Alma Mater Studiorum · Università di Bologna

facoltà di scienze matematiche, fisiche e naturali

corso di laurea triennale in informatica

Refactoring del pacchetto XCSubd: da GLUI a FLTK

Relatore:
Prof. Giulio Casciola

Tesi di Laurea di:
Davide Parise

Anno Accademico 2009-2010

Indice

Elenco delle figure

Introduzione

Attualmente le *superfici di suddivisione* sono uno strumento che permette di risolvere agevolmente problemi legati al raffinamento, scalabilità e rappresentazione di una mesh. Tali superfici vengono definite come il limite di una successione di raffinamenti su una mesh iniziale di punti.

XCSubd è un tool che nasce per lo studio e la sperimentazione di superfici di suddivisione. Si tratta di uno strumento accademico sviluppato negli anni attraverso diversi lavori di tesi. Il suo scopo è quello di fornire funzionalità utili allo studio di schemi di suddivisione di vario tipo e di fornire un valido sistema di visualizzazione per permettere un' analisi accurata e fedele delle superfici generate.
Dalla sua prima implementazione ad oggi sono state apportate molteplici modifiche. Dall' aggiunta di funzioni che implementano diversi algoritmi di suddivisione ad un motore di rendering per la visualizzazione di superfici.

Queste modifiche hanno di sicuro apportato miglioramenti ma hanno anche reso *XCSubd* instabile e, date le sue dimensioni, riuscirne a capire con esattezza il problema diventa difficile.
Con una prima analisi teorica si intuisce che uno dei possibili problemi della sua instabilità è dovuto all' uso della libreria grafica in quanto non più adeguatamente mantenuta ed inoltre presenta gravi problemi di compatibilità verso il basso.
Una possibile soluzione è quella di sostituire la vecchia interfaccia grafica con

una nuova che abbia le stesse qualità e determinate caratteritiche.

Questo lavoro descrive il processo di refactoring effettuato sul pacchetto XCSubd in modo da sostituire la vecchia libreria grafica con una nuova.
Nella prima parte verranno descritte le librerie grafiche in generale, quindi verrà spiegato che cosa sono e a cosa servono e, in più, verrà effettuato un confronto tra la libreria attuale e quella scelta.
In una seconda parte verrà mostrato il lavoro di refactoring e, in dettaglio, i vari passi di sostituzione.
Infine verranno discussi i risultati ottenuti e verranno proposti consigli sulle possibili modifiche future.

Capitolo 1

Librerie Grafiche

L' *interfaccia grafica utente* (graphical user interface), in breve GUI, comunemente detta interfaccia grafica, è lo strato di un' applicazione software che si occupa del dialogo con l' utente del sistema utilizzando un ambiente grafico e svincolando l' utente dall' obbligo di imparare una serie di comandi.

Una *libreria software* è un insieme di funzioni di uso comune. Lo scopo delle librerie software è quello di fornire una vasta collezione di funzioni di base pronte per l' uso, evitando al programmatore di dover riscrivere ogni volta le stesse funzioni.

Adesso è chiaro che le *librerie grafiche* sono una raccolta di funzioni che facilitano la creazione di un' interfaccia grafica utente.

1.1 GLUI

Il pacchetto *XCSubd* utilizza la libreria grafica *GLUI* nella versione 2.35. Qui di seguito verranno spiegate le caratteristiche e i problemi riscontrati di questa libreria.

GLUT (OpenGL User Toolkit) è una libreria grafica per le applicazioni OpenGl. Provvede a delle semplici interfacce e gestice eventi di finestre, del mouse, della tastiera e di altri dispositivi di input. Oltre a dare la possibilità di creare pop-up menu, GLUT include funzioni per scrivere testo e disegnare primitive grafiche come sfere, tori, ecc.

GLUI, come descritto in [RADE99], è una libreria grafica basata su GLUT e scritta in C++, provvede a controlli come buttons, checkboxes, radio buttons, spinners e listboxes per applicazioni OpenGl.
Il maggior difetto che presenta GLUI è quello di non essere aggiornata, infatti la versione 2.35, pur essendo l' ultima versione, è stata rilasciata nel Luglio del 2006. Inoltre presenta difetti di compatibilità verso il basso.

1.2 FLTK

La nuova libreria dovrà avere determinate caratteristiche oltre che offrire le stesse funzionalità.
In particolare la nuova libreria dovrà essere leggera, per quanto riguarda l'utilizzo della cpu, aggiornata, multipiattaforma e, cosa più importante, dovrà essere compatibile con GLUT ed OpenGL.
È stata scelta la libreria grafica *FLTK*, acronimo di "Fast Light ToolKit", come nuova libreria poiché possiede le caratteristiche appena menzionate.

Rispetto ad altre librerie grafiche come Qt, Gtk, WxWidget, la libreria *FLTK* è migliore in "velocità" e "leggerezza" ([LUCAR9]); di seguito ne viene spiegato il motivo.

FLTK è un toolkit (raccolta di funzioni utili che semplificano la programmazione di GUI) che non include funzionalità aggiuntive rispetto a quelle strettamente necessarie all' implementazione degli elementi d' interfaccia.

Perciò FLTK non include funzioni per gestire ad esempio stringhe, file, thread, ma include soltanto funzioni per la gestione di menu, bottoni, barre di scorrimento ed in generale tutti gli elementi necessari alla realizzazione e gestione di finestre.
Questa caratteristica offre il vantaggio di avere un prodotto finale contenuto nelle dimensioni e, dato che FLTK è pensato per una compilazione statica (cioè l' eseguibile finale contiene anche il binario di tutte le funzioni di libreria), si ottiene un eseguibile stand-alone e cioè che il codice prodotto per funzionare non richiederà l' installazione della libreria su una macchina diversa da quella su cui è stato compilato.

I programmi scritti utilizzando il toolkit FLTK, oltre che ad essere leggeri come descritto prima, sono anche competitivi in velocità rispetto ad altre librerie, inteso come il tempo di avvio di un' applicazione, il refresh delle finestre, il tempo di passaggio delle callback, ecc.

Il toolkit è multipiattaforma e quindi le interfacce prodotte saranno esportabili senza alcuna modifica per i sistemi Unix/Linux, Windows, MacOS, OS2.
Inoltre si integra ottimamente con le librerie OpenGL e GLUT.

Un utile strumento integrato in FLTK è FLUID, acronimo di "Fast Light User Interface Designer", un editor grafico di interfacce scritto utilizzando la libreria FLTK e usato per produrre con semplicità codice FLTK.

1.3 FLTK VS GLUI

Per mettere a confronto le due librerie (FLTK e GLUI) e quindi per essere sicuri di una possibile sostituzione, oltre che a procedere in modo letterale

tramite documentazioni, si sono eseguiti test pratici.

Nella libreria GLUI, come in FLTK, vengono inclusi degli esempi sia per verificarne il corretto funzionamento sia come punto di partenze per l' utilizzatore ed anche per mostrarne le potenzialità.
Il metodo seguito è stato quello di tradurre gli esempi allegati nella distribuzione di GLUI in programmi equivalenti che utilizzano la libreria FLTK.

Da questi test è stato possibile affermare che la traduzione di un programma che utilizza la GLUI come libreria grafica ad uno equivalente che utilizza FLTK è possibile ed in più la traduzione è lineare (non complicata, facile).
Inoltre è emerso che i programmi scritti in GLUI per la visualizzazione di oggetti 3D utilizzando GLUT ed OpenGL si possono tradurre lasciando inalterate le porzioni di codice di GLUT che provvedono alla gestione grafica e alla gestione degli eventi, in questo modo si delega la responsabilità di gestire gli eventi per la visualizzazione di oggetti 3D a GLUT stessa.

La traduzione è avvenuta "in-linea" , cioè:

- Sostituendo una chiamata a una funzione GLUI con una equivalente in FLTK.
- Aggiornando esplicitamente alcune variabili che in GLUI venivano aggiornate automaticamente.
- Cambiando i nomi delle funzioni che gestiscono le callback dell' interfaccia grafica. Questo è dovuto al fatto che FLTK prevede un prototipo di funzione diverso da quello che prevede GLUI.

Capitolo 2

Utilizzo di *FLTK*

In questo capitolo vengono introdotte le nozioni base sull' utilizzo del toolkit FLTK traducendo alcune parti della documentazione allegata ([MEMS09]).

2.1 Il primo programma

Tutti i programmi devono includere il file <FL/FL.H>. Inoltre il programma deve includere un file di intestazione per ciascuna classe FLTK che utilizza. Viene mostrato il semplice programma "Hello, World!" che utilizza FLTK per visualizzare una finestra.

