi non ancora aggiunti all'albero e la seconda i nodi attualmente presenti nell'albero. Il metodo `randomTree` prende in input un integer che rappresenta il numero di nodi che l'albero deve contenere.

Inizialmente, tutti i nodi (eccetto la radice) sono aggiunti alla lista availableNodes. La radice, che ha codice identificativo 1, viene aggiunta alla lista connectedNodes, poiché punto di partenza dell'albero.

Dopodiché, tramite un ciclo while, ogni nodo contenuto in availableNodes è assegnato come figlio ad un nodo scelto casualmente da connectedNodes, per poi essere rimosso da availableNodes ed essere aggiunto a connectedNodes.

Riassumendo, questo codice genera un albero casuale collegando nodi in maniera sequenziale e registra ogni connessione in un file. L'albero risultante è un albero non diretto senza cicli, dove ogni nodo, eccetto la radice, ha esattamente un genitore, garantendo che la struttura sia effettivamente un albero.

Le dimensioni di input scelte per condurre l'analisi sperimentale sono le seguenti:

```
int dim = {100, 500, 1000, 2000, 3000, 5000, 8000, 10000, 20000,
   30000, 40000, 50000};
```

Per ogni dimensione considerata, sono state generate casualmente dieci diverse istanze. I risultati presentati in seguito rappresentano quindi la media di questi casi.

In ogni esperimento, è stato misurato il tempo necessario per determinare il layout tridimensionale dei nodi.

4.2 Presentazione dei Risultati

Proseguiamo adesso con l'analisi nello specifico di ogni algoritmo, focalizzando l'attenzione sui tempi di esecuzione, il volume occupato e, nel caso del TreeMap, sull'Aspect Ratio.

NB: Il rapporto di forma, o aspect ratio, di un rettangolo è definito come il rapporto tra la sua larghezza e la sua altezza. Questo valore, essendo un numero puro, può essere espresso come un numero decimale.

LAYER TREE DRAW 3D

La misurazione dei tempi di esecuzione ed il volume occupato, eseguendo l'algoritmo layerTreeDraw3D, portano a questi risultati numerici:

dim	[ms]	[volume]
100	0,1	2470
500	0,1	31080
1000	0,2	68320
2000	0,2	148740
3000	0,4	272630
5000	0,6	482400
8000	0,7	1043080
10000	1,5	1534580
20000	2,6	3258450
30000	6,6	7676900
40000	9,5	12675000
50000	11,3	19155000

Per gli input di dimensioni minori, si osserva che il tempo di esecuzione varia leggermente, mantenendosi tra 0,1 e 0,2 millisecondi. Man mano che la dimensione dell'input aumenta, i tempi di esecuzione sembrano seguire una tendenza di incremento più lineare. Il grafico dell'andamento prodotto da tali dati è il seguente:

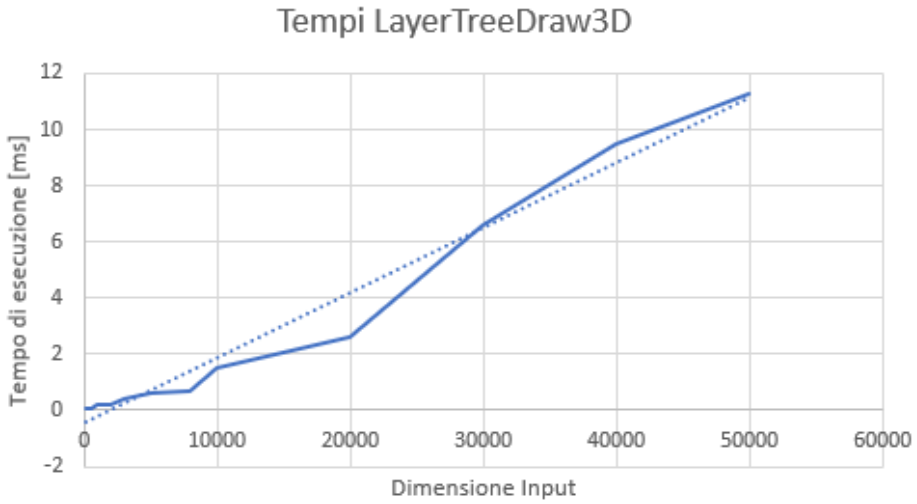

Figura 4.1: Tempi di esecuzione di `LayerTreeDraw3D` al variare della dimensione

L'analisi dell'occupazione spaziale media, derivata dai dati raccolti, rivela una tendenza che si colloca tra una crescita quadratica e una crescita cubica, ovvero tra $O(n^2)$ e $O(n^3)$. Secondo la teoria, l'algoritmo in esame era stato caratterizzato da un'occupazione di volume proporzionale a $O(n^2 \cdot h)$, dove n indica il numero di nodi nell'albero e h la sua altezza. In scenari in cui l'albero risulta fortemente sbilanciato, l'altezza dell'albero può raggiungere una complessità di $O(n)$. Pertanto, si prevede che l'algoritmo mostri una crescita del volume occupato che varia da quadratica a cubica, in linea con il comportamento teorico previsto.

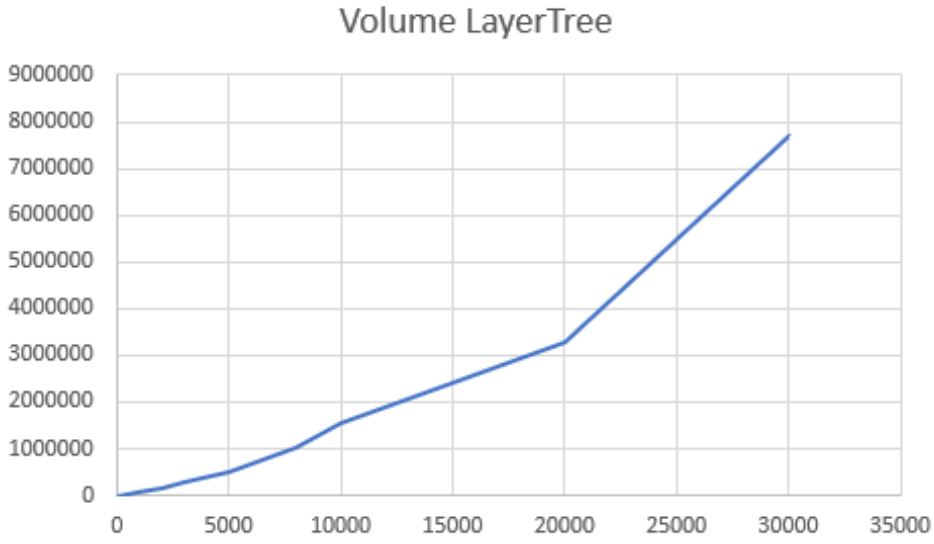

Figura 4.2: Occupazione di volume LayerTreeDraw3D

SLICE AND DICE

I test sperimentali condotti sull'algoritmo Slice and Dice hanno condotto ai seguenti risultati:

dim	[ms]	[volume]	[AR]
100	0,1	890	6,20
500	0,1	6150	9,04
1000	0,2	14300	13,25
2000	0,2	30400	17,10
3000	0,3	51000	23,21
5000	0,9	87500	27,97
8000	1,7	151200	34,35
10000	1,8	196000	41,60
20000	3,5	424000	55,71
30000	6,1	675000	57,3
40000	8,3	900000	69,31
50000	9,8	1125000	85,51

Similmente all'algoritmo discusso in precedenza, lo Slice and Dice manifesta una performance quasi costante per insiemi di dati di dimensioni ridotte. Questo comportamento è dovuto al fatto che, con pochi nodi da elaborare, le operazioni richieste sono minimali e quindi il tempo di esecuzione rimane basso e relativamente uniforme. Tuttavia, man mano che la dimensione dell'albero aumenta, si osserva che i tempi di esecuzione crescono in modo lineare, conformemente alle previsioni teoriche.

