Università degli studi di Padova
Corso di laurea magistrale in Ingegneria Informatica

Elaborazione di Dati Tridimensionali
Relazione del progetto finale

CNC Simulator

Alberto Franzin 1012883
Nicola Gobbo 1014195

Docente:
Prof. Emanuele Menegatti

AA 2011-12

Indice

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.

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 file di configurazione. Per evitare problemi di questo tipo ed aumentare la flessibilità della configurazione, si consiglia di scomporre il file attuale in due parti: un

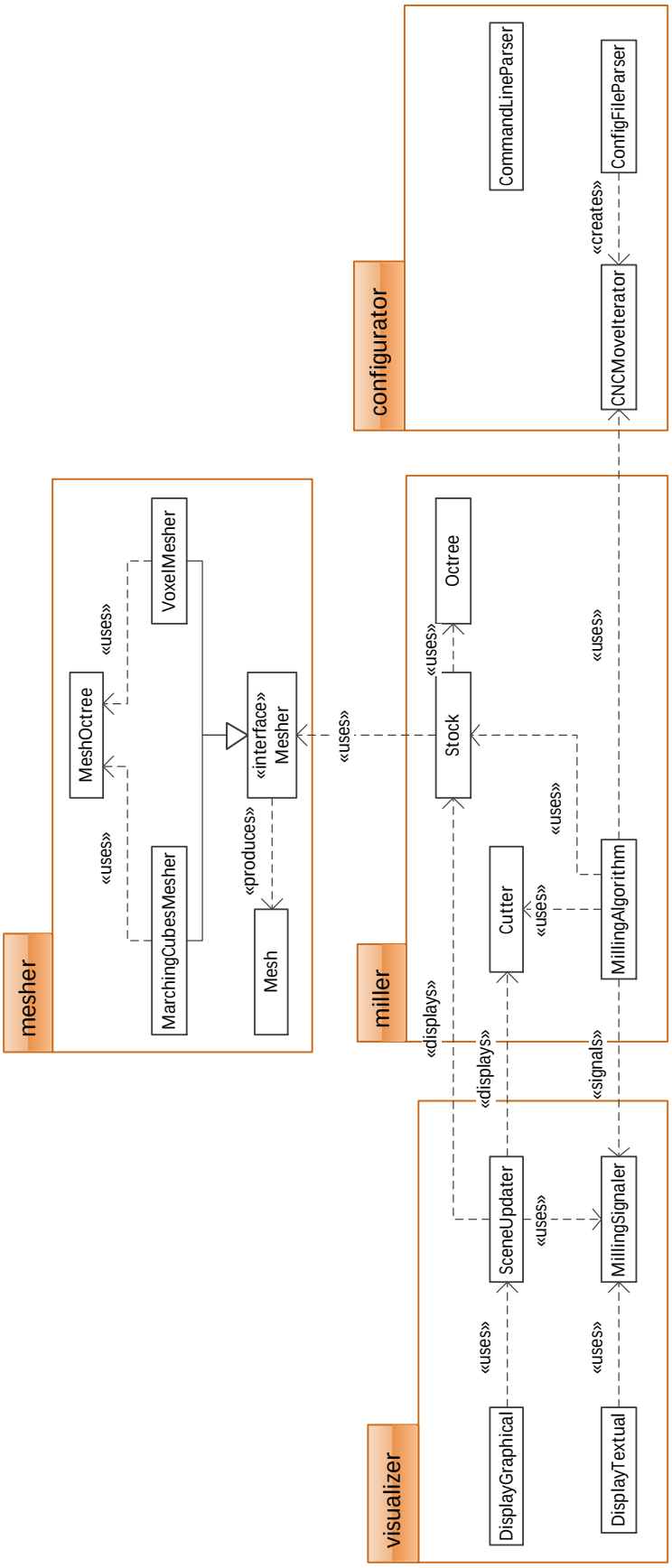

Figura 1: UML dei moduli e delle classi principali del software.

primo file contenente la configurazione della fresatura, codificata in formato XML e un secondo file contenente la lista dei "punti" che, dovendo essere letta in modo sequenziale, può mantenere l'attuale formato.

scrivere del custom cutter da qualche parte, dicendo che basta creare una nuova classe per il calcolo della distanza e tutto va a suo posto

2.3 Miller

Il *miller* è il componente che 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.

