Alma Mater Studiorum
Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

CORSO DI LAUREA IN INFORMATICA

XCSurf:
riprogettazione dell'interfaccia utente

Relatore:
Chiar.mo Prof. Giulio Casciola

Tesi di Laurea di:
Matteo Galisi

Anno Accademico 2009-2010

Indice

Introduzione **5**

1 Panoramica **7**

1.1 Cenno storico su XCModel 7

1.2 Problematiche 11

1.2.1 Il *backing-store* 11

1.2.2 Il ciclo degli eventi 13

1.2.3 *Pipeline* e *double-buffering* 14

1.2.4 Problemi legati alle operazioni pesanti 15

2 Grafica Interattiva **17**

2.1 X-Window System. 17

2.1.1 L'interfaccia. 19

2.1.2 Cenno storico. 19

2.1.3 X-Org ed XFree86. 21

2.1.4 Il modello *client/server*. 23

2.2 Le librerie. 25

2.2.1 La gestione delle finestre. 27

2.3 Paradigmi ed ambienti per la programmazione grafica. 30

2.3.1 Grafica accelerata con *Direct Rendering*. 30

2.3.2 OpenGL. 32

2.4 Le entità fondamentali della grafica interattiva. 36

2.4.1 Il *backing-store* in **X-Window** ed in ambiente **MicroSoft**. 37

2.4.2 La *pipeline grafica*. 38

2.4.3 Le PixMap. 39
2.4.4 Il *double buffering*. 40
2.4.5 Grafica interattiva ad eventi. 41
2.4.6 Coda e ciclo degli eventi. Interrogazione e *call-back functions*. Le **GUI**. 43

3 XCModel **47**
3.1 La *suite* XCModel. 47
3.2 La Libreria XTools. 50
3.2.1 Funzioni XTools. 51
3.2.2 Il progressivo sviluppo di XTools. 54
3.2.3 Le fasi risolutive. 56
3.2.4 Le **PixMap** nei pacchetti di **XCModel**: il *disallineamento* di **XCSurf**. 59
3.3 La riprogettazione dell'interfaccia utente in **XCSurf**. 61
3.3.1 Interventi. 66
3.4 Conclusioni. 68
3.4.1 Gli sviluppi futuri. 69

Introduzione

Il **Progetto XCModel**, che da dieci anni si sviluppa e si perfeziona col contributo di varie persone, consente la progettazione di scene **3D** e la resa fotorealistica. Esso si appoggia sulle funzionalità grafiche di **X-Window** in **Linux**: si tratta di un ambiente in costante evoluzione, dovuta sia alle continue innovazioni *hardware* che ai cambiamenti degli *standard*.

XCModel è composto da vari pacchetti, ognuno dei quali specializzato nel trattare un particolare aspetto della modellazione o della resa (curve, superfici, *textures*, luci, ...) e che si presentano come applicazioni dotate di una sofisticata interfaccia grafica. Tutte attingono alle funzioni della **Libreria XTools**, che fornisce numerose primitive grafiche facenti uso delle potenzialità di **X-Window**.

I recenti cambiamenti nelle specifiche di **X-Org**, ed in particolare la *destandardizzazione* del *backing-store*, ha obbligato il *team* di **XCModel** ad una consistente riprogettazione di **XTools** e di tutti i pacchetti che su di esso s'appoggiano. S'è colta l'occasione per un massiccio *debugging* e per modificare i pacchetti in base alle impressioni emerse nel corso degli anni da parte degli utilizzatori, primi tra tutti gli studenti del corso di **Grafica** tenuto dal Prof. Giulio Casciola, *project-manager* di **XCModel**: questa è un'ulteriore testimonianza del continuo adattamento e dell'aggiornabilità del Progetto!

Nella fattispecie, ci si occuperà delle modifiche effettuate in particolare

al pacchetto **XCSurf**, finalizzato alla creazione ed alla modifica di curve e superfici **3D** da utilizzarsi successivamente per la creazione delle scene: tale pacchetto è di fatto risultato il più avulso dalle specifiche di **XTools**, il che ha costretto ad un massiccio intervento al codice del programma, ma d'altro canto ha consentito una *standardizzazione* del pacchetto in linea con gli altri, in particolare ridisegnandone l'interfaccia.

Nel primo capitolo si effettuerà una veloce panoramica dell'ambiente **XCModel** e delle problematiche emerse dai cambiamenti degli *standard* di **X-Window**.

Il secondo capitolo affronterà un'ampia analisi dell'*hardware*, del *software* e dei paradigmi che caratterizzano la **grafica interattiva**, e gli ambienti necessari per interagire con essa ed esaminando i vari obiettivi raggiungibili.

Infine, il terzo capitolo analizzerà nel dettaglio le modifiche effettuate ai pacchetti di **XCModel** ed in particolare ad **XCSurf** per l'adattamento ai nuovi *standard*, nel rispetto delle politiche e delle linee guida dettate dalla grafica interattiva.

Capitolo 1

Panoramica

1.1 Cenno storico su XCModel

Da circa dieci anni, all'Università di Bologna esiste una *suite* di programmi denominata **XCModel** e supervisionata dal Prof. Giulio Casciola (*Fig. 1.1*).

Figura 1.1: XCModel.

Essa include una vasta serie di strumenti per la progettazione di ambienti tridimensionali fotorealistici, partendo dalla definizione di curve **NURBS** da utilizzare per la creazione di superfici che a loro volta andranno collocate nella scena.

I principali programmi di cui **XCModel** si compone sono:

- **XCCurv** (*Fig. 1.2*), per la creazione, gestione e modifica di curve **NURBS** bidimensionali

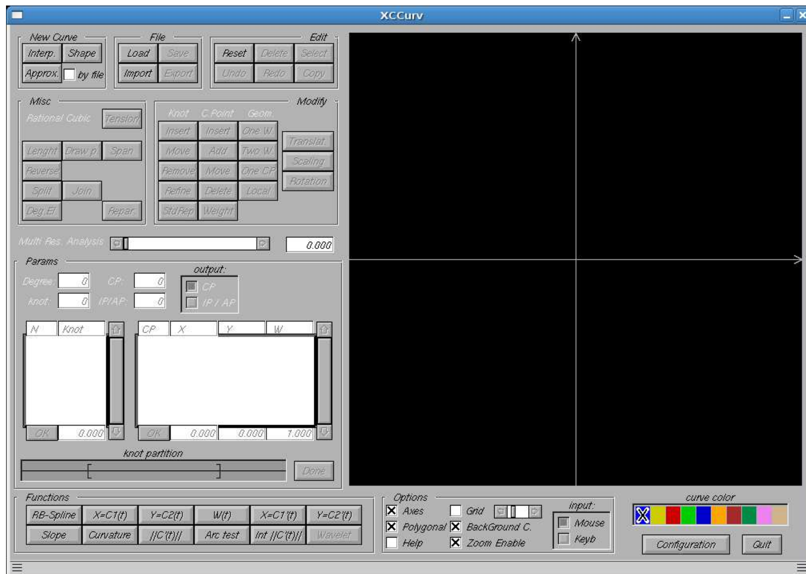

Figura 1.2: XCCurv.