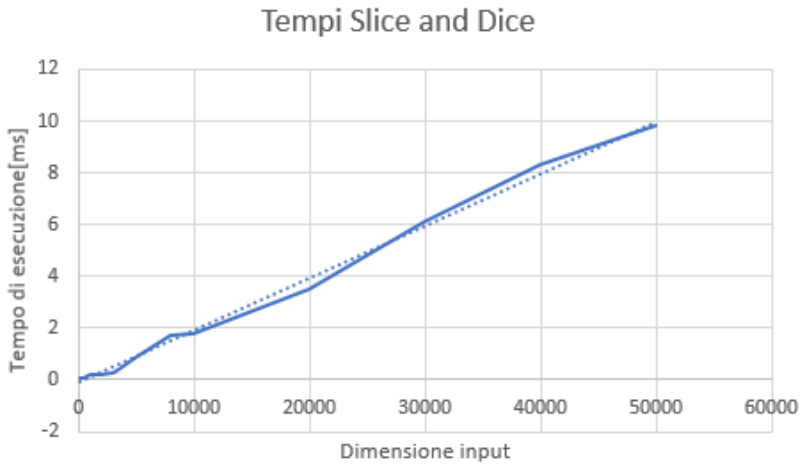

Figura 4.3: Tempi di esecuzione Slice and Dice

L'analisi dell'occupazione spaziale dei dati, raccolti in fase di test, suggerisce che l'andamento cresce più rapidamente di una funzione lineare, seppur non seguendo esattamente una crescita quadratica pura su tutta la gamma di input. Basandoci sui dati forniti, l'andamento della complessità sembra situarsi tra lineare e quadratica.

I risultati osservati riflettono le aspettative teoriche relative agli algoritmi basati sulla logica dei TreeMap, i quali presentano un'occupazione spaziale proporzionale a $O(n \cdot h)$, dove n è il numero dei nodi dell'albero e h ne rappresenta l'altezza. Nello scenario peggiore, in cui l'albero sia altamente sbilanciato, l'altezza può crescere linearmente con il numero dei nodi, rendendo h proporzionale a $O(n)$. Pertanto, in tale contesto, l'occupazione volumetrica dell'algoritmo varierebbe da una crescita lineare $O(n)$ a una quadrato-lineare $O(n^2)$, a seconda della configurazione specifica dell'albero preso in esame.

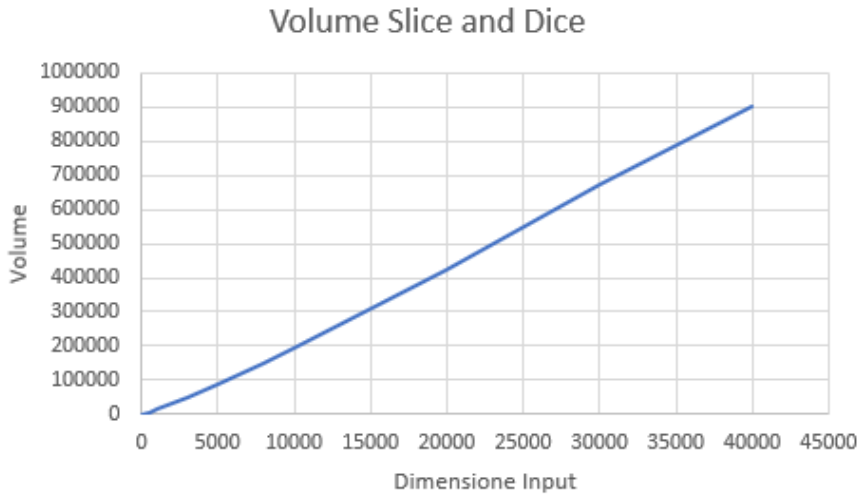

Figura 4.4: Occupazione di volume Slice and Dice

In relazione all'aspect ratio medio dei confini (bounds) generati dall'algoritmo Slice and Dice, si osserva un progressivo deterioramento al crescere della dimensione dell'albero in input. Questa tendenza è coerente con le aspettative teoriche relative alla natura dell'algoritmo: per sua stessa costruzione, lo Slice and Dice tende a produrre un rapporto di forma (aspect ratio) non ottimale, che si degrada ulteriormente con l'incremento delle dimensioni dell'input.