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>

int main(int argc, char **argv) {
.     Fl_Window *window = new Fl_Window (300,180);
.     Fl_Box *box = new Fl_Box (20,40,260,100,"Hello, World!")
.     box-> box ( FL_UP_BOX );
```

```
.       box-> labelsize (36);
.       box-> labelfont ( FL_BOLD + FL_ITALIC );
.       box-> labeltype ( FL_SHADOW_LABEL );
.       window-> end ();
.       window-> show (argc, argv);
.       return Fl::run ();
}
```

Dopo aver inserito i file header necessari, il programma instanzia una classe utile alla creazione di una finestra. Tutti i widget creati dopo diventeranno automaticamente figli di questa finestra.
*Fl_Window *window = new Fl_Window (300,180);*

Dopo viene creato un box con la scritta "Hello, World!". FLTK aggiunge automaticamente il nuovo box nella finestra.
*Fl_Box *box = new Fl_Box (20,40,260,100, "Hello, World!");*

Successivamente viene impostato il tipo di box e la dimensione, il tipo di font, e lo stile del label.
box-> box (FL_UP_BOX);
box-> labelsize (36);
box-> labelfont (FL_BOLD+FL_ITALIC);
box-> labeltype (FL_SHADOW_LABEL);

Diciamo a FLTK che non si vogliono aggiungere altri widget alla finestra.
Window->end();

Infine, mostriamo la finestra e si entra nel loop di FLTK.
Window-> show (argc, argv);
return Fl::run ();

Il programma visualizzarà la finestra mostrata in Figura 2.1. Si può uscire dal programma chiudendo la finestra o premendo il tasto *ESC*.

Figura 2.1: Output del programma "Hello, World!".

2.1.1 Creare i Widget

I widget sono creati utilizzando l' operatore *new* del linguaggio C++. Per la maggior parte dei widget gli argomenti da passare al costruttore sono:
Fl_Widget(x,y,larghezza,altezza,etichetta);

I parametri x ed y determinano dove la finestra o il widget verrà posizionato sullo schermo.
I parametri *altezza* e *larghezza* determinano la dimensione in pixel della finestra o del widget.
etichetta è un puntatore a una stringa di caratteri oppure *NULL* ed imposta il testo del widget.

2.1.2 Ridisegnare dopo la modifica degli attributi

Quasi tutte le coppie di *Set/Get* sono molto veloci. Il metodo *Set* non chiama il metodo *redraw* per ridisegnare la scena, bisogna chiamarlo esplicitamente. Questo riduce enormemente le dimensioni del codice e il tempo di esecuzione. Le uniche eccezioni sono *value()* che chiama *redraw()* e *label()* che chiama *redraw_label()* se necessario.

2.1.3 Visualizzare la finestra

Il metodo *show* mostra il widget o la finestra. Per le finestre si può anche prevedere il passaggio di parametri da riga di comando per permettere agli utenti di personalizzare l' aspetto, la dimensione e la posizione delle finestre.

2.1.4 Il Loop del Main

Tutte le applicazioni FLTK (e la maggior parte delle applicazioni GUI in generale) sono basati su un modello di elaborazione di eventi semplici. Le azioni dell' utente come il movimento del mouse, clic sul pulsante e l' attività della tastiera generano eventi che vengono inviati all' applicazione. L' applicazione può ignorare gli eventi o gestirli.

Le applicazioni FLTK devono periodicamente controllare (*Fl::check()*) o attendere (*Fl::Wait()*) gli eventi oppure usare *Fl::run()* per entrare in un ciclo di gestione standard degli eventi. Chiamare Fl::run() è equivalente al seguente codice:
while(Fl::wait());
Fl::run() non ritorna fino a quando tutte le finestre sotto il controllo di FLTK non vengono chiuse.

2.2 Widget e Attributi comuni

Qui vengono descritti alcuni dei widget e attributi di FLTK.

2.2.1 Pulsanti

FLTK offre molti tipi di pulsanti:

- Fl_Button - Un pulsante standard.
- Fl_Light_Button - Un pulsante con una luce.
- Fl_Repeat_Button - Un pulsante che si ripete.
- Fl_Return_Button - Un pulsante che viene attivato dal tasto Invio.
- Fl_Round_Button - Un pulsante con un cerchio.
- Fl_Check_Button - un pulsante con una casella di controllo.

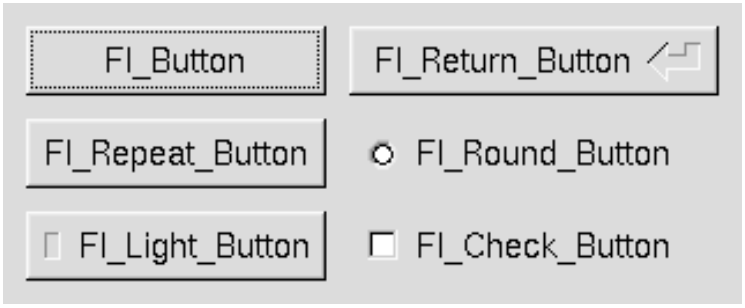

Figura 2.2: Pulsanti di FLTK

Per utilizzare questi pulsanti c' è bisogno del file di intestazione <Fl_xyz_Button.H>. Il costruttore accetta i parametri per definire l' area del rettangolo e, facoltativamente, un' etichetta.
*Fl_Button *button = new Fl_Button (x, y, larghezza, altezza, "etichetta");*
*Fl_Light_Button *lbutton = new Fl_Light_Button(x, y, width, height);*

*Fl_Round_Button *rbutton = new Fl_Round_Button(x, y, width, height, "label");*
A ogni bottone è associato un *tipo* che consente di comportarsi come pulsante, interruttore o scelta esclusiva.
Il metodo *value()* restituisce, per i pulsanti interruttore e quelli che consentono una scelta, lo stato attuale.

2.2.2 Testo

FLTK fornisce widget per la visualizzazione e ricezione del testo.

- Fl_Input - Una sola riga di testo come input.
- Fl_Output - Una sola riga di testo come ouput.
- Fl_Multiline_Input - più righe di testo come input.