- **XCSurf** (*Fig. 1.3*), per la creazione, gestione e modifica di curve **NURBS** tridimensionali e di superfici, sia *ex novo* che a partire da curve e superfici esistenti, mediante strumenti dedicati come *skinning*, *swinging*, *tubular*, *ecc.* (*cfr.* [**XCSURF**])

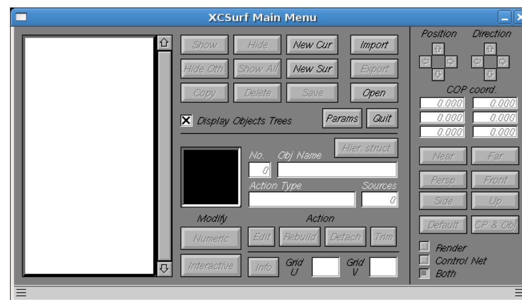

Figura 1.3: XCSurf.

- **XCBool** (*Fig. 1.4*), per effetturare operazioni *booleane* sulle superfici (unione, intersezione, differenza, *ecc.*)

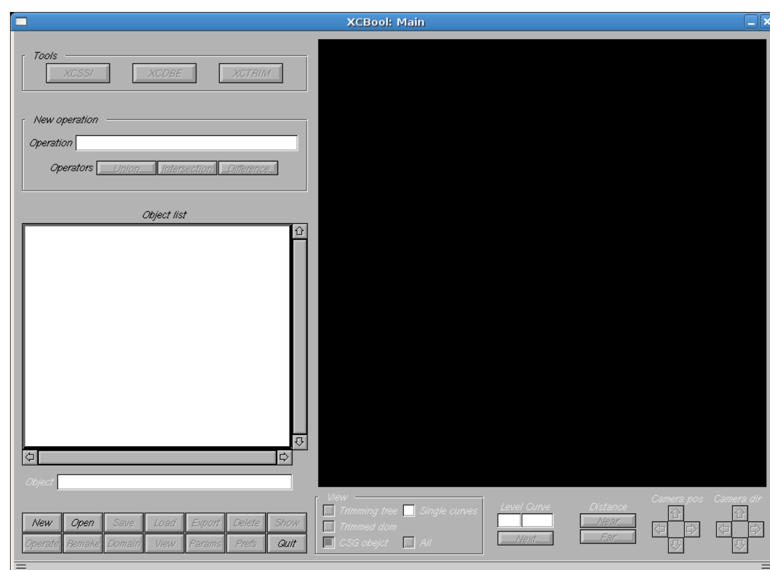

Figura 1.4: XCBool.

- **XCRayt** (*Fig. 1.5*), per l'inserimento degli oggetti creati in una scena, con la possibilità di colorarli, *texturarli* ed effettuare un *rendering* fotorealistico tramite **ray-tracing**

- **XCView** (precedentemente denominato **XMovie**, *Fig. 1.6*), per la definizione di filmati consistenti di varie scene già rese e mostrate in sequenza, nonché per la conversione di **textures** da utilizzare in **XCRayt**

Ognuno dei suddetti elementi sfrutta le potenzialità di **XTools**, una libreria che s'avvale delle funzionalità di **X-Window** per fornire metodi veloci ed efficaci finalizzati alla creazione d'un sistema a finestre completo ed efficace.

XCModel è definito come un **aCADemic modeling/rendering system**, ossia un sistema di **Computer Aided Design** a livello universitario: rivolto *agli* studenti e gestito *dagli* studenti.

XCModel s'appoggia sul Sistema Operativo **Linux**, s'avvale della sua

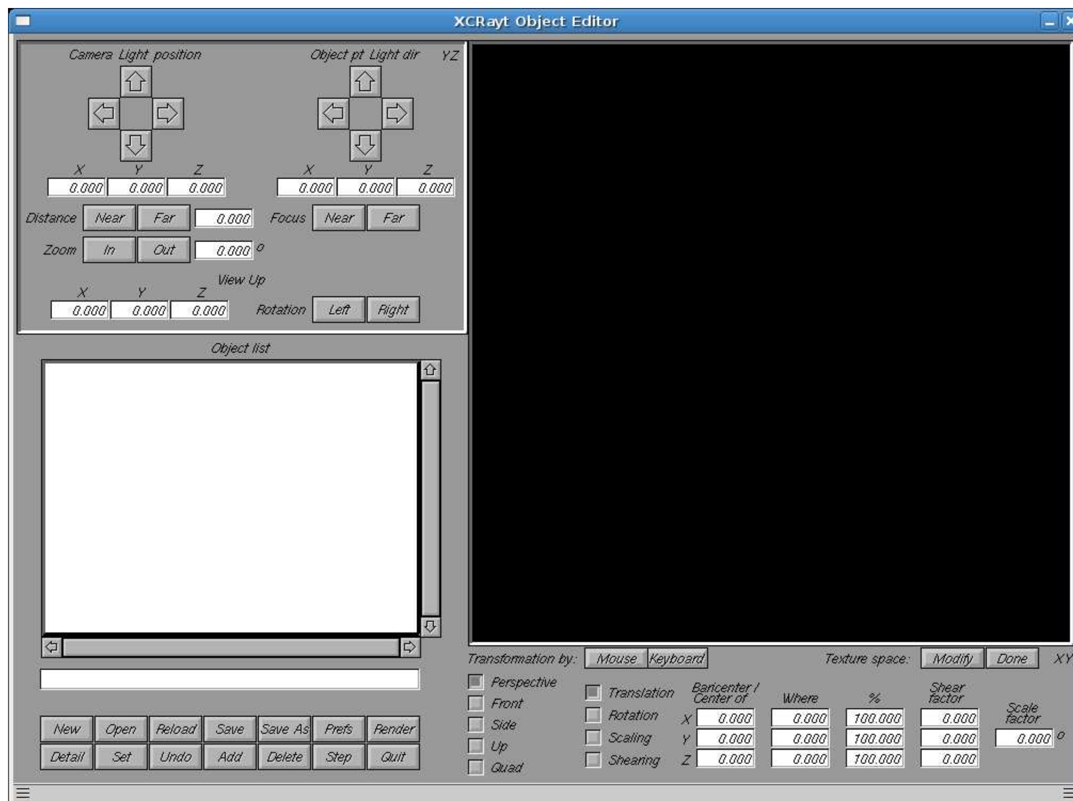

Figura 1.5: XCRayt.

