



UNIVERSITÀ DEGLI STUDI DI PADOVA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Elaborazione di Dati Tridimensionali
Relazione del progetto finale

CNC Simulator

Alberto FRANZIN Nicola GOBBO
1012883 1014195

Docente:
Prof. Emanuele MENEGATTI

AA 2011-12

Indice

1	Descrizione del problema	3
2	Descrizione dei moduli implementati	4
2.1	UML del moduli	4
2.2	Configurator	4
2.3	Miller	6
2.4	Mesher	9
2.5	Visualizer	12
3	Strumenti usati, prerequisiti e istruzioni	14
3.1	Strumenti usati	14
3.2	Prerequisiti	14
3.3	Istruzioni	14
4	Esempi di lavorazione	16
4.1	Modalità testuale	16
4.2	Modalità grafica box	17
4.3	Modalità grafica mesh	17
4.4	Confronto tra box e mesh	17
4.5	Prestazioni grafiche	19
4.6	Carico relativo dei moduli	20
5	Conclusioni	24
	Bibliografia	25

1 Descrizione del problema

Il progetto consiste nel realizzare un software che simuli una macchina a controllo numerico (CNC - Computer Numerical Control) per la fresatura di blocchi di materiale a forma di parallelepipedo rettangolo.

Le specifiche date richiedono che il progetto sia eseguibile sia in ambiente Microsoft Windows (Visual Studio) che Linux ed è stato fornito un diagramma UML con le principali classi da implementare. Il simulatore dovrà accettare in ingresso un file di testo contenente la configurazione degli agenti -ovvero le specifiche della punta della fresa e del blocco- e una lista di *posizioni*: decine di valori che rappresentano la roto-traslazione del blocco di materiale e della fresa nello spazio. Questo file, assieme ad un valore numerico che esprime la precisione della lavorazione, costituisce l'input per il software che dovrà essere in grado di elaborare i movimenti richiesti, asportare le porzioni di blocco corrette, e mostrare a video l'avanzamento della fresatura.

2 Descrizione dei moduli implementati

Sin dalle prime fasi della progettazione il programma è stato suddiviso in moduli, ognuno dei quali è implementato come una libreria, la quale comunica con le altre tramite interfacce fissate. Questa scelta è stata dettata sia dalla necessità di una efficace suddivisione del lavoro tra i programmatori che dalla volontà di rendere intercambiabili i moduli, per sostituirli con versioni più efficienti o debug-oriented.

2.1 UML del moduli

Per meglio comprendere le scelte progettuali fatte, viene presentato in figura 1 il diagramma UML dei moduli e delle classi principali che compongono il software.

2.2 Configurator

`configurator` è il modulo che si occupa di leggere i dati di ingresso e trasformarli in strutture dati comprensibili al resto del programma. Le fonti da cui attinge le informazioni sono la linea di comando, attraverso la classe `CommandLineParser`, e il file di configurazione, attraverso la classe `ConfigFileParser`.

`CommandLineParser` deve tutta la sua flessibilità nell'acquisizione della linea di comando alla libreria `boost program_options` di cui la classe è un semplice wrapper. Altro discorso va fatto per `ConfigFileParser`, classe scritta ad-hoc, in quanto il file da interpretare era di tipo *plain-text* non strutturato. Per garantire una certa flessibilità al contenuto del file, questa classe permette di:

- invertire la posizione delle sezioni `[PRODUCT]` e `[TOOL]`, fermo restando che la sezione `[POINTS]` deve rimanere l'ultima del file;
- gestire correttamente linee vuote o di commento, ovvero righe in cui il primo carattere non di spaziatura è "#";
- gestire parametri opzionali come la presenza o meno della direttiva `COLOR` nella sezione `[TOOL]`.

Individuata la sezione `[POINTS]`, `ConfigFileParser` demanda il compito di interpretare la lista delle posizioni a `CNCMoveIterator`. Questa classe estende `std::istream_iterator`, che a sua volta incarna il pattern `InputIterator`, proprio del C++: può perciò essere usata come un iteratore che, ad ogni dereferenziazione, legge la riga successiva del file e la interpreta come una coppia di roto-traslazioni, ritornando al chiamante queste informazioni con oggetti opportuni.

2.2.1 Sviluppi futuri.

L'implementazione della classe `ConfigFileParser` utilizza solo funzioni definite nello standard C++03 ma, nonostante questo, esistono delle incongruenze nella gestione dei file testuali da parte di Windows e Linux, dovute in primo luogo al diverso marcatore di fine riga dei due sistemi operativi. Queste incongruenze interessano per lo più la gestione dei parametri opzionali e, in ambiente Windows, portano ad una errata interpretazione del

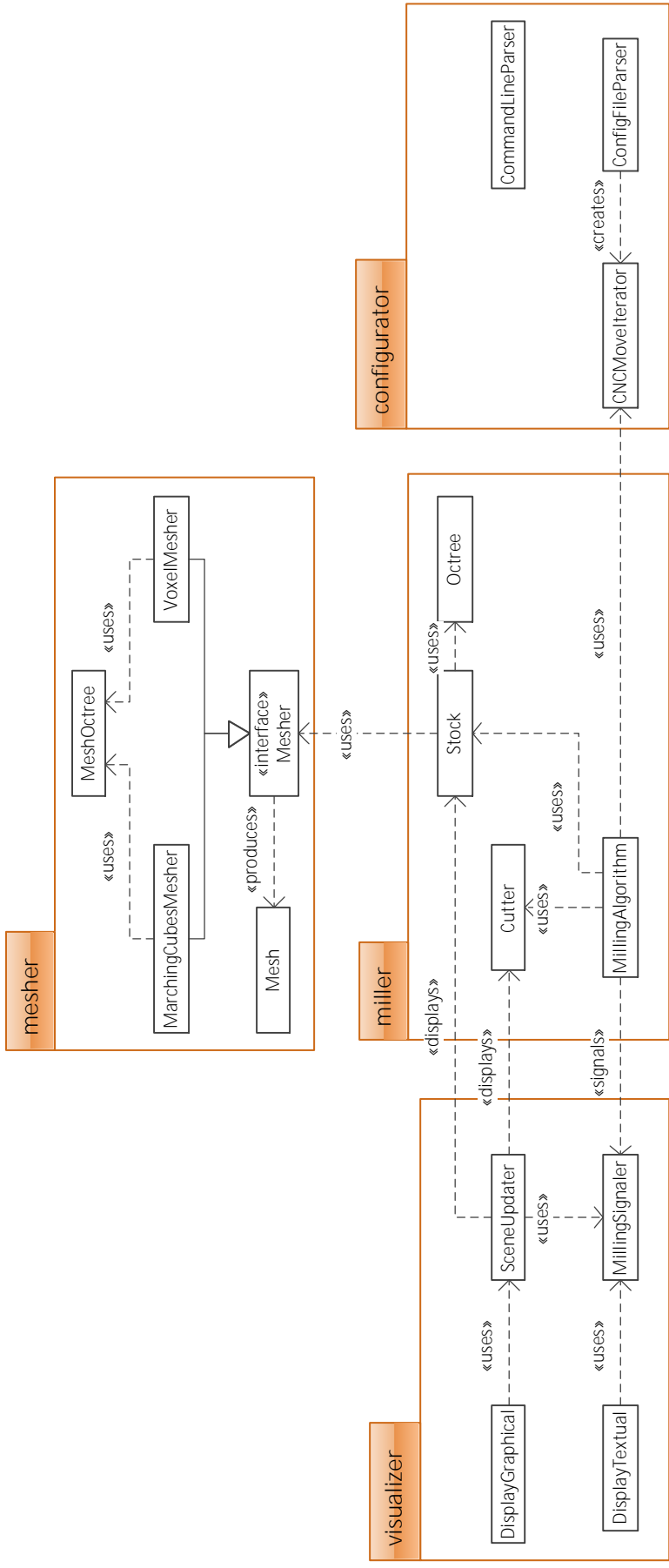


Figura 1: UML dei moduli e delle classi principali del software.

file di configurazione. Per evitare problemi di questo tipo ed aumentare la flessibilità della configurazione, si consiglia di separare in due documenti distinti quanto attualmente contenuto in uno unico: un primo file servirà a descrivere la configurazione della fresatura, codificata in formato XML, mentre il secondo conterrà la lista delle mosse che, dovendo essere letta in modo sequenziale, può mantenere l'attuale formato.