- Fl_Multiline_Output - Una sola riga di testo come output.
- Fl_Text_Display - Visualizza più righr di testo.
- Fl_Text_Editor - Modifica di più righe di testo.
- Fl_Help_View - Visualizza testo HTML.

Il metodo *value()* viene utilizzato per impostare o ottenere il testo che viene visualizzato. Quando la stringa viene impostata il widget se la copia in una prorpia struttura interna.

2.2.3 Valutatori

A differenza dei widget di testo, i valutatori utilizzano numeri al posto delle stringhe. FLTK fornisce i seguenti valutatori.

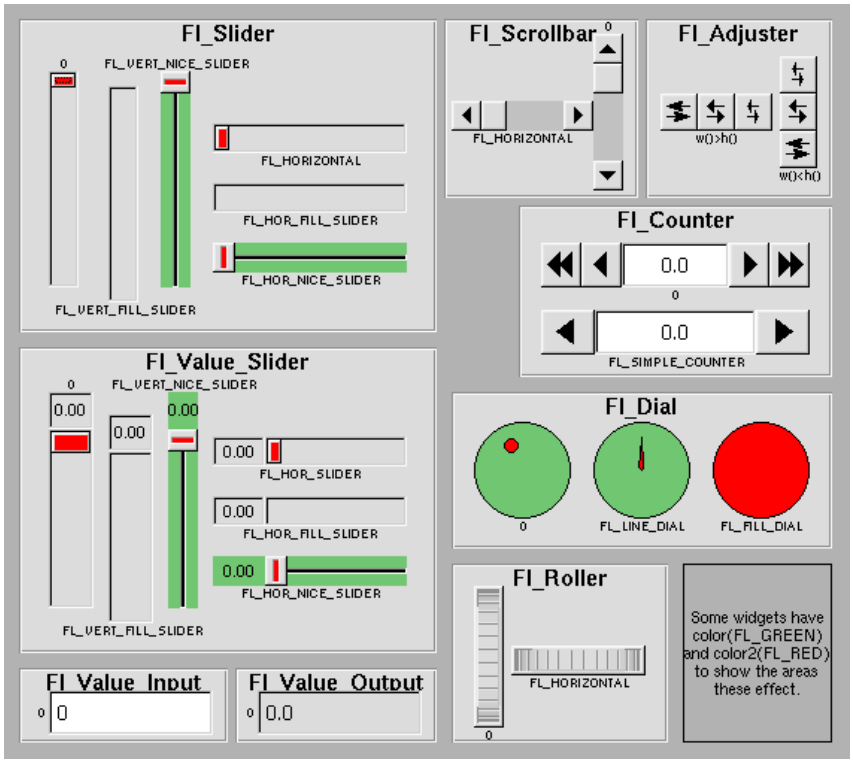

Figura 2.3: Valutatori presenti in FLTK

- Fl_Counter - Un widget con i tasti freccia che indica il valore corrente.
- Fl_Dial - Un cerchio.
- Fl_Roller - Un widget come SGI.
- Fl_Scrollbar - Un widget scrollbar standard.
- Fl_Slider - Una barra di scorrimento con una manopola.
- Fl_Value_Slider - Un cursore che indica il valore corrente.

Il metodo *value()* ottiene e imposta il valore corrente. I metodi *minimum()* e *maximum()* impostano l' intervallo dei valori.

2.2.4 Gruppi

La classe *Fl_Group* è utilizzata come un contenitore di widget. Oltre a raggruppare pulsanti per la scelta escusiva i gruppi vengono utilizzati per incapsulare finestre, schede e finestre di scorrimento.

2.2.5 Tipi di Box

I tipi di box disponibili in FLTK vengono mostrati nella Figura 2.4.

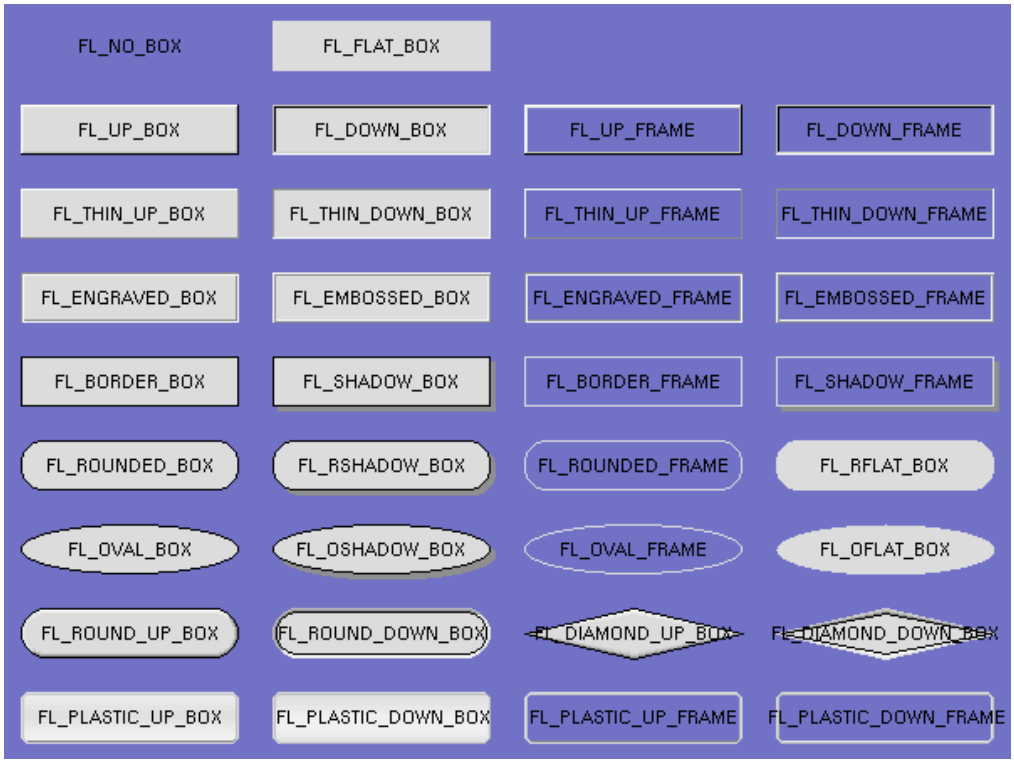

Figura 2.4: Tipi di box presenti in FLTK

2.2.6 Callback

Le callback sono funzioni che vengono chiamate quando il valore di un widget cambia. Alla funzione di callback viene inviato il puntatore del widget che ha cambiato valore e un puntatore a dati che si vogliono fornire.
La funzione di callback è generalmente:
*void xyz_callback(Fl_Widget *w, void *data)*{
...
}
L' associazione alla funzione invece avviene in questo modo:
int data;
button->callback(xyz_callback,&data);
Normalmente le callback vengono eseguite solo quando il valore del widget cambia. È possibile modificare questa impostazione tramite il metodo *when()*.

2.3 Gli Eventi

Ogni volta che un utente sposta il puntatore del mouse, fa un clic su un pulsante, o preme un tasto, un evento viene generato e inviato all' applicazione. Gli eventi possono anche provenire da altri programmi.

Gli eventi sono identificati con un intero passato come parametro al metodo virtuale *Fl_Widget::Handle()*. Altre informazioni sull' evento più recente vengono memorizzati in posizioni statiche e acquisite chiamando il metodo *Fl:: *_event()*. Queste informazioni statiche restano valide fino al verificarsi del prossimo evento.

Esiste una macro per ogni evento da gestire. Gli eventi che si possono gestire sono quelli del mouse, del focus della finestra, della tastiera e dei widget. I widget possono indicare che non sono interessati a gestire un evento restituendo 0 nel metodo *handle()* ed FLTK invia altrove l' evento.

La maggior parte degli eventi vengono inviati direttamente al metodo handle della finestra principale. La finestra ha la responsabilità di propagare l' evento ai figli. Alcuni eventi vengono inviati da FLTK direttamente ai figli.

2.4 Estendere FLTK

Nuovi widget vengono creati estendendo una classe esistende di FLTK, di solito Fl_Widget per controllo (interagire con l' utente) e Fl_Group per la composizione di widget.
Si possono estendere anche classi di widget esistenti per cambiare l' aspetto dei widget stessi. Per esempio i pulsanti sono tutti sottoclasse di Fl_Button, l' unica differenza è il codice che disegna l' aspetto.

La sottoclasse può estendere direttamente Fl_Widget o una sua sottoclasse. Fl_Widget ha solo quattro metodi virtuali che potrebbe essere necessario ridefinirne.

Il costruttore della classe deve avere i seguenti argomenti:
*MyClass(int x, int y, int w, int h, cont char *l = 0);*
Il costruttore deve chiamare il costruttore della classe base e passargli gli stessi parametri.

Il metodo virtuale *int Fl_Widget::handle(int event)* gestisce ogni evento passato al widget. Gli eventi sono identificati dal parametro intero. Altre informazioni sull' evento vengono memorizzate in posizioni statiche e bisogna chiamare i metodi Fl::*_event() per recuperarle. Queste informazioni rimangono valide fino a quando non si verifica un altro evento.

2.5 Utilizzo di OpenGl in FLTK

Il modo più semplice per utilizzare OpenGl è estendere la classe *Fl_Gl_Window*. La sottoclasse deve implementare il metodo *draw* che utilizza le chiamate OpenGl per disegnare sul display. Il programma dovrà chiamare *redraw* se vuole ridisegnare il display ed FLTK provvederà a chiamare il metodo draw(); Nella sottoclasse estesa bisogna includere il file <FL/gl.h>.

La sottoclasse di Fl_Gl_Window deve fornire:

- Una definizione di classe.
- Un metodo draw().
- Un metodo handle se c' è bisogno di gestire input da parte dell' utente.

La definizione della sottoclasse:

```
class MyWindow: public Fl_Gl_Window {
.     void draw();
.     int handle(int);

public:
.     MyWindow(int X, int Y, int W, int H, const char *L)
.       : Fl_Gl_Window(X, Y, W, H, L) {}
};
```

Il metodo *draw()* è dove effettivamente avviene il disegno attraverso le primitive OpenGl:

```
void MyWindow::draw(){
.     if(!valid()){
.         ... inizializza proiezione, viewport, ecc
.         ... dimensione della finestra è in w () e h ().
.         ... valid() viene modificata da FLTK dopo la funzione draw()
```

```
.     }
.     ... disegno ...
}
```

Il metodo handle gestisce gli eventi del mouse e della tastiera per la finestra. Quando il metodo handle() viene chiamato, il contesto OpenGL non è configurato. Se il display cambia, si dovrebbe chiamare redraw() e lasciare a draw() il compito di ridisegnare utilizzando le primitive OpenGl. Non si devono chiamare le funzioni di disegno OpenGL da dentro il metodo handle().

```
int MyWindow::Handle(event int){
.     switch (event){
.     case FL_PUSH:
.        ... tasto del mouse premuto ...
.        ... posizione in Fl::event_x() e Fl::event_y()
.        return 1;
.     case FL_DRAG:
.        ... mouse spostato con il tasto premuto ...
.        return 1;
.     case FL_RELEASE:
.     ...
.     }
}
```

Capitolo 3

Analisi del codice di *XCSubd*

In questo capitolo verranno spiegati i passi eseguiti prima di effettuare il refactoring del pacchetto XCSubd.
Verrà mostrato prima lo schema logico del pacchetto e quindi le varie relazioni e dipendenze tra classi, in seguito verrà spiegato il metodo utilizzato per rendere indipendente il pacchetto XCSubd dalla libreria GLUI.
Nei prossimi capitoli verranno trattati il procedimento di sostituzione e il testing del prodotto ottenuto.

3.1 Schema logico di *XCSubd*

Per verificare la possibilità di un' operazione di refactoring sul pacchetto XCSubd si è effettuata un' analisi del codice contenuto nello stesso. Questa analisi è mirata a ricavarne lo schema logico e quindi le varie relazioni e dipendenze tra classi e, in particolar modo, le dipendenze tra la libreria grafica attuale GLUI e il restante codice presente nel pacchetto.

Dall' analisi è risultato che il pacchetto ha forti dipendenze con GLUI e coinvolge molte classi.
Per eliminare queste dipendenze si è effettuato un primo procedimento per

isolare l' utilizzo di GLUI, elimanando la relazione tra le classi che creano l'interfaccia grafica con le rastanti.
Il procedimento successivo, che discuteremo nel prossimo paragrafo, è stato quello di rendere il pacchetto indipendente da GLUI, commentando ogni riga di codice che fa riferimento alla libreria GLUI iniziando dall' "include" della libreria stessa.
Il risultato ottenuto da questa prima fase è un pacchetto compilabile, contenente le stesse funzionalità, ma privo di interfaccia grafica.

Per non appesantire la lettura non entreremo nel dettaglio delle varie classi ma verrà mostrato un piccolo schema riassuntivo con solo le relazioni e dipendenze tra le classi e la libreria GLUI.
Subito dopo verranno descritte con maggior dettaglio le classi coinvolte e le modifiche effettuate.

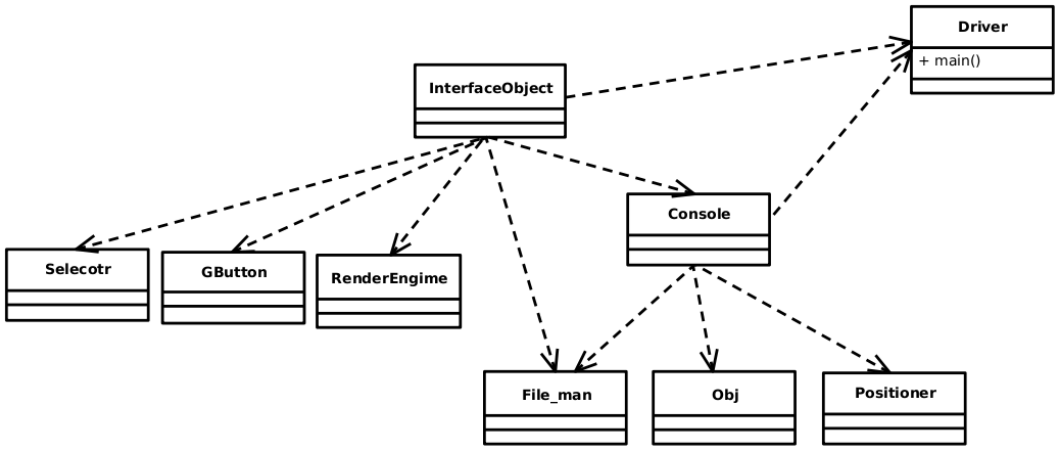

Figura 3.1: Relazione tra la classe InterfaceObject e le restanti

Adesso lo scopo principale è quello di isolare le classi contenute nella cartella interface dal resto del programma.
Questa cartella contiene tutti i metodi e le classi per creare l' interfaccia grafica utente utilizzando la libreria GLUI. La classe principale che si occupa della

creazione dell' interfaccia grafica è *InterfaceObject* quindi bisogna isolare l'utilizzo di questa classe dal resto del pacchetto.
Come si nota dalla Figura 3.1, invece di essere un modulo separato ed indipendente, ha molte relazioni e dipendenze con classi esterne. Inoltre le classi coinvolte non sono isolate ma hanno anche esse una relazione con altre classi del pacchetto, dove, queste relazioni, si estendono in modo gerarchico con il resto delle classi.
Il risultato è di avere la maggior parte delle classi con una dipendenza, anche se indiretta, dalla libreria GLUI.
Quella che presenta con maggior enfasi questo problema è la classe *Console* che, come si vede in figura, dipende dalla classe *Interface*, e la "classe" Obj ha una relazione con la classe Console. Obj in realtà non è una classe ma è una raccolta di classi collegate tra loro in modo gerarchico e rappresentano la maggioranza delle classi di tutto il pacchetto.
Per rendere meglio l' idea nella Figura 3.2 viene mostrata l' ereditarietà tra classi Obj contenute nella cartella coresystem [ONTE07]. Le classi mostrate in figura vengono nuovamente estese dalle classi contenute nella cartella surfaceclass.

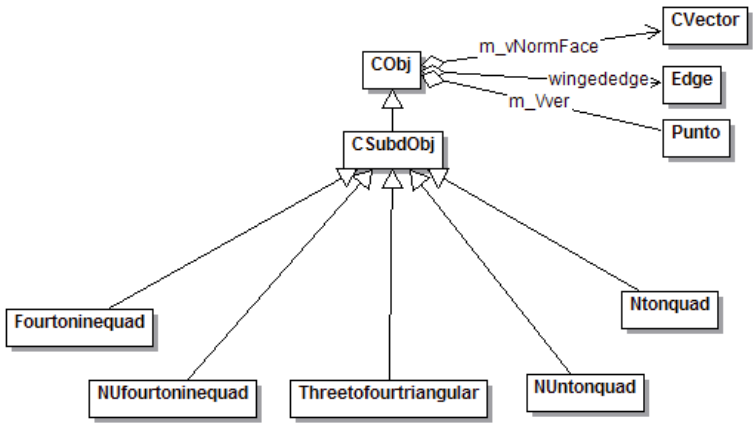

Figura 3.2: Ereditarietà delle classi contenute in Coresystem.

Qui di seguito vengono analizzati i metodi usati per avere la classe InterfaceObject indipendente dal resto del pacchetto insieme ad una breve descrizione della dipendenza stessa:

- *Console* -> Registra i log del programma come ad esempio il tipo di suddivisione effettuata e il tempo impiegato.

  Dipendenza:
  Utilizza un' istanza di IterfaceObject. Imposta "Console->text", variabile interna di una classe appartenente alla libreria GLUI, per ottenere l' output su video.

  Refactoring:
  Si è lasciato Console come classe generica che effettua il log su una stringa passata come parametro.
  Si è esteso Console con la classe ConsoleGui che richiama i metodi della Superclasse passando come stringa quella della console "Console->text" per effettuare l' update per la visualizzazione su schermo.
  In questo modo si sono risolte tutte le dipendenze da questa classe.
  Se si vuole ottenere il log basta recuperarlo dalla classe Console.

- *File_Man* -> Carica da file oggetti obj.

  Dipendenza:
  Utilizza un' istanza di InterfaceObject per ottenere il valore della variabile autoweight.

  Refactoring:
  La variabile viene passata come parametro alla funzione che lo richiede

e salvata in una variabile interna alla classe per usi successivi.

- *Positioner* -> Classe che viene utilizzata per i movimenti degli oggetti 3D visualizzati sullo schermo.

  Dipendenze:
  Utilizza un' istanza di InterfaceObject ma viene solamente dichiarata e mai riferita(utilizzata).

  Refactoring:
  Delete dell' istanza.

- *Obj* -> Intesa come raccolta di classi che estendono la classe base CObj. Si occupano delle suddivisioni e rappresentazione del modello.

  Dipendenze:
  Utilizza un' istanza di Console.

  Refactoring:
  Come tutte le classi che dipendono da Console la dipendenza è stata risolta tramite la generilizzazione della stessa.

- *Selector* -> Provvede ad algoritmi per la selezione di punti appartenenti ad oggetti 3D disegnati sullo schermo. Questo procedimento viene denominato picking.

  Dipendenze:
  Utilizza un' istanza di InterfaceObject per ottenere il valore della variabile viewRotate. Questo parametro indica lo spostamento effettuato

dall' oggetto 3D.

Refactoring:
Viene spostato il settaggio del movimento in un altro modulo eliminando l' uso della variabile.
Per effettuare il picking si potrebbe usare uno dei tanti algoritmi che si trovano in rete perché l' implementazione attuale non è corretta.

- *Render_Engime* -> Classe per il rendering di oggetti 3D.

  Dipendenze:
  Utilizza un' istanza di InterfaceObject per prelevare le variabili getpred, getLightPosition e viewRotate.

  Refactoring:
  Viene spostato l' uso di viewRotate in un altro modulo (quello che si occupa della gestione degli eventi del mouse e di disegno attraverso GLUT) e sono state aggiunte due variabili interne alla classe per memorizzare i parametri getpred e getLightposition.

- *GButton* -> Viene solamente usata da InterfaceObject per aggiungere bottoni e non interferisce con il resto delle classi.

  Dipendenze:
  Viene utilizzata solamente da InterfaceObject.

  Refactoring:
  nessuno.

- *Driver* -> Contiene il main del programma e i metodi per gestire gli eventi associati a GLUT e all' interfaccia grafica.

  Dipendenze:
  Tutte le classi.

  Refactoring:
  Nessuno.

Dopo questo primo lavoro di refactoring si ottiene un pacchetto che è indipendente dalla realizzazione dell' interfaccia grafica utente, ma la libreria grafica utilizzata per creare l' interfaccia è ancora GLUI.

3.2 Eliminazione GLUI

Il procedimento successivo, prima di creare la nuova interfaccia grafica, è quello di eliminare l' utilizzo della libreria GLUI dal pacchetto XCSubd.

I due file di interesse a questo procedimento sono:

- *InterfaceObject* che contiene la classe InterfaceObject utile all' implementazione dell' interfaccia grafica attraverso l' utilizzo della libreria GLUI.

- *Driver* che, oltre a contenere il main, provvede anche ad implementare i metodi utili alla gestione delle callback che gestiscono gli eventi provenienti dall' interfaccia grafica (inviati dalla libreria GLUI), ed inoltre contiene anche i metodi che gestiscono gli eventi associati a GLUT.

Una modifica che viene subito in mente è quella di riscrivere le due classi eliminando la possibilità di riutilizzo del codice.
Il motivo risiede nel fatto che InterfaceObject contiene l' implementazione vera e propria dell' interfaccia grafica e il codice contenuto nel file Driver fa un utilizzo massiccio della classe InterfaceObject sia per gestire GLUT sia per GLUI.

Controllando con attenzione la struttura della classe InterfaceObject, e l'uso fatto nel file Driver, si nota però che può avvenire un' astrazione della classe stessa, in quanto molti metodi e variabili, usati da Driver, si possono generalizzare alla creazione di interfacce grafiche adatte al pacchetto XCSubd, delegando la costruzione dell' interfaccia grafica ad una sottoclasse.
In questo modo, non solo si offre la possibilità di scegliere la libreria grafica preferita, ma si ottiene un ottimo riutilizzo del codice.

Questa astrazione fa si che il file Driver utilizza solo i metodi e le variabili della classe astratta, lasciando, se si vuole, come unica relazione con la libreria grafica, quella dell' impementazione delle callback.
