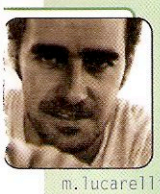
Non solo QT su queste pagine: per chi ha necessità di spazio e leggerezza, ecco FLTK, un toolkit che mantiene davvero quello che promette. Ecco qualche esempio di utilizzo e alcuni consigli per impiegarlo al meglio!

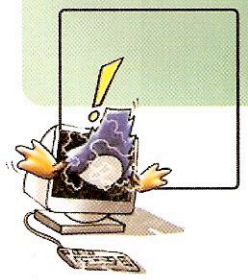
Interfacce grafiche veloci e leggere con il toolkit FLTK



Matteo Lucarelli

m.lucarelli@oltrelinux.com

Ingegnere aeronautico e programmatore dall'età di 11 anni, lavora come consulente nella progettazione e prototipazione di sistemi innovativi di monitoraggio e sicurezza. Mototurista e musicista dilettante, non disdegna, a tempo perso, l'attività di web designer. Quando possibile preferisce lavorare con strumenti open-source (e rilasciare il codice sotto GPL).



Questo articolo si propone di presentare la libreria FLTK *(the Fast Light Toolkit)* e di darne una introduzione all'uso nella programmazione di interfacce grafiche. Verranno illustrati i campi applicativi ed relativi vantaggi d'uso di questo toolkit alternativo ai più blasonati QT e Gtk. La trattazione è indirizzata a programmatori o ad aspiranti tali.

Per la completa comprensione si presuppone quindi una conoscenza di base delle tecniche di programmazione in linguaggio C/C++.

Cos'è un toolkit

FLTK (l'autore suggerisce la pronuncia pronuncia "fulltick") è una libreria per la realizzazione di interfacce grafiche. Librerie come questa (oltre ad FLTK ricordiamo le Qt, le Gtk e le WxWidget, ma ce ne sono molte altre), normalmente chiamate toolkit, sono, in generale, raccolte di funzioni (e/o classi) utili a semplificare la programmazione di GUI (Graphic User Interface) mettendo a disposizione del programmatore una serie di funzionalità "chiavi in mano" tramite cui realizzare e gestire finestre e oggetti grafici correlati. Tra queste funzionalità si trovano, per intenderci, bottoni, barre di scorrimento, tab, menu ed in generale quant'altro necessario a realizzare a gestire interfacce utente.

La necessità di tali librerie, almeno nel modo Unix, nasce sostanzialmente dalla eccessiva difficoltà di scrittura di codice che si interfacci direttamente ad X, ciò a causa dell'orientamento di basso livello delle Xlib, che ne rende la sintassi decisamente meno intuitiva e più laboriosa rispetto a qualsiasi toolkit.

Ulteriore comune vantaggio è la portabilità del codice che, demandando alla libreria la gestione delle differenze tra un sistema e l'altro, può essere solitamente ricompilato senza modifiche tra piattaforme differenti. E' quindi ovvio concludere che per qualsiasi progetto che preveda oggetti grafici più articolati di una finestra e qualche bottone, l'uso di un toolkit è praticamente una scelta obbligata.

Naturalmente tale scelta condizionerà, oltre allo stile di programmazione, anche un'altra serie di aspetti tra cui il comportamento dinamico dell'applicazione e, non ultima, la sua estetica.

Cercheremo di spiegare perché, tra le varie offerte possibili, meriti di essere preso in considerazione anche questo piccolo progetto open source.

Caratteristiche "fast&light"

Rispetto alle già citate "concorrenti", FLTK gioca le sue carte migliori in quanto a velocità e, soprattutto, a leggerezza: prima di tutto, FLTK è un toolkit che fa solo il toolkit, e cioè non include alcuna funzionalità aggiuntiva rispetto a quelle strettamente necessarie alla implementazione degli elementi di interfaccia.

Tra le funzioni e le classi disponibili non troveremo quindi, come invece avviene in molti altri casi, funzioni di gestione stringhe e file, funzioni di comunicazione, servizi di gestione thread o altro. Vi si trovano invece finestre, bottoni, barre di scorrimento e progress-bar, menu, ed in generale tutti gli elementi necessari alla realizzazione ed alla gestione di finestre.

La figura 1 mostra un assortito esempio delle primitive messe a disposizione del programmatore. La varietà di oggetti disponibili nella libreria FLTK è sufficiente ad un ampio ventaglio di esigenze. Viene inoltre estesa dalle ampie varia-

RIQUADRO 1

Ecco il famoso "Hello World!" realizzato con FLTK!

