



UNIVERSITÀ DEGLI STUDI DI PADOVA

SCUOLA DI INGEGNERIA

Corso di Laurea in ingegneria dell'informazione

**STUDIO DI UN ALGORITMO AD ALTE PRESTAZIONI
PER COLLISION DETECTION SU POINT CLOUD**

Laureando

Fabio Rossetto

Relatore

Prof. E. Menegatti

ANNO ACCADEMICO 2012/2013

Indice

1	Introduzione	2
1.1	Stato dell'arte	4
1.1.1	Klein and Zachmann	4
1.1.2	Klein and Zachmann [7]	6
1.1.3	Pan et al. [12]	8
2	Strutture dati per la collision detection	13
2.1	<i>Bounding Volume Hierarchies</i>	14
2.2	K-d tree	16
2.3	Octree	19
3	Descrizione dell'algoritmo attuale	22
4	Descrizione dell'algoritmo proposto	25
4.1	Attraversamento della struttura dati	25
4.2	Il metodo <code>intersectBox</code>	26
4.2.1	Richiamo teorico: il <i>Separating Axis Theorem</i>	27
4.2.2	Preparazione	28
4.2.3	L'algoritmo di intersezione	29
4.3	Setting ottimale dell'Octree	34
5	Confronto delle prestazioni	36
5.1	Dettagli implementativi	36
5.2	La simulazione	37
5.3	Tempo di esecuzione sul numero di punti	38
5.4	Reiezione in assenza di collisioni	38
5.5	Preprocessing	40
5.6	Occupazione di memoria	41
6	Conclusioni	44

Sommario

Lo scopo di questa tesi è ottimizzare un algoritmo già esistente per la determinazione di collisioni tra un braccio robotico e l'ambiente esterno. L'algoritmo va a inserirsi in un software per il *bin picking* come elemento fondamentale per la determinazione del punto di presa. Una volta individuato l'oggetto da estrarre, è necessario provare diverse posizioni del braccio per trovare quella più adatta all'estrazione del prodotto. Per ogni posizione provata, bisogna verificare che il braccio stia afferrando l'oggetto senza collidere con altre parti del contenitore. Dato che l'operazione viene iterata più volte all'interno del processo, è essenziale poter determinare la presenza di una collisione tra il robot e il contenitore in tempi ridotti.

La visione del robot avviene per scansione laser, per cui il problema si riduce alla determinazione di intersezioni tra la descrizione geometrica del braccio robotico e una *point cloud* che rappresenta l'ambiente in cui il robot si muove.

Verrà presentato lo stato dell'arte sugli algoritmi e strutture dati utilizzati per la collision detection, con particolare attenzione per quelli pensati specificamente per point cloud. Verrà presentato l'algoritmo attuale, mettendo in luce gli aspetti che ne compromettono l'efficienza. Verrà proposto un nuovo algoritmo, che fa uso di un *Octree* per accelerare il processo di collision detection. Infine, verranno confrontate le performance dei due algoritmi.

Si mostrerà come l'algoritmo proposto abbia complessità asintotica inferiore a quello attuale, e permetta un miglioramento delle prestazioni di oltre venti volte, richiedendo tempi di preprocessing e occupazione di memoria aggiuntivi sufficientemente ridotti.

Capitolo 1

Introduzione

L'algoritmo presentato in questa tesi è stato richiesto dall'azienda IT-Robotics come miglioramento di quello implementato attualmente nel framework per il bin picking *Smart Pick 3D*.

Il *bin picking* è il processo che permette l'estrazione di oggetti disposti alla rinfusa all'interno di un contenitore da parte di un robot manipolatore. Il processo di bin picking è costituito di tre fasi:

1. **Riconoscimento del contenitore e dei pezzi in esso contenuti.** Viene effettuata una scansione laser del contenitore. Questa produce come output un'immagine tridimensionale dell'ambiente sotto forma di *point cloud*, che altro non è che un set di punti disposti in uno spazio tridimensionale. A questo punto bisogna selezionare dalla point cloud i cluster che rappresentano i prodotti da estrarre, determinandone posizione ed orientamento.
2. **Determinazione della posizione di presa.** Una volta individuati gli oggetti all'interno del contenitore, bisogna scegliere quale sia meglio estrarre e determinare la posizione di presa per poter afferrare l'oggetto saldamente senza collidere con il resto del contenitore. Per l'estrazione di un singolo oggetto sono necessari diversi tentativi prima di poter trovare una posizione adatta.
3. **Motion planning.** Una volta determinata la posizione di presa, è necessario determinare un percorso che permetta al robot di estrarre l'oggetto dal contenitore senza collidere con altri oggetti o con il contenitore stesso.

La collision detection è essenziale specialmente nella seconda e terza fase. In particolare nella seconda, per ogni posizione di presa testata bisogna verificare sia l'assenza di collisioni tra il braccio robotico e l'ambiente, sia l'effettivo contatto tra il gripper e l'oggetto da estrarre.

Anche se il problema della collision detection è essenziale per molte applicazioni, come videogiochi, motori fisici e software di simulazione in genere, non c'è molta letteratura riguardo la collision detection su point cloud. Usualmente gli algoritmi di collision detection sono pensati per mesh e presuppongono una descrizione geometrica esatta degli oggetti. Le point cloud sono frutto della rilevazione di sensori, e contengono inevitabilmente un certo margine d'incertezza. Sono stati proposti alcuni algoritmi per ovviare a questo problema, e verranno descritti in questo capitolo.

Da notare un altro aspetto importante delle point cloud, che le differenzia significativamente dalle mesh: anche assumendo che siano sufficientemente precise da potersi considerare una descrizione perfetta dell'oggetto che rappresentano, per loro natura le point cloud sono una versione discretizzata delle superfici scansionate: di queste non si ha una descrizione continua, ma solo un insieme di punti posti sulla superficie. Esistono algoritmi che permettono la ricostruzione della superficie sotto forma di mesh a partire dalla descrizione per punti [11] [4], ma si tratta di algoritmi complessi e con tempi di esecuzione lunghi (in [10] viene descritto un algoritmo di surface reconstruction con tempi di esecuzione su un moderno computer di almeno 30 secondi per point cloud da 400 mila punti). Come si mostrerà nel seguito, la ricostruzione della superficie non è essenziale per la determinazione delle collisioni. Tuttavia, è evidente che l'assenza di una descrizione esplicita della superficie renda particolarmente difficile la determinazione di intersezioni tra point cloud poichè la condizione di collisione tra due oggetti è usualmente identificata con l'intersezione tra le rispettive superfici.

1.1 Stato dell'arte

1.1.1 Klein and Zachmann

Klein and Zachmann sono stati i primi ad esporre un algoritmo di collision detection per point clouds. Oltre ad un algoritmo per la determinazione di intersezioni, in questo articolo si propone anche una struttura dati per l'accelerazione del processo di collision detection.

Test di intersezione nelle foglie. La superficie della point cloud è descritta come lo zero insieme della funzione implicita $f : \mathbb{R}^3 \rightarrow \mathbb{R}$, che descrive la distanza di un punto $\mathbf{x}$ dal piano dato dal punto $\mathbf{a}(\mathbf{x})$ e la normale $\mathbf{n}(\mathbf{x})$. Quindi, $f(\mathbf{x})$ è definita come:

$$f(\mathbf{x}) = \mathbf{n}(\mathbf{x}) \cdot (\mathbf{a}(\mathbf{x}) - \mathbf{x}) \quad (1.1)$$

Il punto $\mathbf{a}(\mathbf{x})$ è la media pesata:

$$\mathbf{a}(\mathbf{x}) = \frac{\sum_{i=1}^N \theta(\|\mathbf{x} - \mathbf{p}_i\|) \mathbf{p}_i}{\sum_{i=1}^N \theta(\|\mathbf{x} - \mathbf{p}_i\|)} \quad (1.2)$$

dove $\theta(d)$ è un'opportuna *weight function*, di solito gaussiana. Il vettore normale $\mathbf{n}(\mathbf{x})$ è calcolato con il metodo dei *moving least squares* [1]. Date le point cloud P_A e P_B , con le rispettive funzioni implicite $f_A(\mathbf{x})$ e $f_B(\mathbf{x})$, esiste un'intersezione tra P_A e P_B se esistono due punti $\mathbf{p}_1, \mathbf{p}_2 \in P_A$ tali che:

$$\begin{aligned} f_B(\mathbf{p}_1) &> 0 \\ f_B(\mathbf{p}_2) &< 0 \end{aligned} \quad (1.3)$$

o viceversa. Vale a dire, esiste un'intersezione tra le due point cloud se esistono punti di una che si trovano su lati differenti della superficie dell'altra.

La struttura dati: *Point cloud hierarchy*. Per rendere più rapido il processo e scartare velocemente insiemi di punti distanti dalla zona di collisione viene utilizzato un partizionamento gerarchico dello spazio. I punti della point cloud sono organizzati in un albero binario in cui ad ogni nodo è associato un sottoinsieme della point cloud. Ad ogni passaggio della costruzione dell'albero l'insieme contenuto nel nodo corrente viene diviso in due, e i due sottoinsiemi risultanti sono assegnati ai figli. L'insieme di punti è diviso in modo da minimizzare il volume dei *Bounding Volume* (BV) dei figli [15]. La scelta di quale bounding volume usare è libera: ad esempio, si possono usare delle box *axis-aligned*, delle sfere o dei k-DOP.

Ad ogni nodo interno A è associato uno *sphere covering* (fig. 1.2): se il sottoalbero radicato in A ha foglie $L = \{L_i\}$, ciascuna contenente l'insieme di punti $P_A = \{P_i\}$, lo sphere covering di A è un set di sfere $K = \{K_i\}$ di raggio uguale. Lo sphere covering viene costruito trovando il set di sfere con volume minimo, con il vincolo che l'involuppo di P_A sia contenuto interamente in K . La ricerca dei centri delle K sfere e del loro raggio ottimale viene fatta con un algoritmo randomizzato, prendendo i punti di P_A più vicini a dei punti scelti a caso nel BV di A. L'uso dello sphere covering permette di ridurre ancor di più la possibile zona di collisione tra le due point cloud.

Riassumendo, l'algoritmo proposto da Klein and Zachmann consiste di tre fasi:

1. Fai una prima rozza scrematura delle zone in cui non può esserci una collisione attraversando contemporaneamente gli alberi delle due point cloud e verificando l'intersezione tra i bounding volume. Il passo due verrà eseguito solo sulle foglie di una point cloud che intersecano foglie dell'altra.
2. Riduci ulteriormente l'area in cui può esser presente una collisione escludendo i punti contenuti in sfere di uno *sphere covering* che non intersecano lo *sphere covering* dell'altra nuvola.

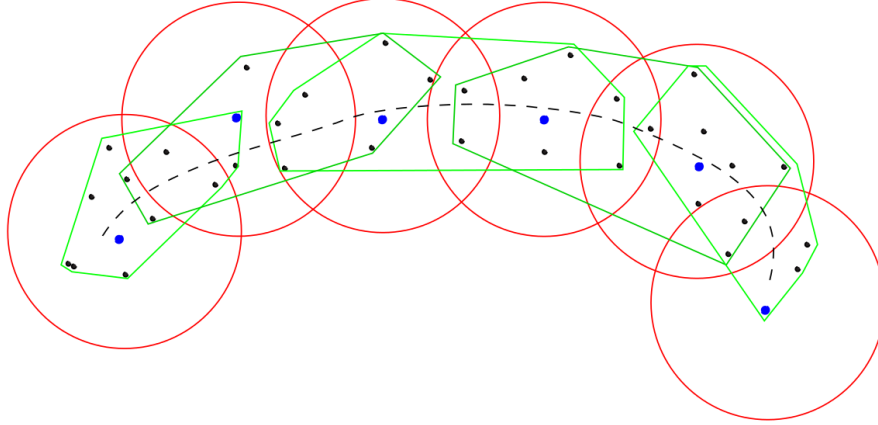


Figura 1.1: *sphere covering*

3. Nelle sfere delle due point cloud che si intersecano, stima la presenza di una collisione scegliendo punti a caso di una point cloud e verificando se oltrepassano la superficie dell'altra, utilizzando l'equazione 1.3.

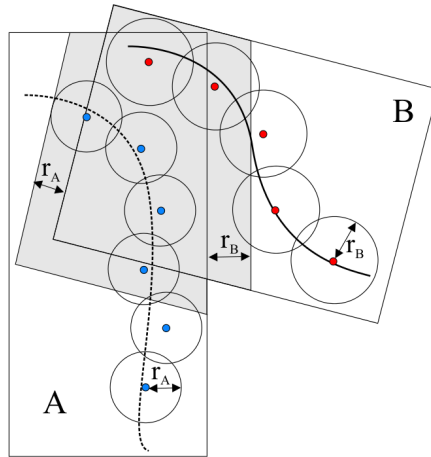


Figura 1.2: Usando i *Bounding Volumes* e lo *sphere covering* è possibile escludere velocemente parti della superficie dalla ricerca di collisioni

Un interessante aspetto dell'algoritmo di Klein and Zachmann è che fornisce un risultato best effort: se c'è ancora tempo, l'algoritmo può spenderne di più per aumentare l'accuratezza della collision detection, scegliendo casualmente un maggior numero di punti nella terza fase, e calcolando approssimativamente la distanza media tra le due point cloud.

Sebbene non venga menzionato dall'autore, dovrebbe essere possibile estendere l'algoritmo alla determinazione di intersezioni tra point cloud e mesh, seppur con

qualche modifica. A differenza di ricostruire la superficie della point cloud con i minimi quadrati, questa sarebbe già nota da principio in maniera esatta.

Lo svantaggio principale dell'algoritmo proposto in questo articolo è il lungo tempo di preprocessing richiesto per la creazione della *point cloud hierarchy* e dello *sphere covering*: per una nuvola di soli 200mila punti sono necessari più di due minuti.

1.1.2 Klein and Zachmann [7]

Questo algoritmo è un miglioramento da parte degli stessi autori di quello appena presentato.