potenza e versatilità e ne segue gli sviluppi e le modifiche. Ciò comporta vantaggi e svantaggi, riassumibili nei due punti seguenti:

- **XCModel** è sempre aggiornato ed in linea col progresso dell'*hardware* e del *software*

- qualora vengano fatte delle scelte implementative dipendenti dalle funzionalità offerte e garantite dal Sistema Operativo, occorrerà un notevole sforzo di modifica qualora dette funzionalità dovessero cambiare.

Quest'ultima ipotesi s'è verificata quando sono cambiati gli *standard* di **X-Window**, definendo una serie di *problematiche* da analizzare e superare: vediamole!

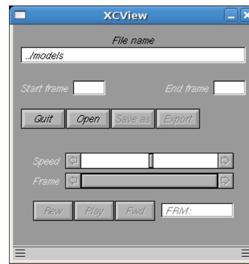

Figura 1.6: XCView.

1.2 Problematiche

1.2.1 Il *backing-store*

Figura 1.7: Una finestra ne copre un'altra: è necessario preservare le parti nascoste per ridisegnarle quando ridiverranno visibili.

Storicamente, **XCModel** decise di basarsi sul *backing-store* di **X11**. Il *backing-store* consente l'automatica memorizzazione (da parte del sistema **X-Window** dei contenuti delle finestre affinché, in caso di copertura delle stesse

da parte di altri elementi (finestre diverse, bordi, ...), esse possano essere correttamente ripristinate (cioè ridisegnate) quando la finestra tornerà visibile (*Fig. 1.7*). Avere la garanzia che sia il sistema a finestre utilizzato ad occuparsi di questo delicato e frequente aspetto consente di adottare una politica particolarmente snella nella gestione delle finestre. **X11** dapprima caldeggiò considerevolmente l'uso del *backing-store* da parte dei programmatori, poi però cominciò a sconsigliarlo (a seconda dei tempi e delle mode).

XCModel nacque quando **X11** consigliava il *backing-store*: vi si basò consistentemente, anche in sede di cambiamenti massicci. Due o tre anni fa, **X-Org** dichiarò che il *backing-store* non avrebbe più costituito lo *standard*. Apparentemente, ciò può sembrare un vincolo di poco conto: sarebbe stato sufficiente elencare tra i requisiti di configurazione dell'**X-Window** per l'utilizzo di **XCModel** l'attivazione del *backing-store*, ma ci fu una **differenza chiave**: dapprima, **X-Org** attivava il *backing-store* già prima del *mapping* delle finestre (non solo, quindi, dopo averle effettivamente mostrate a schermo), mentre dalla versione 7.1.3 la memorizzazione è stata subordinata alla visualizzazione. Per la precisione va specificato, come in [**TRON07**] e [**CLEM08**], che le indicazioni di configurazione in merito al *backing-store* sono dei *suggerimenti* (*hints*): il **Server** non è vincolato a rispettarle. Bisogna tener conto di ciò e dell'esatto comportamento del **Server** al fine di poter prevedere la situazione e l'eventuale bisogno di un *redraw* a carico dell'applicazione. Se la memorizzazione è svolta dal *sistema grafico*, infatti, il **Server** non segnala eventi di **Expose** (i quali permetterebbero all'applicativo di conoscere la necessità del ridisegno). A livello di configurazione, il *backing-store* può essere attivato in tre modalità:

- **NotUseful** (il *default*) segnala all'**X Server** che non è necessario conservare i contenuti
- **WhenMapped** richiede il salvataggio delle immagini presenti in finestre già *mappate* (e non necessariamente in quelle create ma non attive)
- **Always** domanda di preservare tutti i contenuti delle finestre, attive ed

inattive (notare come ciò, se rispettato, comporti la totale assenza di eventi **Expose**)

Come conseguenza di quanto osservato, **XCModel** smise di funzionare e richiese un massiccio *refactoring*, specialmente **XCSurf**, il cui stato d'aggiornamento era ritardato rispetto agli altri pacchetti.

Per prima cosa, è stata elaborata una *patch* per **XTools** (*cfr.* [**PREM10**]): l'aspetto apparentemente prioritario sembrava essere la gestione del *refresh* delle finestre in caso di *expose*, ossia quel che prima veniva demandato all'**X-Window** grazie al *backing-store*, ma non fu l'unico, come si vedrà immediatamente.

1.2.2 Il ciclo degli eventi

XTools non permette solo l'elaborazione e la gestione d'interfacce grafiche, ma anche l'interazione *ad-hoc* col sistema a finestre. Infatti, il lavoro col *software* con esso creato (**XCModel** *in primis*) permette, tipicamente, di rapportarsi direttamente con gli oggetti *resi*, ruotàndoli, traslàndoli, ingrandéndoli e modificàndone attributi e proprietà: per questi motivi, bisogna prevedere che le periferiche (tastiera e *mouse*) possano agire sulle immagini contenute nelle varie aree, non solo con elementi fissati a priori (bottoni, barre, ...). Operazioni come queste richiedono un ciclo degli eventi dedicato, che di fatto *impedisce* il *refresh* della finestra, qualora esso non sia a carico dell'**X Server**.

Per questi motivi, la versione *patchata* di **XTools** è dotata d'un sistema che permetta il corretto *refresh* delle finestre pur mantenendo la possibilità di catturare gli eventi di *input*: `ProcessGUIEventsUntil`. Questa funzione, dal nome esplicativo, coordina un unico ciclo degli eventi che catturi sia le mosse dell'utente sia eventuali *expose* che rendano necessario il ridisegno della finestra interessata. A dire il vero, già precedentemente erano presenti dei metodi legati all'*expose*, detti `SetWindowEXScript`, che avrebbero permesso di collegare ad un tale evento una funzione dedicata (*i.e.* il *redraw*), ma non era necessario utilizzarli.