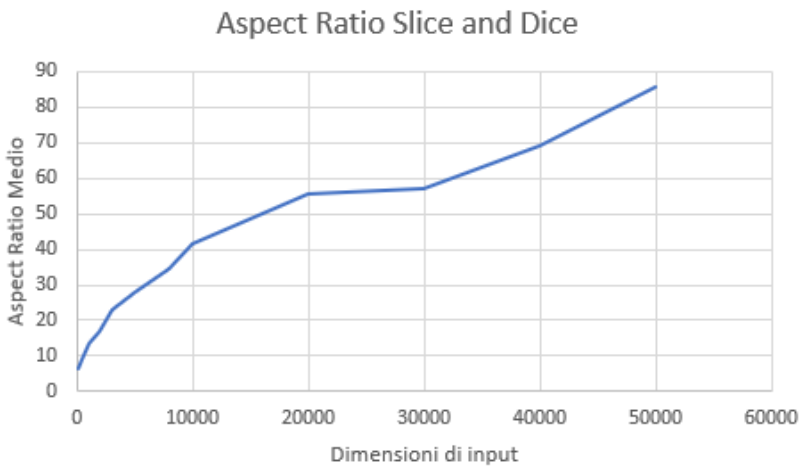

Figura 4.5: Aspect Ratio medio Slice and Dice

SPLIT

I test sperimentali sull'algoritmo di Split conducono ai seguenti risultati:

dim	[ms]	[volume]	[AR]
100	0,4	890	2,12
500	0,4	6150	1,98
1000	0,4	14300	2,11
2000	0,3	30400	2,10
3000	0,5	51000	2,04
5000	0,8	87500	2,14
8000	1,1	151200	2,09
10000	1,4	196000	2,08
20000	3,5	424000	2,09
30000	10,4	675000	2,11
40000	10,7	900000	2,09
50000	13,8	1125000	2,13

Per input di piccole dimensioni, il tempo di esecuzione rimane pressoché costatante, all'aumentare delle dimensioni i tempi di esecuzione sembrano avere un andamento di tendenza lineare. Tuttavia, a partire da 30000 nodi, si nota un discreto degradamento dei tempi di esecuzione.

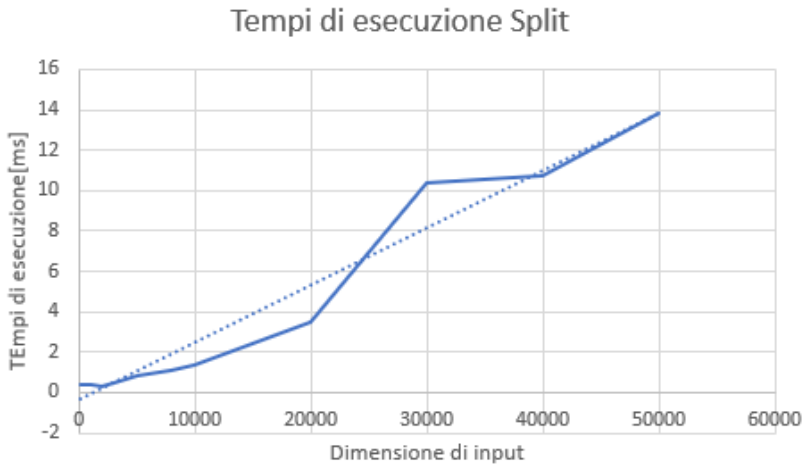

Figura 4.6: Tempi di esecuzione Split

Allo stesso modo dello Slice and Dice, l'analisi dell'occupazione spaziale dei dati raccolti in fase di test suggerisce che, l'andamento cresce più rapidamente di una funzione lineare ma che quanto una funzione quadratica.