Riportiamo, a titolo di esempio, il codice della classica applicazione "hello word", che realizza una finestra, contenente una progress-bar animata, completa di richiesta di conferma alla chiusura. Il codice che segueva salvato in un file di nome `hello.cxx`, e compilato con il comando:

```
$ fltk-config --compile hello.cxx
```

Sorgente dell'applicazione di esempio:

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/fl_ask.H>
#include <FL/Fl_Button.H>
#include <FL/Fl_Progress.H>

Fl_Window * window;

// prima funzione "callback" nella main verrà associata alla finestra
void window_cb(Fl_Widget*,void*)
{
   if (!fl_ask("Non preferisci premere il bottone?"))
       window->hide();
}

// funzione "callback" per il bottone (viene chiamata al click)
void button_cb(Fl_Widget* button, void* arg)
{
   button->label("Bye bye, Word!");

   // realizza un conto alla rovescia con la progress-bar
   Fl_Progress* bar=(Fl_Progress*)arg;
   for (int i=10000; i>0 ; i--){
       bar->value(i);
       Fl::check();
   }
   window->hide();
}

int main(int argc, char **argv)
{
   window = new Fl_Window(300,170,"helloword");
   window->callback(window_cb);

   Fl_Progress *bar= new Fl_Progress(20,10,260,10);
   bar->maximum(10000);
   bar->minimum(0);
   bar->value(10000);

   Fl_Button *but = new Fl_Button(20,40,260,100,"Hello, World!");
   but->callback(button_cb,bar);

   window->show(argc,argv);
   return Fl::run();
}
```

zioni "sul tema" disponibili tra le proprietà configurabili di ogni oggetto. Se questo non fosse ancora sufficiente è possibile, facilmente, arricchire ulteriormente il ventaglio di oggetti disponibili scrivendo personalmente delle estensioni ad hoc.

A tal proposito il sito del progetto (`fltk.org`), riporta, nella sezione link, una nutrita varietà di oggetti di uso più o meno generico, frutto del lavoro degli utenti più entusiasti.

Per quanto riguarda le doti di leggerezza è necessaria una distinzione fra due aspetti differenti: il primo è sostanzialmente legato alla reattività del risultato finale, e parliamo di tempi di avvio delle applicazioni, velocità di refresh delle finestre, ecc., aspetto che può sicuramente essere annoverato tra i punti di forza del toolkit.

C'è però un altro aspetto legato alla leggerezza che merita attenzione: FLTK è pensato per una compilazione statica, cioè tale che l'eseguibile finale contenga anche il binario di tutte le funzioni di libreria utilizzate. In tal modo il codice prodotto non richiederà l'installazione della libreria per funzionare su una macchina diversa da quella su cui è stato compilato, non avrà quindi alcuna dipendenza da soddisfare, o almeno non a causa dell'uso di FLTK (si parla in questo caso di eseguibile stand-alone).

Tale caratteristica è potenzialmente sfruttabile con qualsiasi libreria ma, in pratica, è necessario fare i conti con la conseguente crescita in dimensione dell'eseguibile, che, ricordiamo, includendo la libreria ne acquista anche il peso.

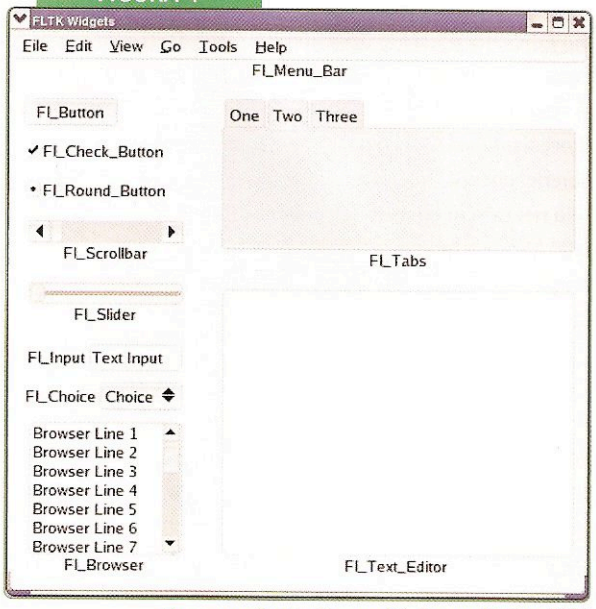
Da questo punto di vista FLTK si comporta egregiamente, "appesantendo" l'eseguibile finale al massimo di qualche centinaio di Kb, e facendolo in modo molto granulare, cioè includendo correttamente solo il binario relativo alle funzioni realmente utilizzate, caratteristica che denota, tra l'atro, una buona pulizia nella scrittura del codice.

Per farsi un'idea basti pensare che il programma **fluid**, che include ogni possibile elemento della libreria, pesa circa 500KB (tale cifra è naturalmente approssimativa, vista la quantità di differenti sistemi supportati).

Probabilmente FLTK non è adatto per applicazioni che richiedano interfacce dati complesse o laddove siano necessari componenti di alto livello (come widget di gestione database), e trova il suo campo di applicazione elettiva in sistemi embedded o comunque in applicativi leggeri (a smentire questa affermazione ci pensa EDE, un completo desktop costruito su una versione modificata della libreria).

E' inoltre esclusivamente indirizzato agli sviluppatori C/C++

FIGURA 1



Un esempio degli oggetti disponibili in FLTK (in questo caso con la finitura "plastic")

RIQUADRO 2

Il loop di esecuzione

Il codice di tutti gli esempi proposti si conclude con l'istruzione:

```
return Fl::run();
```

che richiama il loop principale di esecuzione. Tale loop è sostanzialmente un ciclo infinito di smistamento e gestione degli eventi (click, ridisegno, timer, ecc) ed è necessario a "tenere in vita" l'applicazione fino alla chiusura dell'ultima delle finestre create. Sistemi analoghi sono presenti in qualsiasi toolkit grafico. In particolare nell'FLTK può essere sostituito dal codice equivalente:

```
while ( Fl::wait() );
```

oppure da un qualsiasi ciclo custom che richiami periodicamente la funzione `check()` o `flush()`.

```
while ( condizione )
{
   // ... istruzioni ...
   Fl::check();
}
```

In questo ultimo caso il vantaggio è dato dalla possibilità di gestire la priorità di esecuzione delle istruzioni in funzione delle esigenze del progetto, evitando ad esempio, che eventi relativi al disegno dell'interfaccia interferiscano con operazioni di maggiore priorità, oppure sincronizzando con precisione il refresh della finestra con eventi particolari.

(anche se esistono dei progetti non ufficiali di estensione ad altri linguaggi). Ricordiamo infine che il toolkit funziona sotto Unix/Linux, Windows, MacOS e OS2, e che quindi le interfacce prodotte saranno esportabili senza alcuna modifica su tutti i sistemi operativi citati.

Una ultima nota è relativa all'ottima integrazione con le librerie OpenGL: utilizzando FLTK sarà possibile, con il minimo sforzo, godere di tutte le caratteristiche di questa splendida libreria grafica (stavolta inteso come libreria per la realizzazione di effetti grafici).

Download e installazione

Nonostante siano disponibili pacchetti precompilati per tutte le maggiori distribuzioni, essendo questo articolo orientato alla programmazione, consigliamo caldamente l'installazione dai sorgenti. Puntate quindi il browser su `www.fltk.org` e nella sezione software vi verrà data la possibilità di scegliere se scaricare la versione stabile 1.1.6 oppure una preview della prossima versione 2.0 (questo articolo è stato scritto basandosi sulla versione 1.1.6, anche se le indicazioni date non cambieranno di molto con il nuovo rilascio stabile).

Dopo aver scaricato il pacchetto dei sorgenti i passi da seguire sono i soliti `configure` / `make` / `make install`. Come sempre, al momento del `configure`, è possibile abilitare alcuni switch per modificare il risultato della compilazione. Fra questi i programmatori potrebbero essere interessati a `--enable-debug`, per compilare le informazioni di debug all'interno della libreria (e quindi in fase di debug poter entrare anche nelle funzioni di libreria), ed `--enable-shared`, per ottenere anche una versione a condivisa della libreria (cioè linkabile dinamicamente).

Facciamo notare come, in virtù di quanto già detto a proposito del link statico, gli eseguibili di esempio presenti nel pacchetto funzioneranno anche prima della installazione nel sistema. Sarà quindi possibile, dopo la compilazione, entrare nella sottodirectory `test/` per dare un'occhiata agli eseguibili presenti, decidendo in un secondo tempo se la libreria fa al caso nostro e quindi procedere all'installazione.

Va notato infine che il supporto necessario alle librerie OpenGL verrà abilitato solo nel caso in cui queste siano già installate nel sistema al momento della configurazione.

Oltre ai già citati esempi il pacchetto include la documentazione in formato HTML di tutte le classi e funzioni a disposizione, oltre ad un piccolo tutorial: si trova nella directory `documentation`, viene normalmente installato in `/usr/local/share/doc/fltk`, ed è raggiungibile anche da una voce di menu di FLUID.

L'installazione inoltre renderà il disponibili due nuovi strumen-

ti: `fluid` e `fltk-config`, dei quali parleremo più diffusamente nel seguito.

Studio degli esempi allegati

Una volta completate le necessarie fasi di download ed installazione possiamo immediatamente procedere al primo esperimento di applicazione del toolkit, che ci servirà anche a verificare la buona riuscita della fase di installazione. Come primo passo ci dedicheremo alla compilazione di uno degli esempi allegati. Per studiarlo più comodamente isoleremo l'esempio dalla affollata directory `test/`. Creiamo quindi una directory temporanea e copiamovi il file `symbols.cxx`:

```
$ mkdir directorytemporanea
$ cd directorytemporanea
$ cp percorso/per/i/sorgenti/fltk/test/symbols.cxx .
```

Procediamo quindi alla compilazione con il comando (attenzione agli apici inversi che si ottengono premendo AltGr insieme al tasto apice):

```
$ gcc `fltk-config --cflags` -o
            symbols symbols.cxx `fltk-config -ldflags`
```

E lanciamo l'eseguibile ottenuto:

```
$ ./symbols
```

E' interessante notare come il risultato sia ottenuto con sole 120 linee di codice C.
All'interno del file `symbols.cxx` possiamo già evidenziare alcune delle tecniche fondamentali per la programmazione con FLTK *(a questo proposito si veda anche il codice "Hello Word" nel riquadro 1)*. In particolare va notato come, tutti gli oggetti (bottoni) creati dopo l'istanza della finestra:

```
window = new Fl_Single_Window(width,height,"title");
```

vengano assegnati alla finestra stessa senza necessità di ulteriori specificazioni. Per creare una seconda finestra sarà necessario chiudere la costruzione della prima con l'istruzione:

```
window->end();
```

Tale istruzione non è necessaria in questo caso, visto che la finestra costruita è l'unica dell'applicazione.
Gli include iniziali inoltre possono sembrare un po' verbosi, si tenga comunque presente che tale granularità serve a rendere più efficiente il linking della libreria.
Per quanto riguarda una spiegazione delle funzioni di callback si veda il riquadro 3.
Procediamo ora alla compilazione di un esempio che utilizzi le librerie openGL (naturalmente questo esempio è applicabile solo nel caso in cui nel vostro sistema tali librerie siano installate). Creiamo quindi una nuova directory e copiamovi i due file sorgenti `config.h` e `cube.cxx` (quest'ultimo si trova nella sottodirectory "test"):

RIQUADRO 3

Le funzioni "callback"

Ad uso di quanti non avessero mai affrontato l'uso di un toolkit riportiamo una breve spiegazione del significato del termine callback. A differenza di quanto avviene in un codice "sequenziale", la programmazione di interfacce prevede che l'applicazione per la maggior parte del suo tempo, stia sostanzialmente inattiva, in attesa di eventi utente.
Tale compito è svolto da un ciclo che svolge la funzione di *pompa* dei messaggi, si occupa cioè di richiamare l'appropriata funzione al realizzarsi di determinati eventi di interfaccia (tipicamente "click" del mouse" o pressione di tasti).
Per tale motivo, prima di mettere in loop l'interfaccia, è necessario indicare il desiderato comportamento tramite l'associazione oggetto-evento-funzione.
Ad esempio la linea:

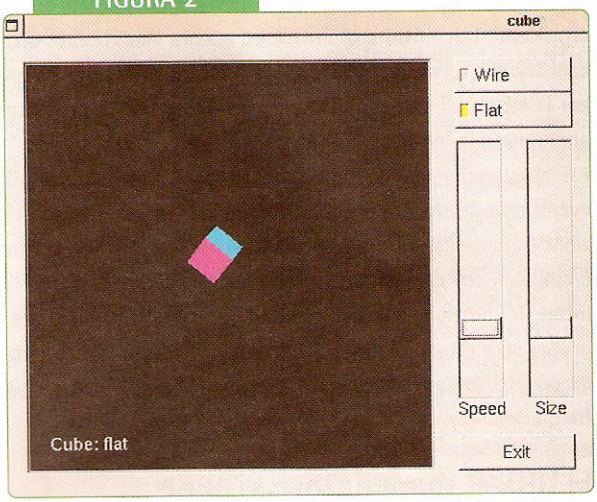
```
button->callback(cb_bottone);
```

indica alla *pompa* dei messaggi che la funzione `cb_bottone` dovrà essere chiamata alla pressione del bottone `button`. Per poter essere chiamata correttamente la funzione callback dovrà avere un prototipo prestabilito, che nel caso di FLTK, per gli eventi di interfaccia è:

```
void Fl_Callback(Fl_Widget* w, void* p);
```

dove all'argomento `w` viene passato il puntatore all'oggetto chiamante (quindi una stessa funzione può essere chiamata da più widget) mentre l'argomento `p` è a disposizione per il passaggio di dati, e va indicato al momento dell'associazione.
Le funzioni utilizzate a questo scopo vengono definite "callback" perché realizzano una chiamata "all'indietro", cioè possono essere utilizzate per "invertire" l'ordine gerarchico all'interno del codice (che richiederebbe che sia il codice runtime ad utilizzare delle funzioni di libreria, e non il viceversa). Questa tecnica viene utilizzata anche in altri casi di gestione eventi esterni, come ad esempio la ricezione di messaggi ethernet.

FIGURA 2



L'applicazione di esempio "cube", inclusa nei sorgenti fltkfig.