Ci fu anche l'utilizzo di **GUI** multiple, ossia un sistema gerarchico di gestione dell'*input* e del *refresh*, ma questo enfatizzò l'annidamento dei cicli degli eventi, con mancato aggiornamento degli esterni quando in esecuzione i più interni, quindi s'è deciso di limitarsi ad una sola **GUI**.

1.2.3 *Pipeline* e *double-buffering*

Cercando una soluzione ai sovraccitati problemi, è lecito chiedersi come si comportino altre librerie grafiche, quale ad esempio **OpenGL**. Dette librerie sono dotate di *pipeline* (*Fig. 1.8*): l'operazione di ridisegno avviene in continuazione (cioè ad ogni fotogramma) ed il problema del *refresh* viene automaticamente scavalcato.

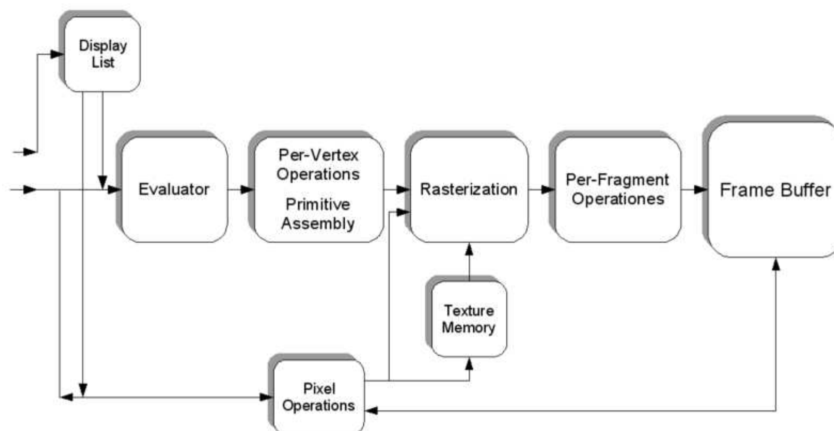

Figura 1.8: Schema operativo della *pipeline*.

Inoltre, il *rendering* **3D** sfrutta l'accelezione grafica: non è possibile che **XCModel** possa avvalersi di tale facilitazione *hardware*. Considerando dunque che **XCModel** nasce con certi obiettivi:

- è un progetto accademico che prevede la stesura di **algoritmi** da parte dei suoi sviluppatori
- dev'essere facilmente gestibile e modificabile dagli studenti

- non deve richiedere specifiche risorse *hardware*

s'è optato per una soluzione esclusivamente *software*.

Tipico delle applicazioni **3D** è anche il *double-buffering*, ossia l'uso contemporaneo di due *buffer*:

- uno per ospitare il fotogramma correntemente visualizzato
- l'altro per il disegno del successivo, onde non dover assistere alla composizione progressiva della scena in fase di *rendering*.

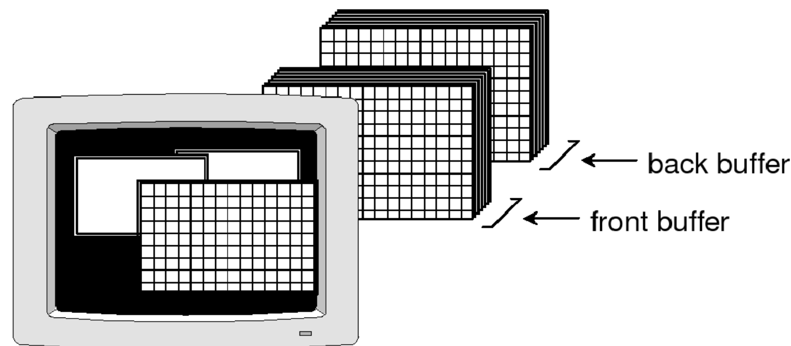

Figura 1.9: Il *double-buffering*.

Tale filosofia è stata in qualche modo ereditata anche nel nuovo **XCSurf** grazie all'uso di **PixMap** per la resa dell'oggetto e per i punti di controllo: tutte le operazioni di disegno avvengono su **PixMap** e solo alla fine vengono copiati nella finestra in uso. Infatti, ci si avvale di facilitazioni come la memorizzazione d'un modello già reso da usarsi in caso di *redraw*, senza doverlo ricalcolare. Due livelli d'aggiornamento: **ricalcolo solo se necessario**.

1.2.4 Problemi legati alle operazioni pesanti

Nella filosofia ormai *standard* di *backing-store* a carico dell'applicazione e di utilizzo di un'unica **GUI**, sorge un problema. Supponiamo che l'unico processo in esecuzione debba effettuare calcoli complicati e pesanti (con massiccio

uso di risorse di sistema): se detto processo deve occuparsi di tutto, potrebbero esserci ritardi nel *refresh* di una finestra (ciò risulta evidente quando vengono avviati numerosi processi pesanti e si cerca di spostare, ridurre e ripristinare le finestre: gli aggiornamenti ritardano e restano scie e colori sbagliati). Ciò non accadrebbe se il *backing-store* fosse gestito dall'**X-Server**, un processo parallelo chiamato ogni frazione di secondo all'occorrenza. Si può ovviare ricorrendo alla nuova funzione `ProcessGUIGraphicEvents`, che ad intervalli regolari (e *regolabili*) abbandona eventuali processi di calcolo eccessivamente pesanti e, se necessario, si occupa degli aspetti grafici, tra cui il *refresh*.

Per finire, una considerazione: se i comportamenti corretti da adottare in sede di grafica **2D** e **3D** sono quelli or ora citati, per quale motivo è nato il concetto di *backing-store*? Semplicemente, perché in passato si pensava che la grafica fosse per definizione solo 2D, dovendo apparire sullo schermo di un terminale, e si dava per scontato che un'operazione imprescindibile del *rendering* fosse la conversione (prospettica od assonometrica) in due dimensioni prima della visualizzazione.

Al contrario, il futuro della **Computer Graphic**, o quantomeno di **X-Org**, è il **Direct Rendering 3D**, che dà per scontate l'accelerazione grafica e la possibilità di *bypassare* il *software*. Il *backing-store*, pertanto, è annunciato come in via di sparizione.