Per gestire in maniera efficiente l'intero processo, si è usata come struttura dati di appoggio un octree non bilanciato, ovvero un albero di arietà 8 che, come si evince dalla figura 2, segmenta in modo efficace uno spazio tridimensionale. Ogni foglia dell'octree -rappresentante un parallelepipedo di volume detto voxel- memorizza lo stato di erosione dei propri vertici, a cui si aggiungono, per motivi di performance, le informazioni necessarie a calcolare le coordinate dei vertici stessi ed un collegamento alle strutture dati adibite alla visualizzazione grafica del blocco, come spiegato nella sezione 2.4.

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 di milling attraversa quindi l'octree per individuare tutti e soli i voxel che contengono un punto di contatto tra i due oggetti: i rami da percorrere sono scelti in base a diverse funzioni di intersezione che diventano via via meno precise, ma più veloci, man mano che aumenta la profondità e, di conseguenza, il numero di voxel da analizzare. Quando l'algoritmo giunge ad una foglia dell'albero, esso verifica se alcuni dei vertici associati risultano interni alla superficie di taglio del cutter, marcandoli come erosi. Le foglie rimaste prive di vertici vengono quindi eliminate dall'albero, 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,

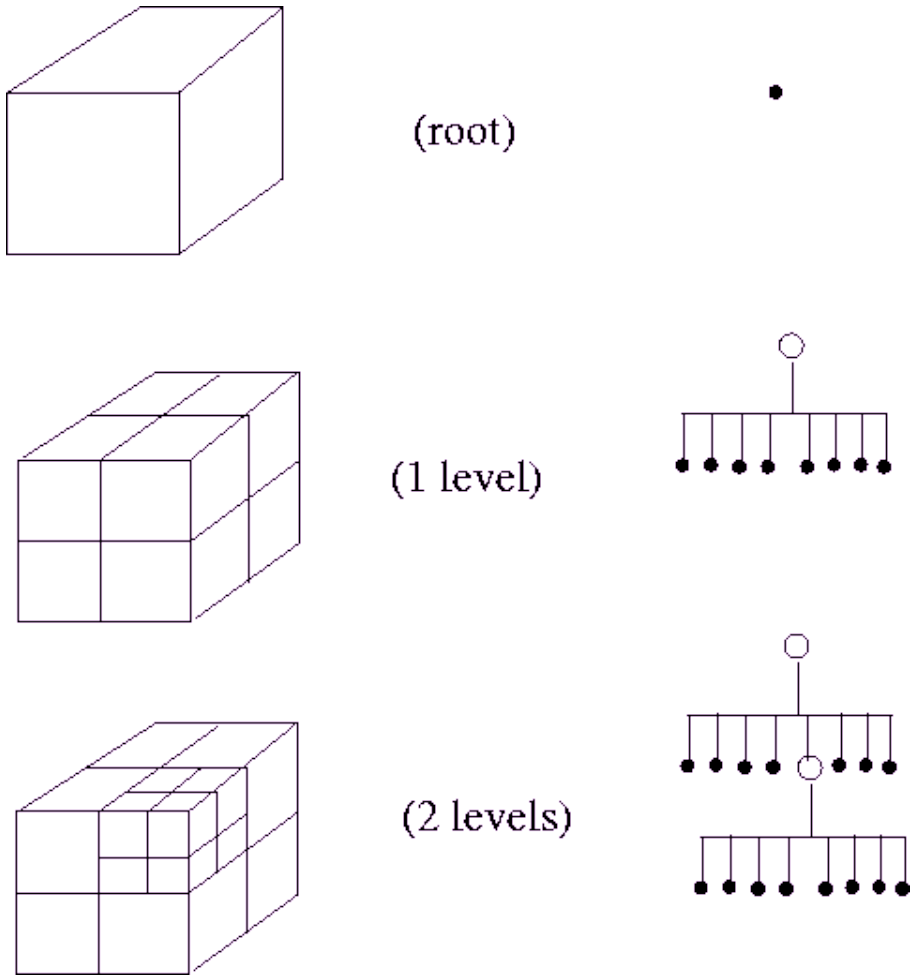

Figura 2: Suddivisione dello spazio tridimensionale tramite albero octree. (fonte immagine: http://www.forceflow.be/wp-content/uploads/2012/04/octree1.gif)

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 evidenziati[1].