È importante evidenziare che la classe astratta appena creata non utilizza librerie grafiche.

Viene riportato un esempio per chiarire meglio il concetto:
`Stato di Partenza:`

- *InterfaceObject* ->
  Contiene la variabile "pippo" dove il suo valore rappresenta la posizione della luce.
  Avviene l' associazione di un metodo "callback" ad un elemento di controllo dell' interfaccia grafica a cui viene passato il parametro "pippo", utile alla gestione della posizione della luce.

- *Driver* ->
  Implementa il metodo “callback”.
  Quando si vuole ricavare il posizionamento della luce, per esempio si vuole ridisegnare la scena, si accede alla variabile “pippo”.

`Stato attuale:`

- *InterfaceObject* ->
  Contiene la variabile “pippo ” dove il suo valore indica la posizione della luce.

- *Interface** ->
  Una unova classe che implementa InterfaceObject e che si occupa di inserire l’ elemento dell’ interfaccia grafica per il controllo della posizione della luce. In questa classe si sposta l’ associazione del metodo per gestire l’ evento dell’ elemento appena creato che va a modificare la variabile “pippo” della superclasse.

- *Driver* ->
  Continua ad utilizzare la variabile “pippo” dichiarata in InterfaceObject.
  Può continuare ad implementare il metodo “callback” o, se si preferisce, spostarlo in un’ altra classe come ad esempio in Interface*.

L’ obbiettivo è stato quello di creare, partendo da InterfaceObject, una classe astratta in modo tale che chi vuole implementare un’ interfaccia grafica per XCSubd può estendere InterfaceObject con una sottoclasse specifica utilizzando il toolkit preferito, e andare a sostituire nel Driver le callback.

Per dimostrare il corretto funzionamento, prima di implementare una nuova interfaccia con l' utilizzo di FLTK, si è spostato in un nuovo file ed in una nuova classe, di nome InterfaceGLUI che estende IntefaceObject, il restante codice appartenente alla precedente classe ed utile all' implementazione dell' interfaccia grafica attraverso l' uso della vecchia libreria GUI. Il contenuto del file Driver è rimasto inalterato.

Visto che questo Test ha avuto un esito positivo, ed avendo separato completamente l' utilizzo della GLUI in un singolo file, si è eliminata, con semplicità e definitivamente, l' utilizzo della stessa all' interno del pacchetto XCSubd andando ad escludere la classe appena creata ed eliminando/commentando le callback dentro il file Driver.
Le funzioni per l' utilizzo di GLUT presenti nel file Driver sono rimaste pressocché inalterate e le modifiche effettuate sono definitive e rimangono tali anche per nuove implementazioni di interfacce grafiche indipendentemente dalla libreria grafica scelta.

La compilazione in questo stato ha prodotto un pacchetto completo di tutte le funzionalità ma privo d' interfaccia grafica.

Capitolo 4

La Nuova Interfaccia Grafica

Dopo aver dimostrato la compatibilità tra GLUI e FLTK, dopo aver eliminate le dipendenze del pacchetto XCSubd dalla libreria grafica utilizzata e dopo aver isolato l' uso di GLUI all' interno di una classe, è avvenuta l'implementazione della nuova interfaccia grafica utilizzando la libreria FLTK.

L' implementazione della nuova interfaccia grafica con l' utilizzo della libreria FLTK è stata fatta in modo da assomigliare a quella precedente sia come aspetto che come implementazione.
La ragione di questa scelta è:

- dal punto di vista *visivo* per semplificare e rendere familiare da subito l' utilizzo del nuovo pacchetto all' utente.

- dal punto di vista *implementativo* per una questione di sicurezza.
  Dato che non esiste una buona documentazione sull' utilizzo delle funzioni e classi che implementano le varie funzionalità di XCSubd si possono commettere errori di cui non si conosce la causa e, visto che XCSubd ha già problemi di instabilità, riconoscere un nuovo difetto/problema dovuto all' implementazione della nuova interfaccia grafica risulterebbe

difficile.

Quindi l' implementazione della nuova interfaccia sarà simile a quella precedente.

Nei prossimi paragrafi analizzeremo in dettaglio l' implementazione della nuova classe InterfaceFLTK e la modifica del file Driver dopo aver chiarito le differenze principali tra le due librerie.

4.1 Concetti Preliminari

Per comprendere meglio di come sia stata implementata la nuova interfaccia grafica bisogna sapere quali sono le maggiori differenze a livello implementativo delle due librerie.
Le differenze vengono schematizzate qui di seguito insieme ad uno o più metodi che le rendono equivalanti.

In GLUI ad ogni componente dell' interfaccia grafica è associata una variabile, chiamata "live-variable", che viene modificata automaticamente per riflettere lo stato del componente, in FLTK bisogna fare una richiesta esplicita per ricavare il valore di una variabile che rappresenta lo stato di un componente.
Si sono usate le callback per emulare il comportamento delle live-variables.

Con GLUI si possono passare alle callback valori (r-value) e quindi il contenuto di variabili, FLTK accetta solo riferimenti (l-value). Con la libreria GLUI, passando degli interi alle funzioni di callback, si ha la possibbilà di creare un' unica funzione di callback per gestire eventi provenienti da diversi componenti, dove, attraverso uno *switch*, è possibile interpretare quale componente ha lanciato l' evento.
Con FLTK c' è bisogno di rami *if-then-else* annidati. In alternativa, per uti-

lizzare uno switch all' interno delle callback per riconoscere quale componente ha provocato l' evento, si possono dichiarare tante variabili per quanti eventi di componenti gestire, e assegnare ai componenti la stessa callback passando una variabile diversa come parametro per identificarli.

Con GLUI il refresh delle finestre avviene in automatico al variare dei controlli, con FLTK si deve fare un refresh esplicito al componente interessato.
Con FLTK si può effettuare esplicitamente il refresh di un componente subito dopo aver gestito, tramite callback, l' evento del componente stesso o di uno in relazione.

Con GLUI il posizionamento dei pulsanti avviene in automatico, con FLTK bisogna specificare la posizione in modo diretto.
Con FLTK, per semplificare il posizionamento (la scelta dei valori da passare alla crezione) di un componente, si può sfruttare la gerarchia delle classi per ricavare la posizione del genitore di un componente e decidere dove posizionarlo rispetto al genitore stesso. Rimane sempre esplicito il posizionamento del genitore.

Chiariti questi concetti è possibile effettuare la creazione della nuova classe che chiameremo *InterfaceFLTK* e modificare le callback contenute nel file *Driver* per la gestione degli eventi dei componenti provenienti dall' interfaccia grafica.

4.2 Implementazione della classe InterfaceFLTK

Il file *InterfaceFLTK* inizia con l' inclusione del file header *Intreface-FLTK.h* che contiene le costanti che indicano il posizionamento dei componenti principali dell' interfaccia grafica sullo schermo, e la dichiarazione della classe InterfaceFLTK che comprende i metodi e le variabili per la creazione e

la gestione dei componenti stessi.
Subito dopo viene implementato il costruttore della classe che, oltre a richiamare il costruttore della superclasse, invoca i metodi CreateInterface, CreateSelectionWin, CreateSettingWin, CreateLightWin, CreateAdaptiveWin, CreateSaveWin. Ognuno di questi metodi provvede alla creazione di una finestra.

Per utilizzare GLUT con FLTK bisogna creare una finestra utilizzando le funzioni messe a disposizione da GLUT.
Quindi, nella finestra principale, si è inizializzata GLUT e si è creata una sottofinestra tramite la chiamata *glutCreateWindow()*.
Una soluzione più elegante è quella di estendere una classe appartenente a FLTK che si occupa della gestione e creazione di finestre per utilizzare GLUT e instanziare questa classe quando ce n' è bisogno.
Il metodo che crea la finestra principale è *CreateInterface*. Questa finestra è composta da un' area centrale per la visualizzazione di superfici e da tre toolbar posizionati in basso, a destra e in alto della finestra principale.
Avviene prima la creazione della finestra, subito dopo l' inizializzazione delle GLUT e la relativa creazione della finestra e, infine, la creazione dei toolbar con i relativi elementi.
Gli altri metodi richiamati nel costruttore provvedono alla creazioni di sottofinestre che estendono funzionalità di XCSubd ma il procedimento di costruzione è uguale per tutte le finestre.

Viene creata la finestra con il comando *new Fl_Window* e dopo, nel blocco *begin() - end()* della finestra appena creata, si aggiungono gli elementi *gruppo* posizionandoli correttamente in un' area della finestra.
Per rendere l' elemento *gruppo* più simile a quello creato con GLUI si sono andati a settare, tramite variabili dell' elemento stesso, delle impostazioni utili a FLTK per visualizzare l' elemento sullo schermo. Queste variabili sono:

- *box*: Utile per impostare il tipo di box. È stato impostato a *FL_BORDER_BOX*.

- *label*: Serve per impostare il testo che comparirà nel box.
- *align*: Utile ad allineare il testo. Questa variabile è stata impostata per visualizzare il testo dentro e centrale al box.
- *labelsize*: Indica la grandezza del label che è stata settata a 18.
- *labeltype* Impostata a *FL_SHADOW_LABEL* indica il tipo di label. In questo modo il testo sarà obreggiato e scuro.

Dopo aver creato i vari gruppi si sono aggiunti i pulsanti relativi. Perciò subito dopo la crezione del gruppo e sempre dentro il blocco *begin() - end()* ma questa volta appartenente al componente gruppo vengono instanziati i vari elementi.
A questi elementi, sempre per renderli più simili a quelli creati con GLUI, si sono impostati la grandezza del label attraverso la variabile *labelsize* impostata a 12 e, per quanto riguarda i checkbox, si è dovuto impostare il parametro *type* a *FL_RADIO_BUTTON* per avere una scelta esclusiva, infine, attraverso la variabile callback, si è impostata la funzione che si deve eseguire per gestire l' evento del componente.

Riporto lo pseudo-codice dell' implementazione:

```
    /* costruisce una finetra */
void CostruisciFinestra(){
/* creazione della finestra */
win = new Fl_Window(x,y,lunghezza,altezza);
win->begin(); /* aggiunta elementi della finestra */
.     group = new Fl_Group(x,y,lunghezza,altezza,label); /* parametri
relativi alla finestra */
.     group->box(FL_BORDER_BOX); /* tipo di box */
.     group->align(FL_ALIGN_*); /* allineamento del label */
.     group->label(label); /* testo del label*/
.     group->labelsize(G_LABELSIZE); /* grandezza del label*/
```

```
.     group->labeltype(G_LABELTYPE); /* tipo di label */
.     group->begin(); /* aggiunta elementi del gruppo */
.      button = new Fl_Button(x,y,lunghezza,altezza,label); /* parametri
relativi al gruppo */
.      button->labelsize(LABELSIZE); /* grandezza del testo dei pulsanti
*/
.      button->callback(button_cb); /* imposta la funzione di callback
del pulsante */

.     button2 = ...

.     group->end(); /* fine aggiunta elementi del gruppo */

.     group2 = ...
.     ...
.     group2->begin();
.     ...
.     group2->end();
.     ...

.     groupN
.   ...
.     win->end(); /* fine aggiunta elementi della finestra */