Un'ulteriore funzionalità non ancora implementata completamente è la gestione di cutter “custom”. Le linee guida fornite non descrivevano nel dettaglio questo requisito, quindi l'ipotesi che è stata fatta riguarda da una parte l'apertura del codice rispetto all'aggiunta di nuovi cutter e, dall'altra, la possibilità di gestire mesh fornite dall'utente. Per quanto concerne l'introduzione di un nuovo modello di fresa, gli sforzi si sono rivolti a limitare le modifiche che andrebbero apportate alle classi esistenti; in merito alle mesh arbitrarie, invece, il `ConfigFileParser` dovrà venir modificato per accettare un ulteriore parametro opzionale nella sezione [TOOL], il quale servirà a specificare il file contenente la mesh da visualizzare.

2.3 Miller

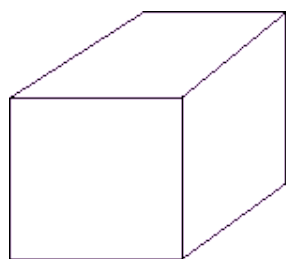
Il *miller* è il componente che, sulla linea di quanto descritto da Yau *et al.* [5], simula la fresatura vera e propria verificando dove e come l'utensile della macchina compenetra il blocco di materiale, determinando la porzione da rimuovere e decidendo se sia o meno necessario attivare il getto d'acqua di pulitura.

Il prodotto da lavorare (classe `Stock`) e l'utensile (classe `Cutter`) sono i principali attori di questo modulo. Per gestire in maniera efficiente il processo di erosione lo stock è stato modellato con una particolare struttura dati chiamata *octree*. Un *octree* è un albero di arietà 8 —in questo caso, non bilanciato— che, come si evince dalla figura 2, permette di segmentare uno spazio tridimensionale in regioni via via più piccole man mano che la aumenta la profondità. Ogni foglia rappresenta quindi un parallelepipedo di volume, detto *voxel*, e su di essa è salvato un valore che identifica quali vertici sono stati asportati dal cutter e quali, invece, sono ancora presenti. Per motivi di performance a ciò si aggiungono le informazioni necessarie a calcolare le coordinate dei vertici stessi ed un collegamento alle strutture adibite alla visualizzazione grafica del blocco, come spiegato nella sezione 2.4.

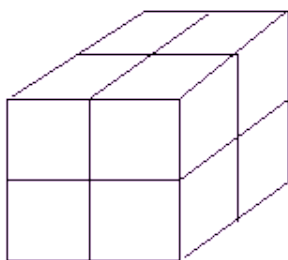
All'interno del modulo di milling il cutter è caratterizzato da due soli parametri: una *funzione di distanza* e una *bounding box*. La funzione di distanza, descritta nell'equazione (1) indica se un punto dello spazio è interno o esterno all'utensile: nel caso sia interno, significa che è stato asportato.

$$distance(P) = \begin{cases} < 0 & \text{se } P \text{ è esterno al cutter} \\ \geq 0 & \text{se } P \text{ è interno al cutter} \end{cases} \quad \text{dove } P \in \mathbb{R}^3 \quad (1)$$

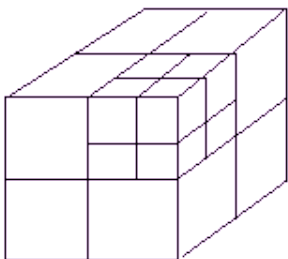
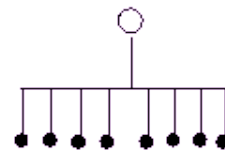
L'ingombro del cutter viene modellato attraverso un parallelepipedo orientato. La scelta di questa forma è dovuta al fatto che anche i voxel che costituiscono lo stock sono dei parallelepipedi: il problema dell'intersezione tra box è molto studiato in letteratura [1], e sono stati individuati algoritmi efficienti per questo tipo di verifica. L'orientamento, invece, serve a poter definire le più piccole dimensioni possibili tali da contenere il cutter: ciò permette di eseguire dei test di intersezione meno rapidi ma anche più accurati.



(root)



(1 level)



(2 levels)

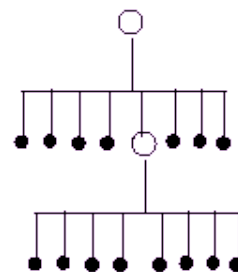


Figura 2: Suddivisione dello spazio tridimensionale tramite albero octree. (fonte immagine: <http://www.forceflow.be/wp-content/uploads/2012/04/octree1.gif>)

2.3.1 Il processo di erosione.

Per ogni “mossa” letta da file, il *miller* converte le due rototraslazioni in una isometria tridimensionale del cutter nei confronti del sistema di riferimento del prodotto. L’algoritmo attraversa quindi l’octree per individuare tutti e soli i voxel che intersecano il volume di riferimento dello stesso cutter: i rami da percorrere sono scelti in base a diverse funzioni di intersezione che diventano via via meno precise, ma più veloci, via via che aumenta la profondità e, di conseguenza, il numero di voxel da analizzare. Giunto ad una foglia dell’albero, l’algoritmo ne enumera i vertici e, per ognuno di essi, invoca la funzione *distance* del cutter: così facendo si marcano i punti interni alla superficie di taglio che, da quel momento in avanti, verranno considerati erosi. Le foglie rimaste prive di vertici vengono quindi eliminate, mentre per le altre, se la profondità massima non è ancora stata raggiunta, l’algoritmo effettua una divisione in otto parti del volume di competenza, aggiungendo un nuovo livello all’albero. Come scelta progettuale si è deciso di non condividere i vertici comuni tra voxel contigui in quanto il concetto di vicinanza spaziale non viene modellato bene dalla struttura octree, soprattutto se sbilanciata. Il costo computazionale necessario a recuperare i voxel “vicini”, infatti, sarebbe stato superiore ai vantaggi portati dalla condivisione dei vertici stessi. Al termine di ogni mossa, il *miller* conteggia la quantità di materiale eroso e non ancora pulito e decide se attivare o meno il getto d’acqua: la scelta viene presa tramite una funzione a doppia soglia, caratterizzata da un “rate di pulitura”, cioè dal volume di detriti che l’acqua riesce a pulire per ogni mossa.

Mostrare a video lo stato dell’erosione comporta uno scambio di informazioni tra *miller* e *mesher* in quanto questi due algoritmi operano in modo indipendente e con diversi tempi di elaborazione. Per gestire in maniera efficiente l’accesso concorrente ai dati, ogniqualvolta una foglia viene cancellata il *miller* la inserisce in una lista opportuna mentre, per le foglie aggiunte o modificate, il percorso da esse alla radice viene evidenziato. Così facendo il processo di *meshing*, dopo aver acquisito il controllo esclusivo dell’octree, potrà ricavare rapidamente tutte e sole le foglie modificate dalla sua ultima visita. Per impedire che il processo di *milling* possa subire starvation dal *mesher*, quest’ultimo viene attivato al più al termine di ogni mossa letta da file e, comunque, non più di 30 volte al secondo: nei casi reali il tempo di attesa forzata del *miller* è ridotto in quanto, un ciclo di rendering impiega molto più tempo della simulazione di una singola posizione e quest’ultima è più lenta dell’attraversamento dell’albero sui percorsi evidenziati¹.

2.3.2 Miglioramenti tentati e sviluppi futuri.

Il processo di *milling* è frutto di più riscritture successive, ognuna delle quali ha sperimentato un modo diverso per rendere più efficiente e veloce l’algoritmo. Il primo approccio scorreva l’albero per livelli sfruttando una coda come struttura dati di appoggio in cui inserire i nodi da analizzare. L’idea di fondo era facilitare la successiva parallelizzazione dell’algoritmo in cui vari thread avrebbero usato la coda per gestire i nodi da controllare, così come mostrato in figura 3.

¹I rapporti tra le durate delle operazioni indicate variano di molto in base alla profondità massima dell’albero e alla “mobilità” dell’utensile, ovvero al numero di voxel modificati ad ogni iterazione.