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: la versione attuale risulta essere la più performante in single-thread ma, dal punto di vista dello spazio occupato, potrebbe essere ulteriormente migliorata sfruttando in maniera più intelligente la ricorsione[2]. Il vero salto di qualità, però, si avrebbe rendendo multi-threaded il processo di erosione. L'esperienza da noi maturata indica che la strada da seguire per ottenere il massimo delle prestazioni è analizzare l'octree usando il pattern Fork-Join[3] congiuntamente a uno thread-pool che permetta il *work-stealing*: in questo modo si conservano tutti i vantaggi della ricorsione, con l'aggiunta di quelli dovuti a un processing embarassingly parallel, in quanto i thread elaborano dati indipendenti gli uni dagli altri.

2.4 Mesher

Il *mesher* è il componente che, a partire dai dati forniti dal *miller*, crea la mesh 3D dell'oggetto lavorato. Le interfacce verso il *mesher* e verso il modulo che visualizzerà la scena sono definite: questo permette la scrittura di *mesher* differenti che possono venir scelti all'avvio del software, ad esempio per visualizzare i voxel non cancellati, piuttosto che una ricostruzione del taglio eseguito, attraverso l'algoritmo marching cubes.

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 visualizzazione del solo 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 e i voxel ivi contenuti.

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. Una volta acquisiti i dati dal *miller*, nella prima fase il *mesher* provvede ad aggiornare la struttura eliminando tutti i voxel non più necessari ed inserendo quelli nuovi o modificati, a patto che siano visibili (vengono cioè ignorate tutte le informazioni riguardanti volumi strettamente interni alla superficie dell'oggetto lavorato). É importante notare che le informazioni manipolate in questa prima fase sono strettamente

[1] I 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.

[2] Attualmente ogni nodo dell'albero contiene un riferimento al padre, retaggio di vecchie implementazioni: questo puntatore può essere rimosso mantenendo comunque la possibilità di "risalire" la struttura dati, durante il completamento delle chiamate ricorsive.

[3] http://www.oracle.com/technetwork/articles/java/fork-join-422606.html

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 $O(\log_8(\#\text{ voxel da visualizzare/max voxel per foglia}))$ per l'inserimento. L'albero così arricchito di informazioni aggiuntive viene quindi ritornato e, al momento della visualizzazione, scatta la seconda fase di *meshing*: per ciascuna foglia modificata nella fase precedente viene ora creata una mesh vera e propria, contenente tutte le informazioni dei voxel di competenza della foglia stessa. 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.

Sviluppi futuri. La figura 3 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*.

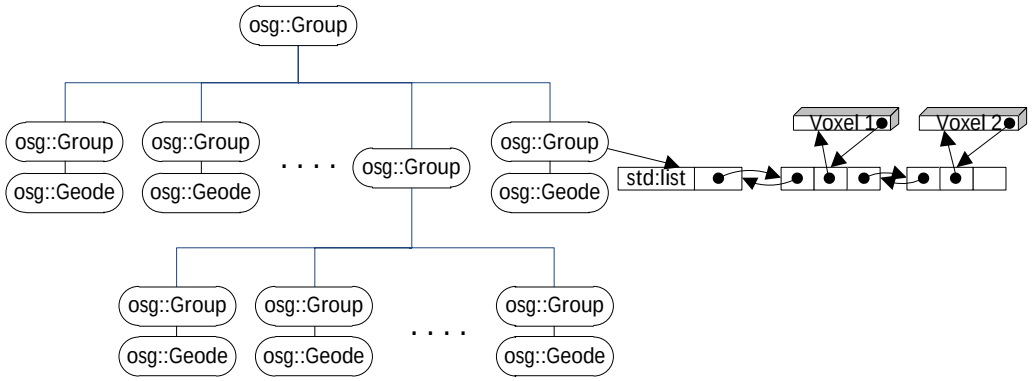

Figura 3: Octree usato dal mesher per visualizzare l'oggetto lavorato.

Un possibile sviluppo futuro consiste nel portare le operazioni di cancellazione e aggiornamento a complessità logaritmica, limitando la prima fase a marcare come "modificate" 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 puntatori con singoli *weak pointer*[4] che puntano al voxel, risparmiando così memoria RAM.

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 4, mostrando un riepilogo dei dati di configurazione e, nel caso sia stato lanciato "in pausa", attende che l'utente lo istruisca sul da farsi.

[4]I 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

```
./CNCSimulator -s 1 -v none -p ../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(y/n)      waste   #analyzed leaves        #purged #updated_data_leaves    #pushed #elapsed time (ms)
```

Figura 4: `CNCSimulator` avviato in modalità testuale.

Durante l'esecuzione la modalità testuale stampa a video una nuova riga 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 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 5. 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.