}
```

Il procedimento è stato semplice, infatti, dopo aver creata la struttura si sono andati ad aggiungere gli elementi come bottoni, menu, barre di scorrimento all' interno delle singole aree.
Per aggiungere degli elementi con FLTK basta semplicemente crearli (istan-

ziarli) dentro un blocco *begin() - end()* del componente che li vuole includere e posizionarli correttamente all' interno della finestra o del componente stesso. In alternativa, dopo averli creati, si possono aggiungere tramite il metodo *add* del componente.
Per quanto riguarda la creazione, e quindi la scelta del tipo, dei singoli elementi che compongono l' interfaccia grafica si è andato a sostituire alla vecchia implementazione che utilizzava GLUI una equivalente in FLTK.
L' output della funzione *CreateInterface* viene mostrata in Figura 4.1.
Si puo' osservare la Figura 4.7 per notare la somiglianza ottenuta all' implementazione che utilizzava la libreria GLUI .

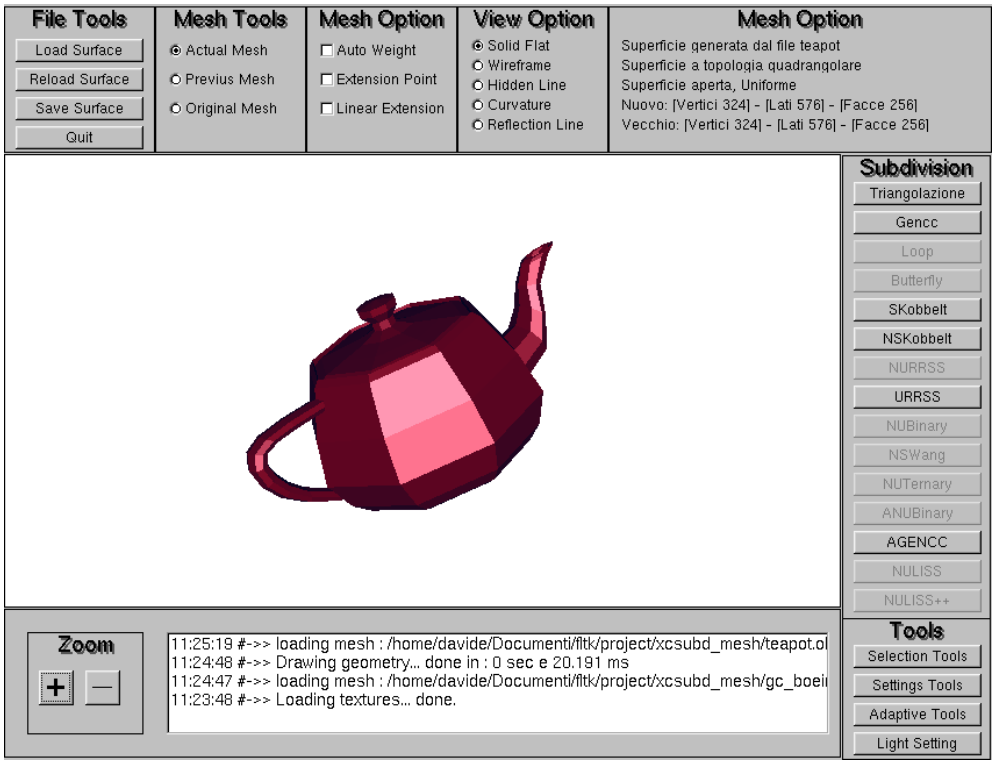

Figura 4.1: Finestra principale ottenuta con la libreria FLTK

Tutte le sottofinstre vengono create all' avvio del programma e nascoste o visualizzate quando se ne richiede l' utilizzo. Queste sottofinestre vengono

implementate usando lo stesso metodo di quello usato per creare la finestra principale.
Qui di seguito vengono elencate le funzioni che creano le sottofinestre con una breve descrizione e un' immagine del risultato prodotto.

CreateSelectionWin crea la sottofinestra utile alla selezione di punti appartenenti agli oggetti visualizzati. È composta da due gruppi di pulsanti, uno che si occupa della selezione di punti e l' altro per la definizione di curve.

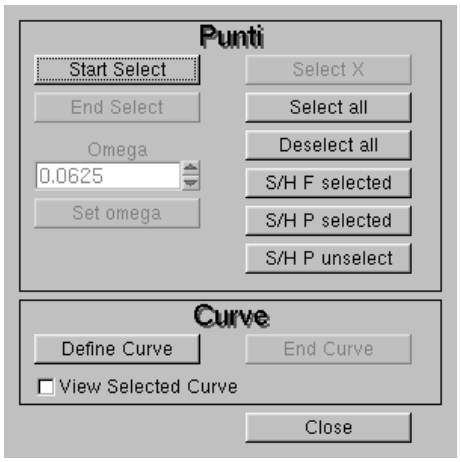

Figura 4.2: Finestra per la selezione ottenuta con la libreria FLTK

CreateSettingWin provvede alla sottofinestra che consente di modificare delle opzioni del programma. Le opzioni che si possono modificare sono la frequenza e la grandezza delle reflection lines, il tipo di materiale o la scelta di una texture, come deve essere mappata la texture, se utilizzare l' antialiasing e la grandezza delle linee nella visualizzazione di oggetti wireframe.

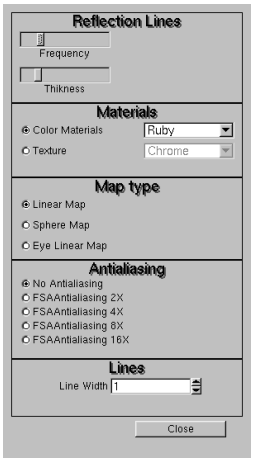

Figura 4.3: Finestra per la selezione ottenuta con la libreria FLTK

CreateLightWin gestisce l' illuminazione della scena, come la posizione della luce o l' attivazione della luce speculare.

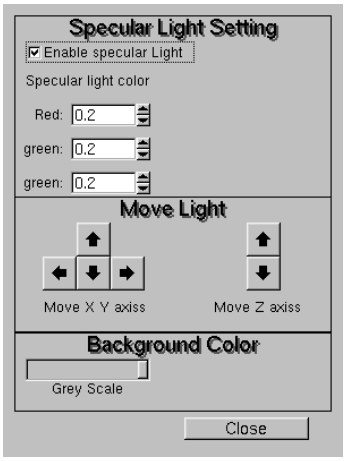

Figura 4.4: Finestra per gestire l' illuminazione ottenuta con la libreria FLTK

CreateAdaptiveWin gestisce i parametri per la tassellazione adattiva. Consente di settare il tipo di selezione adattiva da utilizzare e la soglia di selezione.

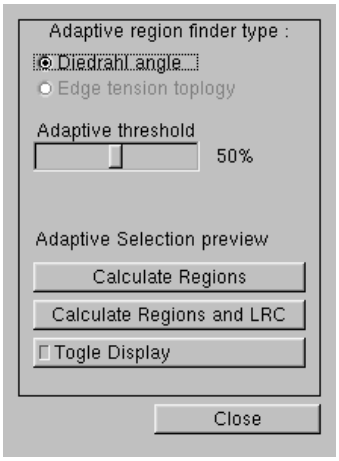

Figura 4.5: Finestra per la selezione ottenuta con la libreria FLTK

CreateSaveWin permette di salvare o esportare le superfici modificate in differenti formati.

Figura 4.6: Finestra per Salvare o Esportare superfici ottenuta con la libreria FLTK

Nella Figura 4.7 viene mostrata l' immagine dell' interfaccia grafica ottenuta utilizzando la libreria GLUI.
Si può notare che la disposizione dei pulsanti e la loro grandezza sono rimasti simili, così come sono rimasti simili la grandezza e il tipo di testo.

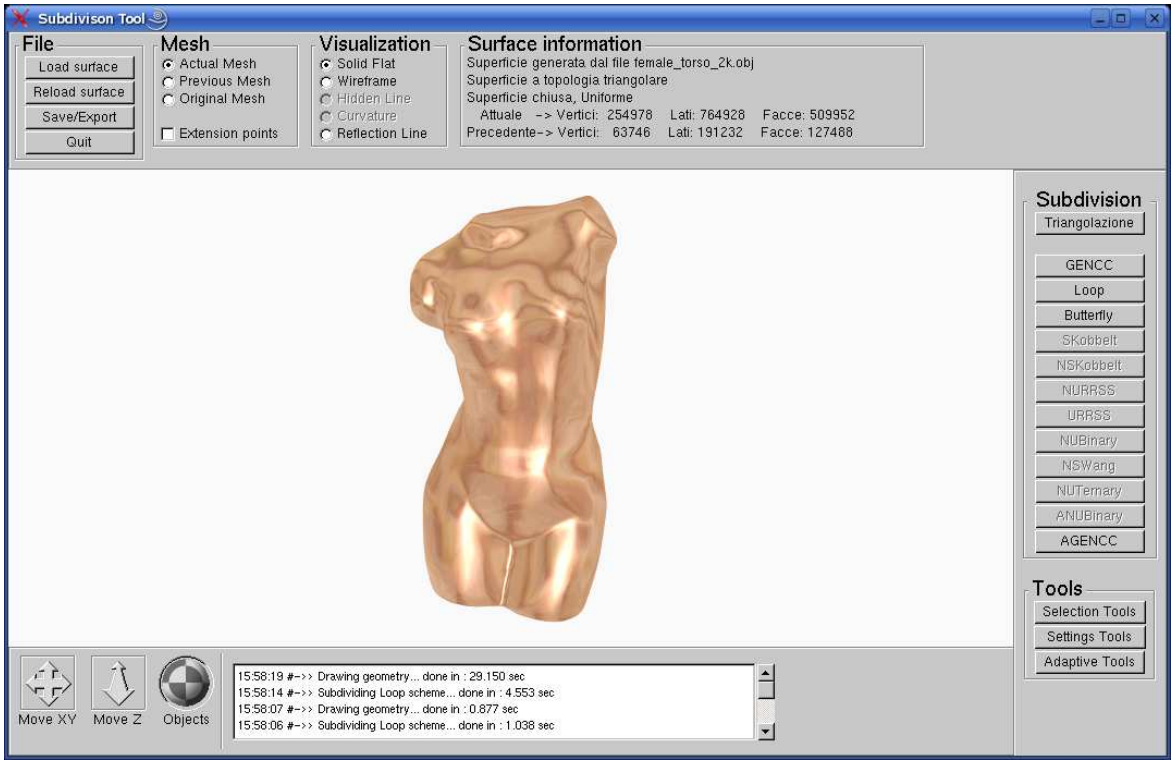

Figura 4.7: Finestra principale ottenuta con la libreria GLUI

4.3 Modifica del file Driver

Implementare un' interfaccia non significa solo costruire bottoni, finestre, menu, ecc ma anche associare ad ogni elemento dell' interfaccia un' azione.
Questo compito viene svolto dalle *callback*.
La gestione delle callback vengono implementate in modo diverso per ogni libreria ma il loro funzionamento rimane identico:

Si associa una funzione per ogni evento dell' interfaccia che si vuole gestire.

Nel pacchetto XCSubd le callback sono implementate nel file Driver ma potevano essere implementate in un file qualsiasi.
In accordo con la scelta presa prima, cioè di avere un' implementazione simile a quella fatta con GLUI, si sono lasciate le implementazioni delle callback dentro il file Driver.
Si è cambiato il prototipo delle funzione, cioè i parametri passati e il tipo restituito, e, per rendere il codice "standard", si sono rinominati i nomi delle funzioni da handler_* a *_cb dove l' asterisco rappresenta una stringa generica e cb ha il significato di callback.
Il corpo delle funzioni di callback è cambiato, soltanto nei casi in cui una funzione gestisce più componenti, per intercettare l' elemento dell'interfaccia che ha provocato l' evento. È stato semplice intercettare quest' elemento dell' interfaccia perché uno dei parametri che viene passato dalla libreria FLTK alla funzione di callback è l' elemento stesso, quindi tramite un *if-then-else*, che confronta il parametro con tutti i possibili elementi, si riesce a risalire all'elemento interessato.

Riporto uno pseudo-codice di esempio per chiarire il concetto.

```
    /* funzione che crea i pulsanti per muovere le luci */
function MuoviLuci{
.       /* creo i pulsanti */
.       muoviDestra = new Fl_Button(x,y..);
.       muoviSinistra = new Fl_Button(x,y,..);
.       ...
.       /* associo la stessa funzione di callback */
.       muoviDestra->callback(muoviLuci_cb);
.       muoviSinistra->callback(muoviLuci_cb);
.       ...  }
```