La constatazione che gli algoritmi Split e Slice and Dice condividano una simile occupazione spaziale, nonostante le loro differenze metodologiche, non si presenta come una sorpresa al termine dell'analisi. La ragione fondamentale di questa equivalenza spaziale risiede nel fatto che, indipendentemente dal metodo di divisione spaziale adottato, lo spazio totale disponibile per l'allocazione non cambia e rimane intrinsecamente legato alle dimensioni dell'albero in input. In altre parole, sebbene Split e Slice and Dice approccino la partizione dello spazio in maniere concettualmente diverse, entrambi operano all'interno degli stessi confini definiti dall'input. Questo significa che le variazioni nella tecnica di divisione, sebbene possano influenzare l'aspetto visuale e l'efficienza di navigazione del TreeMap, non alterano la quantità totale di spazio occupato dai nodi dell'albero.

Relativamente all'Aspect Ratio medio, il metodo Split dimostra di mantenere un rapporto di forma inferiore e più uniforme rispetto al metodo Slice and Dice, in linea con le previsioni teoriche. Questa differenza si spiega con le caratteristiche intrinseche della logica di funzionamento del metodo Split, che è progettato per ottimizzare la proporzione tra altezza e larghezza dei rettangoli, risultando in layout più equilibrati.

L'efficienza nell'aspect ratio osservata, deriva dall'impiego di una funzione che assicura una suddivisione equilibrata dei gruppi di nodi nell'ambito dell'algoritmo di Split. Questa funzione, centrale per l'approccio di divisione dello spazio, è cruciale per mantenere proporzioni armoniose tra i diversi nodi e gruppi di nodi, influenzando direttamente l'aspect ratio dei rettangoli generati.

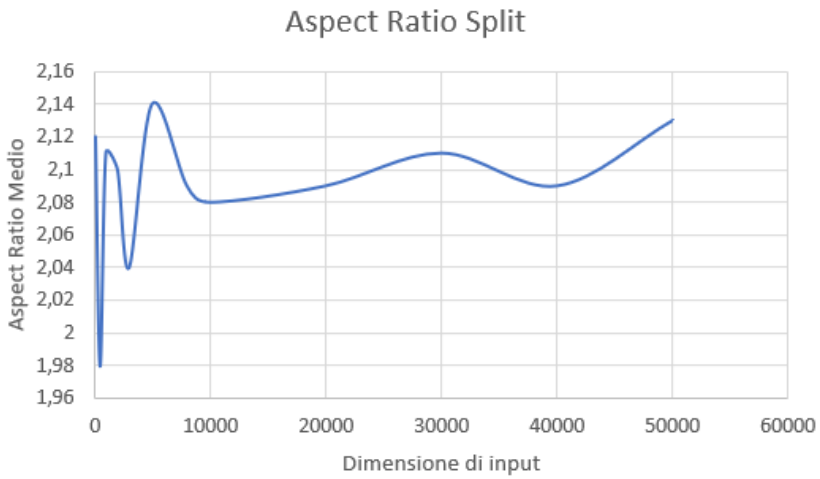

Figura 4.7: Aspect Ratio medio Split

4.3 Commenti e Interpretazioni

I risultati sperimentali confermano le previsioni teoriche riguardanti la complessità degli algoritmi.

Conforme alle previsioni teoriche, discusse nei capitoli precedenti, gli algoritmi basati sulla logica del treemap dimostrano un'occupazione spaziale che cresce linearmente in funzione della dimensione dell'albero in input. Questo indica un'efficiente gestione dello spazio proporzionale al numero di nodi. Al contrario, l'algoritmo LayerTreeDraw3D evidenzia un comportamento differente, con un'area di occupazione che segue un andamento quadratico rispetto al numero di input, suggerendo una maggiore richiesta di spazio al crescere della complessità dell'albero.