Capitolo 2

Grafica Interattiva

2.1 X-Window System.

L'**X-Window System**, noto anche come **X11** o semplicemente **X**, è il gestore grafico *standard* dei sistemi **Unix** (*Fig. 2.1*, *cfr.* [**JON89**]).

Figura 2.1: Il *logo* di **X-Window**.

Normalmente, viene supportato da tutti i produttori di *hardware* grafico, ed utilizzato secondariamente anche da altri Sistemi Operativi (come ad esempio **Mac OS X**) per l'utilizzo di programmi nati esplicitamente per esso, come i noti **OpenOffice.org** (*Fig. 2.2*) e **GIMP** (*Fig. 2.3*).

X-Window, come successivamente si vedrà, nacque una trentina d'anni fa e fu un progetto assai lungimirante, tant'è che, appunto, viene utilizzato

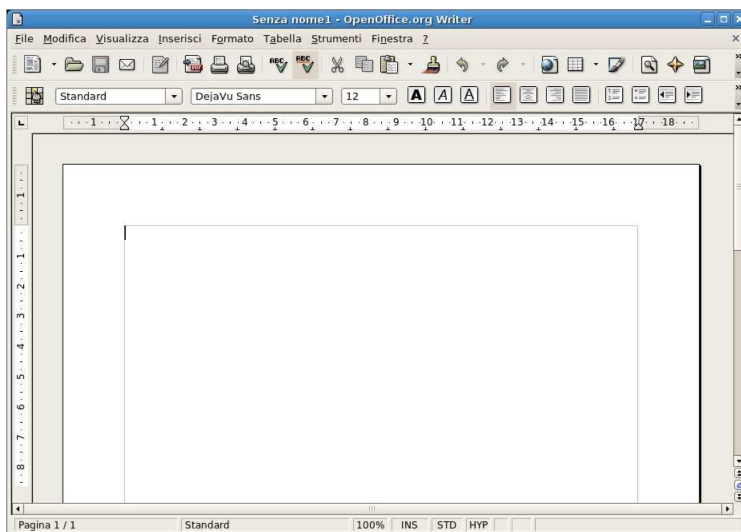

Figura 2.2: **OpenOffice.org Writer**.

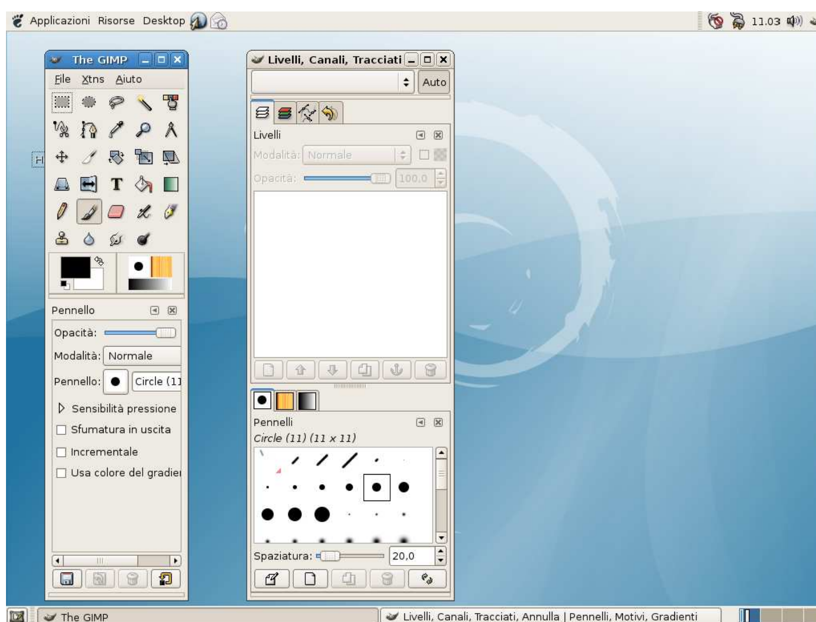

Figura 2.3: The **GIMP**.

ancor oggi, sebbene sia supportato da librerie che agiscono ad alto livello, lasciando ad **X-Window** il minimo indispensabile.

X fornisce l'ambiente e le funzionalità necessari alla creazione di programmi grafici interattivi, consentendo la creazione di interfacce grafiche e la gestione di finestre, dell'*input* e dell'*output*. Tuttavia, non definisce come primitive grafiche i classici elementi d'interfaccia quali i pulsanti, le caselle, le barre di scorrimento, *ecc.*, bensì delega tali oggetti ai singoli applicativi, qualora ne necèssitino.

Un aspetto assai peculiare di **X**, che lo differenzia da altri gestori grafici, è il **modello client/server**, che conferisce all'ambiente la *trasparenza di rete* necessaria per utilizzare comodamente i singoli pacchetti di un certo applicativo anche su *host* differenti.

2.1.1 L'interfaccia.

Dal momento che, come osservato, **X** non fornisce elementi per la creazione d'interfacce utente (se non punti, segmenti e colori), la loro implementazione concede completa libertà agli sviluppatori, e ciò si nota nelle profonde differenze delle svariate interfacce **X** realizzate nel corso del tempo e con varie risorse *hardware*.

La gestione delle finestre (controllo, posizionamento, visualizzazione) è competenza del *software* che s'appoggia su **X**, detto **Window Manager**. Esempi famosi in merito sono:

- **KWin** (per **KDE**)
- **MetaCity** (per **GNOME**)
- **Window Maker**

2.1.2 Cenno storico.

I primi approcci al sistema grafico che col tempo portò ad **X** risalgono agli anni ottanta del secolo scorso, all'Università di Stanford, ove Brian Red e Paul