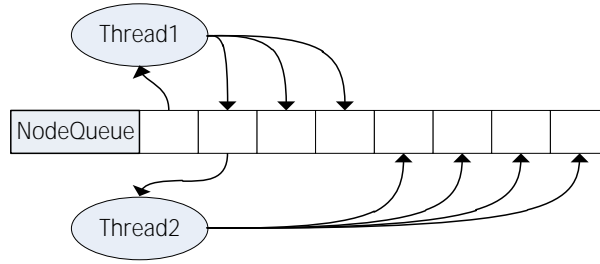


Figura 3: Schematizzazione della prima idea di algoritmo parallelo che risolve il milling.

Affrontare una simile strada avrebbe comportato un uso meno efficiente della memoria: l’inserimento dei nodi in una struttura dati in rapida espansione avrebbe comportato molti più accessi in RAM rispetto ad una semplice visita *depth-first* in quanto la gerarchia di cache degli attuali elaboratori non sarebbe stata sfruttata appieno. La speranza era che il parallelismo riuscisse a sopperire a questo problema strutturale e, anzi, ad essere più rapido nell’esecuzione. Questo purtroppo non è avvenuto e, dai confronti con le implementazioni di altri colleghi, le prestazioni risultavano molto inferiori. Le ipotesi sulle motivazioni di questo “fallimento” sono state molteplici, ma la più plausibile risiede nell’aver scelto un approccio errato alla parallelizzazione, che male si adatta alla particolare struttura del problema: i nodi che devono venir analizzati sono molti ma, il lavoro da compiere su ognuno di essi è poco, per cui l’impianto necessario a gestire i thread introduce un overhead troppo elevato.

L’esperienza così maturata ha permesso di individuare un approccio migliore nella parallelizzazione del problema, analizzando l’octree attraverso il pattern Fork-Join² congiuntamente a uno thread-pool che permetta il *work-stealing*. Così facendo si conservano tutti i vantaggi della visita *depth-first*, senza relegare i thread all’analisi di singoli nodi, e beneficiando sia della possibilità di scrivere codice ricorsivo che di un processing *embarrassingly parallel*.

Purtroppo per carenze di tempo e per la mancanza di librerie che implementano questo paradigma, si è optato per la classica versione ricorsiva. Questa risulta essere la più performante in single-thread ma, dal punto di vista dello spazio occupato, l’implementazione potrebbe essere ulteriormente migliorata, sfruttando i vantaggi offerti dalla ricorsione: attualmente, per esempio, ogni nodo dell’albero contiene un riferimento al padre, retaggio delle vecchie realizzazioni; questo puntatore può essere rimosso mantenendo comunque la possibilità di “risalire” la struttura dati durante il completamento delle chiamate ricorsive.

2.4 Mesher

Il *mesher* è il componente che, a partire dai dati forniti dal processo di fresatura, crea la mesh 3D dell’oggetto lavorato. Le interfacce di comunicazione con il *mill* e il modulo che visualizzerà la scena sono state definite in maniera precisa: ciò permette la scrittura di *mesher* differenti che —scelti all’avvio del software— permettono di rappresentare i voxel non cancellati, piuttosto che una ricostruzione del taglio eseguito. In quest’ultimo caso il mesher si avvale dell’algoritmo Marching Cubes, descritto in sezione 2.4.2.

²<http://www.oracle.com/technetwork/articles/java/fork-join-422606.html>

La mesh è costruita con gli strumenti messi a disposizione dalla libreria OpenSceneGraph: si tratterà quindi di un albero sui cui rami e sulle cui foglie sono contenute tutte le informazioni necessarie alla rappresentazione grafica dell'oggetto lavorato. Per la struttura dell'albero si è scelto nuovamente un octree non bilanciato in cui le foglie, stavolta, contengono una molteplicità di voxel: quando il numero di oggetti contenuti in una foglia raggiunge un valore di soglia, viene aggiunto un nuovo livello all'albero, ripartendo tra i nuovi figli così creati il volume di competenza ed i voxel precedentemente contenuti. Questa scelta permette di ottimizzare l'uso delle risorse grafiche in quanto limita l'estensione dell'albero della scena e diminuisce il numero di mesh da rappresentare, aumentandone la dimensione.

Il processo di *meshing* si articola in due fasi: una prima fase di aggiornamento dell'albero della scena e una seconda fase in cui vengono calcolate le mesh vere e proprie. Nella prima fase il *mesher*, dopo aver acquisito i dati dal *miller*, provvede ad aggiornare la struttura eliminando tutti i voxel non più necessari ed inserendo quelli nuovi o modificati: l'inserimento, però, è subordinato ad un test di "visibilità" il quale provvede a scartare tutti quei volumi che risultano essere strettamente interni alla superficie da visualizzare. È importante notare che le informazioni manipolate in questa prima fase sono legate ai voxel contenuti nell'octree del *miller* e vengono salvate in uno "spazio utente" messo a disposizione dagli oggetti di OpenSceneGraph. Ciò permette di ottenere una complessità computazionale $O(1)$ in cancellazione e aggiornamento e tendente a $O(\log_8(\# \text{ voxel da visualizzare} / \text{max voxel per foglia}))$ per l'inserimento.

L'albero così arricchito di dati aggiuntivi viene quindi ritornato e, al momento della visualizzazione, scatta la seconda fase di *meshing* a cura di una funzione di callback presente su ogni nodo foglia dell'albero. Questa procedura parte dai dati dei volumi di competenza e si incarica di calcolarne la mesh vera e propria, la quale verrà salvata, e mai più ricalcolata fintanto che i relativi voxel non subiscono variazioni. Da quanto detto si evince che la mesh di una foglia —costituita da centinaia di voxel— viene ricostruita da zero anche nel caso uno solo di questi volumi sia stato modificato. A prima vista ciò sembra uno spreco di risorse, ma, alla luce del "principio" di località della lavorazione, si ha che nella maggior parte dei casi le mesh da ricalcolare sono state pesantemente modificate: ciò si traduce in un basso overhead dovuto alle porzioni rimaste inalterate.

2.4.1 Meshing tramite box

Un modo per rappresentare lo stato della lavorazione consiste nel mostrare direttamente i voxel non ancora totalmente erosi dalla fresa: così facendo si esegue un'approssimazione in quanto ogni volume viene considerato totalmente presente quando, nella realtà, delle porzioni —ovvero alcuni vertici— potrebbero esser già stati asportati.

L'algoritmo che crea questo tipo di mesh si avvaleva inizialmente del costruito **Box** messo a disposizione da OpenSceneGraph, ma le prestazioni ottenute erano scadenti; le motivazioni risiedevano sia nel numero di facce "nascoste" che venivano comunque create, sia nel modo in cui la libreria grafica gestisce le **Box** stesse, ovvero come fossero tante piccole mesh e questo, come già visto, non è una situazione ottimale. L'implementazione successiva ha quindi cercato di rimediare a questi due problemi disegnando, per ogni parallelepipedo, tutti e soli i lati visibili, e accorpando quelli afferenti allo stesso nodo foglia in un'unica mesh, di maggiori dimensioni.

2.4.2 L'algoritmo Marching Cubes

Marching Cubes [2] è un algoritmo per estrarre una mesh poligonale a partire da un insieme di voxel, ovvero, data la rappresentazione di un volume, ne ricava la superficie approssimata per mezzo di poligoni —nella fattispecie triangoli. Tali approssimazioni sono date dalla combinazione e rotazione di 15 configurazioni note a priori, definite nell'articolo citato e riportate in figura 4.

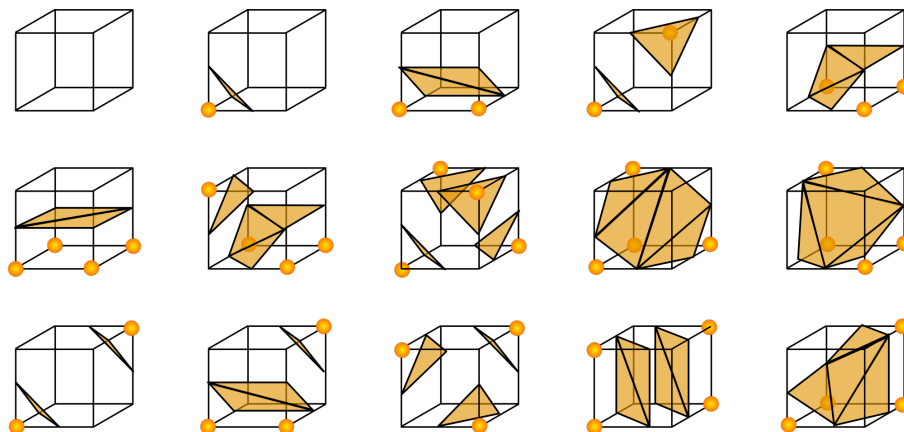


Figura 4: Le 15 configurazioni originali di Marching Cubes.