L'ultima considerazione riguarda il rapporto di forma utilizzato dagli algoritmi Treemap. In linea con le aspettative teoriche, l'algoritmo Slice and Dice manifesta un rapporto di forma meno buono rispetto allo Split, tendenza che si accentua con l'incremento della dimensione di input.

Al contrario, l'algoritmo Split riesce a mantenere un rapporto di forma più costante, approssimativamente intorno a 2, anche quando si gestiscono alberi di grandi dimensioni. Questo risultato si ottiene attraverso una scelta accurata del metodo di divisione per i gruppi di nodi durante il processo di split, evidenziando l'importanza di un approccio ben calibrato nella suddivisione dello spazio per garantire un aspect ratio armonioso nelle visualizzazioni risultanti.

Un rapporto di forma adeguato è cruciale per migliorare la leggibilità di un modello tridimensionale, poiché le coordinate dei nodi sono determinate utilizzando come riferimento il centro del loro confine. Mantenere un rapporto di forma equilibrato consente una distinzione spaziale più evidente tra i nodi, facilitando così la leggibilità e l'interpretazione del modello.

Capitolo 5

Conclusioni e Sviluppi Futuri

La visualizzazione immersiva, attraverso la creazione di ambienti tridimensionali dettagliati, permette agli utenti di immergersi completamente nell'analisi dei dati, migliorando significativamente le capacità interpretative e di lettura che in formati bidimensionali tradizionali potrebbero non essere immediatamente evidenti. Questo approccio apre nuove prospettive nell'esplorazione e nella comprensione di strutture dati complesse, potenziando la capacità analitica e decisionale in diversi campi applicativi.

Questa tesi esplora il processo di progettazione e implementazione di algoritmi per la visualizzazione grafica di alberi in ambienti di realtà immersiva. L'approccio migliora, sostanzialmente, la comprensione delle strutture dati, grazie alla capacità di navigare interattivamente all'interno dell'albero in un contesto immersivo. Questa modalità di visualizzazione trasforma l'apprendimento di concetti complessi in un'esperienza intuitiva, rendendo l'assimilazione delle informazioni più diretta e immediatamente accessibile.

L'obiettivo e la motivazione di questa tesi era l'implementazione di un metodo, che permettesse di visualizzare strutture dati, in questo caso alberi, in realtà immersiva.

Un percorso promettente per la ricerca futura potrebbe includere la conduzione di studi mirati a quantificare il miglioramento nella leggibilità e nella comprensione delle strutture ad albero, generate mediante l'algoritmo descritto in tesi e visualizzandole tramite la realtà immersiva.

Possibili sviluppi futuri potrebbero essere l'implementazione dell'algoritmo squarified, il quale sfrutta la logica del treemap, conferendo ai bound un aspect ratio ancora più ottimizzato rispetto allo split. In tal modo si migliorerebbe la leggibilità tridimensionale rispetto agli algoritmi implementati.

Ringraziamenti

Vorrei evitare dei ringraziamenti strappalacrime dove citare tutte le persone che mi sono state accanto, questo perché l'unico motivo di lacrime dovrebbe essere per tutte le serate perse per stare davanti al computer per completare questa tesi.

A parte gli scherzi passiamo ai veri ringraziamenti.

È essenziale menzionare il ruolo fondamentale che la mia famiglia ha avuto nel mio percorso accademico, offrendomi l'opportunità di perseguire gli studi universitari, garantendomi la libertà di scegliere e fornendomi un supporto incrollabile in ogni mia decisione. La loro presenza ha avuto un impatto incommensurabile: nei momenti di difficoltà, hanno saputo regalarmi un sorriso rassicurante e, quando occorreva, non hanno esitato a richiamarmi all'ordine.