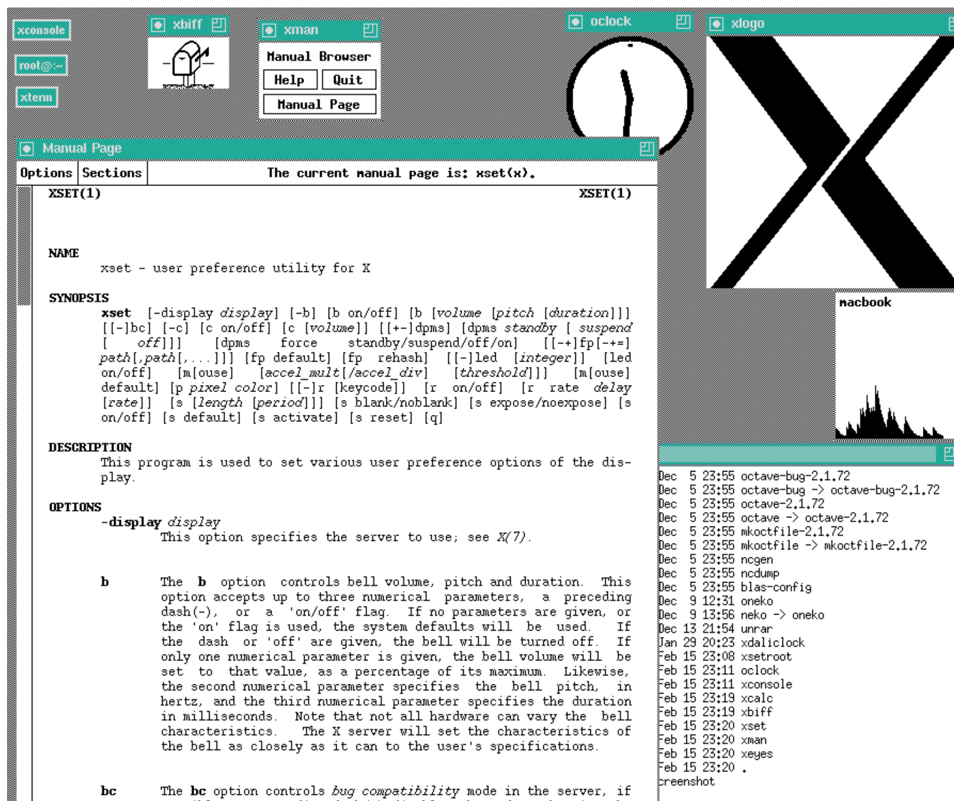

Figura 2.4: Il **Window Manager TWM**: come tutti, esso implementa autonomamente gli strumenti per la creazione di un'interfaccia.

Ascente realizzarono un'interfaccia chiamata **W** (da **Window**). A riguardo, *cfr.* [**XWINDOW**].

Successivamente, dopo i progressivi avanzamenti, il sistema cominciò a chiamarsi **X** (secondo l'usanza di denominare le successive evoluzioni di un programma con lettere consecutive dell'alfabeto, come per il **Linguaggio C**). Un aspetto fondamentale, apportato nel 1984 da Bob Scheifler e da Jim Gettys, fu il passaggio da un protocollo sincrono quale era **W** ad uno asincrono. Si enfatizzò sulla portabilità in varie piattaforme e perciò **X** vide numerosi miglioramenti nelle sue versioni successive, tant'è che nel 1987, all'undicesima versione del sistema asincrono, cioè **X11**, le varie aziende che ne facevano uso, tra cui la Sun MicroSysems, la IBM e la Hewlett Packard, si riunirono

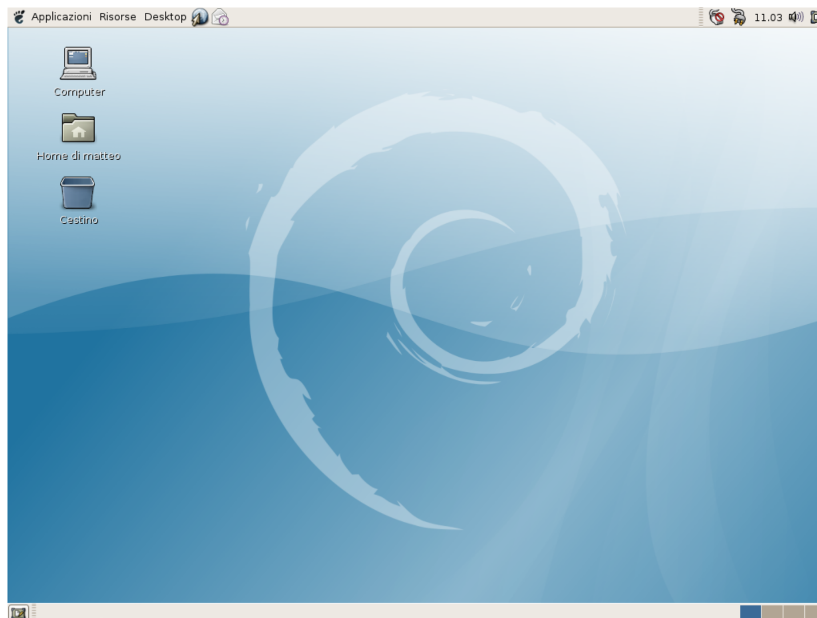

Figura 2.5: **GNOME DeskTop** per **Debian Linux Etch**.

nel consorzio **X-Org** (*Fig. 2.6*) per definirne le specifiche.

2.1.3 X-Org ed XFree86.

Negli anni successivi, il cuore di **X11** non subì significativi miglioramenti, a parte l'aumentata portabilità sulle macchine Sun, e pertanto il consorzio **X-Org** cominciò ad indebolirsi. Al tempo della *release* 6.6 di **X** subentrò **XFree86** (gioco di parole basato su **libero** e su **X386**, **X Three Eight Six**), che auspicava di migliorare significativamente **X** in un breve tempo, ma anche qui i risultati lasciarono a desiderare, pertanto le aziende bloccarono i finanziamenti ed il progetto dovette continuare solo grazie a volontari (*cfr.* [**XORG**]). Un altro problema era legato all'inadeguatezza di **X** agli usi grafici avanzati (*ambienti desktop*, *grafica tridimensionale in tempo reale*, ...), a cui si ovviò fornendo **toolkit**, cioè librerie che non solo aggiungessero funzioni, ma anche semplificassero la programmazione di applicativi grafici. Tra questi si possono ricordare:

Figura 2.6: Il *logo* di **X-Org**.

- **Motif**
- librerie **QT**, su cui s'appoggia **KDE**
- librerie **GTK**, nate da **GIMP** e successivamente sviluppate nelle **GTK+**

Il rovescio della medaglia nell'aggiunta dei **toolkit** fu smascherare le limitazioni interne di **X**, prime tra tutte la lentezza e la pesantezza. Ciò spinse **XFree86**, attorno all'anno 2000, ad una massiccia riprogettazione, snellendo il codice ed aggiungendo diverse funzionalità come l'*anti-aliasing* ed il supporto per i *fonts* e per la **grafica 3D**. Ciò non bastò, però, per non allontanare dal progetto alcuni validi collaboratori come Keith Packard.