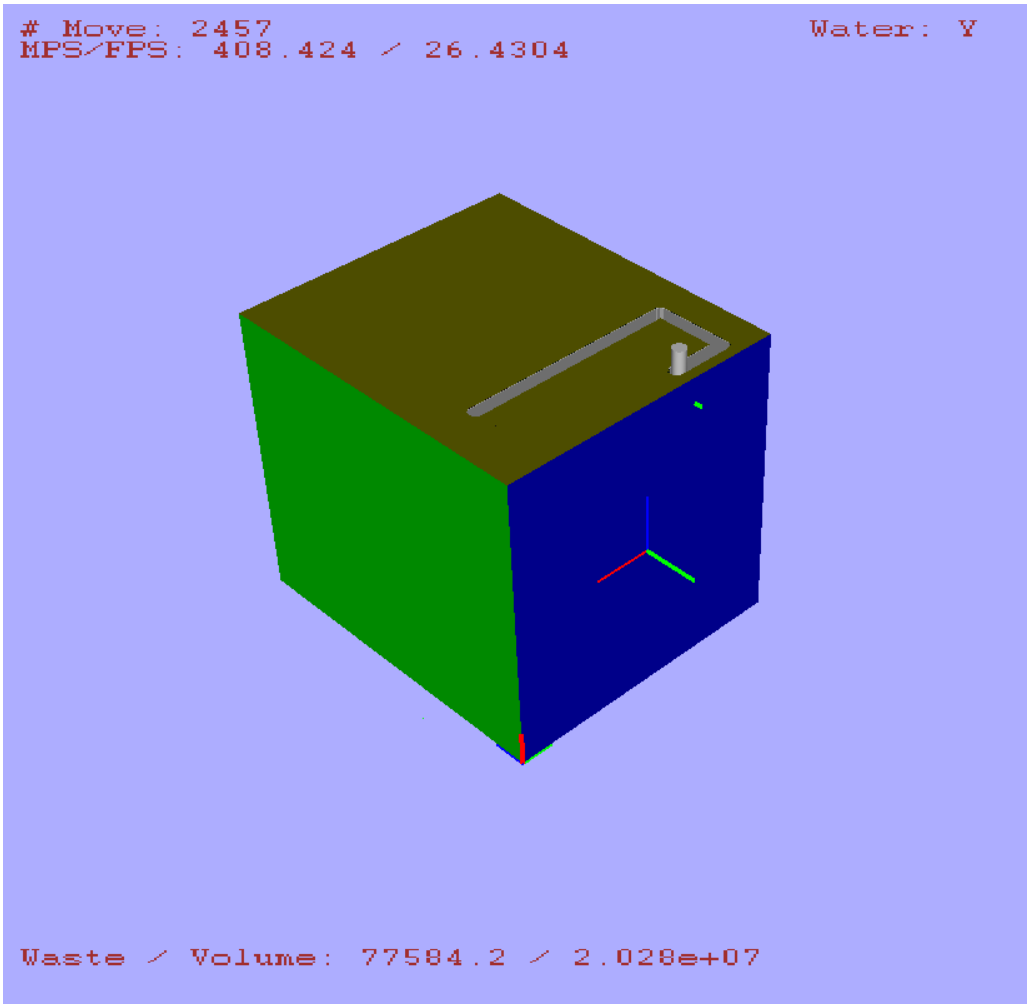

Figura 5: `CNCSimulator` avviato in modalità grafica.

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 abbiamo 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 ($\geq$ 2.6).

3.2 Prerequisiti

Il progetto è stato sviluppato usando le seguenti librerie:

- Boost ($\geq$ 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.

3.3 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.

4 Esempi di lavorazione

Mostriamo 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, uno con oltre 7000 posizioni, e l'altro con oltre 20000. I tempi riportati si riferiscono a test effettuati 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`. Per la profilazione invece si è reso necessario compilare on modalità `debug`. Abbiamo effettuato varie prove per ciascuna configurazione, e abbiamo qui riportato i valori medi.

4.1 Modalità testuale

Vediamo innanzitutto (tabelle 1 e 2) come si comporta il simulatore quando viene eseguito in modalità solo testuale.

Dimensione Voxel	Tempo `[s]`	Memoria `[MiB]`
3	0.517	-[a]
2.5	0.543	-
2	1.632	17.2
1.5	1.607	-
1	8.437	70.0
0.5	67.691	334.7

Tabella 1: Test con file `positions.txt`.