Prima di tutto, è importante notare che questo algoritmo può essere combinato con qualsiasi struttura dati d'accelerazione, poichè va invocato solo vicino l'area di collisione. Ad esempio, se si usa una struttura dati ad albero per escludere parti della superficie, questo algoritmo va invocato solo alle foglie.

L'algoritmo sfrutta un grafo di prossimità, più precisamente, un *r-Sphere-of-Influence Graph* (r-SIG) [8].

In maniera informale, un grafo a sfera d'influenza connette punti che sono vicini tra loro, dove la definizione di vicino dipende dalla densità locale dei punti. L'algoritmo 1 mostra la procedura per costruire un r-SIG con $r = 1$, mentre la figura 1.4 mostra un SIG su una point cloud bidimensionale di otto punti.

Algorithm 1: Algorithm for creating a SIG

Data: a set of points A (will be the nodes of the graph)
Result: set of edges
foreach p_i *in* A **do**
 compute the distance d_i to the nearest neighbour of p_i ;
 foreach $p_j \neq p_i$ *in* A **do**
 compute the distance d_j to the nearest neighbour of p_j ;
 if $\|p_i - p_j\| \leq d_i + d_j$ **then**
 create connection between p_i and p_j
 end
 end
end

Vogliamo determinare se esiste una intersezione tra le point cloud A e B . Si assuma di aver raggiunto le foglie dell'attraversamento gerarchico. Ci troviamo nella regione di intersezione delle foglie dei rispettivi alberi delle due point cloud. Per ogni punto $p_i \in A$ rappresentiamo il vicinato di p_i come una sfera c_i centrata in p_i . Se esiste un'intersezione tra A e B , vicino all'intersezione tra $f_A(\mathbf{x})$ e $f_B(\mathbf{x})$

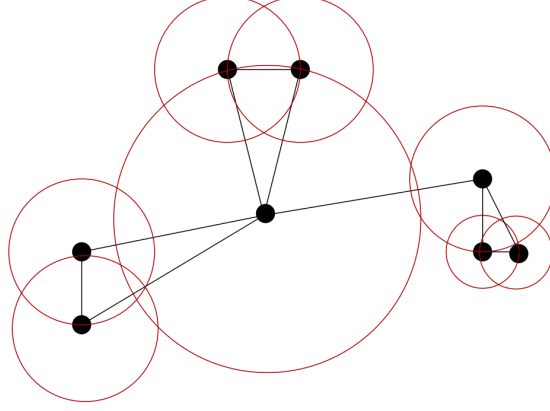


Figura 1.3: Rappresentazione grafica di un 1-SIG. I raggi delle circonferenze sono le distanze dei punti dal loro *nearest neighbour*. Quando due circonferenze si intersecano, viene creata una connessione tra i rispettivi centri.

devono esserci almeno due punti di A che risiedono su lati differenti di $f_B(\mathbf{x})$. La presenza di un'intersezione vicino a p_i può essere determinata calcolando $f_B(\mathbf{p}_i) \cdot f_B(\mathbf{p}_j)$ per ogni p_j in c_i . Se $f_B(\mathbf{p}_i) \cdot f_B(\mathbf{p}_j) \leq 0$, p_i e p_j sono su lati differenti di $f_B(\mathbf{x})$, per cui esiste un'intersezione. Grazie al grafo di prossimità costruito in precedenza, trovare i vicini di p_i può essere fatto velocemente con una ricerca breadth-first.

Il punto chiave di questo approccio è che il grafo permette di sfruttare la coerenza spaziale per selezionare punti vicini tra loro, invece di campionare casualmente i punti come proposto in Klein and Zachmann [6].

Infine va notato che per poter generare l'r-SIG, le distanze minime tra i punti devono essere note e deve essere precalcolata la matrice delle distanze del grafo.

1.1.3 Pan et al. [12]

In questo articolo viene proposto un algoritmo di machine-learning basato sulla *Support Vector Machine* (SVM). L'idea è quella di riformulare il problema di collision detection come un problema di classificazione in due classi.

Collision detection probabilistico. Dati due insiemi di punti $X = \{\mathbf{x}_i\}_{i=1}^n$ $Y = \{\mathbf{y}_j\}_{j=1}^m$ diciamo che sono *linearmente separabili* se esiste un piano $\mathbf{w} \cdot \mathbf{x} - b = 0$ che può completamente separare i due insiemi. In termini più formali, X e Y sono linearmente separabili se:

$$\mathbf{w} \cdot \mathbf{x}_i = -\mathbf{w} \cdot \mathbf{y}_j \quad (1.4)$$

$\forall i \in [1, n]$ e $j \in [1, m]$. Se le posizioni dei punti sono note esattamente, e assumendo che la superficie implicita delle due point clouds sia convessa¹, la presenza di una collisione può essere stimata determinando se i due insiemi sono linearmente separabili, vale a dire, trovando il piano che massimizza la distanza da entrambe le point cloud, se esiste. Se non esiste, è presente una collisione.

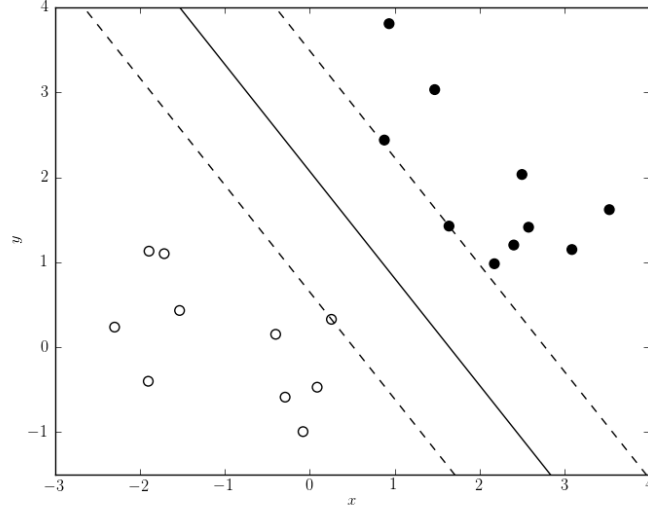


Figura 1.4: Problema standard di classificazione a due classi basato su SVM. I punti bianchi e neri appartengono alle due differenti point cloud. La linea nera massimizza la distanza tra i due insiemi di punti, le linee tratteggiate definiscono il margine tra i due insiemi. La larghezza del margine è $\frac{\|\mathbf{w}\|}{2}$.

Massimizzare la distanza tra due insiemi è un problema di ottimizzazione quadratica che può essere espresso come segue:

$$\begin{aligned} & \underset{\mathbf{w}, b}{\text{minimize}} && \frac{1}{2} \|\mathbf{w}\|^2 \\ & \text{subject to} && c_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1, \quad \forall 1 \leq i \leq n. \end{aligned} \tag{1.5}$$

dove i due insiemi di punti sono stati definiti come (p_i, c_i) con $c_i = 1$ se $p_i \in X$ e $c_i = -1$ se $p_i \in Y$.

¹Se la superficie implicita non è convessa, potrebbe non esistere un piano che separa i due insiemi di punti, anche se i due insiemi non stanno collidendo. In questo caso, il problema è comunque risolvibile, ma al posto di un piano è necessario usare una superficie non-lineare. In queste circostanze si può usare il *kernel trick* per linearizzare il problema di classificazione mappando non esplicitamente i punti dei due insiemi in uno spazio hermitiano a dimensionalità più elevata in cui i dati acquisiscano una struttura che permetta la classificazione lineare.

Ora, si prenda in considerazione anche il rumore sulla posizione dei punti. Il problema di ottimizzazione diventa trovare il piano tra le due point cloud che minimizza il *rischio* che un punto si trovi sul lato sbagliato del piano di separazione, vale a dire nel semipiano contenente l'altra point cloud. E' possibile stimare la probabilità di una collisione valutando questo rischio.

Matematicamente, il problema di ottimizzazione 1.5 diventa:

$$\begin{aligned}
& \underset{\mathbf{w}, b, \xi_i}{\text{minimize}} && \frac{1}{2} \|\mathbf{w}\|^2 + \lambda \sum_{i=1}^n \xi_i \\
& \text{subject to} && c_i(\mathbf{w}^T \mathbf{x}'_i + b) \geq 1 - \xi_i, \forall 1 \leq i \leq n. \\
& && \xi_i \geq 0, \forall 1 \leq i \leq n.
\end{aligned} \tag{1.6}$$

Dove $\mathbf{x}_i$ è la posizione esatta del punto e $\mathbf{x}'_i$ è quella misurata. Questa formulazione minimizza l'upper bound dell'errore di classificazione. A differenza di 1.5, 1.6 permette la presenza di punti sul lato sbagliato del piano, quindi 1.6 è risolvibile anche se i due insiemi di punti non sono separabili. Per ogni punto $\mathbf{x}_i$, ξ_i indica se $\mathbf{x}'_i$ è nel lato corretto ($0 \leq \xi_i \leq 1$) o sbagliato ($\xi_i > 1$) del piano (vedi figura 1.6). La probabilità che $\mathbf{x}_i$ risieda nel lato corretto, data la posizione di $\mathbf{x}'_i$, è inversamente proporzionale a ξ_i . Si possono quindi sfruttare i parametri ξ_i per stimare la probabilità di una collisione. Il parametro λ dipende dalla precisione delle misurazioni, e indica quanto peso dare ai punti che risiedono sul lato sbagliato del piano.

Struttura dati d'accelerazione. L'approccio presentato finora ha alcuni vantaggi. La complessità temporale della SVM è $O(n^3)$, dove $n = n_1 + n_2$ è il numero di punti totale di entrambe le point clouds. Inoltre, avere due point clouds con differenti dimensioni e quantità di punti può portare ad avere training sets sbilanciati e differenti errori di separazione. Per evitare questi problemi Jia Pan propone l'uso di una *Bounding Volume Hierarchy* (BVH) simile all'*Axis Aligned Bounding Box Tree* (AABB tree), con la principale differenza che l'attraversamento dell'albero è stocastico anzichè deterministico. La BVH è costruita con un approccio top-down nel seguente modo:

1. la radice dell'albero e' una box allineata con gli assi che contiene l'intera point cloud. Trattandosi di un AABB-tree, anche ogni nodo interno è una box *axis-aligned*.
2. per ogni nodo interno, calcola la matrice di covarianza dei punti all'interno del *bounding volume* (BV):

$$\mathbf{C} = \frac{1}{m} \sum_{i=1}^m (\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T$$

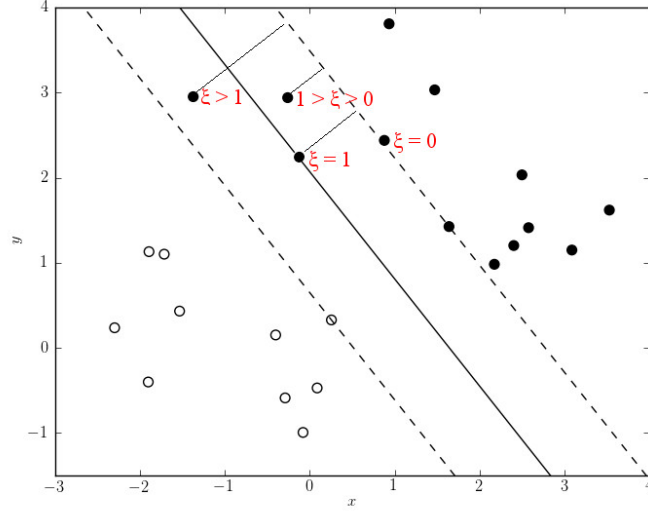


Figura 1.5: Error-wise two-class classification problem. Più grande è ξ_i , più lontano è il punto dal margine della sua point cloud. Nota che i punti bianchi e neri rappresentano le posizioni misurate dei punti, non quelle reali.

dove $\{\mathbf{x}_i\}_{i=1}^m$ sono i punti nel BV e $\bar{\mathbf{x}}$ è la loro media. La matrice $\mathbf{C}$ è una matrice 3×3 .

3. calcola i tre autovalori $\sigma_1, \sigma_2, \sigma_3$ della matrice $\mathbf{C}$ e definisci la *variazione* del BV come:

$$\sigma_n = \frac{\sigma_1}{\sigma_1 + \sigma_2 + \sigma_3}$$

assumendo che sia $\sigma_1 \leq \sigma_2 \leq \sigma_3$. σ_n è compreso tra 0 e $\frac{1}{3}$: se $\sigma_n = \frac{1}{3}$ significa che i punti sono isotropicamente distribuiti nel volume, se $\sigma_n = 0$ significa che risiedono tutti su uno stesso piano.

4. Se il numero di punti nel nodo è maggiore di n_{max} o σ_n è sopra σ_{max} (dove n_{max} e σ_{max} sono soglie fissate), dividi il BV in due lungo l'asse maggiore (i.e. l'autovettore corrispondente all'autovalore σ_3) e assegna i due BV così ottenuti ai figli. Ripeti il procedimento dal punto 2 per i figli appena creati.

Normalmente gli AABB tree sono costruiti in modo che nodi differenti non possano contenere la stessa primitiva. Tuttavia per le point cloud questo può portare a falsi negativi (i.e. mancate collisioni), come mostrato in figura 1.7. Due BV di due differenti point cloud potrebbero non intersecarsi anche se i punti in essi

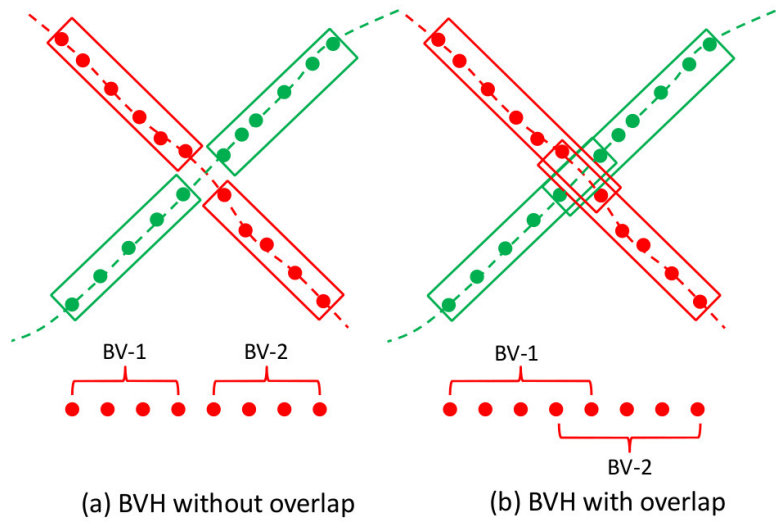


Figura 1.6: I punti verdi e rossi sono le point cloud dei due oggetti. In a) non c'è sovrapposizione e l'intersezione non è individuata. In b) la regione di sovrapposizione permette l'individuazione della collisione.

contenuti si intersecano. Per evitare che questo accada, è necessario che i BV si sovrappongano leggermente (per l' 1%-5% delle loro dimensioni).