Nonostante tutto, le aziende videro in **XFree86** l'agognata opportunità per portare **Linux** sui *desktop*. Inviarono pertanto vari programmatori per partecipare al progetto, ma ciò comportò un problema d'incompatibilità tra la licenza **GPL** su cui era basato **XFree86** e le neo subentrate politiche aziendali. Fu così che vari sviluppatori abbandonarono **XFree86** facendo rinascere **X-Org** e presentando, a partire dal 2004 fino ad oggi, nuove *release* che:

- corressero vari *bug*
- introdussero la gestione dell'**IPv6**

- revisionarono i moduli per la gestione dei **fonts**

- inserirono nuovi metodi per la resa e la gestione dei cursori

- permisero l'uso di finestre traslucide ed ombreggiature

- previdero architetture per l'accelerazione grafica

Soprattutto, il codice venne modularizzato per favorire lo sviluppo futuro e vennero costantemente aggiornati i *driver* per l'*hardware* grafico.

Fu così che, dopo un decennio di sviluppo discontinuo e di mancanza di solide basi, **X-Window** divenne un punto fermo del panorama dei gestori grafici nel mondo informatico di oggi e di domani.

2.1.4 Il modello *client/server*.

X segue il **modello *client/server*** (*Fig. 2.7*), ovverosia esiste un **X Server** in grado di eserguire operazioni a livello grafico, ed uno o più programmi **Client** che vi rivolgono richieste da soddisfare. L'**X Server** riceve l'*input* dell'utente (da tastiera, *mouse*, *touchscreen* e simili) ed indirizza le richiesta di *output* alle finestre.

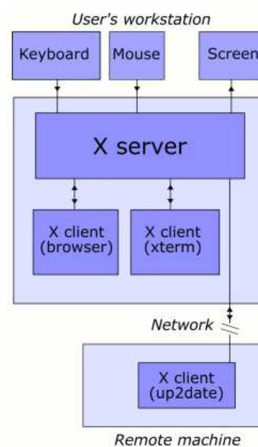

Figura 2.7: Il **modello Client/Server**.

La comunicazione tra **X Server** e **Client** è trasparente rispetto alla rete, in quanto i protocolli possono risiedere sulla stessa macchina o su macchine diverse con architetture e Sistemi Operativi differenti. È anche possibile la comunicazione criptata via **SSH**.

Nel concreto, l'**X Server** può essere:

- del *software* di sistema che controlla la grafica d'un computer (caso più comune)
- una componente *hardware* dedicata, cioè una macchina dotata del solo *hardware* necessario all'esecuzione dell'**X Server** (il cosiddetto **X Terminal**), che verrà richiamata dai programmi che ne necèssitino
- un programma che invia l'*output* ad un altro sistema grafico

Una delle particolarità terminologiche del *modello client/server* di **X Window** è che spesso i due termini sembrano scambiati rispetto al loro uso consueto, che prevede un **Client** locale ed un **Server** remoto: con **X**, infatti, non è desueto che l'**X Server** giri sulla macchina locale operando la gestione dell'*input* e dell'*output*, mentre il **Client** che ne usa i servizi sia remoto, magari *on-line*. Questo perché l'assegnazione dei termini è stata fatta ponendo al centro il programma, non la persona. *Cfr.* a riguardo [**XWINDPA**] **ed** [**XWINDCP**].

Nell'ambito della *Computer Graphics*, la scelta del **modello Client/Server** non è uno *standard* diffuso, basti pensare al altre architetture e/o linguaggi di programmazione.

- In **MicroSoft QuickBASIC** o **QBASIC**, esistono istruzioni per commutare lo schermo dalla modalità *testuale* alla modalità *grafica* e viceversa: in *modo grafico* diviene possibile assegnare un colore ai singoli *pixel*.
- Nel **BASIC** del **Commodore 64** la grafica s'implementa modificando le singole mappe *8*x*8* dei caratteri sullo schermo, verificando come debba cambiare il colore di ogni *pixel* e modificando di conseguenza detta

mappa. In aggiunta od in sostituzione di questo metodo, è possibile avvalersi di otto immagini speciali dette **sprites**: la loro definizione, il colore e la posizione sono potenzialmente indipendenti, possono essere in **multi-color** o **monocromatiche ad alta risoluzione** e sono sensibili ad operazioni specifiche su di esse, come ad esempio il controllo delle collisioni. Nella *Fig. 2.8* è possibile visualizzare gli *screenshot* di tre giochi per il **Commodore 64** facenti uso sia di *sprites* che di grafica a caratteri ridefiniti.

- In **Turbo Pascal** della **Borland** si entra nel *modo grafico* definendo il **driver grafico** e le relative opzioni, quindi agendo su *pixel* ed aree con le procedure e le funzioni dedicate.

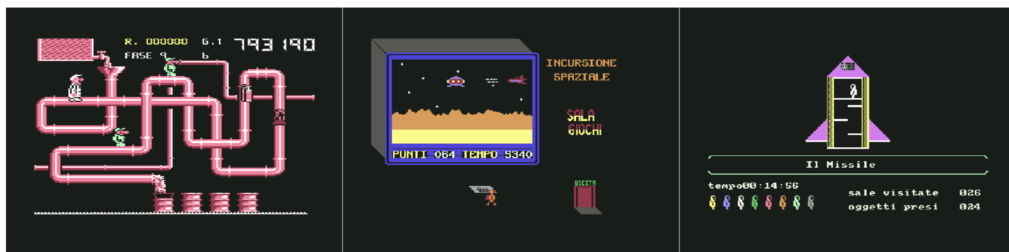

Figura 2.8: **SuperPipeline II** (**Tubature**), **Lazy Jones** (**Sala Giochi**) e **Jet Set Willy II** (**Ponzie**).

2.2 Le librerie.

Sebbene, come notato, **X-Window** non sia dotato dei classici elementi d'interfaccia per un sistema interattivo, fornisce una strumentazione ampia e completa per la gestione grafica, con ricche librerie di funzioni in grado di svolgere operazione che vanno dall'accendere singoli *pixel* sullo schermo al catturare le coordinate delle periferiche d'interazione col sistema, dalla tastiera al *mouse*, dal *modem* alla lavagnetta grafica ([**NYE92**], [**SCHEIF**], [**GETSCH**]).