Marching Cubes lavora su un parallelepipedo alla volta per il quale ha a disposizione un array di 256 ($= 2^{\#}$ di vertici di un voxel) possibili configurazioni, differenziate sulla base dei vertici interni ed esterni alla superficie da ricostruire. Ognuna di queste configurazioni prevede la definizione di uno o più triangoli, i cui vertici risiedono sugli spigoli del parallelepipedo stesso. La posizione di questi punti può essere scelta in base ad una funzione che “pesa” la vicinanza della superficie da rappresentare ai vertici del parallelepipedo ma, nell'implementazione realizzata, viene sempre usato il punto medio dello spigolo: questa scelta permette di risparmiare molta memoria sull'octree del *millier* ed è ulteriormente motivata dal fatto che l'aumento della precisione nella visualizzazione dovrebbe essere ottenuto solo riducendo la dimensione del voxel usato nella simulazione.

2.4.3 Sviluppi futuri.

La figura 5 mostra che garantire performance $O(1)$ nella cancellazione e nell'aggiornamento di dati costa molto in termini di spazio occupato, in quanto necessita di una lista doppiamente concatenata e, per ciascun voxel salvato, di una coppia di *shared pointer*. Un possibile sviluppo futuro consiste nel portare le operazioni di cancellazione e aggiornamento a complessità logaritmica, limitando la prima fase a marcare come “modified” quelle liste di voxel interessate dai cambiamenti; sarà poi la seconda fase che si occuperà di riallineare il contenuto dell'albero con lo stato di fatto. Così facendo le liste possono essere sostituite da array e le coppie di *shared pointer* con singoli *weak pointer*³ che puntano al voxel, risparmiando così memoria RAM.

³I concetti di *shared* e *weak pointer* sono mutuati dalla libreria boost e spiegati nella relativa documentazione, reperibile all'url http://www.boost.org/doc/libs/1_52_0/libs/smart_ptr/smart_ptr.htm

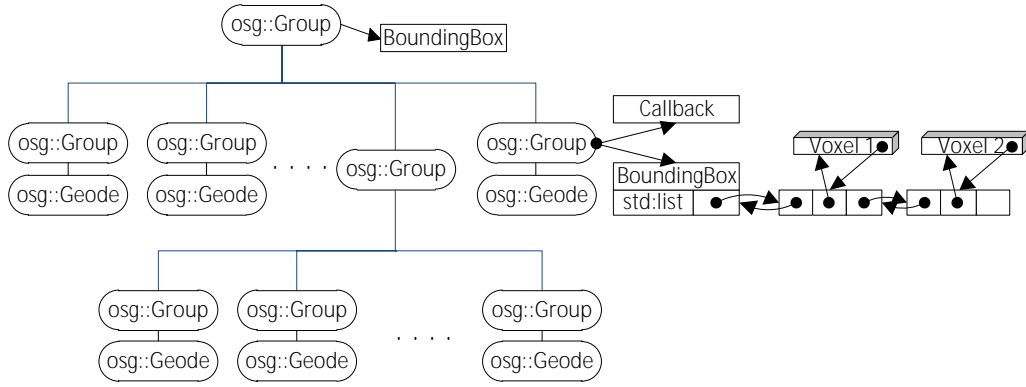


Figura 5: Octree usato dal mesher per visualizzare l’oggetto lavorato.

Una strada che non è stata praticata consiste nell’uso di CUDA[®] per le operazioni di meshing. Date le peculiarità del calcolo parallelo su scheda grafica⁴, la generazione delle mesh potrebbe trarre vantaggio dalla potenza di calcolo messa a disposizione da queste librerie, a patto di prestare attenzione a come i dati vengono scambiati tra la memoria di sistema e quella grafica: questi trasferimenti, infatti, possono diventare velocemente un collo di bottiglia per l’algoritmo.

2.5 Visualizer

Il visualizzatore è il componente che, interfacciandosi con l’utente, sincronizza gli altri moduli software e mostra i progressi dell’elaborazione in corso. Esso può operare in modalità grafica oppure testuale.

Nella modalità testuale il software si presenta come in figura 6, mostrando un riepilogo dei dati di configurazione e, nel caso sia stato lanciato “in pausa”, attende che l’utente lo istruisca sul da farsi. Durante l’esecuzione questa modalità stampa a video una nuova riga

```
./CNCSimulator -v none -s 1 ../positions.txt
*****
***** CNCSimulator *****
**** Nicola Gobbo & Alberto Franzin ****
***** v0.1 *****
Setup info:
  Position file: ../positions.txt
  Cutter: CYLINDER(diameter=30; height=40)
  Stock: STOCK(extent=[260 260 300];maxDepth=9;minBlockSize=[0.507812 0.507812 0.585938])
#move  water      waste #analyzed #purged #updated #pushed #elapsedTime(ms)
1      n          179.655    4240      0      3120    1120      12.066
2      n           0         3120      0      3120      0       1.527
3      n           0         3120      0      3120      0       1.375
4      y          359.311    6240    2269    3971      0       3.875
5      n           0         3971      0      3971      0       1.589
6      n           0         3971      0      3971      0       1.549
7      y          359.311    7901    2269    4822     810     11.551
8      y           0         4822      0      4822      0       2.334
```

Figura 6: CNCSimulator avviato in modalità testuale.

ad ogni “mossa” completata. Le informazioni ivi contenute riguardano: il lavoro svolto dal *miller*, espresso in numero di foglie analizzate, aggiunte o cancellate, la quantità

⁴Un algoritmo che sfrutta in maniera ottimale il pipelining offerto dal calcolo su GPU riduce al minimo il numero di salti condizionati, e usa in modo intensivo i dati forniti in input: questi sono i motivi per cui è stata abbandonata l’idea di sfruttare CUDA[®] anche nel *milling*.

di materiale eroso, l'eventuale necessità di attivare il getto d'acqua e il tempo speso nell'elaborazione della mossa.

Il simulatore lanciato in modalità grafica esibisce una finestra simile a quella in figura 7. Oltre a mostrare lo stato dell'erosione in tempo reale permette all'utente di interagire con l'esecuzione in corso, mettendola in pausa, facendola avanzare di alcune mosse alla volta o disabilitando momentaneamente l'aggiornamento della scena per dedicare tutte le risorse hardware alla fresatura.