Inoltre, questo AABB tree non garantisce di includere completamente la superficie della point cloud. Questo significa che con una piccola probabilità l'algoritmo deve controllare la presenza di collisioni anche tra nodi che non si intersecano. Questo è il motivo per cui l'attraversamento è stato definito probabilistico.

Per ultimo bisogna notare che, come cita lo stesso autore, quest'algoritmo è adatto per la determinazione di collisioni anche con mesh poligonali, che sono trattate come point clouds la cui descrizione è conosciuta in maniera esatta.

Capitolo 2

Strutture dati per la collision detection

La ricerca di collisioni tra due oggetti è un'operazione costosa. Per questo motivo è necessario utilizzare delle strutture dati che accelerino il processo scartando rapidamente oggetti e parti di oggetti che non possono contenere collisioni.

In computer grafica le scene solitamente prevedono la presenza di più oggetti in movimento, per cui in letteratura è comune dividere il processo di collision detection in due fasi [14, section 17.5.1]:

1. **Broad phase:** in questa fase si determina quali degli oggetti presenti sulla scena potrebbero entrare in collisione. L'uso di una struttura dati in questa fase ha lo scopo di evitare un confronto $O(n^2)$ tra tutti gli oggetti della scena a due a due per determinare quali potrebbero collidere.
2. **Narrow phase:** in questa fase si determina se due oggetti effettivamente collidono. Una struttura dati può essere utile per scartare porzioni dei due oggetti che non sono prossime all'area di collisione.

Per questa tesi sono d'interesse solo le strutture dati adatte alla narrow phase, in quanto nel nostro caso di studio è presente un solo oggetto in movimento (il robot manipolatore).

Esistono numerose strutture dati per il partizionamento dello spazio, che possono essere divise in due categorie principali:

- **Gerarchiche:** lo spazio (o l'oggetto) viene diviso in parti via via più piccole ed organizzate in un albero. La ricerca delle collisioni viene fatta attraversando l'albero e riducendo sempre di più la possibile area di collisione man mano che si scende in profondità. La struttura gerarchica permette di scartare rapidamente porzioni dello spazio ogni volta che si scende di un livello. A questa categoria appartengono ad esempio:

- K-d tree
- Octree
- *Bounding Volume Hierarchies*

Ciascuna di queste tre strutture dati verrà descritta in dettaglio nelle prossime sezioni.

- **Non gerarchiche:** a questa categoria appartengono le *grid* e lo *spatial hashing*. In questo caso lo spazio viene diviso in celle uguali che poi vengono organizzate in forma tabulare. Questo tipo di strutture dati sono particolarmente efficaci nella broad phase perchè permettono di isolare la ricerca di collisioni solo ad oggetti che si trovano dentro la stessa cella.

Nella scelta di quale struttura dati utilizzare, bisogna tener conto dei seguenti fattori:

- In linea generale, più una struttura dati permette di ricercare rapidamente le collisioni, maggiore sarà l'occupazione di memoria richiesta. Inoltre, aumenterà anche il tempo di preprocessing. Se ad esempio si utilizza una struttura ad albero, partizionare lo spazio in celle molto piccole rende l'algoritmo più veloce, ma l'occupazione di memoria cresce esponenzialmente con la profondità dell'albero e il tempo necessario per la creazione è pure proporzionale.
- strutture dati complesse possono essere molto efficienti, ma difficili da mantenere ed aggiornare. Una struttura dati può non essere robusta alla modifica dei dati su cui è costruita, al punto da richiedere il ricalcolo dell'intera struttura dati ad ogni aggiornamento della scena (si veda, ad esempio, il caso del k-d tree).

2.1 *Bounding Volume Hierarchies*

Le *Bounding Volume Hierarchies* (BVH) sono una classe di alberi con le seguenti caratteristiche:

- ogni nodo interno è costituito da un *Bounding Volume* (BV) che ingloba geometricamente tutte le primitive del suo sottoalbero.
- il BV di un nodo è contenuto in quello del padre.
- le primitive si trovano nelle foglie.

L'idea è incapsulare le primitive in solidi più semplici che le approssimino, per i quali la verifica di intersezione possa essere fatta velocemente. Una BVH si dice *overlapping* se la stessa primitiva può essere contenuta (in senso geometrico) nel BV di nodi differenti, *non-overlapping* se due BV allo stesso livello sono disgiunti. In generale la ricerca di collisioni con una BVH viene fatta come mostrato nell'algoritmo 2: si attraversano contemporaneamente gli alberi di due oggetti, se si trovano foglie che si intersecano si esegue il confronto tra le primitive contenute in quelle foglie.

Algorithm 2: `traverse(A,B)`

```

if A and B do not overlap then
  └ return
if A and B are leaves then
  | return intersection of primitives enclosed by A and B
else
  | foreach children A[i] of A do
  | | foreach children B[j] of B do
  | | | └ traverse(A[i],B[j])
  |

```

Le BVH si differenziano in base alla strategia per la creazione dell'albero e al solido utilizzato come bounding volume. La scelta della BV è soggetta ad un importante trade-off: più un BV è complesso più fitterà bene la primitiva che contiene, per cui l'attraversamento dell'albero proseguirà solo dove una collisione è effettivamente molto probabile. Al contrario, un BV semplice avrà un volume molto maggiore di quello della primitiva contenuta, per cui verrà rilevata la possibilità di una collisione molto più spesso. Come conseguenza, l'attraversamento dell'albero si dilungherà anche dove non sono realmente presenti collisioni. Di contro, solidi complessi richiedono più tempo per la verifica di intersezione, allungando il tempo dell'attraversamento. In poche parole, a seconda del BV scelto, si possono fare o molti confronti rapidi, o pochi confronti lenti.

Bounding volume comunemente utilizzati sono:

- **Sfere.** La verifica di intersezione è immediata: se la distanza tra i centri è maggiore della somma dei raggi, allora le sfere non si intersecano. Se però la primitiva si estende molto in una direzione rispetto alle altre, la sfera avrà un volume molto maggiore del solido in essa contenuto.
- **AABB** ovvero *axis-aligned bounding box*. L'AABB è un parallelepipedo allineato con il sistema di riferimento. Anche in questo caso il test d'intersezione è molto semplice. Siano $\mathbf{a}^{min}, \mathbf{a}^{max}$ e $\mathbf{b}^{min}, \mathbf{b}^{max}$ i vertici più vicino

e più distante dall'origine delle due box, queste sono disgiunte se e solo se $a_i^{min} > b_i^{max}$ o $b_i^{min} > a_i^{max}$ su ognuna delle tre componenti. Come per le sfere, anche in questo caso il fitting può essere molto lasco se la primitiva non è allineata con gli assi.

- **OBB** ovvero *oriented bounding box*. Sono parallelepipedi come gli AABB, ma hanno il grado di libertà in più di poter essere orientati. In questo caso il test di intersezione è più complesso, ma comunque abbastanza rapido, ed è trattato nel dettaglio nella sezione 4.2. Se la primitiva è convessa, l'OBB garantisce un buon fitting.
- **k-DOP** ovvero *discrete oriented polytope*. Il DOP è un politopo convesso costruito prendendo k piani orientati all'infinito e facendoli scorrere fino a che non collidono con l'oggetto da includere. L'AABB ad esempio è un caso particolare di DOP (6-DOP).

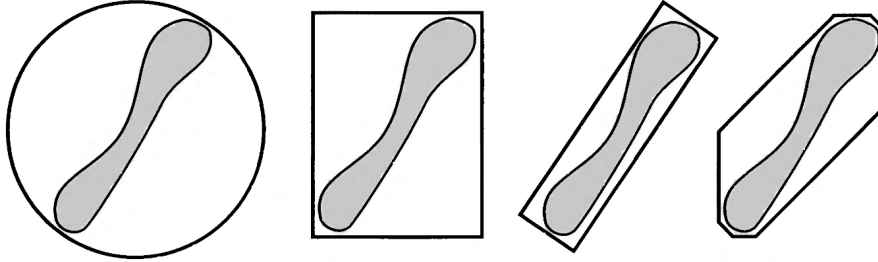


Figura 2.1: Nell'ordine, una sfera, un AABB, un OBB e un DOP. E' evidente come OBB e DOP garantiscano un fitting migliore [2].

2.2 K-d tree

Il k-d tree è stato proposto per la prima volta in Bentley [3]. Questa struttura dati è particolarmente adatta alle point clouds, in quanto pensata specificamente per l'organizzazione di punti k-dimensionali. Si può pensare al k-d tree come un albero binario in cui ogni nodo interno genera implicitamente un piano allineato con gli assi che divide lo spazio in due sotto-spazi. I punti nel lato destro del piano appartengono al sottoalbero destro, i punti nel lato sinistro del piano appartengono al sottoalbero sinistro.

Ciò che caratterizza un k-d tree è la strategia adottata per la generazione del piano: poichè esistono diversi modi di scegliere un piano separatore allineato con gli assi, esistono altresì modi differenti di costruire un k-d tree. Il metodo classico ha i seguenti vincoli:

- Scendendo nell'albero, cicla l'asse usato per scegliere il piano separatore. Vale a dire, in un albero 3-dimensionale, la radice genera un piano allineato all'asse x, i suoi figli piani allineati all'asse y, i figli dei figli allineati all'asse z e i figli dei figli dei figli di nuovo allineati con l'asse x e così via.
- Il piano scelto è la mediana dei punti del sottoalbero rispetto alle coordinate sull'asse usato per generare il piano separatore. Questo equivale ad ordinare i punti in ordine crescente rispetto alla coordinata scelta, e mettere la metà inferiore dei punti nel sottoalbero di sinistra e la metà superiore nel sottoalbero di destra. Si noterà la somiglianza del k-d tree con un albero binario di ricerca, di cui è di fatto un caso particolare.

Avendo scelto la mediana, i due sottoalberi conterranno esattamente lo stesso numero di punti, garantendo che l'albero finale sia bilanciato, cioè con foglie tutte a circa la stessa profondità. Il fatto di generare alberi bilanciati è proprio il motivo per cui questa strategia di generazione dei piani separatori è la più usata. Un albero bilanciato garantisce una profondità logaritmica e quindi anche tempi per l'attraversamento logaritmici. Si noti tuttavia che l'essere bilanciato non è una caratteristica intrinseca del k-d tree, ma una conseguenza della strategia scelta per costruirlo.

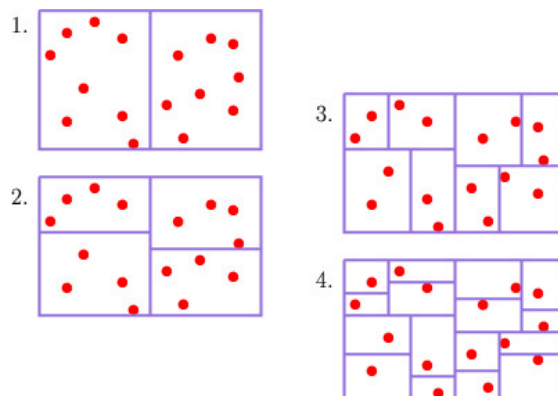


Figura 2.2: Construction of a k-d tree selecting the median of the points

Vantaggi:

1. adatto all'implementazione su point clouds.
2. veloce *nearest neighbour search* ($O(n \log n)$)
3. profondità dell'albero logaritmica garantita

4. l'albero è bilanciato e con un numero di punti limitato all'interno delle foglie. Questo fa sì che la complessità asintotica del k-d tree (per la ricerca di un punto) sia sicuramente logaritmica.

Svantaggi:

1. l'aggiunta di punti può portare ad alberi sbilanciati
2. alberi sbilanciati solitamente hanno performance peggiori
3. ribilanciare un k-d tree può essere un'operazione difficile e, a meno di non usare versioni più sofisticate del k-d tree (*Adaptive k-d tree*), richiede la ricostruzione completa dell'albero.
4. per le ragioni del punto precedente, se le primitive su cui è costruito l'albero sono dinamiche, in generale l'albero va ricostruito ogni volta che un elemento si muove
5. la costruzione di un k-d tree è un'operazione costosa. Con l'algoritmo descritto sopra trovare la mediana richiede l'ordinamento dei punti, quindi complessità $O(n \log n)$. Esistono algoritmi di selezione lineari nel tempo, ma sono comunque troppo costosi se applicati ad ogni nodo su tutti i punti in esso contenuti. Per ovviare a questo problema si usa un'euristica per il calcolo della mediana basata sulla scelta di un sottoinsieme casuale di punti, che si suppone avere mediana approssimativamente simile a quella di tutti gli n punti. Questo sistema solitamente funziona bene, ma ovviamente non garantisce la creazione di alberi bilanciati.
6. la libreria **PCL** offre metodi di *Nearest Neighbour Search* basati su k-d tree, ma fa uso della libreria esterna **FLANN** (*Fast Library for Approximate Nearest Neighbors*). Questa libreria è fortemente ottimizzata e molto veloce, ma è pensata esclusivamente per ricerche NN, e non mette a disposizione metodi per l'attraversamento dell'albero, e non può quindi essere riutilizzata per lo scopo di questa tesi.
7. a seconda del modo in cui viene implementato il k-d tree, questo può avere un elevato *aspect ratio*[13]. L'*aspect ratio* è il rapporto tra l'estensione nelle varie dimensioni di una cella. Celle con un elevato *aspect ratio* sono indicate in letteratura come celle *magre*, perchè sono molto estese in una direzione rispetto alle altre. La presenza di celle magre diventa un problema quando si eseguono ricerche approssimate di *nearest neighbour*. Una ricerca approssimata cerca i punti in un anello largo ϵ a distanza R dal punto di cui si

vogliono conoscere i vicini. Come si può intuire dalla figura 2.3, elevati aspect ratio rendono necessaria l'analisi di un gran numero di celle per questo tipo di ricerche.