```
$ mkdir directorytemporanea2
$ cd directorytemporanea2
$ cp percorso/per/i/sorgenti/fltk/config.h .
$ cp percorso/per/i/sorgenti/fltk/test/cube.cxx .
```

Modifichiamo il file `cube.cxx`, sostituendo gli apici alle virgolette nella riga 28 (questo solo perchè abbiamo spostato il file config.h nella stessa directory):

```
#include "config.h"
```

Procediamo quindi alla compilazione impartendo il comando (sostanzialmente equivalente al precedente):

```
fltk-config --use-gl --compile cube.cxx
```

Il risultato ottenuto, visibile in figura 2, è quantomeno sorprendente per un sorgente di soli 5.4K. Naturalmente in questo caso parte del merito va alle librerie OpenGL, infatti, come è possibile vedere alla linea 53 del file `cube.cxx`, le funzionalità delle OpenGL sono disponibili semplicemente derivando la finestra principale dall'oggetto `Fl_Gl_Window`, e sovrascrivendone la funzione draw:

```
class myglwindow : public Fl_Gl_Window {
  void draw();
  [...]
}
```

Utilizzando la tecnica illustrata è possibile studiare comodamente gli esempi allegati, che coprono praticamente tutti gli aspetti dell'uso della libreria, e possono essere preziosi per fugare eventuali dubbi sull'uso delle funzioni e degli oggetti a disposizione.

Il comando fltk-config

Come visto dagli esempi di compilazione del precedente paragrafo, e come molti altri prodotti del genere, il tookit include un comodo comando (più esattamente è uno shell-script) per generare i flag necessari al compilatore ed al linker, che permette di semplificare notevolmente la scrittura delle istruzioni di compilazione con FLTK. Tale script viene normalmente installato nella directory `/usr/local/bin`, che dovrà quindi far parte del vostro PATH. Digitando semplicemente:

```
$ fltk-config
```

verranno elencate tutte le possibilità di tale comando. Si apprenderà quindi che all'interno del comando per il compilatore sarà possibile inserire semplicemente:

```
`fltk-config --ldstaticsflags -use-gl`
```

(anche qui attenzione gli apici inversi) per ottenere tutti i flag necessari al linking statico delle librerie, in questo caso includendo anche le OpenGL.
In un `makefile` invece si utilizzerà qualcosa di simile alla seguente forma:

```
CFLAGS=`fltk-config --cxxflags`
LDFLAGS=`fltk-config --ldstaticsflags`
```

per realizzare l'inclusione statica a codice C++.
Nei casi più semplici, ovvero dove non si preveda il linking con altre librerie ad esclusione dell'FLTK, sarà addirittura possibile scrivere semplicemente:

```
fltk-config --compile nomesorgente.c
```

per impartire il comando di compilazione completo. Infine il comando:

```
fltk-config --api-version
```

permette di verificare la versione del toolkit installata nel sistema in uso.

Le funzioni statiche

Oltre a tutti gli oggetti grafici, che vengono appunto trattati da codice come oggetti, FLTK prevede una classe statica ("Fl") per le funzioni globali. La comprensione di alcune di queste funzioni (si veda anche il riquadro 2) non è sempre immediata, ma è necessaria per approcciarsi correttamente ad alcune delle funzionalità disponibili.

Facciamo notare come una buona parte di queste funzioni accetti come argomento l'indirizzo di una ulteriore funzione definita da noi. Tale tipo di funzione, chiamata *callback function*, verrà appunto chiamata dal loop principale al verificarsi di desiderati eventi (riquadro 3). Per evitare incertezze ricordiamo che in C i puntatori a funzione vengono definiti con una sintassi che può apparire "strana", visto che è necessario specificare anche il prototipo della funzione stessa.

Ad esempio la forma:

```
int has_idle( void(*cb)(void*), void* );
```

Indica che la funzione `has_idle` accetta come argomento (oltre ad un `void*`) l'indirizzo di una funzione (`*cb`) che deve essere definita:

```
void nostra_cb(void*)
```

Le prime funzioni statiche che si renderanno necessarie sono `Fl::flush` ed `Fl:check`, che servono a passare momentaneamente l'esecuzione al ciclo principale per verificare la necessità di eventuali refresh degli oggetti grafici.

Tali funzioni vanno inserite periodicamente all'interno di cicli o di sequenze di codice particolarmente lunghe per evitare il congelamento della finestra.

Molto importante è anche `Fl::args` : questa funzione va chiamata ad inizio programma ed accetta gli stessi parametri tipici della `main` (gli argomenti della riga di comando), che verranno passati anche alla finestra principale; in questo modo è possibile accettare "in automatico" alcuni degli argomenti standard delle applicazioni X, come dimensioni e posizione iniziale, rendendo la nostra applicazione compatibile con le specifiche dei diversi window manager.

Le funzioni statiche servono anche nella gestione di eventi differiti o periodici (timer). Tramite `Fl::add_timeout` è possibile di impostare un timer "one-shot", allo scadere del quale verrà chiamata la funzione callback assegnata, `Fl::repeat_timeout` serve invece ad impostare un timer che verrà ripetuto indefinitamente. I timer possono essere cancellati prima dello scadere tramite `Fl::remove_timeout`.

Molto comoda può essere anche `Fl::add_fd`, che accetta come argomenti il descrittore di un file ed ancora l'indirizzo di una callback che verrà chiamata quando il file indicato sarà pronto per la lettura di dati, permettendo così di gestire al momento appropriato la lettura di dati in arrivo tramite pipe o socket.

`Fl::add_handler` aggiunge una callback a cui vengono passati eventi non gestiti dagli altri oggetti FLTK e, insieme alla funzione `Fl::event_key()`, che ritorna il codice dell'ultimo tasto premuto, viene utilizzata per gestire gli shortcut di tastiera.

Infine le funzioni `Fl::set_font`, `Fl::background` e `Fl:foreground` permettono di cambiare rapidamente l'aspetto di tutti gli oggetti dell'applicazione, modificando i colori principali, i font, ecc.

FLUID, il modellatore visuale

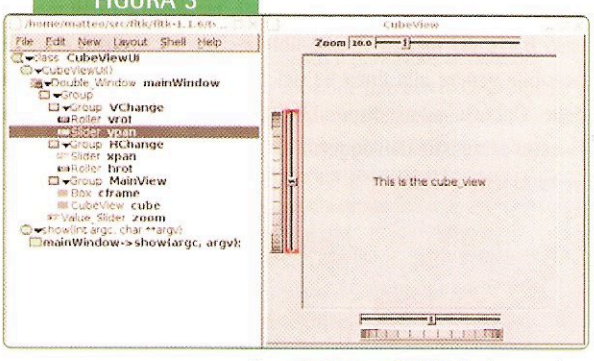
Abbiamo finora presentato alcuni esempi di utilizzo "diretto" del toolkit, cioè ottenuti scrivendo direttamente il codice necessario. Naturalmente tale sistema incontra i suoi limiti nel momento in cui si vogliano creare interfacce di una certa complessità (oppure se siete stati viziati dall'uso di "certi" ambienti visuali). In tale ed in altri casi, comunque, l'uso di uno uno strumento che permetta di creare l'interfaccia "a colpi di mouse" risulta estremamente comodo.

La distribuzione standard della libreria include FLUID *(Fast Light User Interface Designer)*, che è appunto un modellatore visuale di interfacce dedicato all'FLTK (figura 3).

In perfetto spirito con il resto del toolkit FLUID fa solo questo. Non aspettatevi quindi di potervi gestire un intero progetto come potreste fare ad esempio con kdevelop, perché FLUID non è pensato a questo scopo.

Le possibilità di editing del codice, infatti, sono limitate allo stretto indispensabile a linkare le funzionalità di codice esterno, non essendo dotato di alcun ausilio alla digitazione come

FIGURA 3



Il modellatore FLUID, incluso nei sorgenti della libreria FLTK

RIQUADRO 3

9 consigli per utilizzare al meglio FLUID

1 Per iniziare la costruzione di una finestra è necessario prima inserire una nuova funzione (new → code → function), oppure una nuova classe (new → code → class) ed un metodo della stessa. Il codice per costruire la finestra verrà quindi posto all'interno della funzione specificata. Si noti che in tale modo FLUID supporta sia il codice C che C++. La distinzione è data appunto dall'includere o meno oggetti di tipo `class`.

2 FLUID non crea autonomamente una nuova directory di progetto. Allo stesso modo non fornisce automaticamente l'estensione convenzionale "`.fl`" al proprio file. Si dovrà quindi provvedere manualmente.

3 Il tasto di cancellazione è `[Ctrl]+[x]`, cioè "taglia". La pressione del tasto `[Canc]` non sortisce alcun effetto.