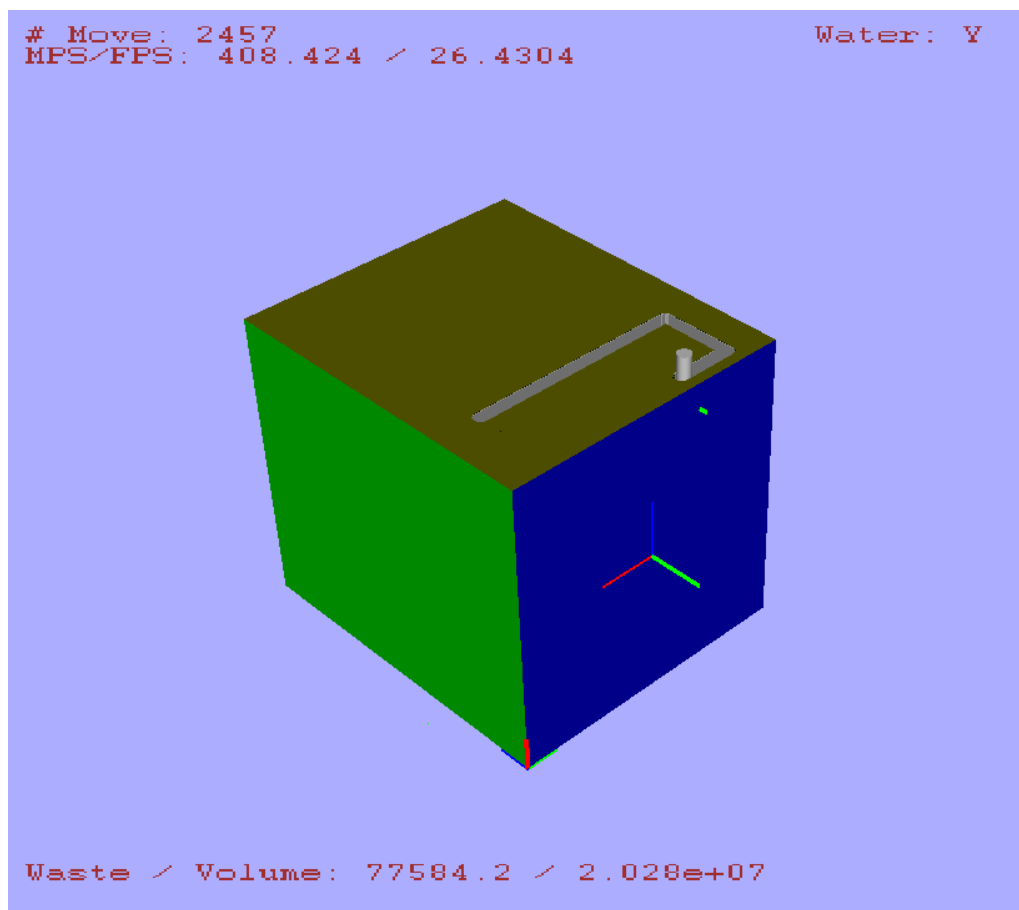


Figura 7: CNC Simulator avviato in modalità grafica.

Come già detto il modulo *visualizer* ha il compito di aggiornare la scena e per fare questo attende che il *miller* completi almeno una mossa: questa attesa dura al più una quantità fissata di tempo, necessaria a garantire un numero di fotogrammi al secondo compreso tra 20 e 30. Nel caso in cui l'erosione venga completata in tempo, l'algoritmo procede ad aggiornare l'albero complessivo della scena richiedendo al cutter e allo stock le rispettive mesh —che verranno quindi riposizionate in base ai parametri della mossa corrente— e ricalcolando le varie quantità mostrate all'utente.

3 Strumenti usati, prerequisiti e istruzioni

3.1 Strumenti usati

Il progetto è stato sviluppato in C++. Per lo sviluppo in ambiente Linux è stato usato Eclipse su Ubuntu 12.04, mentre per l'ambiente Windows è stato usato Visual Studio 2010. Lo strumento usato per la compilazione è CMake (≥ 2.6).

3.2 Prerequisiti

Il progetto è stato sviluppato usando le seguenti librerie:

- Boost (≥ 1.48): è una libreria che fornisce diverse funzioni per molteplici scopi, come ad esempio gestione dei thread e gestione dei parametri.
- Eigen ($\geq 3.1.1$): è una libreria che mette a disposizione funzioni di algebra lineare;
- OpenSceneGraph ($\geq 3.0.0$): è un framework che permette di interfacciarsi alle librerie OpenGL in maniera semplificata ed efficiente [3][4].

3.3 Istruzioni

Il codice è disponibile all'indirizzo <http://code.google.com/p/edt-finalproject-nand/>.
Per compilare il progetto bisogna seguire i seguenti passi:

1. portarsi nella cartella `/path/del/progetto/`;
2. lanciare il comando `cmake flags source/CMakeLists.txt`, dove i `flags` di compilazione possono essere:
 - `-G"Visual Studio 10"` per la compilazione in ambiente Windows;
 - `-G"Unix Makefiles"` per la compilazione in ambiente Linux;
 - `-D CMAKE_BUILD_TYPE=Debug` per compilare in modalità `debug`;
 - `-D CMAKE_BUILD_TYPE=Release` per compilare in modalità `release`;
3. lanciare il comando `make` per compilare.

Per eseguire il progetto, lanciare il comando
`/path/del/progetto/CNCSimulator opzioni file_positions`
dove:

- le opzioni possono essere:
 1. `-s x`, dove `x` è la dimensione minima dei voxel. Minore è `x`, maggiori saranno la precisione della simulazione e il tempo impiegato per completare l'esecuzione;
 2. `-v box|mesh|none`, per specificare il tipo di visualizzazione, rappresentando i voxel come cubi (`box`), approssimando in maniera più precisa il taglio con l'algoritmo MarchingCubes (`mesh`) o in modalità solo testuale (`none`);
 3. `-p` lancia il simulatore in pausa;
 4. `-f x`, dove `x` indica il rate di apertura del getto d'acqua per la rimozione dei detriti in eccesso;

5. `-t x`, dove `x` indica la quantità di materiale da rimuovere prima di attivare il getto d'acqua;
6. `-h`, per visualizzare il menu di help completo.

- `file_positions` è il file contenente i movimenti da riprodurre.

In fase di esecuzione, è possibile modificare il comportamento del simulatore interagendo con esso mediante i seguenti comandi, visibili sul terminale premendo `h`:

- `r` esegue la simulazione alla massima velocità possibile, aggiornando il visualizzatore quando possibile;
- `1` il miller esegue una mossa e poi si pone in pausa;
- `2` il miller esegue 10 mosse e poi si pone in pausa;
- `3` il miller esegue 50 mosse e poi si pone in pausa;
- `p` pausa;
- `t` blocca/attiva gli aggiornamenti delle informazioni visualizzate;
- `k` termina il milling;
- `h` mostra il menu di help;
- `ESC` esce dal simulatore.

4 Esempi di lavorazione

Vengono mostrati ora alcuni esempi di lavorazione, per vedere come i tempi di esecuzione catalogati variano a seconda dei parametri scelti.

Le prove sono state effettuate usando i due file messi a disposizione per i testing, chiamati `positions.txt` e `positions2.txt`. Il primo file contiene oltre 7000 posizioni, e riproduce una fresatura limitata ad una ristretta porzione di blocco, generando quindi un octree poco bilanciato. Il secondo file invece contiene oltre 20000 posizioni e riproduce una fresatura che scava “tutto attorno” al blocco, generando un octree più bilanciato, e risultando quindi più onerosa dal punto di vista computazionale. In figura 1 sono presentati gli esiti delle due lavorazioni. I tempi riportati si riferiscono a test effettuati

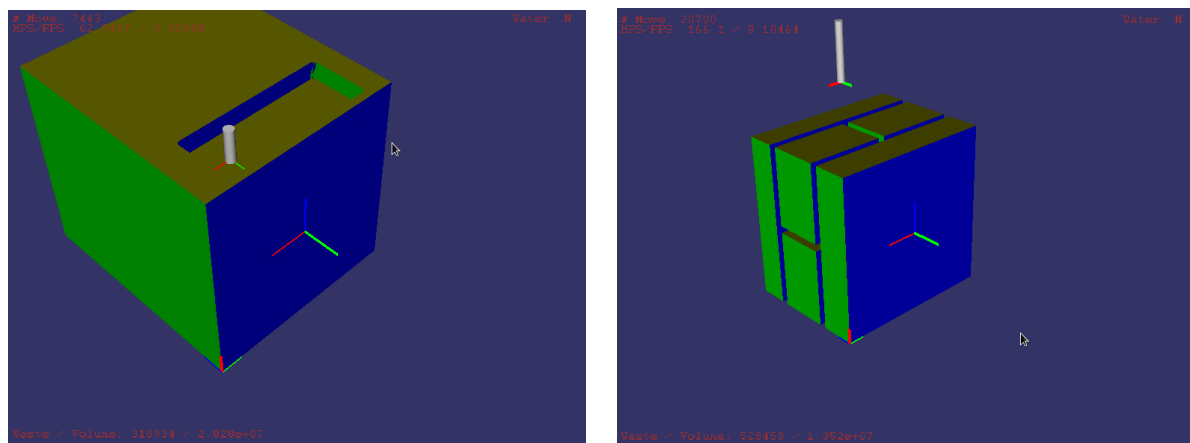


Tabella 1: Risultato delle lavorazioni contenute nei file `positions.txt` e `positions2.txt`.

usando una macchina con Ubuntu Linux 12.04 a 64 bit, con processore Intel i7 a 1.73 GHz e 4GB di RAM, con il simulatore compilato in modalità `release` e lanciato, salvo dove indicato, per completare la lavorazione alla massima velocità possibile. Per il profiling, invece, si è reso necessario compilare in modalità `debug`. Per ciascuna configurazione sono state eseguite varie prove, le quali, una volta mediate, hanno prodotto i risultati qui presentati.