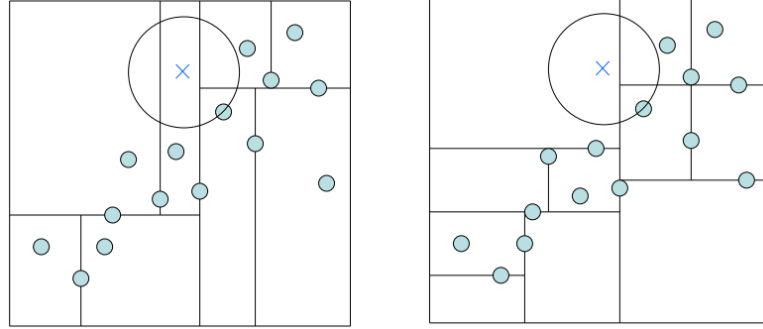


Figura 2.3: A sinistra un k-d tree con celle magre, a destra con celle grasse. La ricerca di vicini nella palla di raggio R richiede la visita di quattro celle per il k-d tree di sinistra, di solo tre nel k-d tree di destra

2.3 Octree

Gli octree sono stati sviluppati per la prima volta da Laboratory and Meagher [9] e, nella loro semplicità, si rivelano utili in molte circostanze. Inoltre, si adattano facilmente sia all'uso su mesh che su point cloud.

Un octree è un albero in cui ogni nodo interno è un cubo ed ha esattamente otto figli o nessuno, tutti cubici. Gli otto figli di un nodo sono degli ottanti, cioè otto cubi ottenuti tagliando il nodo secondo 3 piani ortogonali tra loro e alle facce del cubo e passanti per il centro del cubo stesso.

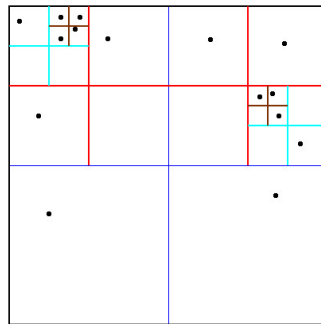


Figura 2.4: Quadtree con capacità 1. Il quadtree funziona allo stesso modo di un octree, ma in due dimensioni.

Esistono principalmente due modi per costruire un octree, a seconda dei vincoli che vengono posti sui nodi.

Nodi con capacità limitata: ogni nodo può contenere al più una quantità fissata di punti. Se ne contiene di più, viene diviso in ottanti. Partendo dalla radice, che contiene l'intera point cloud, se questa contiene più del numero massimo di punti, viene divisa in ottanti. Si ripete l'operazione per ogni ottante dividendolo in sotto-ottanti fino a quando il numero di punti all'interno del nodo non è inferiore alla soglia. In questo modo si garantisce che le foglie contengano un numero di punti inferiore ad un valore fissato, ma non si ha alcuna garanzia sulla profondità dell'octree, che può aumentare se zone della point cloud sono molto dense (i.e. molti punti in poco spazio).

Nodi con dimensione limitata: ogni nodo interno non può avere lati più corti di un valore prefissato. Partendo dalla radice, che anche in questo caso contiene l'intera point cloud, se questa ha lato minore del valore fissato o non contiene punti, l'algoritmo si ferma, altrimenti suddivide lo spazio in ottanti e ripete l'operazione sui figli. Si procede ricorsivamente fino a quando un nodo non ha dimensione inferiore a quella fissata o è vuoto. Con questa strategia non c'è un upper bound al numero di punti in una foglia, ma se la radice (che contiene l'intera point cloud) ha dimensione l_{root} e la dimensione fissata per le foglie è l_{leaves} , allora la profondità dell'albero ha upper bound

$$d \leq \lceil \log_2 \frac{l_{root}}{l_{leaves}} \rceil$$

questa strategia di creazione è quella adottata nell'implementazione dell'octree fornita dalla libreria PCL.

Vantaggi:

1. adatto all'uso su point cloud
2. l'octree è implementato nativamente in PCL e sono disponibili metodi per l'attraversamento, per cui può essere riutilizzato per lo scopo di questa tesi.
3. essendo ogni nodo cubico, l'*aspect ratio* è costante e pari a 1.
4. se si usa il primo approccio, si ha un limite noto al numero di punti in una foglia (sui quali va eseguita una ricerca lineare). Se si usa il secondo approccio, si ha un limite alla profondità dell'albero
5. tempo di preprocessing inferiore rispetto al k-d tree, grazie alla strategia triviale di creazione.

6. robustezza agli inserimenti/cancellazioni. L'inserimento di un nuovo punto si fa come in un albero di ricerca, con un attraversamento breadth-first dell'albero verificando ad ogni livello in quella degli otto ottanti si trova il punto. Raggiunta una foglia, viene aggiunto il punto nella foglia e, eventualmente, si divide la foglia in ottanti qualora l'aggiunta del punto avesse fatto superare la soglia di capacità fissata.

Svantaggi:

1. la semplicità dell'octree lo rende in generale più lento rispetto al k-d tree. Il k-d tree può, a differenza dell'octree, scegliere dove posizionare il piano separatore, il che permette a quest'albero di essere sempre bilanciato. L'octree di contro, salvo il caso improbabile in cui i punti siano distribuiti isotropicamente nello spazio, non è bilanciato, e la sua profondità varia a seconda della distribuzione dei punti nella nuvola, a prescindere dalla strategia di creazione scelta.
2. Un k-d tree garantisce profondità logaritmica e upper bound al numero di punti in una foglia. Un octree costruito con la prima strategia ha un upper bound al numero di punti in una foglia, ma non garantisce profondità logaritmica. Al contrario, un octree costruito con la seconda strategia garantisce un limite alla profondità, ma non garantisce limiti al numero di punti in una foglia.

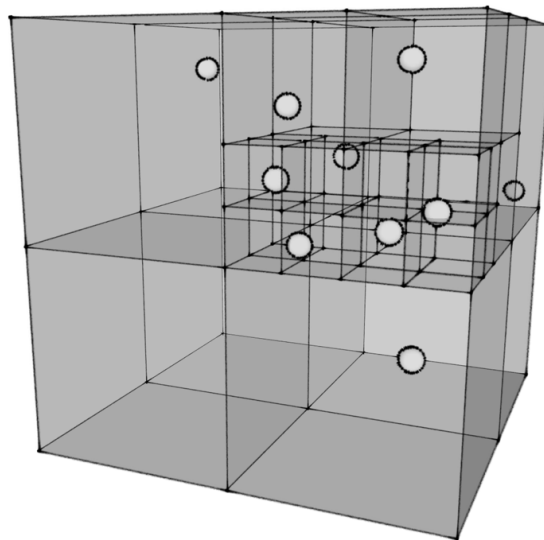


Figura 2.5: Octree

Capitolo 3

Descrizione dell'algoritmo attuale

In questo capitolo verrà descritto brevemente come il problema di collision detection è affrontato ora.

L'algoritmo in discussione in questa tesi è inserito in un software per la determinazione di posizioni di presa per un braccio robotico. Il braccio robotico è descritto con l'ausilio di due classi: la classe `Gripper` e la classe `GripperComponent`.

- La classe `GripperComponent` rappresenta una parte del braccio robotico. Ciascun componente è rappresentato in maniera semplificata come una scatola orientata. La scatola è descritta attraverso un vettore delle dimensioni lungo i tre assi e una matrice 4x4 di trasformazione affine nell'implementazione fornita dalla libreria `Eigen`. Si ricorda che una matrice di trasformazione affine è una matrice del tipo

$$\mathbf{R} = \begin{pmatrix} b_{00} & b_{01} & b_{02} & t_u \\ b_{10} & b_{11} & b_{12} & t_v \\ b_{20} & b_{21} & b_{22} & t_w \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

in cui la matrice 3x3 costituita dalle prime 3 righe/colonne descrive una trasformazione lineare, e l'ultima colonna una traslazione. Trasformazioni affini possono essere concatenate producendo queste matrici esattamente come nel caso di trasformazioni lineari.

È importante tener presente che la posizione ed orientazione del componente non è riferita al sistema di riferimento assoluto, ma al sistema di riferimento del braccio robotico. In questo modo per applicare una trasformazione al braccio è sufficiente moltiplicare la matrice di trasformazione per le matrici di ciascun componente.

La classe contiene anche alcune informazioni secondarie (come il nome e il colore del componente)

- La classe **Gripper** rappresenta l'intero braccio robotico. Contiene un vettore di **GripperComponent** e una matrice di trasformazione che descrive la posizione del braccio rispetto al sistema di riferimento assoluto. Inoltre, la classe contiene anche una bounding box orientata che incapsula l'intero gripper. L'utilità di questa scatola verrà descritta più avanti.

L'implementazione attuale sfrutta il filtro **cropBox** dalla libreria *PCL*. La funzione **CropBox::applyFilter()** itera semplicemente attraverso i punti della cloud cercando quelli che si trovano all'interno di una scatola specificata in input. Una descrizione più dettagliata è riportata nello pseudocodice seguente:

Algorithm 3: **CropBox::applyFilter()** algorithm

```

Data: input point cloud P
minimum vertex of the box m
maximum vertex of the box M
4x4 transformation matrix T of the box
Result: v vector of points inside the box
foreach p in P do
    // get the point position respect to the crop box coordinate system
     $\mathbf{p}_{rel} = T^{-1}\mathbf{p}$ 
    // check if p is inside the box.  $e_1, e_2, e_3$  is the canonical base
    if  $(\mathbf{p}_{rel} - \mathbf{m}) \cdot e_1 < 0$  OR  $(\mathbf{p}_{rel} - \mathbf{m}) \cdot e_2 < 0$  OR  $(\mathbf{p}_{rel} - \mathbf{m}) \cdot e_3 < 0$  then
        | continue;
    end
    if  $(\mathbf{p}_{rel} - \mathbf{M}) \cdot e_1 > 0$  OR  $(\mathbf{p}_{rel} - \mathbf{M}) \cdot e_2 > 0$  OR  $(\mathbf{p}_{rel} - \mathbf{M}) \cdot e_3 > 0$  then
        | continue;
    end
    v.add(p);
end
return v;

```

Per determinare se un punto si trova all'interno della scatola, per prima cosa ne viene calcolata la posizione rispetto al sistema di riferimento della scatola. Ora il punto si trova all'interno della scatola solo se è maggiore del vertice minimo su ogni componente e minore del vertice massimo su ogni componente.

L'algoritmo di ricerca delle collisioni viene eseguito in due parti:

1. Utilizza **cropBox** specificando in input la scatola che ingloba l'intero braccio robotico per determinare un primo set di punti che potrebbero collidere con il braccio.

2. Per ogni componente, riutilizza lo stesso filtro per determinare quali dei punti trovati al punto precedente sono effettivamente all'interno del componente.

Questo approccio può essere sufficientemente efficace per point cloud piccole, ma richiede l'iterazione almeno una volta su tutti i punti della nuvola, per una complessità asintotica $O(n)$.

Capitolo 4

Descrizione dell'algoritmo proposto

La soluzione proposta in questa tesi sfrutta un albero di ricerca per garantire una complessità $O(\log n)$ sul numero di punti della point cloud per la collision detection. In un primo momento ci si era orientati sull'uso di un k-d tree come struttura dati d'accelerazione. Durante la fase di collision detection la point cloud non subisce alcuna modifica, per cui l'uso di un k-d tree sarebbe stato adeguato in questo scenario. Tuttavia, non esistono librerie che permettano la costruzione e l'attraversamento di k-d tree per point cloud. Si è quindi preferito orientarsi su un Octree, di cui esiste un'implementazione utilizzabile nella libreria PCL. Come si mostrerà nel capitolo 5, questa struttura dati garantisce comunque un miglioramento delle performance decisamente significativo.

D'ora in avanti si farà riferimento all'algoritmo proposto in questo capitolo come algoritmo Octree.

4.1 Attraversamento della struttura dati

L'algoritmo 4 mostra lo pseudo-codice per l'attraversamento dell'octree. Come si potrà notare, l'algoritmo è una versione particolare dell'algoritmo di attraversamento generico riportato nel capitolo 2.

Per ogni componente del gripper viene fatto un attraversamento *depth-first* dell'octree partendo dalla radice. Ogni volta che si giunge in un nodo interno, si verifica se questo interseca il componente. Poichè ogni nodo interno di un octree è un cubo allineato con gli assi (*voxel*), i suoi vertici minimo e massimo sono sufficienti per descriverlo (cfr. capitolo 3). Se effettivamente c'è un'intersezione, si continua con l'attraversamento, altrimenti si salta la visita del sottoalbero e si ritorna al padre del nodo corrente. La struttura ad albero garantisce che se un nodo non interseca un volume pure i suoi figli non lo intersecheranno. Quando si raggiunge una foglia, bisogna fare un controllo lineare di tutti i punti in essa

contenuti per verificare quanti di questi siano anche all'interno del componente. Questi punti verranno aggiunti alla lista di quelli in collisione. L'algoritmo sfrutta due metodi ausiliari: il metodo `component.pointsInside` per determinare i punti in collisione e il metodo `component.intersectBox` per verificare se il nodo e il componente si stanno intersecando. Il primo metodo è del tutto simile al metodo `CropBox` descritto nel capitolo precedente, il secondo verrà illustrato nel dettaglio nella prossima sezione.