4 A proposito di cancellazione: agite con attenzione, FLUID è privo di *undo* (anche se, alla pressione dei tasti `[CTRL][z]`, una ironica dialog ci informa che non è "ancora" implementato). Per fortuna (vedi punto precedente) è possibile ri-incollare immediatamente quanto cancellato. Vale comunque l'abitudine di salvare spesso.

5 I nuovi elementi vengono aggiunti in modo gerarchico, quindi quando si vuole inserire un elemento è necessario assicurarsi che nella finestra principale sia selezionata la corretta posizione od il gruppo di appartenenza. Ricordiamo che per gruppo si intendono tutti gli oggetti che ne possono contenere altri (finestre, tab, ecc).

6 L'ordine degli elementi nella finestra principale rispecchia l'ordine nel codice, quindi non è ininfluente. Per spostare gli oggetti è sufficiente selezionarli e premere poi i tasti `[F2]` o `[F3]`.

7 In alcune versioni il salvataggio del codice associato al comando *shell execute* (che serve ad eseguire un comando di shell) non funziona come dovrebbe, infatti salva i file nella home dell'utente e non nella cartella di progetto. E' quindi meglio salvare tramite l'apposita voce di menu prima di ogni compilazione e ricorrere ad un terminale per impartire i comandi al compilatore. Tale sistema risulta comunque più pratico anche perché è possibile visualizzare più comodamente l'output di eventuali errori.

8 Il pulsante "resizable" (nella scheda GUI della finestra "proprietà"): serve a definire se l'oggetto sarà ridimensionabile dall'utente oppure no. FLTK gestisce il ridimensionamento in modo quasi completamente automatico, ma non sempre intuitivo. Tramite questo flag ed un attento uso dei gruppi è possibile definire facilmente qualsiasi comportamento desiderato.

9 Caselle "extra code" (nella scheda C++ della finestra "proprietà"): è possibile inserire del codice aggiuntivo per l'inizializzazione dell'oggetto, eseguito quindi al momento della sua creazione. In tale codice ci si riferirà all'oggetto tramite il puntatore `o`, a prescindere dal nome da noi assegnatogli. Quindi per definire il valore iniziale, ad esempio ad una progress-bar, sarà necessario inserire in "extra code" `o->value(100)`.

autocompletamento ed evidenziazione sintattica del codice. FLUID è quindi pensato per affiancare altri strumenti più votati all'editing ed alla gestione del progetto: potrete quindi continuare ad usare il vostro ambiente preferito, sia esso vi, Eclipse oppure un semplice editor ed una shell.

Proprio a causa dell'essenzialità delle funzioni, lo sviluppo in FLUID è estremamente veloce. Oltretutto le interfacce sviluppate in questo modo sono molto "maneggevoli" e non introducono alcuna complicazione legata all'ambiente di sviluppo (cioè non vi riempirà la directory dei sorgenti di misteriosi file utili solo ad esso).

Il file generato da FLUID è un semplice file di testo che, tramite lo stesso FLUID, può essere "tradotto" nei due canonici file di codice C (.c e .h) o C++ , ed è agevolmente modificabile anche direttamente. Tale possibilità si dimostra preziosa, ad esempio, per allineare rapidamente una serie ripetitiva di elementi, o per imporre posizioni e dimensioni in modo calcolato (per non parlare della possibilità di automatizzare la creazione di interfacce, successivamente modificabili in modo visuale).

Il codice generato risulta inoltre piuttosto pulito, chiaro quanto basta per potere a sua volta essere editato direttamente (al limite anche solo per farsi suggerire una soluzione complessa). Citiamo inoltre la possibilità di richiamare l'agile manuale della libreria in formato html (incluso nel pacchetto dei sorgenti), la possibilità di memorizzare un comando da passare alla shell (ad esempio il comando di compilazione) e la possibilità di esportare tutte le stringhe dell'interfaccia in un file di testo, utile per l'eventuale traduzione del progetto.

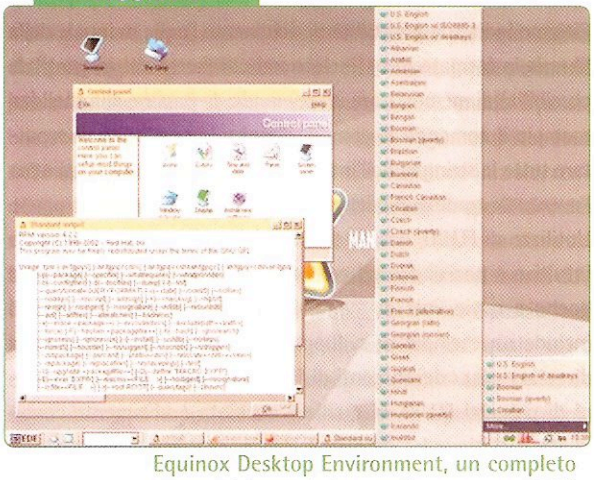
L'utilizzo di FLUID, basato sull'uso del mouse, è estremamente intuitivo e non ci dilungheremo quindi nella spiegazione delle poche e ovvie voci di menu. Il tutorial allegato ai sorgenti include un esempio di sviluppo passo passo, ed inoltre nella directory `test/` dei sorgenti di FLTK sono presenti alcuni esempi di progetti FLUID, ai quali un'occhiata è vivamente consigliata. Nel riquadro in queste pagine potrete trovare alcuni consigli per utilizzare FLUID al meglio.