4.1 Modalità testuale

Si mostra innanzitutto (tabella 2) come si comporta il simulatore quando viene eseguito in modalità testuale. Con voxel grandi i tempi di esecuzione sono molto ridotti e molto simili, mentre con voxel più piccoli i tempi crescono compatibilmente con un fattore 8 (l'arietà dell'octree). L'assenza di alcuni valori di memoria in tabella è causata dal fatto che la lavorazione contenuta nel file `positions.txt`, con voxel grandi, termina prima che sia possibile leggere la quantità di RAM occupata. Osservando però i valori rilevati per lo stesso file con voxel più piccoli, e paragonandoli con i valori osservati nella lavorazione riportata nel file `positions2.txt`, si può inferire che l'occupazione si limita a pochi MB. Da questo ragionamento si può dedurre che con voxel grandi l'albero generato è poco profondo e viene analizzato molto velocemente, con un impatto poco significativo

Dimensione Voxel	file positions.txt		file positions2.txt	
	Tempo [s]	Memoria [MB]	Tempo [s]	Memoria [MB]
3	0.517	-	2.446	22.7
2.5	0.543	-	2.558	12.2
2	1.632	17.2	11.478	90.6
1.5	1.607	-	11.613	95.0
1	8.437	70.0	99.785	463.4
0.5	67.691	334.7	973.191	2764.8

Tabella 2: Riepilogo dei test effettuati in modalità testuale.

sul tempo di esecuzione totale. Al diminuire della dimensione dei voxel, però, l’octree diventa più profondo, e la sua scansione occupa una parte via via più consistente del tempo di esecuzione totale.

Per quanto riguarda i tempi rilevati per la lavorazione del file `positions2.txt` con dimensione voxel pari a 0.5, si segnala che la lavorazione è stata rallentata dalle operazioni di swap causate dall’elevata quantità di memoria richiesta.

4.2 Modalità grafica box

La modalità grafica definita “box”, introdotta in sezione 2.4.1, non sfrutta Marching Cubes per la rappresentazione dei voxel, ma rappresenta ciascuno di essi con un parallelepipedo. L’esecuzione caricherà quindi l’interfaccia grafica di OSG e al suo interno verranno visualizzate le varie fasi della lavorazione: il rendering, come detto, sarà basato su cubi di varie dimensioni che rappresentano i voxel dell’octree.

4.3 Modalità grafica mesh

La terza modalità di visualizzazione è quella che utilizza l’algoritmo Marching Cubes (si veda la sezione 2.4.2) per estrarre la mesh tridimensionale dai voxel. Come nella modalità `box`, anche in questo caso viene lanciato il visualizzatore di OSG, ma la lavorazione viene resa in maniera più precisa mediante l’approssimazione calcolata da Marching Cubes. Nella sezione successiva viene fatto un confronto tra le due modalità appena presentate.

4.4 Confronto tra box e mesh

Si pone ora a confronto la visualizzazione della lavorazione con le due modalità di visualizzazione, `box` e `mesh` (le immagini sono zoomate per apprezzare le differenze).

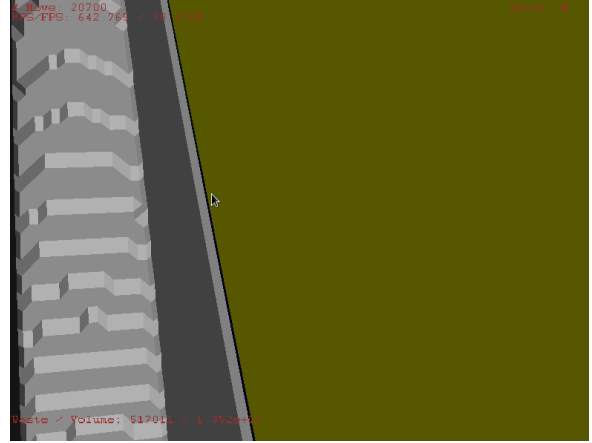
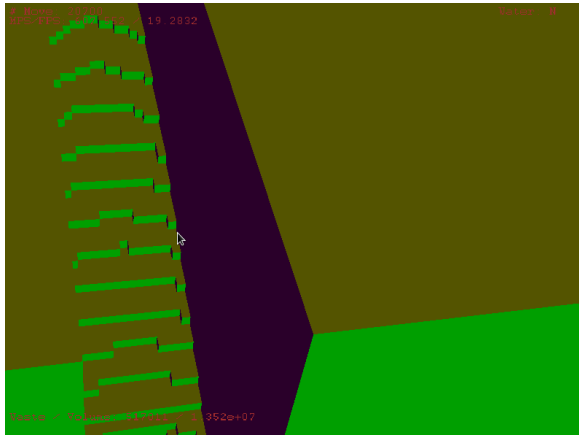


Tabella 3: Confronto tra modalità **box** e modalità **mesh** con dim. voxel pari a 2.

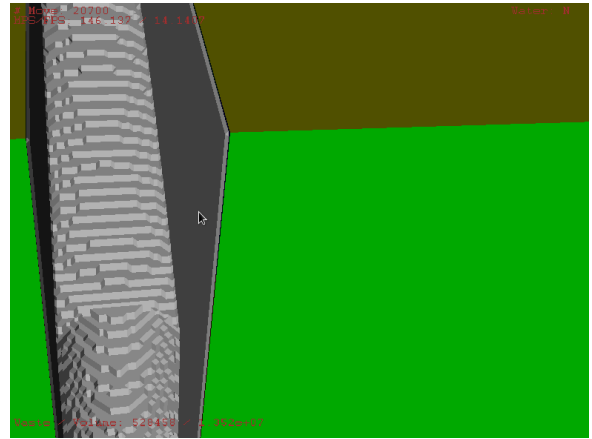
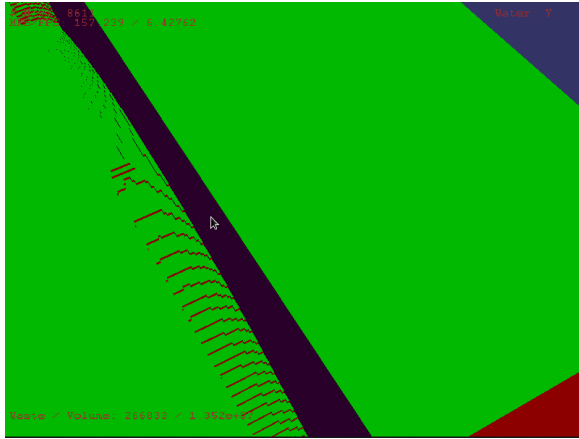


Tabella 4: Confronto tra modalità **box** e modalità **mesh** con dim. voxel pari a 1.

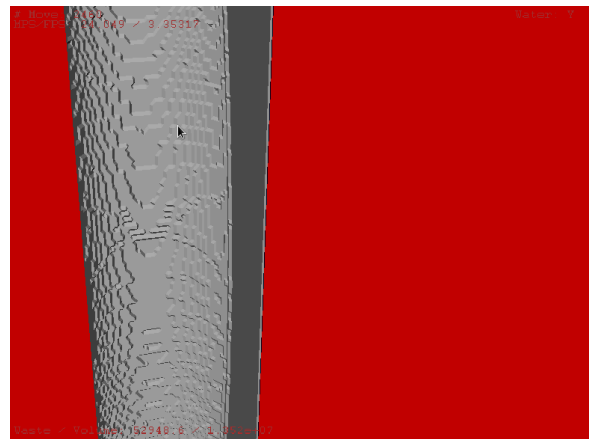
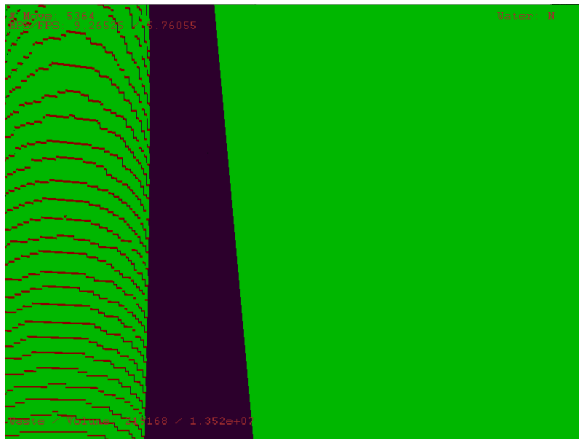


Tabella 5: Confronto tra modalità **box** e modalità **mesh** con dim. voxel pari a 0.5.