Algorithm 4: `Gripper::getObjectCollisionPointsOtree`

Input:
`gripper`: istanza della classe `gripper`;
`octree`: octree già inizializzato sulla point cloud;
`M`: matrice di rototraslazione del `gripper`;
Output: $\mathbf{v}_{in}$ vettore dei punti dentro al `gripper`;

```

begin
     $\mathbf{v}_{in} = []$ ;
    foreach component in gripper do
        it = depthFirstIterator(octree);
        while it  $\neq$  NULL do
            if it.isLeafNode() then
                 $\mathbf{v}_{leaf} = \text{it.getPointsInNode}()$ ;
                 $\mathbf{v}_{in}.\text{add}(\text{component.pointsInside}(\mathbf{M}, \mathbf{v}_{leaf}))$ ;
                continue;
            else
                 $\mathbf{p}_{min}, \mathbf{p}_{max} = \text{it.getBoundingBoxOfNode}()$ ;
                if component.intersectBox( $\mathbf{p}_{min}, \mathbf{p}_{max}, \mathbf{M}$ ) then
                    continue;
                else
                    it.skipChildren();
        return v;

```

4.2 Il metodo `intersectBox`

Per decidere se visitare i figli del nodo corrente o no è necessario verificare se esiste un'intersezione tra il voxel del nodo e il componente. Si tratta quindi di trovare un algoritmo per determinare l'intersezione tra due parallelepipedi orientati arbitrariamente l'uno rispetto all'altro (OBB/OBB intersection). L'algoritmo scelto

per questa verifica è stato esposto per la prima volta in [5]. L'algoritmo ha come supporto teorico il *Separating Axis Theorem* (abbr. *SAT*), di cui viene data una breve spiegazione nella sezione seguente.

4.2.1 Richiamo teorico: il *Separating Axis Theorem*

Data la seguente definizione:

Definizione 1. Due insiemi A e B in $\mathbb{R}^n$ si dicono **strettamente separabili** se risiedono in semispazi aperti disgiunti. Cioè se, dato un iperpiano $a_1x_1 + \dots + a_nx_n = \alpha$, $A \subset a_1x_1 + \dots + a_nx_n < \alpha$ e $B \subset a_1x_1 + \dots + a_nx_n > \alpha$ o viceversa. A e B si dicono **fortemente separabili** se risiedono in semispazi chiusi e disgiunti. Cioè se $\exists \epsilon > 0$ t.c. $A \subset a_1x_1 + \dots + a_nx_n \leq \alpha - \epsilon$ e $B \subset a_1x_1 + \dots + a_nx_n \geq \alpha + \epsilon$ o viceversa.

Il teorema *Sat* dice che:

Teorema 1. Dati due insiemi A e B non vuoti, disgiunti, convessi e compatti in uno spazio euclideo n -dimensionale, esiste un iperpiano h che separa fortemente i due insiemi. Inoltre, esiste un iperpiano p ortogonale ad h tale che le proiezioni ortogonali di A e B su p sono compatte e disgiunte.

Da questo teorema ne segue un altro più specifico:

Teorema 2. Dati due politopi A e B convessi in $\mathbb{R}^3$ essi sono disgiunti se e solo se esiste un piano p che separa i due politopi parallelo ad una faccia di uno dei due politopi o a uno spigolo di A e a uno spigolo di B .

Se anzichè prendere in considerazione il piano di separazione consideriamo l'asse ad esso ortogonale, otteniamo quest'ultimo enunciato:

Teorema 3. Dati due politopi A e B convessi in $\mathbb{R}^3$ essi sono disgiunti se e solo se esiste una retta L ortogonale ad una faccia di uno dei due politopi o a uno spigolo di A e a uno di B tale che le proiezioni ortogonali di A e B su L sono intervalli compatti disgiunti. L è detto *asse di separazione* (Separating axis) di A e B .

In altre parole, dati due poliedri aventi rispettivamente F_A e F_B facce e S_A e S_B spigoli ¹, per verificare se si intersecano è sufficiente un set opportuno di $F_A + F_B + S_A \cdot S_B$ assi su cui proiettare ortogonalmente i due poliedri. Se in nessuno di questi assi i due intervalli dati dalla proiezione si intersecano, allora i due poliedri sono disgiunti.

¹Sarebbe più giusto parlare di *orientazioni* di facce e spigoli in quanto un test eseguito su due assi tra loro paralleli dà lo stesso risultato. Di conseguenza facce parallele vanno contate una sola volta. Per chiarire le idee, un cubo ha 6 facce e 12 spigoli, ma ha solo tre orientazioni per le facce e 3 orientazioni per gli spigoli

4.2.2 Preparazione

Indichiamo con A il voxel dell'octree e con B la bounding box del componente.

Per prima cosa, bisogna adottare un sistema di riferimento conveniente e calcolare la posizione delle due box rispetto a questo sistema. Scegliamo un sistema centrato nel centro del voxel e con gli assi allineati ad esso. Si ricordi che si sta cercando l'intersezione tra un voxel di un octree e la bounding box di un componente: mentre la bounding box può essere disposta arbitrariamente, il voxel è sempre allineato con gli assi, per cui la sua matrice di rotazione è l'identità e pertanto non è necessaria alcuna rotazione per allineare gli assi del voxel con quelli del nuovo sistema.

Il metodo riceve in input:

- **T**: la matrice di rototraslazione del componente nel sistema di riferimento assoluto
- h, l, d : altezza, larghezza e profondità della bounding box del componente
- $\mathbf{v}_{min}$ e $\mathbf{v}_{max}$: i vertici massimo e minimo del voxel.

Mentre l'algoritmo richiede i seguenti parametri:

- **R**: la matrice di rototraslazione del componente nel nuovo sistema di riferimento
- $\beta = (\beta_u, \beta_v, \beta_w)$: estensione del componente lungo i tre assi
- $\alpha = (\alpha_u, \alpha_v, \alpha_w)$: estensione del voxel lungo i tre assi

Per estensione di A e B si intende semplicemente metà della dimensione lungo i tre assi. Per cui

$$\alpha = \frac{1}{2}(\mathbf{v}_{max} - \mathbf{v}_{min})$$
$$\beta = \frac{1}{2}(l, h, d)$$

La traslazione del voxel rispetto l'origine è la posizione del suo centro nel sist. di riferimento assoluto. Quindi è:

$$\mathbf{t}_A = \mathbf{v}_{min} + \frac{1}{2}(\mathbf{v}_{max} - \mathbf{v}_{min})$$

E la matrice di rototraslazione di A è:

$$\mathbf{T}_A = \begin{pmatrix} \mathbf{I} & \mathbf{t}_A \\ 0 & \dots & 1 \end{pmatrix}$$

Essendo A allineato con gli assi.

La matrice $\mathbf{R}$ si ottiene come

$$\mathbf{R} = \mathbf{T}_A^{-1} \mathbf{T}_B$$

4.2.3 L'algoritmo di intersezione

Un parallelepipedo ha tre differenti orientazioni per le facce e tre differenti direzioni per gli spigoli. Questo significa che per verificare la presenza di intersezioni sono necessarie 15 verifiche. 3 per le facce di A, 3 per le facce di B e 9 per tutte le coppie di spigoli di A e B. Assumiamo un sistema di riferimento centrato nel centro di A formato dagli assi di A. In questo modo gli assi di A sono la tripletta ortonormale $\mathbf{a}^u = (1, 0, 0)$, $\mathbf{a}^v = (0, 1, 0)$ e $\mathbf{a}^w = (0, 0, 1)$. B e' rototraslato rispetto ad A, e la sua posizione puo' essere descritta con la seguente matrice di trasformazione affine:

$$\mathbf{R} = \begin{pmatrix} b_{00} & b_{01} & b_{02} & t_u \\ b_{10} & b_{11} & b_{12} & t_v \\ b_{20} & b_{21} & b_{22} & t_w \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{b}^u & \mathbf{b}^v & \mathbf{b}^w & \mathbf{t} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Dove $\mathbf{b}^u, \mathbf{b}^v, \mathbf{b}^w$ sono gli assi ortonormali di B nel sistema di riferimento di A e $\mathbf{t}$ e' la traslazione di B rispetto ad A. I vettori $\boldsymbol{\alpha} = (\alpha_u, \alpha_v, \alpha_w)$ e $\boldsymbol{\beta} = (\beta_u, \beta_v, \beta_w)$ rappresentano le dimensioni di A e B lungo i loro tre assi. I valori $\alpha_u, \alpha_v, \alpha_w, \beta_u, \beta_v, \beta_w$ sono sempre positivi per definizione.

I 15 assi su cui va effettuato il test sono quindi:

1. I tre assi di A: $\mathbf{a}^u, \mathbf{a}^v$ e $\mathbf{a}^w$
2. I tre assi di B: $\mathbf{b}^u, \mathbf{b}^v, \mathbf{b}^w$
3. I nove assi $\mathbf{c}^{ij} = \mathbf{a}^i \times \mathbf{b}^j, \forall i \in \{u, v, w\}$ e $\forall j \in \{u, v, w\}$

Dato un generico asse separatore $\mathbf{s}$, $|\mathbf{t} \cdot \mathbf{s}|$ è la proiezione della distanza tra A e B su $\mathbf{s}$. Le proiezioni di A e B su $\mathbf{s}$ sono due intervalli, e i centri di A e B vengono proiettati nel punto medio dei due intervalli. I raggi degl'intervalli (cioè la metà della loro lunghezza) si calcolano come

$$r_A = \sum_{i \in \{u, v, w\}} \alpha_i |\mathbf{a}^i \cdot \mathbf{s}| \quad (4.1)$$

$$r_B = \sum_{i \in \{u, v, w\}} \beta_i |\mathbf{b}^i \cdot \mathbf{s}| \quad (4.2)$$

Poichè $\mathbf{t}$ è la distanza tra i centri di A e B, $|\mathbf{t} \cdot \mathbf{s}|$ è la distanza tra i centri degli intervalli ottenuti dalle proiezioni di A e B. Se

$$|\mathbf{t} \cdot \mathbf{s}| > r_A + r_B \quad (4.3)$$

allora i due intervalli non si intersecano.

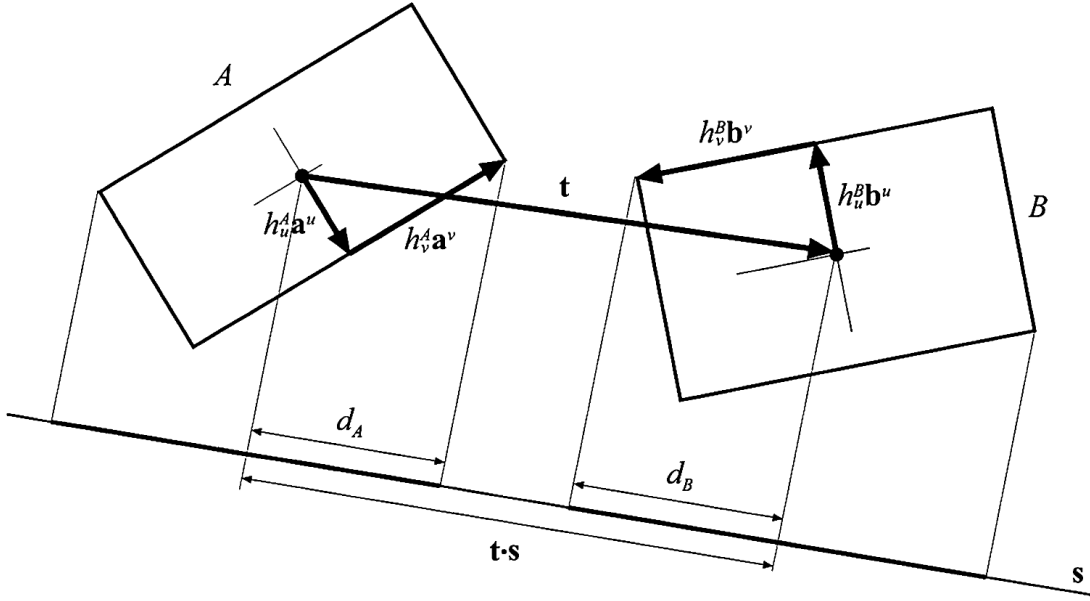


Figura 4.1: Separating axis theorem: la proiezione sull'asse della distanza tra i due centri è maggiore della somma delle semi dimensioni, quindi senz'altro i due poligoni sono separabili

Rimane da analizzare come semplificare l'equazione 4.3 nei tre casi.