```
/* funzione nel Driver per gestire gli eventi */
function muovi_cb(Fl_Widget *w){
.       if(w == muoviDestra)
.         muovi verso destra le luci
.       else if(w == muoviSinistra)
.         muovi verso sinistra le luci
}
```

Queste sono state tutte le operazioni di refactoring avvenute nel file Driver semplificate grazie alla creazione della classe astratta InterfaceObject.

4.4 Scelte Effettuate

Le scelte effettuate, che vengono elencate qui di seguito, sono mirate al riutilizzo del codice e prese in base al metodo di sostituzione *in-linea*.

La scelta di creare una classe astratta come punto di partenza nella creazione dell' interfaccia è stata di grande aiuto per l' operazione di refactoring ma si sarebbe preferito costruire l' interfaccia utilizzando metodi di una ipotetica "libreria XCSubd" e non integrare l' uso dell' interfaccia con il resto del pacchetto.

Si e' scelto di usare i due metodi esposti prima per intercettare gli eventi nelle callback, utilizzando sia *switch* che rami *if-then-else*, con lo scopo di dimostrare quali dei due è più conveniente usare.
Come aspetto personale preferisco associare callback ad un numero ristretto di componenti e utilizzare i rami *if-then-else* poiché il codice risulta più chiaro e, in alcuni casi, più breve perché si risparmia la dichiarazione di molte varia-

bili.

Il posizionamento dei vari elementi che compongono l' interfaccia è avvenuta sia provando a utilizzare il genitore come riferimento e posizionare i genitori esplicitamente, e sia posizionando il componente manualmente.
Nel primo caso il posizionamento è più semplice, nel secondo è più preciso.
Sarebbe stato più chiaro e molto più predisposto ad azioni di refactoring creare una classe contente gruppi di componenti costruiti con un determinato aspetto e posizionando la classe all' interno della finestra principale. Ogni classe avrebbe avuto le proprie caratteristiche e sarebbe più gestibile la composizione che avverrebbe tramite classi e non con singoli elementi.
Sarebbe stato utile l' uso del tool *FLUID* per posizionare correttamente e con semplicità tutti i vari componenti, sempre dividendoli per classi.

Per quanto riguarda la gestione di GLUT si è voluta integrarla appositamente in una finestra e delegare tutta la sua gestione ai metodi implementati nel file Driver.
Si sarebbe preferito delegare la gestione di GLUT ad una classe che estende *Fl_Glut_Window*, una classe particolare di FLTK adatta ad interagire con GLUT.

Capitolo 5

Test e Bugs del pacchetto *XCSubd*

I test sono stati effettuati su tutte e due le versioni del pacchetto XCSubd con sistemi operativi Linux. Le distribuzioni utilizzate sono state:

- Ubuntu 8.04 hardy.
- Debian 5.00 etch.

La libreria FLTK è adatta al porting sui più diffusi sistemi operativi come Windows e MacOs ma su di essi non è avvenuto nessun test.
La versione di XCSubd che utilizzava GLUI era compatibile ai sistemi Windows perciò il pacchetto è compatibile con questo sistema operativo indipendentemente dalla libreria grafica utilizzata.
Un' operazione futura molto importante sarà quella di creare l' eseguibile del pacchetto per i sistemi Windows.

La compilazione delle due versioni del pacchetto XCSubd ha prodotto, dopo numerosi tentativi, un file eseguibile di 5.2M contro i 5.0M della versione che utilizzava GLUI stranamente a quanto ci si aspettava dato che uno dei vantaggi principali nell' utilizzo della libreria FLTK è quello di ottenere

un eseguibile ridotto nelle dimensioni.

L' esecuzione di XCSubd con l' utilizzo delle GLUI ha prodotto differenti risultati sui differenti sistemi operativi utilizzati.
Sul sistema Debian l' esecuzione del software produceva la finestra di lavoro che ci si aspettava mentre su Ubuntu, installata su un' architettura a 64bit, la finestra non veniva visualizzata correttamente. In particolare veniva mostrata soltanto la toolbar superiore e rallentava notevolmente tutte le altre applicazioni in esecuzione sul pc, e gli unici pulsanti visualizzati non rispondevano agli eventi, come se i bottoni non avessero funzionalità.
La risposta dell' applicazione a modifiche della finestra, come resize, spostamento, iconize, era lenta e provocava il blocco, per alcuni secondi, dei dispositivi di input come mouse e tastiera.

Come soluzione si è provato a creare l' interfaccia (naturalmente utilizzando GLUI) con solo la toolbar superiore, quella che si visualizzava correttamente, ed escludere quella inferiore e laterale. In questo modo l' esecuzione non rallentava gli altri programmi e la risposta della finestra a modifiche, ma gli eventi dei pulsanti continuavano a non venire gestiti.
Risulta difficile individuare la causa di questo malfunzionamento.
Il problema puo' risiedere nel come è stata implementata l' interfaccia e quindi di come è stata utilizzata la libreria GLUI, e non nella libreria stessa, in quanto gli esempi allegati con la libreria funzionano correttamente su tutte e due le distribuzioni.
Oppure gli esempi allegati non utilizzano funzionalità della libreria GLUI che invece vengono utilizzate da XCSubd e quindi il problema è legato alla libreria stessa.

L' esecuzione di XCSubd utilizzando la libreria FLTK non ha dato problemi di visualizzazione e interazione degli elementi dell' interfaccia in entrambe le distribuzioni, ed è risultata più veloce nel gestire la visualizzazione di ogget-

ti 3D anche di forme complesse.

Durante l' utilizzo del software nella fase di testing, alcune volte XCSubd terminava la sua esecuzione con l' output su terminale che informava un *segmentation fault*.

La terminazione non corretta di un programma in esecuzione più temuta dai programmatori è di sicuro con un *segmentation fault*, perché è difficile individuarne la causa, si sa solo che il programma ha fatto riferimento a una zona di memoria non valida, non accessibile.

In XCSubd il *segmentation fault* è dovuto nella maggior parte dei casi a:

- Una mancanza di controllo sull' inizializzazione di puntatori a variabili prima del loro utilizzo, andando così a dereferenziare un puntatore nullo.

- Cicli, con i costrutti *for* anche annidati, controllati da un indice dove, con l' indice, si accede ad elementi di array. L' indice viene modificato tramite operazioni complicate anche attraverso combinazioni di elementi dell' array stesso e di altri. Non avendo un corretto uso dell' indice, in quanto non e' determinabile staticamente il suo valore, si può utilizzare un elemento dell' array inesistente, quindi utilizzare un puntatore nullo.

- L' utilizzo eccessivo di cast statici su puntatori a classi può provocare l'utilizzo di un oggetto incompleto o la perdita di informazioni.

Per ridurre i segmentation fault è possibile aggiungere soltanto più controlli ai puntatori utilizzati, ma non è possibile ridurli nei cicli *for* perché si dorvrebbero cambiare gli algoritmi di suddivisione, e neanche per i cast perché si dovrebbe sostituire la rappresentazione del modello. Tutte queste

sono operazioni che produrrebbero un nuovo software.

Sempre durante l' utilizzo del software XCSubd, si è osservato che cambiando i parametri di "specular light setting" nel pannello "light setting" si cambia il tipo di materiale e non il colore della luce.
Invece nel pannello "Selection Tool" l' operazione di selezione produceva segmentation fault perché veniva utilizzata una variabile non inizializzata e, "SelectX", sempre nello stesso pannello, alcune volte entra in loop.
Anche "NsSwang" portava a segmentation fault se eseguito più di una volta consecutivamente, per evitarlo si salta un passo di suddivisione e viene segnalato l' errore sul terminale.

Capitolo 6

Conclusioni

Questo lavoro di tesi descrive il processo avvenuto per effettuare un' operazione di refactoring sul pacchetto *XCSubd*.

Per comprendere meglio il lavoro svolto vengono introdotte le librerie grafiche in generale, che cosa sono e a cosa servono.
Il processo di sostituzione ha inizio con un' analisi del codice del pacchetto *XCSubd* per evidenziare la porzione di codice da modificare o sostituire, subito dopo è avvenuta l' implementazione della nuova interfaccia con i relativi test.

Se si vuole approfondire l' utilizzo di una libreria grafica si possono sempre leggere le documentazioni e i manuali che si trovano in rete. Quello che si è voluto descrivere con maggior dettaglio è la metodologia usata per confrontare le librerie grafiche.
Questa metodologia è adatta al confronto di qualsiasi libreria grafica e si basa su una prima analisi teorica, utile per conoscere le caratteristiche, seguita da una traduzione degli esempi allegati nella libreria da sostituire, che conferma la possibilità di effettuare una sostituzione.

Dall' analisi del codice è emerso che il pacchetto XCSubd ha molta dipendenza tra moduli non affini. Si sono andate ad osservare con maggior dettaglio

le classi utili nel creare l' interfaccia grafica e si sono modificate in modo da renderle indipendenti dal resto del programma.

Dopo essere sicuri che le librerie sono compatibili, ed ottenuto uno schema di sostituzione dagli esempi delle librerie tradotti, ed avendo l' interfaccia grafica indipendente dal resto del pacchetto, è stato semplice e ripetivo sostituire una libreria con un' altra.
Inoltre la sostituzione è avvenuta con la certezza di un esito positivo, poi dimostrato dai test effettuati.

Il pacchetto XCSubd ha un problema di organizzazione logica del codice che sembra essere avvenuta nel momento in cui le dimensioni del pacchetto sono cresciute considerevolmente.
Questo è accettabile se si pensa che XCSubd è nato e progettato con lo scopo di sperimentare una nuova tecnica di suddivisione di superfici, infatti le sue origini risalgono a una raccolta di classi che implementavano nuovi algoritmi di suddivisione di superfici.
Adesso, nella sua versione 1.4, la sua funzionalità va oltre la semplice sperimentazione di superfici di suddivisione, infatti, grazie al lavoro di tesisti, si è arrivati, partendo da algoritmi teorici di suddivisione, ad un pacchetto completo, in grado di fornire un riscontro visivo di alta qualità, senza includere le innumerevoli opzioni che si possono settare ed al considerevole numero di algoritmi di suddivisione messi a disposizione.

Il pacchetto XCSubd non è ancora adatto per la distribuzione ad un pubblico che non conosce le tecniche di suddivisione di superfici perché azioni non previste non vengono gestite, ma danno come risultato il crash del programma o un risultato errato. Però possiede tutte le potenzialità e funzionalità per far utilizzare e conoscere queste nuove tecniche di suddivisione di superfici anche ad un pubblico “curioso” oltre che a quello esperto.

Bibliografia

[XCMODEL] G.Casciola. *xcmodel: a system to model and render NURBS curves and surfaces* User's guide (1999), `http://www.dm.unibo.it/~casciola/html/xcmodel.html`

[MEMS09] M.Sweet, P.Earls, M.Melcher, B.Spitzak *FLTK 1.1.10 Programming Manual*, (2009). `http://www.fltk.org/documentation.php/doc-1.1/toc.html`

[RADE99] P.Rademacher *A GLUT-Based User Interface Library*, (1999).

[ONTE07] F.Dell'Onte *Tassellazione adattiva di superfici di suddivisione in XCSubd*, (2007).

[MAGR08] A.Magrini *Implementazione di schemi di suddivisione non uniformi in XCSubd*, (2008).

[SASS05] L.Sassi *XCSubd: un ambiente per l' analisi e sperimentazione di superfici di suddivisione*, (2005).

[LUCAR9] M.Lucarelli *Interfacce Grafiche Veloci e leggere con il toolkit Fltk*, Linux&C - Anno 9 - Numero 59