In 3 si può osservare la differenza nell'approssimazione della fresatura da parte del cilindro in modalità **box** (a sx) e **mesh** (a dx) con dimensione minima dei voxel pari a 2. Si può notare come, nella modalità **mesh**, l'algoritmo Marching Cubes approssimi meglio la lavorazione, pur mantenendo visibile la struttura “a cubi” del rendering.

In 4 e 5 invece si vede la stessa lavorazione, effettuata con dimensione dei voxel pari a, rispettivamente, 1 e 0.5. Via via che la dimensione dei voxel diminuisce, entrambe le modalità, ovviamente, approssimano in maniera sempre più precisa la fresatura. Tuttavia, mentre la modalità **box** approssima la lavorazione in modo sempre più preciso ma lasciando comunque visibile la “quadrettatura”, l'algoritmo Marching Cubes che lavora alla stessa profondità dell'octree fornisce risultati sempre più precisi e realistici.

Si vede invece dalla tabella 6 come l'implementazione con Marching Cubes possa risultare competitiva rispetto alla mera visualizzazione di cubi in lavorazioni brevi e che comportino la generazione di un octree poco bilanciato e poco profondo. All'aumentare della profondità dell'albero, invece, la semplicità della generazione dei Box di OSG risulta più veloce. Per lavorazioni più complesse, che richiedono un octree più completo, l'approccio **box** è più veloce rispetto alla generazione della **mesh** in tutti i test effettuati.

La tabella 6 permette anche di constatare un'altra proprietà del processo di simulazione, legata alla “località” della simulazione: il file `positions.txt` contiene poco più di un terzo delle posizioni di `positions2.txt` ma i tempi di esecuzione non differiscono dello stesso ordine di grandezza, anzi, diminuendo la dimensione dei voxel sembra che queste quantità tendano a divergere⁵. Questo comportamento è dovuto al fatto che la prima simulazione interessa una porzione piuttosto ristretta dell'intero prodotto e attraversa ripetutamente zone in cui la maggior parte del volume è già stato rimosso, analizzando di conseguenza molti meno nodi dell'albero; nel secondo caso, invece, la fresa si sposta su porzioni sempre nuove del blocco, creando un octree più bilanciato e quindi con un numero maggiore di nodi da verificare – e visualizzare.

Si segnala che, per dimensioni dei voxel sufficientemente grandi, la lavorazione termina in tempi talmente brevi che risulta impossibile misurare un tempo attendibile. Inoltre, i tempi per le lavorazioni con octree più profondo sono da considerarsi al netto del tempo di swap, che può essere più o meno consistente a seconda della quantità di memoria RAM a disposizione.

4.5 Prestazioni grafiche

Vengono riportate ora (tabella 7) le prestazioni grafiche per le due modalità. Tali valori sono presi dai test effettuati sul file `positions.txt`, al termine della lavorazione. Il valore **max** nella colonna **# mosse per aggiornamento** indica che il simulatore lavora alla massima velocità possibile, mentre negli altri casi il visualizzatore viene aggiornato dopo un numero fissato di passi.

⁵Con un piccolo abuso di notazione è possibile scrivere $\frac{\text{tempi positions.txt}}{\text{tempi positions2.txt}} \xrightarrow{\text{voxel size} \rightarrow 0} 0$

Analizzando i risultati ottenuti, si può notare come per profondità dell’octree non troppo elevate, la lavorazione al massimo della velocità possibile penalizzi le prestazioni del visualizzatore, pur mantenendole più che buone. I FPS sono costantemente sopra i 25, ad eccezione della lavorazione alla velocità massima con octree poco profondo: tuttavia, come si è già avuto modo di notare analizzando i tempi di esecuzione, i test effettuati in queste condizioni sono poco significativi, in quanto la lavorazione termina troppo presto per avere valori realistici. A dimostrazione di quanto detto è sufficiente considerare che il numero di movimenti al secondo risulta essere circa il doppio delle mosse realmente eseguite, con il visualizzatore che si apre a lavorazione già conclusa.

Si vede inoltre come l’aggiornamento ogni 10 posizioni dia risultati migliori—in termini di FPS—rispetto agli altri (quando disponibile, ogni 50, ogni aggiornamento). Infine, è interessante notare come la visualizzazione in modalità **mesh** comporti una visualizzazione più rapida rispetto alla modalità **box**, a parità di parametri: questo è probabilmente indice di un minor numero di poligoni da rappresentare nella modalità *mesh*, tale da compensare l’overhead dovuto a Marching Cubes.

Per dimensione dei voxel pari a 0.5 invece si osserva come la lavorazione diventi talmente onerosa da penalizzare in maniera consistente anche l’aggiornamento della scena video: le prestazioni del rendering quando l’octree è molto profondo sono infatti le peggiori.

4.6 Carico relativo dei moduli

È interessante infine analizzare cosa succede *all’interno* di un’esecuzione del simulatore. Per farlo, osserviamo il grafico delle chiamate in figura 8. Questo grafico è stato ottenuto a partire da una lavorazione in modalità **mesh** con dimensione dei voxel pari a 2, con gli strumenti di debug e profiling **valgrind**, **callgrind** e **kcachegrind**.

Il modulo che lavora per la maggior parte del tempo è chiaramente il Miller, che consuma circa i 3/4 di cicli processore impiegati in totale. Il metodo `CyclicRunnable::run()` chiama infatti `MillerRunnable::doCycle()` per 7443 volte, ovvero il numero di iterazioni dell’algoritmo di milling. Questa è chiaramente l’operazione più onerosa eseguita dall’algoritmo, e quella in cui si dovrebbero concentrare eventuali sforzi di ottimizzazione per ottenere un prodotto più performante.

La visualizzazione grafica, che inizia con l’invocazione del metodo `Display::draw()` è invece responsabile del 15% circa del tempo di calcolo.

Dimensione Voxel	file <code>positions.txt</code>		file <code>positions2.txt</code>	
	Tempo con <code>box</code> [s]	Tempo con <code>mesh</code> [s]	Tempo con <code>box</code> [s]	Tempo con <code>mesh</code> [s]
3	-	-	2.381	2.690
2.5	-	-	2.510	2.581
2	2.090	1.988	12.719	16.221
1.5	2.093	1.971	12.581	19.051
1	10.267	9.807	126.827	127.073
0.5	81.466	94.821	1552.065	1684.734

Tabella 6: Confronto delle prestazioni tra modalità **box** e **mesh**.

Dimensione Voxel	# mosse per aggiornamento	box		mesh	
		MPS	FPS	MPS	FPS
3	max	14866.3	1.99735	14301.8	1.92151
3	50	883.677	40.1294	1181.59	47.3080
3	10	309.825	57.1947	294.285	55.3144
3	1	15.7666	30.4380	19.3040	32.8477
2	max	4782.33	3.21263	4591.99	35.1164
2	50	1091.91	32.2764	1232.29	48.8411
2	10	302.094	43.5912	313.589	57.7210
2	1	21.9418	31.2309	17.6811	31.7073
1	max	790.810	5.41869	811.415	27.6904
1	50	562.534	13.7240	584.776	35.4339
1	10	160.020	19.2419	276.251	46.4686
1	1	16.8491	20.1293	21.0637	32.6215
0.5	max	87.5204	2.22240	76.4261	7.04729
0.5	50	61.7659	6.52264	78.0658	8.25444
0.5	10	47.7159	5.91720	62.8882	10.8912
0.5	1	6.01965	7.00878	12.8600	14.8038

Tabella 7: Framerate in modalità **box** e **mesh**.