Devo a loro non solo la persona che sono oggi ma anche i valori che incarno, quali la costanza, la determinazione e un'etica del lavoro instancabile.

Un ringraziamento speciale va a mia nonna, la cui fede e preghiere incessanti hanno accompagnato ogni tappa del mio viaggio accademico.

I ringraziamenti non sarebbero completi senza menzionare gli amici e colleghi che hanno condiviso con me questo tratto del cammino. Un ringraziamento speciale va a tutti quegli amici che hanno reso questo periodo della mia vita un'esperienza condivisa, tra momenti di ansia e preoccupazione.

È doveroso fare una menzione per Giovanni, il mio fedele compagno di progetti, insieme al quale ho vissuto gran parte del mio percorso universitario. Altrettanto doverosa è la menzione a Tommaso, che ho avuto la fortuna di incontrare all'università e, a parte essere un mio collega universitario, è diventato uno dei miei più cari amici con cui ho potuto condividere passioni ed interessi.

Un caloroso grazie va anche a Piero, Andrea e Giacomo, compagni di innumerevoli giornate di studio e pause pranzo universitarie, con cui ho condiviso non solo i banchi di scuola ma anche momenti di vita quotidiana.

Per ultima, ma non meno importante, desidero esprimere la mia più profonda gratitudine a Ilaria, a cui dedico con affetto questa tesi. È stata la mia roccia in questi anni, la presenza costante e il sostegno più grande, sempre al mio fianco. Il suo aiuto inestimabile, le lunghe serate trascorse insieme sui libri e il tempo prezioso che mi ha dedicato, specialmente nei momenti più ardui, sono stati fondamentali. La sua forza e il suo supporto sono stati essenziali per il mio percorso; senza di lei, non sarei arrivato dove sono ora.

List of Algorithms

Elenco delle figure

Bibliografia

[1] Wikipedia . Visualizzazione dell'informazione, 07 2015. URL: https://it.wikipedia.org/wiki/Visualizzazione_dell%27informazione.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, and Livio Colussi. *Introduzione agli algoritmi e strutture dati*. Mcgraw-Hill, 2010.

[3] Adrian Rusu. 5 tree drawing algorithms, 2010. URL: https://cs.brown.edu/people/rtamassi/gdhandbook/chapters/trees.pdf.

[4] Willy Scheibel, Daniel Limberger, and Jürgen Döllner. Survey of treemap layout algorithms. *Survey of Treemap Layout Algorithms*, 12 2020. doi:10.1145/3430036.3430041.

[5] Huy Nguyen. Exploring the squarified tree map algorithm with reasonml (part 1) — huy.dev, 03 2019. URL: https://www.huy.dev/squarified-tree-map-reasonml-part-1-2019-03/.

[6] Stephen Kobourov. 2 force-directed drawing algorithms, 2010. URL: https://cs.brown.edu/people/rtamassi/gdhandbook/chapters/force-directed.pdf.

[7] Pietro Prosperi. Realtà aumentata (ar), virtuale (vr) e mista (mr): Differenze ed elementi in comune — scuole digitali, 11 2020. URL: https://www.scuole-digitali.it/2020/09/08/realta-aumentata-ar-virtuale-vr-e-mista-mr-differenze-ed-elementi-in-comune/.

[8] Blender 2.92.0 python api documentation — blender python api. URL: https://docs.blender.org/api/current/index.html.

[9] Decoding quaternion rotations in blender: A simplified guide, 11 2023. URL: https://cgcookie.com/posts/what-are-quaternion-rotations-in-blender.

[10] Giovanna Guidone. Hamilton e i quaternioni: una rivoluzione in algebra — pearson, 11 2020. URL: https://it.

pearson.com/aree-disciplinari/scienze-matematica/articoli/hamilton-quaternioni-una-rivoluzione-algebra.html.

[11] Unity . Create with vr. URL: https://learn.unity.com/course/create-with-vr.