Test I - $\mathbf{s} = \mathbf{a}^k$: analizziamo solo il caso $k = u$, essendo gli altri analoghi. $|\mathbf{t} \cdot \mathbf{s}|$ diventa:

$$|\mathbf{t} \cdot \mathbf{s}| = |\mathbf{t} \cdot \mathbf{a}^u| = |t_u|$$

L'equazione 4.1 si semplifica notevolmente:

$$r_A = \sum_{i \in \{u, v, w\}} \alpha_i |\mathbf{a}^i \cdot \mathbf{a}^u|$$

$$r_A = \alpha_u |\mathbf{a}^u \cdot \mathbf{a}^u| = \alpha_u$$

E l'equazione 4.2 diventa:

$$\begin{aligned} r_B &= \sum_{i \in \{u,v,w\}} \beta_i |\mathbf{b}^i \cdot \mathbf{a}^u| \\ r_B &= \beta_u |b_{00}| + \beta_v |b_{01}| + \beta_w |b_{02}| \end{aligned}$$

E quindi 4.3 si riduce a

$$|t_u| > \alpha_u + \beta_u |b_{00}| + \beta_v |b_{01}| + \beta_w |b_{02}| \quad (4.4)$$

Test II - $\mathbf{s} = \mathbf{b}^k$: come prima, analizziamo solo il caso $k = u$, essendo gli altri analoghi. $|\mathbf{t} \cdot \mathbf{s}|$ diventa:

$$|\mathbf{t} \cdot \mathbf{s}| = |\mathbf{t} \cdot \mathbf{b}^u| = |b_{00}t_u + b_{10}t_v + b_{20}t_w|$$

L'equazione 4.1 diventa:

$$\begin{aligned} r_A &= \sum_{i \in \{u,v,w\}} \alpha_i |\mathbf{a}^i \cdot \mathbf{b}^u| \\ r_A &= \alpha_u |\mathbf{a}^u \cdot \mathbf{b}^u| + \alpha_v |\mathbf{a}^v \cdot \mathbf{b}^u| + \alpha_w |\mathbf{a}^w \cdot \mathbf{b}^u| \\ r_A &= \alpha_u |b_{00}| + \alpha_v |b_{10}| + \alpha_w |b_{20}| \end{aligned}$$

Mentre, in maniera speculare a prima, l'equazione 4.2 si semplifica molto:

$$\begin{aligned} r_B &= \sum_{i \in \{u,v,w\}} \beta_i |\mathbf{b}^i \cdot \mathbf{b}^u| \\ r_B &= \beta_u |\mathbf{b}^u \cdot \mathbf{b}^u| = \beta_u \end{aligned}$$

E quindi 4.3 diventa:

$$|b_{00}t_u + b_{10}t_v + b_{20}t_w| > \alpha_u |b_{00}| + \alpha_v |b_{10}| + \alpha_w |b_{20}| + \beta_u \quad (4.5)$$

Test III - $\mathbf{s} = \mathbf{a}^k \times \mathbf{b}^l$: scegliamo il caso $k = u, l = v$. Anch'ora una volta gli altri casi si deducono facilmente da questo.

$$\begin{aligned} |\mathbf{t} \cdot \mathbf{s}| &= |\mathbf{t} \cdot (\mathbf{b}^v \times \mathbf{a}^u)| = \\ &= |\mathbf{t} \cdot ((1, 0, 0) \times (b_{01}, b_{11}, b_{21}))| = \\ &= |\mathbf{t} \cdot (0, -b_{21}, b_{11})| = \\ &= |t_w b_{11} - t_v b_{21}| \end{aligned}$$

La presenza del prodotto vettoriale rende un pò più complicate le formule per r_A e r_b :

$$d_A = \sum_{i \in \{u,v,w\}} \alpha_i |\mathbf{a}^i \cdot \mathbf{s}| = \sum_{i \in \{u,v,w\}} \alpha_i |\mathbf{a}^i \cdot (\mathbf{b}^v \times \mathbf{a}^u)|$$

Il risultato del prodotto misto $\cdot, \times$ non cambia in seguito ad una permutazione ciclica dei fattori. Vale a dire:

$$\mathbf{a}^i \cdot (\mathbf{b}^v \times \mathbf{a}^u) = \mathbf{b}^v \cdot (\mathbf{a}^u \times \mathbf{a}^i)$$

e quindi:

$$\begin{aligned} d_A &= \sum_{i \in \{u,v,w\}} \alpha_i |\mathbf{b}^v \cdot (\mathbf{a}^u \times \mathbf{a}^i)| \\ &= \alpha_u |\mathbf{b}^v \cdot (\mathbf{a}^u \times \mathbf{a}^u)| + \alpha_v |\mathbf{b}^v \cdot (\mathbf{a}^u \times \mathbf{a}^v)| + \alpha_w |\mathbf{b}^v \cdot (\mathbf{a}^u \times \mathbf{a}^w)| \\ &= \alpha_v |\mathbf{b}^v \cdot \mathbf{a}^w| + \alpha_w |\mathbf{b}^v \cdot \mathbf{a}^v| \\ &= \alpha_v |b_{21}| + \alpha_w |b_{11}| \end{aligned}$$

$$\begin{aligned} d_B &= \sum_{i \in \{u,v,w\}} \beta_i |\mathbf{b}^i \cdot \mathbf{s}| = \sum_{i \in \{u,v,w\}} \beta_i |\mathbf{b}^i \cdot (\mathbf{b}^v \times \mathbf{a}^u)| \\ &= \sum_{i \in \{u,v,w\}} \beta_i |\mathbf{a}^u \cdot (\mathbf{b}^i \times \mathbf{b}^v)| \\ &= \beta_u |\mathbf{a}^u \cdot (\mathbf{b}^u \times \mathbf{b}^v)| + \beta_v |\mathbf{a}^u \cdot (\mathbf{b}^v \times \mathbf{b}^v)| + \beta_w |\mathbf{a}^u \cdot (\mathbf{b}^w \times \mathbf{b}^v)| \\ &= \beta_u |\mathbf{a}^u \cdot \mathbf{b}^w| + \beta_w |\mathbf{a}^u \cdot \mathbf{b}^u| \\ &= \beta_u |b_{02}| + \beta_w |b_{00}| \end{aligned}$$

In conclusione 4.3 diventa:

$$|t_w b_{11} - t_v b_{21}| > \alpha_v |b_{21}| + \alpha_w |b_{11}| + \beta_u |b_{02}| + \beta_w |b_{00}| \quad (4.6)$$

Si assuma che $\mathbf{R}_{ij}$ rappresenti il numero alla riga $i + 1$ e colonna $j + 1$ della matrice $\mathbf{R}$, quindi per esempio $\mathbf{R}_{13} = t_v$. Inoltre si assuma α_i la $i+1$ -esima componente di α , per cui $\alpha_0 = \alpha_u$, $\alpha_1 = \alpha_v$, $\alpha_2 = \alpha_w$. Si usi la stessa notazione per β . Date queste definizioni, sfruttando l'aritmetica modulare si può ottenere una forma generica dei casi I, II e III.

Definiamo

$$i \rightarrow k = (i + k) \mod 3$$

Per esempio $0 \rightarrow 1 = 1$ ma anche $2 \rightarrow 2 = 1$.

Nel caso I, si ottengono le seguenti formule valide per $\mathbf{s} = \mathbf{a}^k, \forall k \in \{u, v, w\}$ al variare di i in $[0, 2]$:

$$\begin{aligned} |\mathbf{t} \cdot \mathbf{s}| &= |\mathbf{R}_{i3}| \\ r_A &= \alpha_i \\ r_B &= \beta_0 |\mathbf{R}_{i0}| + \beta_1 |\mathbf{R}_{i1}| + \beta_2 |\mathbf{R}_{i2}| \end{aligned}$$

Nel caso II le formule diventano:

$$\begin{aligned} |\mathbf{t} \cdot \mathbf{s}| &= |\mathbf{R}_{03}\mathbf{R}_{0i} + \mathbf{R}_{13}\mathbf{R}_{1i} + \mathbf{R}_{23}\mathbf{R}_{2i}| \\ r_A &= \alpha_0 |\mathbf{R}_{i0}| + \alpha_1 |\mathbf{R}_{i1}| + \alpha_2 |\mathbf{R}_{i2}| \\ r_B &= \beta_i \end{aligned}$$

Mentre nel caso III le nove combinazioni possono essere espresse al variare di i e j come:

$$\begin{aligned} |\mathbf{t} \cdot \mathbf{s}| &= |\mathbf{R}_{i \rightarrow 1j} \mathbf{R}_{i \rightarrow 23} - \mathbf{R}_{i \rightarrow 2j} \mathbf{R}_{i \rightarrow 13}| \\ r_A &= \alpha_{i \rightarrow 1} |\mathbf{R}_{i \rightarrow 2j}| + \alpha_{i \rightarrow 2} |\mathbf{R}_{i \rightarrow 1j}| \\ r_B &= \beta_{j \rightarrow 1} |\mathbf{R}_{ij \rightarrow 2}| + \beta_{j \rightarrow 2} |\mathbf{R}_{ij \rightarrow 1}| \end{aligned}$$

A conclusione di questa sezione è opportuno fare due osservazioni:

1. I valori assoluti degli elementi di $\mathbf{R}$ vengono usati quattro volte ciascuno (una volta nei test I e II e due volte nel test III), per cui computare $|\mathbf{R}|$ avente gli stessi elementi di $\mathbf{R}$ ma in valore assoluto riduce del 75% il calcolo dei valori assoluti.
2. Secondo Gottschalk et al. il numero di operazioni richieste per l'intero algoritmo è di circa 200, a cui vanno aggiunte quelle necessarie per calcolare la matrice $\mathbf{R}$ nel corretto sistema di riferimento. Tuttavia questa stima è al caso peggiore, che è quello in cui sia effettivamente presente una collisione. Se le box non collidono, l'algoritmo conclude per l'assenza di intersezioni prima di raggiungere la fine. Per questo stesso motivo, l'ordine in cui sono stati presentati i tre test (che è anche l'ordine in cui vanno poi implementati) è rilevante. I 9 test del caso III richiedono più calcoli (110 operazioni) e andrebbero eseguiti per ultimi, in modo che vengano eseguiti meno frequentemente. Al contrario, il test I è il più rapido (30 operazioni) e andrebbe quindi eseguito per primo.

4.3 Setting ottimale dell'Octree

Come già discusso nella sezione 2.3 la costruzione dell'octree richiede la specificazione di un parametro che fissa la dimensione minima per i voxel dell'albero. Lo scopo di questa sezione è determinare quale sia il valore ottimale per tale parametro, che d'ora in poi chiameremo *risoluzione*. Ci aspettiamo il seguente comportamento:

- al diminuire della risoluzione, il tempo per la collision detection diminuisce. La ricerca dei punti di collisione all'interno delle foglie è lineare. Riducendo la risoluzione si aumenta il numero di foglie, diminuendo conseguentemente il numero di punti in ciascuna foglia. Tuttavia, non diminuisce per sempre: per risoluzioni molto basse, l'albero diventa così profondo che il costo dell'attraversamento (con le verifiche di intersezione) diventa predominante sulla ricerca lineare nelle foglie.
- all'aumentare della risoluzione, il tempo di preprocessing aumenta, così come l'occupazione di memoria. Questo ovviamente perchè più la risoluzione è bassa più sono i nodi dell'albero.

Il grafico 4.2 mostra i tempi di collisione in funzione della risoluzione dell'octree per point cloud larghe circa 1000 e con un numero di punti all'interno variabile. Si può notare come i tempi raggiungano rapidamente un minimo per risoluzioni cento volte inferiori a quelle della cloud, per poi risalire. Sperimentalmente la risoluzione ottimale è quella che garantisce un numero medio di punti nelle foglie dell'ordine di $10 - 10^2$. Si noti che la quantità media di punti nelle foglie è indipendente dal numero totale di punti, mentre dipende dalla densità dei punti. Il numero di punti in una point cloud è variabile, mentre non lo è significativamente la densità, che dipende dal passo di campionamento della point cloud ed è pertanto un parametro costante. Questo implica che anche il valore ottimale per la risoluzione può essere mantenuto costante anche per cloud con un numero diverso di punti.

Il grafico 4.3 mostra invece i tempi di creazione dell'octree in funzione della risoluzione, nelle stesse condizioni sperimentali precedenti. Come previsto, il tempo di creazione ha un andamento iperbolico.

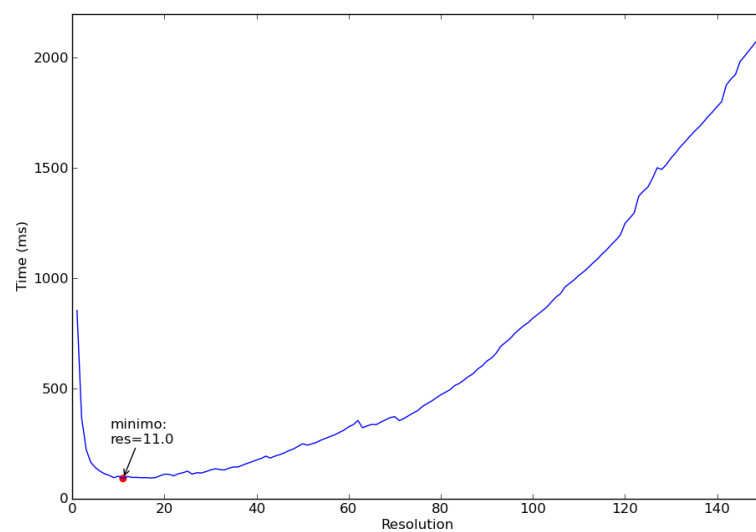


Figura 4.2: Tempi di collision detection in funzione della risoluzione

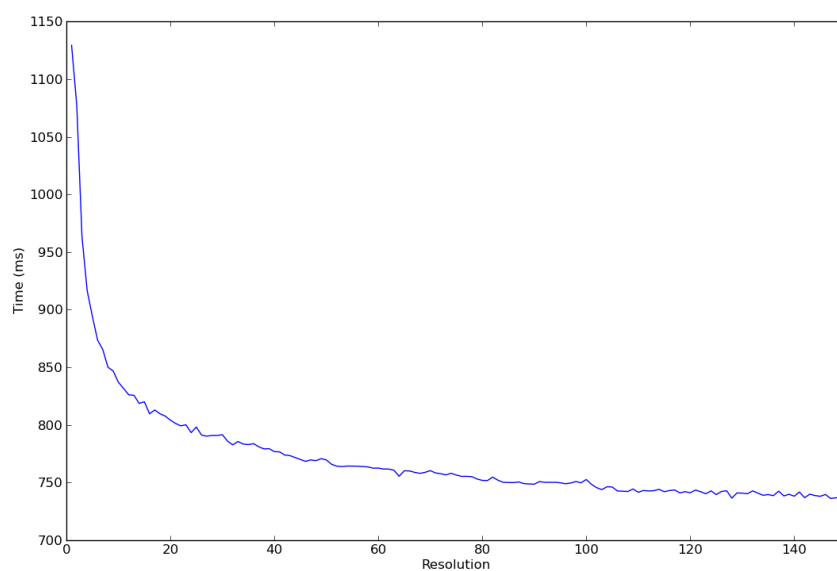


Figura 4.3: Tempi di collision detection in funzione della risoluzione

Capitolo 5

Confronto delle prestazioni

In questo capitolo verranno illustrate alcune caratteristiche peculiari per quanto riguarda le performance degli algoritmi presi in esame, supportate da risultati numerici. I dati presentati in questo capitolo sono stati ottenuti da una simulazione, i cui dettagli verranno descritti nella sezione seguente.