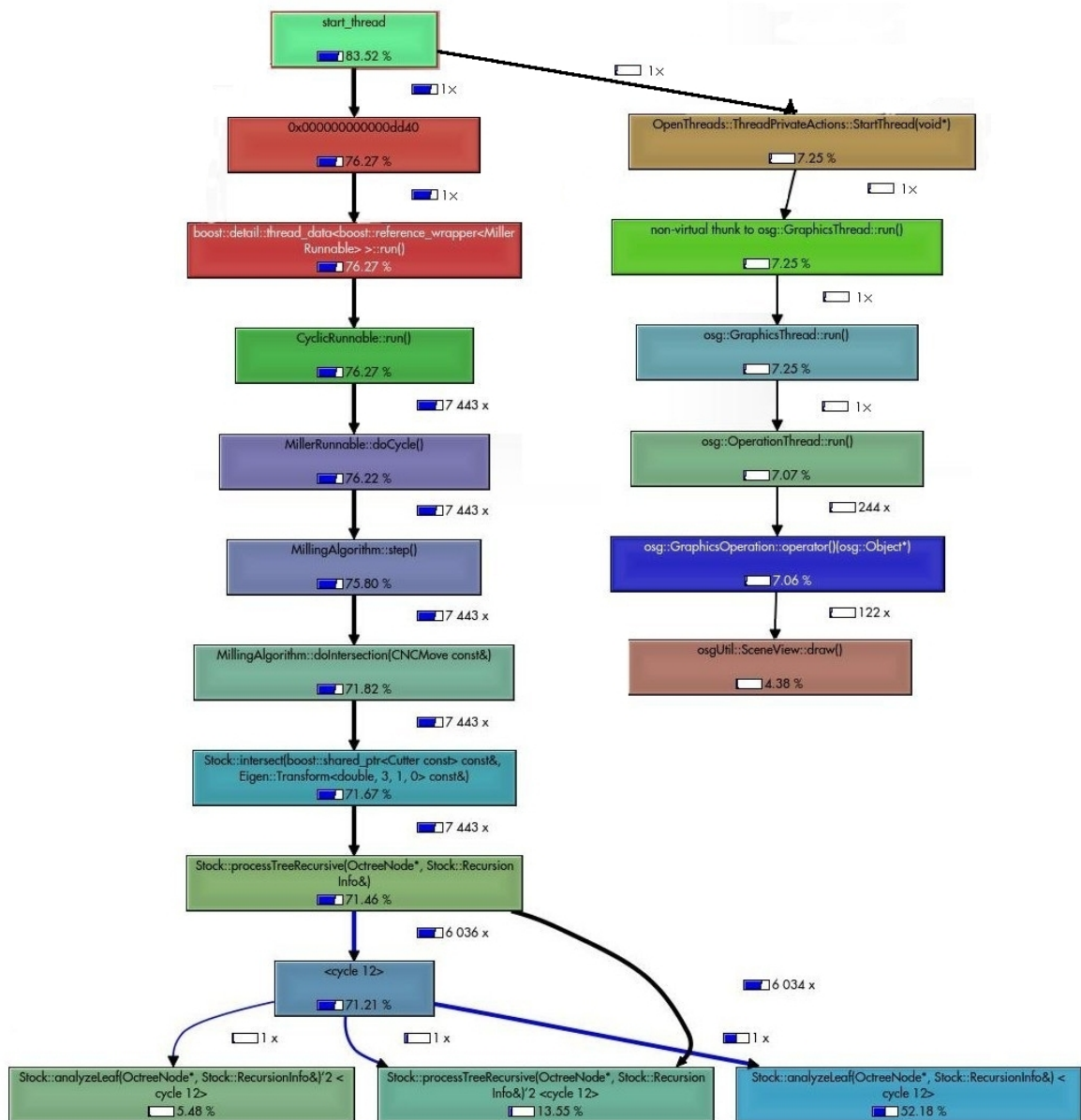


Figura 8: Grafico delle chiamate del simulatore.

Per quanto riguarda i rimanenti moduli, se il configuratore viene eseguito solo preliminarmente alla lavorazione e con compiti di setup, e ci si aspetta quindi un impatto limitato sul tempo totale, si può rilevare invece come in questa prova il Meshing sia assente dal grafo, segno di come anche il suo impatto sia limitato rispetto al Milling, conseguenza dell'efficienza dell'algoritmo Marching Cubes e della bontà della sua implementazione. Si segnala tuttavia come questo accada con dimensione minima dei voxel pari a 2: diminuendo la dimensione dei voxel l'impatto dovuto a Marching Cubes aumenta notevolmente, quindi ci si attende di veder comparire anche i metodi del Mesher tra quelli più impattanti sul tempo. Tuttavia, questi test non sono stati effettuati, in quanto l'esecuzione di `callgrind` rallenta di molto il tempo di esecuzione della simulazione, e una run con voxel più piccoli avrebbe richiesto con ogni probabilità diverse ore prima di giungere al termine.

Come si può facilmente evincere dai risultati appena presentati, i metodi che vengono maggiormente invocati sono quelli che permettono l'interazione tra cutter e octree. Il metodo in assoluto più invocato è infatti `CylinderCutter::getDistance()` che misura la distanza tra il cutter (che in tutti i test effettuati era appunto di forma cilindrica) e il voxel più vicino, per determinare se c'è intersezione e quindi rimozione di materiale. Segue, con circa metà invocazioni, la gestione degli smart pointers di Boost e, con poco più di un terzo di chiamate, il metodo `Stock::IntersectionTester::fastInt()` che esegue i calcoli per determinare se c'è intersezione tra cutter e voxel.

5 Conclusioni

Il programma sviluppato mostra come procede passo dopo passo il lavoro di fresatura, permettendo di regolare precisione della simulazione e del rendering e la velocità di visualizzazione.

Con i file di esempio messi a disposizione, le operazioni vengono portate a termine correttamente e con buone prestazioni, nonostante il software non faccia uso di librerie per il calcolo ad alta prestazione quali CUDA[®].

La lavorazione in modalità testuale è, come ci si può ragionevolmente attendere, più veloce rispetto alle due modalità in cui è impiegata anche la grafica. Per lavorazioni poco impegnative dal punto di vista computazionale la modalità **box** e la modalità **mesh** hanno prestazioni comparabili, mentre, via via che l'elaborazione si fa più processor intensive, l'attivazione del modulo Mesher in modalità **mesh** comporta un significativo incremento sia del tempo di calcolo che della memoria necessaria a contenere l'Octree. Questo è il prezzo da pagare per una visualizzazione più realistica della processo di fresatura, tuttavia, l'uso di Marching Cubes permette di alleggerire il carico della visualizzazione mediante OpenSceneGraph.

Come si può facilmente inferire osservando le tempistiche di esecuzione e osservando quali sono i moduli più time-demanding, i fattori che maggiormente influiscono sul tempo di calcolo sono certamente l'estensione dell'area di lavoro e la profondità dell'Octree: mentre quest'ultima è determinata dalla dimensione dei voxel, la prima è invece fortemente dipendente dalla lavorazione. Se la zona interessata dall'erosione è limitata ad una porzione ristretta di blocco (file **positions.txt**) i tempi e l'occupazione di memoria sono molto inferiori ad una simulazione in cui la fresa si sposta su tutto il prodotto (file **positions2.txt**).

A conclusione di questa relazione si vuole evidenziare che, nonostante il lavoro fatto sia stato molto, è comunque possibile intervenire per migliorare ulteriormente il progetto. In primo luogo è possibile incrementare le prestazioni di Miller e Mesher, rendendo il primo parallelo e "rallentando" il secondo per consentire l'uso di strutture dati più leggere e performanti; ci si potrebbe concentrare poi sul miglioramento della gestione dell'input multiplatforma, modificando il formato dei dati di configurazione, e continuare a lavorare su tutti quegli aspetti già delineati nelle trattazioni specifiche dei vari moduli.

Riferimenti bibliografici

- [1] David Eberly. Dynamic collision detection using oriented bounding boxes. <http://www.geometrictools.com/>.
- [2] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.
- [3] Rui Wang and Xuelei Qian. *OpenSceneGraph 3.0: Beginner's Guide*. Packt Publishing, December 2010.
- [4] Rui Wang and Xuelei Qian. *OpenSceneGraph 3 Cookbook*. Packt Publishing, March 2012.
- [5] Hong T. Yau, Lee S. Tsou, and Yu C. Tong. Adaptive NC simulation for multi-axis solid machining. *Computer-Aided Design & Applications*, 2005.