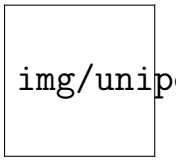

Università degli studi di Padova
Corso di laurea magistrale in Ingegneria Informatica

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

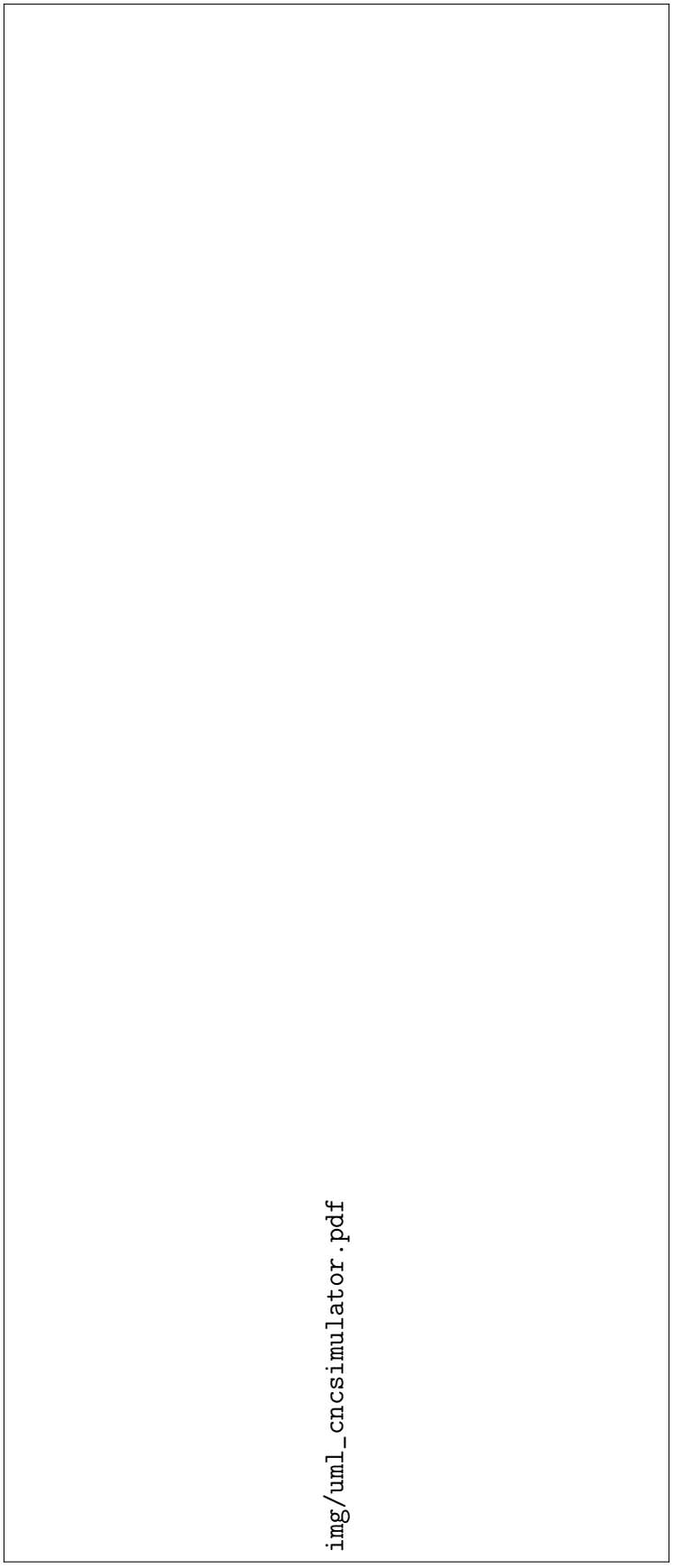

Figura 1: UML dei moduli e delle classi principali del software.

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.

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 e ne determina la porzione da rimuovere.

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.

Figura 2: Suddivisione dello spazio tridimensionale tramite albero octree.

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.

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 evidenziati.

Sviluppi futuri.

2.4 Mesher

Il mesher è il componente che, a partire dalla lavorazione effettuata dal miller, crea la mesh 3D dell'oggetto lavorato, traducendolo quindi in un oggetto tridimensionale da visualizzare.

Quando è pronto a eseguire del lavoro, richiede all'oggetto che rappresenta il prodotto le ultime modifiche effettuate: per tutte le foglie cancellate o aggiornate vengono eseguite le relative operazioni in maniera diretta, sfruttando il puntatore all'oggetto grafico contenuto; quelle nuove, invece, vengono inserite nella scena solo se effettivamente visibili. Per facilitare il processo di visualizzazione, la parte di scena rappresentante il prodotto viene modellata con un octree le cui foglie contengono una molteplicità di voxel da rappresentare: questa scelta permette di ottimizzare l'uso delle risorse grafiche in quanto si diminuisce il numero di mesh, aumentandone la dimensione.

[0]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.

Il mesher esegue l'algoritmo Marching Cubes per creare la mesh a partire dai voxel. Marching Cubes permette inoltre di scartare dalla mesh quei voxel che sicuramente non sono visibili, perché totalmente interni o totalmente esterni (quindi eliminati) alla mesh.

2.5 Visualizer

Il visualizzatore, infine, è il componente che mostra l'elaborazione in corso e il risultato finale. Il programma può operare in due modalità: `mesh` e `box` (in più, è anche disponibile la modalità operativa che esclude la rappresentazione grafica). La prima visualizza la mesh come generata dal Mesher usando Marching Cubes, mentre la seconda sfrutta gli oggetti `Box` di OpenSceneGraph ed è meno accurata, permettendo però di bypassare Marching Cubes.

Il visualizzatore mostra inoltre in tempo reale informazioni relative alla lavorazione, come ad esempio il numero di frame al secondo o l'attivazione o meno del getto d'acqua.

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)
- Eigen ($\geq$ 3.1.1)
- OpenSceneGraph ($\geq$ 3.0.0)

3.3 Istruzioni

Per compilare il progetto bisogna seguire i seguenti passi:

1. portarsi nella cartella `/path/del/progetto/`
2. lanciare il comando `cmake source/CMakeLists.txt`
   - se in Windows, aggiungere il flag `-G"Visual Studio 10"`
   - se in linux, aggiugnere il flag `-G"Unix Makefiles"`
   - se si vuole compilare in modalità `debug`, aggiungere il flag `-D CMAKE_BUILD_TYPE=Debug`
   - se si vuole compilare in modalità `release`, aggiungere il flag `-D CMAKE_BUILD_TYPE=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`, maggiore è la precisione della simulazione
  2. `-v box|mesh|none`, per specificare il tipo di visualizzazione
  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 Esempio di lavorazione

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.