5.1 Dettagli implementativi

L'algoritmo presentato in questa tesi è stato implementato in C++ sfruttando le seguenti librerie esterne:

- **Boost**: set di librerie fondamentale per coprire le carenze della libreria standard. Nel progetto si ricorre spesso agli *smart pointer* forniti da questa libreria.
- **Eigen**: libreria per lo svolgimento di operazioni di algebra lineare.
- **Point Cloud Library**: libreria per il processing di point cloud. Questa libreria offre numerose funzionalità tra cui segmentazione, visualizzazione, filtraggio, *surface reconstruction* e altro.
- **OpenMP**: libreria per il calcolo parallelo.
- **Pugi**: piccola libreria per la lettura di file `.xml`.
- **GTest**: libreria di unit testing sviluppata da Google.

Il codice è stato compilato con GCC con opzioni di ottimizzazione `-O2` sfruttando CMake per la generazione dei Makefile.

5.2 La simulazione

La simulazione è stata eseguita verificando le collisioni tra un braccio robotico e delle point cloud di test.

I tempi di esecuzione su un moderno computer possono subire fluttuazioni a causa di molti fattori. Per garantire dei risultati validi, i due algoritmi sono stati eseguiti circa 35mila volte ciascuno in condizioni il più possibile differenti.

La simulazione è stata condotta come segue:

1. gli algoritmi sono eseguiti ripetutamente su delle point cloud appartenenti ad un dataset di 20 file in formato `.pcd` di dimensioni comprese tra i 700 e i 20000 kB. Le point cloud sono state fornite gentilmente dall'azienda, e sono in tutto simili a quelle su cui l'algoritmo dovrà essere utilizzato realmente. Le nuvole sono frutto di scansioni laser e rappresentano dei cestì contenenti dei prodotti da estrarre.
2. Per ogni file, il numero di punti della nuvola è fatto variare a step di 10000. Il massimo numero di punti per le point cloud prese in esame è $0,7 \sim 1,8 \cdot 10^6$, per cui da ogni file si ottengono tra le 70 e 180 point cloud di dimensioni crescenti.
3. Per ogni point cloud, il gripper è orientato in 4 diversi modi scelti casualmente ad ogni iterazione.
4. Per ogni orientazione, il gripper viene prima posizionato distante dalla nuvola, poi viene fatto avanzare in un numero fissato di passi fino ad intersecare la point cloud. Per questa simulazione è stato scelto un numero di passi pari a 6.
5. Ad ogni passo, l'algoritmo è eseguito due volte, e poi viene fatta la media.

La simulazione è stata eseguita su una macchina avente le seguenti specifiche:

- CPU: Intel®Core™i5 quad core da 3.20 GhZ
- GPU: NVIDIA GeForce GT 220
- RAM: DDR3 4,0 GB
- OS: Ubuntu 13.04 a 64 bit

Il codice che implementa l'algoritmo octree è *single thread*, mentre l'algoritmo CropBox è eseguito in multithreading.

5.3 Tempo di esecuzione sul numero di punti

come già notato l'algoritmo cropBox ha una complessità $O(n)$ sul numero di punti della point cloud. Più precisamente, dopo il primo loop in cui si selezionano i punti della nuvola nella bounding box del Gripper, si eseguono tanti altri loop quanti sono i componenti del Gripper per verificare quali dei punti nella bounding box del gripper sono effettivamente all'interno dei componenti. Quindi, da un punto di vista asintotico, se m è il numero di componenti, l'algoritmo ha complessità $O(n + m \cdot n)$.

L'algoritmo octree attraversa l'albero e una volta giunto in una foglia, fa un confronto lineare sui punti in essa contenuti.

Stabilire la complessità dell'algoritmo octree è difficile: primo, l'albero non è bilanciato e la sua profondità non dipende dal numero di punti ma dalla risoluzione, come spiegato nel capitolo precedente. La ricerca di collisioni prevede un attraversamento parziale dell'albero, per cui a seconda del numero di foglie da visitare il tempo di esecuzione varia. Per ultimo, il tempo di esecuzione dipende fortemente dalla distribuzione dei punti. Senz'altro si può dire che al caso migliore, cioè in assenza di collisioni, la complessità è $O(1)$, mentre al caso peggiore, cioè quello in cui tutti i punti sono in collisione, la complessità è $O(n)$, ma non è possibile stabilire matematicamente la complessità asintotica media dell'attraversamento. Tuttavia, è lecito supporre euristicamente sia logaritmica. Poiché l'attraversamento viene eseguito m volte (una volta per ogni componente), la complessità asintotica dell'algoritmo Octree è $O(m \log n)$.

I risultati riportati in figura 5.1 mostrano come le ipotesi fatte sugli andamenti lineare e logaritmico dei due algoritmi siano ben verificate.

La figura 5.2 mostra nel dettaglio l'andamento logaritmico dell'algoritmo Octree.

Per un numero di punti attorno al milione, l'uso dell'octree garantisce un boost di performance di circa venti volte.

5.4 Reiezione in assenza di collisioni

Una caratteristica fondamentale per un buon algoritmo di collision detection è garantire un'uscita rapida in assenza di collisioni. Non è desiderabile impiegare un tempo prolungato per la ricerca di collisioni se queste non sono presenti. In particolare, nello scenario per cui è stato progettato quest'algoritmo la collision detection serve a determinare la presa di un oggetto. C'è quindi da aspettarsi che i punti in collisione siano pochi o nessuno, rendendo ancora più rilevante questo aspetto.

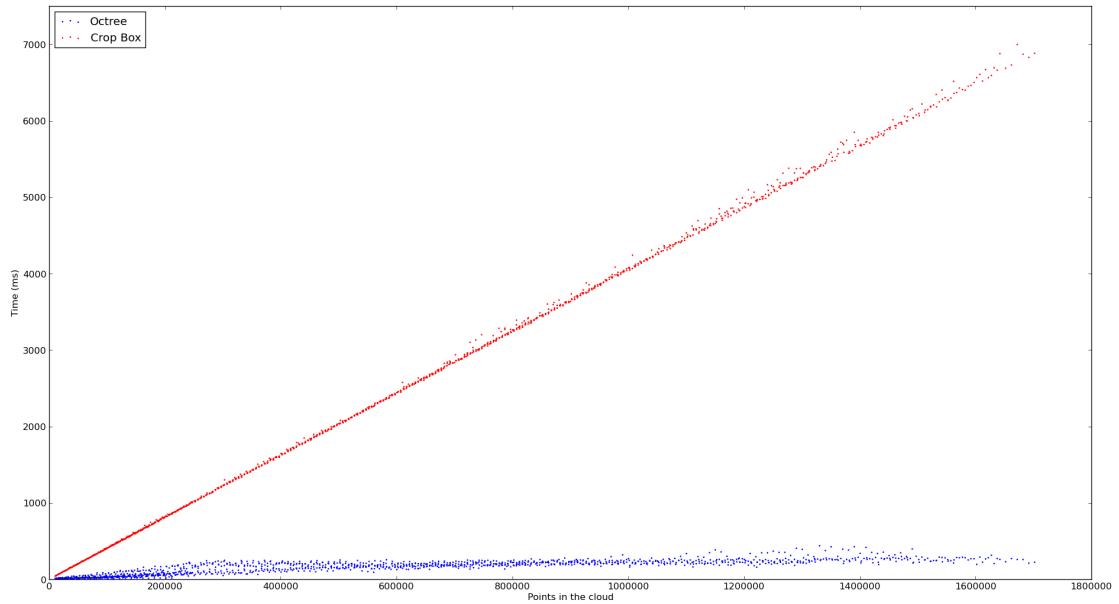


Figura 5.1: Confronto tra i tempi di esecuzione degli algoritmi CropBox e octree in funzione del numero di punti. I punti rossi sono i tempi dell'algoritmo CropBox, quelli di Octree sono in blu.

L'algoritmo CropBox deve iterare almeno una volta su tutti i punti della cloud, a prescindere dalla presenza di collisioni o meno. L'algoritmo Octree invece, in assenza di collisioni, nella maggior parte dei casi non deve iterare neanche su un punto. Nell'attraversamento intersecherà solo nodi interni o foglie vuote (i.e. che non contengono punti) e ritornerà immediatamente. Ad esempio, se il braccio è molto distante dalla nuvola, basterà verificare l'intersezione con la radice. Con una sola verifica di intersezione tra scatole si può escludere la presenza di collisione, a prescindere dal numero di punti contenuti nella point cloud.

Nella figura 5.3 sono riportati i tempi di esecuzione dell'algoritmo CropBox per nuvole da 600mila 800mila punti al variare del numero di punti in collisione.

E' evidente la totale scorrelazione tra le due variabili: l'algoritmo impiega un tempo compreso tra 2.5 e 4 secondi a prescindere dal numero di collisioni. Si osservi invece lo stesso grafico per l'algoritmo Octree, riportato in figura 5.4

In questo caso, si può osservare la dipendenza proporzionale tra tempi e punti in collisione. In assenza di collisioni, l'algoritmo Octree può impiegare anche meno di dieci millisecondi per terminare.

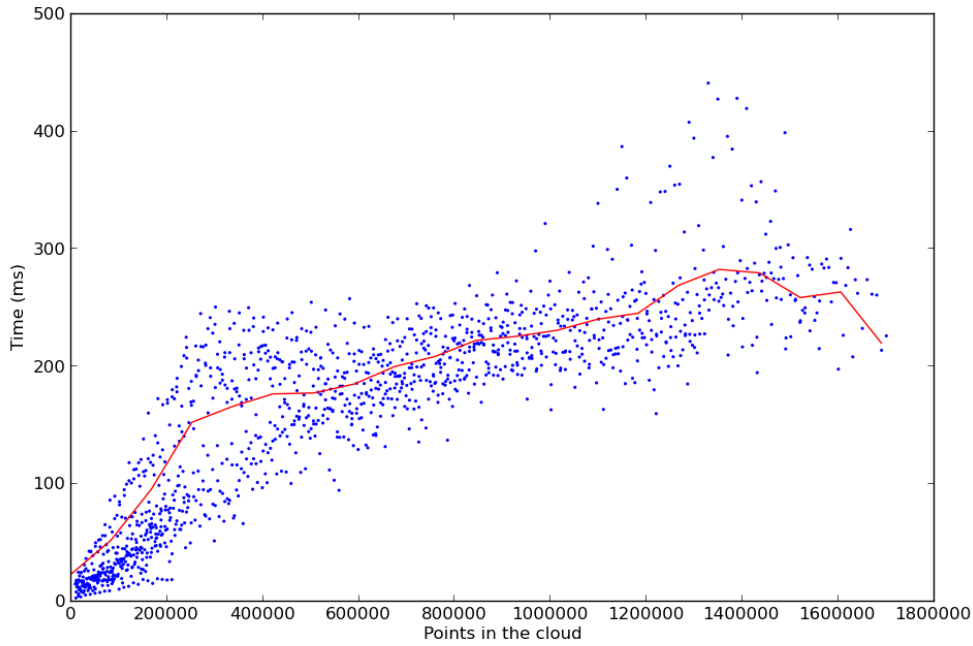


Figura 5.2: Tempi di esecuzione dell'octree sul numero di punti.

5.5 Preprocessing

Il principale svantaggio nell'uso dell'octree è che questo richiede per la creazione un tempo di preprocessing piuttosto lungo. CropBox al contrario non richiede alcun preprocessing. Se il numero di collision detection da fare sulla stessa nuvola è scarso, il tempo guadagnato nella fase di valutazione potrebbe non bilanciare quello speso per la creazione dell'octree. Per questo motivo l'uso dell'octree non è raccomandato in ogni circostanza, ma solo in quelle in cui si prevedono molti confronti sulla stessa nuvola. Nel grafico 5.5 è riportato il tempo di creazione dell'octree in funzione del numero di punti della nuvola, per valori di risoluzione compresi tra 1 e 10.

Si può notare l'andamento lineare, che supera valori di un secondo per point cloud da un milione di punti. Tuttavia, oltre i 500'000 punti l'uso dell'octree diventa vantaggioso anche per una sola ricerca di collisioni, essendo il tempo di preprocessing più il tempo della ricerca inferiore al tempo di ricerca di cropBox.

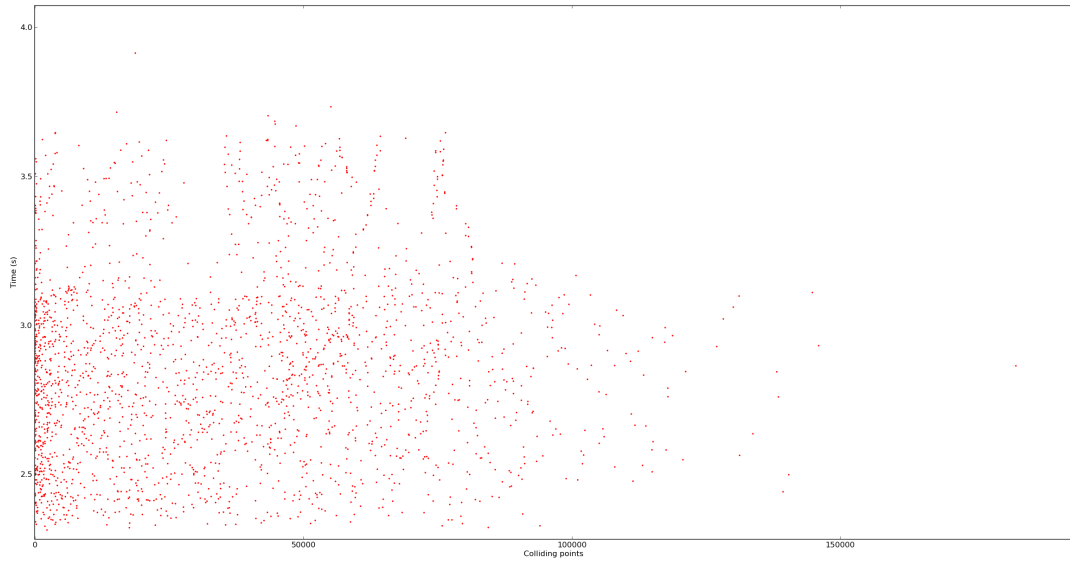


Figura 5.3: Tempi di esecuzione dell'algoritmo CropBox sul numero di punti in collisione per nuvole da 600 800 mila punti.