Alcune linee guida generali

L'esempio visto è stato svolto interamente all'interno di FLUID. Al crescere delle esigenze del progetto, gli spazi offerti dallo strumento iniziano a rivelarsi particolarmente angusti, e si cercherà la via più veloce - qua riporteremo alcune tecniche - per risolvere i problemi legati all'interfaccia conservando la possibilità di lavorare con l'ambiente di sviluppo preferito. Il codice scritto in questo modo permetterà il riutilizzo delle interfacce realizzate, possibilità che viene spesso negata utilizzando ambienti di sviluppo integrati.
In prima fase ci dovremo dedicare alla creazione della finestra ed al posizionamento dei vari oggetti tramite FLUID. L'unica accortezza necessaria è considerare all'inizio se il progetto dovrà essere scritto in C o in C++, inserendo quindi la costruzione della nostra finestra all'interno del metodo di una classe (di solito nel costruttore) oppure in una semplice funzione.
Al termine del design avremo quindi una classe, oppure una funzione, che realizza una finestra priva di funzionalità.
Per aggiungere le desiderate funzionalità procederemo quindi come segue. Nel caso di codice C:

- in FLUID andrà costruita una sola funzione (`make_window` è il nome di default) nella quale andrà posta la finestra e tutti gli oggetti necessari;
- ogni oggetto che deve essere utilizzato dall'esterno, finestra principale compresa, deve essere definito `public` anche se si sta lavorando in linguaggio C. E' necessario inoltre che gli venga assegnato un nome tramite la scheda "C++";
- il `.h` scritto da FLUID deve ovviamente essere incluso dal codice che utilizzerà la finestra;
- per visualizzare la finestra sarà sufficiente richiamare la funzione di costruzione (cioè quella sviluppata in FLUID) e successivamente la funzione `show` della finestra. Naturalmente per comodità la funzione `show` potrà essere inserita direttamente nella funzione di costruzione della finestra. Questo a meno che non siano necessarie particolari inizializzazioni da eseguirsi tra l'istanza della finestra e la sua visualizzazione;
- Le callback potranno essere realizzate esternamente, ovvero in altri moduli di codice, con un prototipo arbitrario (anche se quello prestabilito è adatto alla maggioranza dei casi). Per ogni callback utilizzata andrà aggiunta una dichiarazione `extern` all'interno di FLUID, posta prima della funzione della finestra. Nella casella `callback` dell'oggetto si potrà a questo punto inserire semplicemente la chiamata alla funzione esterna, evitando così di inserire il codice nel poco spazio a disposizione.

FIGURA 4



Equinox Desktop Environment, un completo ambiente desktop sviluppato con FLTK

Nel caso di codice C++ invece le strade possibili sono molte. In particolare andrà deciso a priori se la classe che rappresenta la finestra debba essere "madre" o "figlia" del resto dell'applicazione, vale a dire in quale posizione debba stare nella gerarchia degli oggetti.
Il primo caso sarà preferibile per la classica finestra primaria del progetto, che è di solito strettamente legata alle funzionalità dello stesso e quindi non presenta solitamente la necessità di essere riutilizzata: realizzeremo le funzionalità tramite classi esterne ed il codice aggiunto in FLUID si limiterà alla istanza delle stesse ed alla chiamata di funzioni delle classi istanziate.
Il secondo caso invece (finestra "figlia") sarà preferibile per tutte quelle finestre che rappresentano funzionalità più generiche o secondarie: sarà il codice esterno ad istanziare la finestra, che sfrutterà ancora il meccanismo delle callback-function per notificare gli eventi. In tal caso la finestra realizzata sarà anche facilmente riutilizzabile in altri progetti.

Conclusione

Per farvi un'idea dei risultati ottenibili con FLTK potete dare un'occhiata a qualcuna delle applicazioni linkate nella categoria software del sito ufficiale del progetto e, tra queste, spicca addirittura un completo ambiente desktop: l'Equinox Desktop Environment (`http://ede.sourceforge.net`) con una occupazione di ram complessiva inferiore a quella del solo xterm.
Questo dimostra, se ancora ce ne fosse bisogno, l'efficienza e la leggerezza del toolkit.