[a] termina troppo presto perché io riesca a vedere quanto occupa... TODO spiegare in maniera potabile sta roba - o trovare una maniera per leggere quanto occupa prima che la riga sparisca dal monitor di sistema

Dimensione Voxel	Tempo `[s]`	Memoria `[MiB]`
3	2.446	22.7
2.5	2.558	12.2
2	11.478	90.6
1.5	11.613	95.0
1	99.785	463.4
0.5	973.191	2764.8[a]

Tabella 2: Test con file `positions2.txt`.

[a] ha swappato, non so se per il monitor di sistema cambi qualcosa.

Vediamo come con voxel grandi i tempi di esecuzione sono molto ridotti e molto simili, con poca occupazione di memoria, mentre con voxel più piccoli i tempi crescono compatibilmente con un fattore 8 (l'arietà dell'Octree). Questo è segno che con voxel grandi l'albero generato è poco profondo e viene analizzato molto velocemente, con un impatto poco significativo sul tempo di esecuzione totale. Al diminuire della dimensione dei voxel, l'Octree è invece più profondo, e la sua scansione occupa una parte sempre più consistente del tempo di esecuzione totale.

4.2 Modalità grafica `box`

La modalità grafica `box` è la modalità che non utilizza l'algoritmo MarchingCubes per il taglio dei voxel, ma usa l'oggetto `Box` di OpenSceneGraph per rappresentare ciascun voxel.

4.3 Modalità grafica `mesh`

La terza modalità di visualizzazione è quella che utilizza l'algoritmo MarchingCubes per estrarre la mesh tridimensionale dai voxel.

4.4 Confronto tra `box` e `mesh`

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

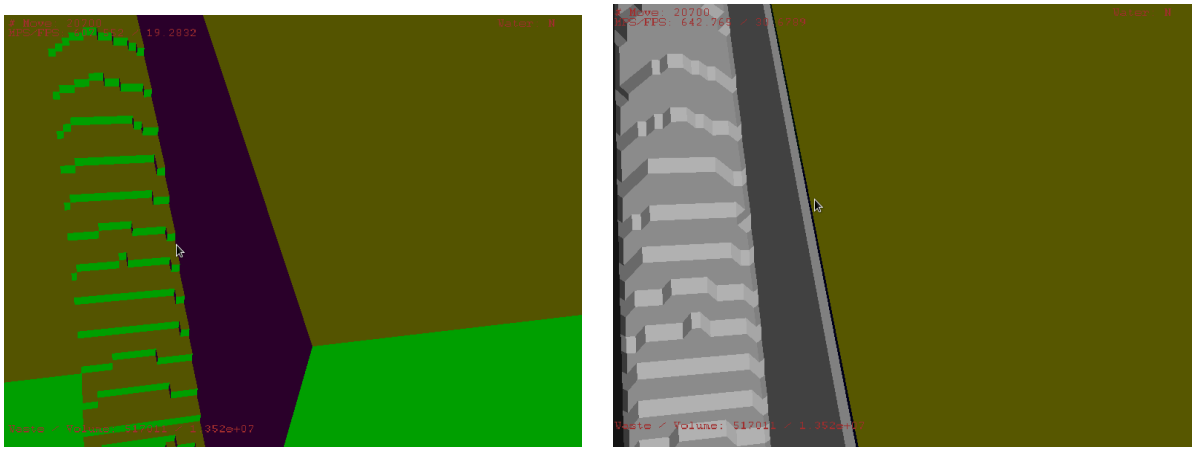

Tabella 3: Confronto tra modalità `box` e modalità `mesh` con dim. voxel pari a 2.

In 3 vediamo 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. Vediamo come, nella modalità `mesh`, l'algoritmo MarchingCubes approssimi meglio la lavorazione, pur mantenendo visibile la struttura "a cubi" del rendering.