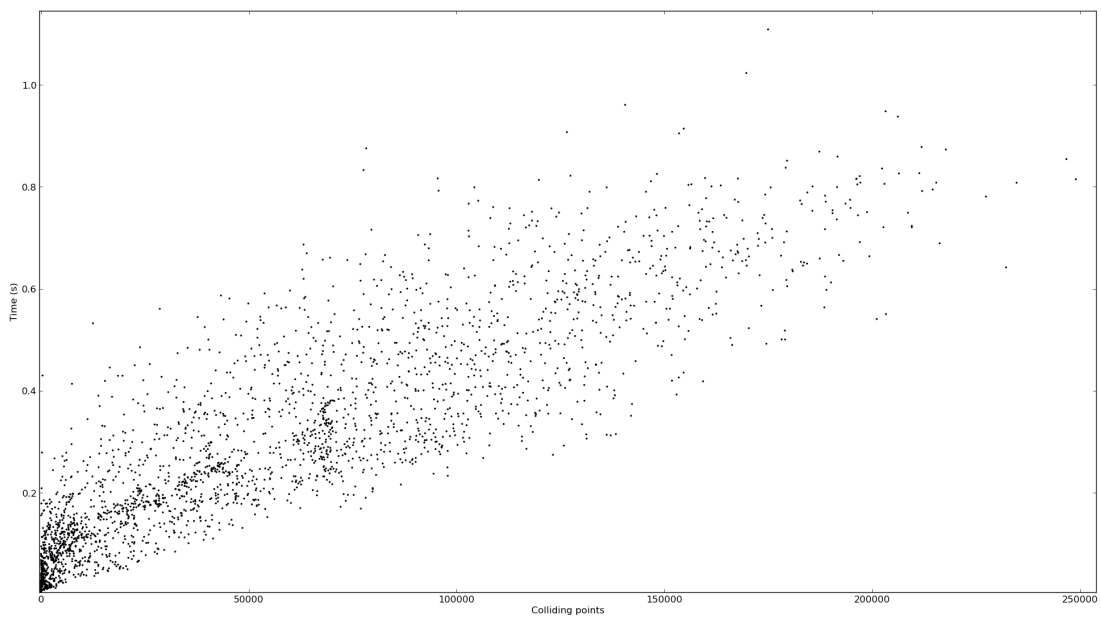


Figura 5.4: Tempi di esecuzione dell'algoritmo Octree sul numero di punti in collisione per nuvole da 600 800 mila punti.

5.6 Occupazione di memoria

Oltre al preprocessing un altro svantaggio dell'octree è che ovviamente richiede occupazione di memoria aggiuntiva. L'implementazione adottata dell'octree utilizza

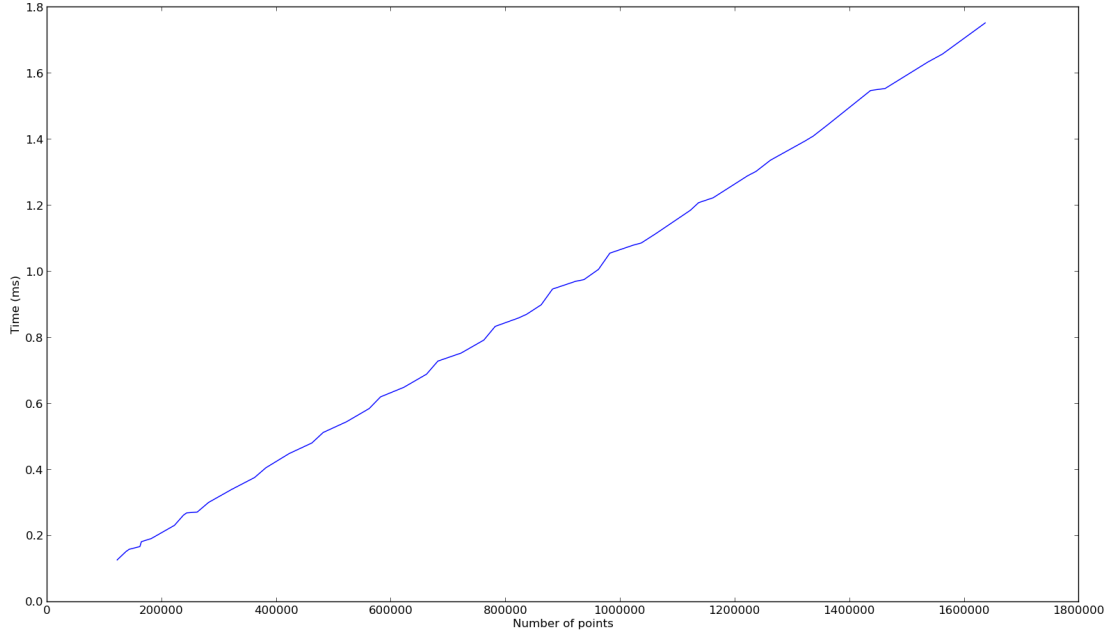


Figura 5.5: Tempo di creazione dell'octree in funzione del numero di punti della cloud

una chiave costituita da tre `unsigned int` per indicizzare i nodi dell'albero. Ogni nodo interno quindi contiene tre interi e i puntatori agli 8 figli. Questo è l'unica occupazione di memoria richiesta per un nodo interno, informazioni ulteriori come la dimensione del voxel possono essere ricavate aritmeticamente dalla chiave. In ogni foglia è salvata una lista di puntatori ai punti della point cloud in essa contenuti. Ciascun punto appartiene esattamente ad una foglia, quindi il numero totale di puntatori salvati nelle foglie è pari al numero di punti della cloud. Supponiamo N_P il numero di punti nella nuvola, n_{int} il numero di nodi interni e n_{leaf} il numero di foglie dell'octree. Su una macchina a 32 bit la memoria occupata dall'octree sarà circa:

$$n_{int} \cdot (3 \cdot 4 + 8 \cdot 4) + n_{leaf} \cdot 3 \cdot 4 + N_P \cdot 4$$

$$44n_{int} + 12n_{leaf} + 4N_P$$

cioè lineare sia sul numero di nodi che sul numero di punti. Il numero di nodi dipende dalla risoluzione e dalla distribuzione dei punti (più la distribuzione dei punti è isotropica, più l'octree è largo e con un numero maggiore di foglie), ma non dal loro numero. Nel grafico 5.6 è riportata una stima dell'occupazione di memoria in funzione della risoluzione. Per risoluzioni ridotte l'occupazione tende ad un asintoto verticale, ma già al valore ottimale per i tempi di collision detection (cioè risoluzione = 11) l'occupazione di memoria è all'accettabile valore di 4 megabyte.

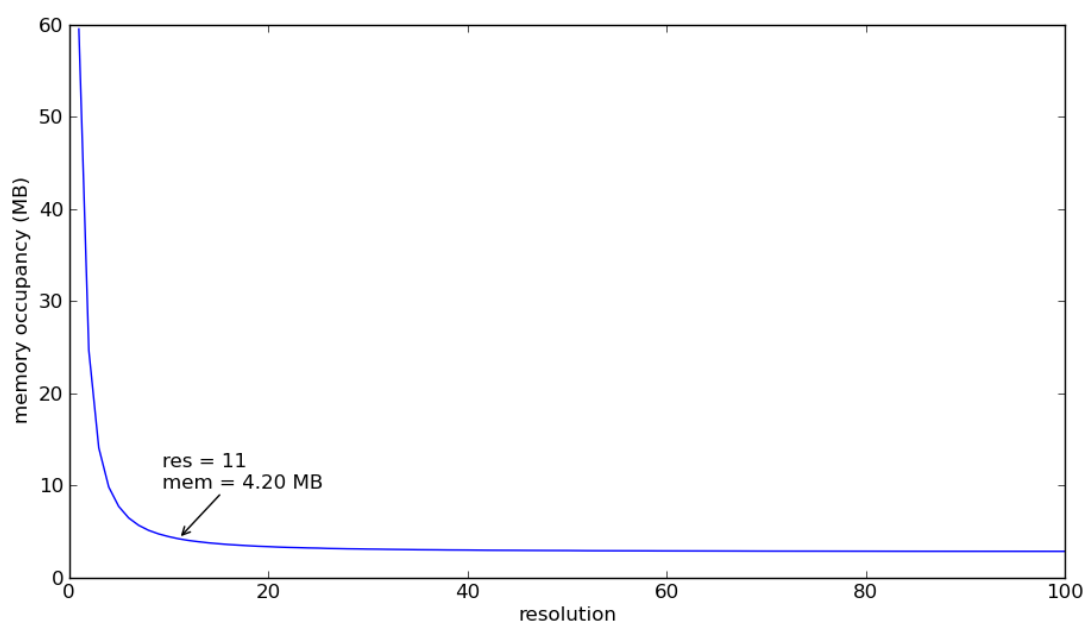


Figura 5.6: Occupazione di memoria al variare della risoluzione

Capitolo 6

Conclusioni

L'algoritmo proposto in questa tesi risolve il problema di collision detection in maniera corretta quanto l'algoritmo precedente.

Il nuovo algoritmo ha complessità asintotica logaritmica, contro quella lineare dell'algoritmo CropBox.

Come mostrato nel capitolo precedente, il miglioramento delle performance è significativo: per nuvole da $\sim 10^6$ punti le performance migliorano di circa venti volte. In assenza di collisioni, il vantaggio è ancora più netto: l'algoritmo octree impiega un tempo indipendente dal numero di punti della cloud, mentre l'algoritmo cropbox itera su tutti i punti in ogni caso. L'uso dell'Octree è in generale preferibile solo in quelle circostanze in cui si debbano eseguire più ricerche di collisioni sulla stessa cloud, tuttavia si è mostrato come già a 500'000 punti l'uso dell'Octree diventi vantaggioso anche per una sola ricerca.

Infine, l'occupazione di memoria aggiuntiva richiesta è accettabile, e dipende solo parzialmente dal numero di punti della point cloud.

Bibliografia

- [1] A. Adamson and M. Alexa. Approximating bounded, non-orientable surfaces from points. In *Proceedings - Shape Modeling International SMI 2004*, pages 243–252, 2004. URL www.scopus.com. Cited By (since 1996): 14.
- [2] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering, Third Edition*. Taylor & Francis, 2011. ISBN 9781439865293. URL <http://books.google.it/books?id=V1k1V9Ra1FoC>.
- [3] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. URL www.scopus.com. Cited By (since 1996): 1152.
- [4] Remondino Fabio. From Point Cloud to Surface: The Modeling and Visualization Problem. In *Proceedings of the International Workshop on Visualization and Animation of Reality-based 3D Models*, volume XXXIV-5/W10, 2003.
- [5] S. Gottschalk, M. C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, pages 171–180, 1996. URL www.scopus.com. Cited By (since 1996): 665.
- [6] J. Klein and G. Zachmann. Point cloud collision detection. *Computer Graphics Forum*, 23(3 SPEC. ISS.):567–576, 2004. URL www.scopus.com. Cited By (since 1996): 16.
- [7] J. Klein and G. Zachmann. Interpolation search for point cloud intersection. In *13th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2005, WSCG'2005 - In Co-operation with EUROGRAPHICS, Full Papers*, pages 163–170, 2005. URL www.scopus.com.
- [8] Jan Klein and Gabriel Zachmann. Proximity graphs for defining surfaces over point clouds. In *Proceedings of the First Eurographics conference on Point-Based Graphics*, SPBG'04, pages 131–138, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association. ISBN 3-905673-09-6.

- doi: 10.2312/SPBG/SPBG04/131-138. URL <http://dx.doi.org/10.2312/SPBG/SPBG04/131-138>.
- [9] Rensselaer Polytechnic Institute. Image Processing Laboratory and D.J.R. Meagher. *Octree Encoding: a New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. 1980. URL <http://books.google.it/books?id=CgRP0AAACAAJ>.
 - [10] Jian Liang, Frederick Park, and Hongkai Zhao. Robust and efficient implicit surface reconstruction for point clouds based on convexified image segmentation. *J. Sci. Comput.*, 54(2-3):577–602, February 2013. ISSN 0885-7474. doi: 10.1007/s10915-012-9674-8. URL <http://dx.doi.org/10.1007/s10915-012-9674-8>.
 - [11] Robert Mencl and Heinrich Müller. Interpolation and approximation of surfaces from three-dimensional scattered data points. In *Proceedings of the Conference on Scientific Visualization, DAGSTUHL '97*, pages 223–, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-7695-0503-1. URL <http://dl.acm.org/citation.cfm?id=789080.789103>.
 - [12] Jia Pan, Sachin Chitta, and Dinesh Manocha. Probabilistic collision detection between noisy point clouds using robust classification. In *International Symposium on Robotics Research*, Flagstaff, Arizona, 08/2011 2011. URL http://www.isrr-2011.org/ISRR-2011//Program_files/Papers/Pan-ISRR-2011.pdf.
 - [13] H. Samet. *Foundations of Multidimensional And Metric Data Structures*. Morgan Kaufmann. Elsevier/Morgan Kaufmann, 2006. ISBN 9780123694461. URL <http://books.google.it/books?id=v0-NRRKHG84C>.
 - [14] Alan Watt. *3D Computer Graphics (3rd Edition)*. Addison-Wesley, 3 edition, 12 1999. ISBN 9780201398557. URL <http://amazon.com/o/ASIN/0201398559/>.
 - [15] G. Zachmann. Minimal hierarchical collision detection. In *ACM Symposium on Virtual Reality Software and Technology, Proceedings, VRST*, pages 121–128, 2002. URL www.scopus.com. Cited By (since 1996): 18.