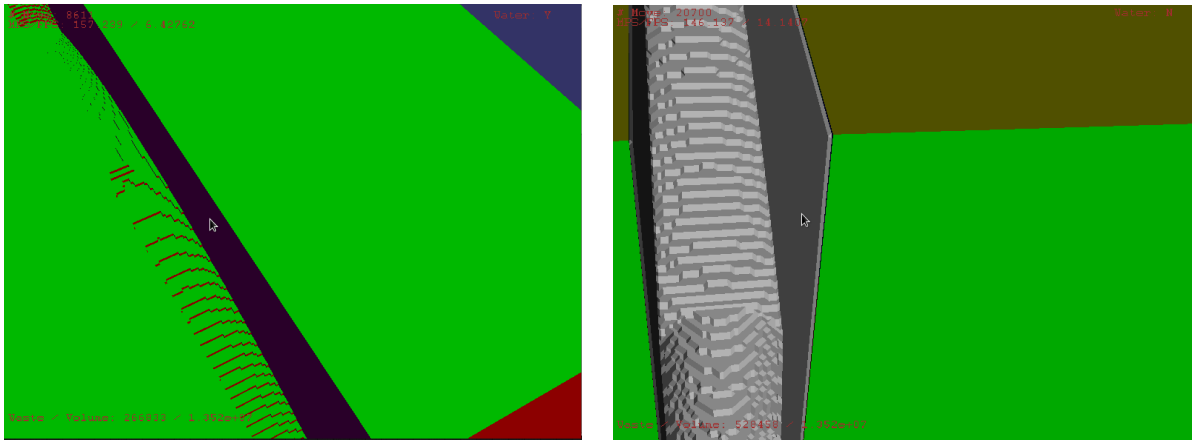

Tabella 4: Confronto tra modalità `box` e modalità `mesh` con dim. voxel pari a 1.

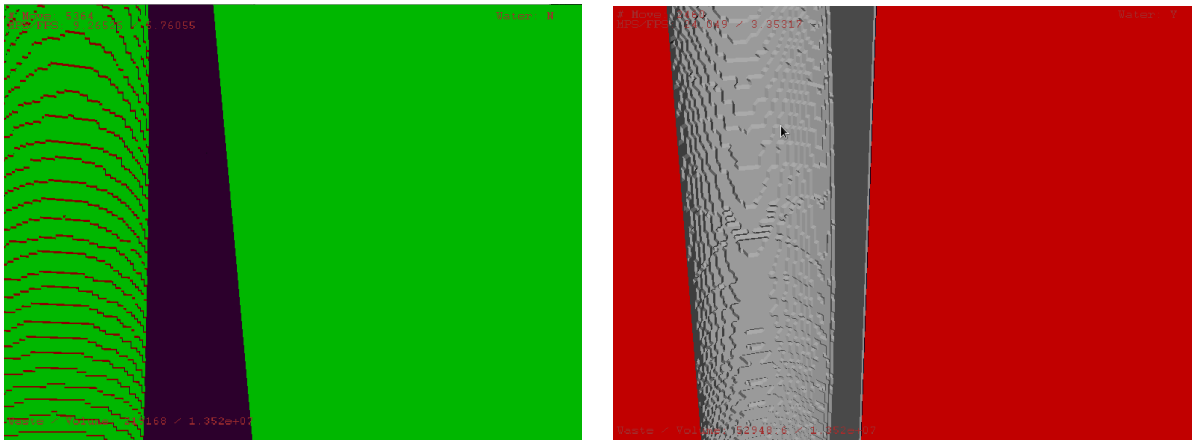

Tabella 5: Confronto tra modalità `box` e modalità `mesh` con dim. voxel pari a 0.5.

In 4 e 5 invece vediamo la stessa lavorazione, effettuata con dimensione dei voxel pari a, rispettivamente, 1 e 0.5. Vediamo come man mano 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 rimane comunque visibile la quadrettatura, l'algoritmo MarchingCubes che lavora alla stessa profondità dell'Octree fornisce risultati sempre più precisi e realistici.

Vediamo, invece, dalle tabelle seguenti (6 ÷ 9) come l'implementazione con MarchingCubes sia più veloce, seppur di poco, in lavorazioni veloci 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.

Dimensione Voxel	Tempo [s]
3	-
2.5	-
2	2.090
1.5	2.093
1	10.267
0.5	81.466

Tabella 6: Test sul file `positions.txt` in modalità `box`.

Dimensione Voxel	Tempo [s]
3	-
2.5	-
2	1.988
1.5	1.971
1	9.807
0.5	94.821

Tabella 7: Test sul file `positions.txt` in modalità `mesh`.

Dimensione Voxel	Tempo [s]
3	2.381
2.5	2.510
2	12.719
1.5	12.581
1	126.827
0.5	arriva...

Tabella 8: Test sul file `positions2.txt` in modalità `box`.

Dimensione Voxel	Tempo [s]
3	2.690
2.5	2.581
2	16.221
1.5	19.051
1	127.073
0.5	1684.734

Tabella 9: Test sul file `positions2.txt` in modalità `mesh`.

5 Conclusioni

Il simulatore mostra come procede passo dopo passo il lavoro di fresatura, permettendo di regolare precisione della lavorazione 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 non sia stato possibile sfruttare CUDA perché nessuno di noi ha a disposizione una scheda grafica Nvidia.