Logicamente, le varie funzioni sono classificate nelle librerie a seconda della *famiglia* di elementi con cui interagire. Fondamentali sono le seguenti:

- **Xlib.h**, la libreria basilare contenente le funzioni per l'accesso al sistema grafico e per la resa delle primitive (punti/*pixel*, segmenti, poligoni, colori, ...)
- **X.h**, con le costanti del protocollo (inclusa automaticamente con **Xlib.h**)
- **Xcms.h**, simboli e *macro* per la gestione dei colori (richiede **Xlib.h**)
- **Xutil.h**, funzioni, tipi e simboli per la comunicazione tra **client** (richiede **Xlib.h**)
- **Xresource.h**, funzioni, tipi e simboli per la gestione delle risorse (richiede **Xlib.h**)
- **Xatom.h**, azioni atomiche predefinite
- **cursorfont.h**, gestione dei **fonts**
- **keysymdef.h** e **keysym.h**, gestione delle mappe caratteri
- **Xproto.h** ed **Xprotostr.h**, funzioni, tipi e simboli basilari del protocollo **X**, necessari per l'implementazione di estensioni
- **Xlibint.h**, funzioni, tipi e simboli per l'implementazione di estensioni (include automaticamente **Xlib.h** ed **Xproto.h**)
- **X10.h**, funzioni, tipi e simboli necessari a mantenere la compatibilità all'indietro

La libreria **Xlib.h**, inoltre, definisce altri utili elementi non prettamente grafici:

- il tipo *booleano*, **Bool**
- l'identificatore nullo universale, **None**

- l'identificatore di risorsa generica, **XID**
- il *puntatore opaco* o *generico*, **XPointer** (equivalente a `char *` o `void *`)

Come quasi sempre capita per le librerie che aggiungono intere categorie di funzionalità (come ad esempio **OpenGL**, *cfr.* più avanti), anche in questo caso vengono stabilite delle convenzioni sintattiche nel prototipo e nella chiamata delle funzioni, nell'uso degli identificativi e nei tipi dei dati. Tra gli altri, si possono ricordare:

- tutte le funzioni di **X** cominciano con una X maiuscola
- le iniziali di tutte le funzioni e di tutti i simboli sono maiuscole
- le *macro* e gli altri simboli composti da più d'una parola hanno ogni iniziale maiuscola, senza *underscores*
- gli elementi delle strutture dati sono in minuscolo, separati da *underscores* se composti da più d'una parola
- nelle liste di argomenti, il parametro *Display* è sempre il primo, le ascisse precedono le ordinate, le larghezze precedono le altezze, le posizioni precedono le dimensioni
- in genere, le posizioni sono di tipo `int`, mentre le dimensioni sono `unsigned int`

2.2.1 La gestione delle finestre.

Come la maggior parte degli altri gestori grafici, **X** fornisce un completo supporto per la gestione delle finestre. Esse sono inserite nell'ambito d'una *struttura ad albero*, e possono essere classificate in tre tipologie, che differiscono per finalità, apparenza e metodi d'inizializzazione.

- La **finestra radice** coincide con lo schermo, tutte le altre finestre sono sue discendenti ed appare sempre in secondo piano rispetto ad esse. Non

presenta **barre del titolo** o **di stato** né pulsanti per chiuderla o ridurla ad icona (chiudere la **finestra radice** equivarrebbe di fatto ad uscire dall'ambiente grafico). Comunemente, la **finestra radice** viene utilizzata come *desktop* dell'ambiente di lavoro e contiene le icone per accedere a vari programmi e documenti. Non è competenza dei programmatori creare la **finestra radice**: si suppone esista già, e se ne può ottenere l'identificativo grazie alla funzione `RootWindow` (*alias* `XRootWindow`).

- Le **finestre *standard***, figlie della **radice** e sorelle tra loro, presentano comunemente una **barra del titolo** con bottoni per chiuderle, ridurle ad icona ed ingrandirle a tutto schermo, una **barra di stato** per mostrare i possibili avvisi all'utente ed eventualmente dei bordi per ridimensionarle. Possono essere spostate e, salvo indicazioni particolari, essere messe l'una davanti all'altra. Al loro interno, i programmatori possono mostrare ciò che desiderano e, se necessario, impostare un'interfaccia utente a propria discrezione.

- Le **sottofinestre** sono di fatto delle aree definite all'interno di una **finestra *standard*** (che per la **sottofinestra** diventa la **finestra genitore**), tipicamente definite per indirizzare l'*output* verso (o catturare l'*input* da) determinate zone. Non hanno **barre del titolo**, **barre di stato** o **pulsanti** e sono gestite completamente dal **genitore**: normalmente, infatti, non possono essere spostate dall'utente, sebbene il programmatore possa prevedere che egli possa interagire con la loro geometria, ma in tal caso dovrà essere l'applicativo ad implementare le reazioni alle varie azioni possibili.

Come già accennato, le finestre costituiscono l'unità fondamentale per la comunicazione tra utente e programma (*input* ed *output*): a tal fine, è possibile associare ad esse le **maschere degli eventi** ed i **contesti grafici**.

- Le **maschere degli eventi** specificano in quali circostanze la finestra debba lanciare segnali al programma che la usa, affinché quest'ultimo

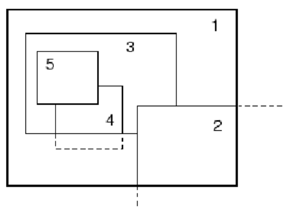

Figura 2.9: Esempi di finestre e relazioni: **1** è la radice, **2** e **3** sono sue figlie e sorelle tra loro (**2** è in parte non visibile), **4** e **5**, sorelle, sono **sottofinestre** di **3** (**4** esce parzialmente).

possa reagire adeguatamente. Per quanto riguarda la descrizione degli eventi, *cfr. Le entità fondamentali della grafica interattiva*, più avanti.

- I **contesti grafici** sono strutture dati che contengono i parametri coi quali verranno effettuate le operazioni grafiche, quali ad esempio i colori, i tipi di tratto ed i *fonts*. Ad esempio, una funzione che permette di disegnare una linea o di scrivere una stringa può prevedere, tra i propri parametri, i colori di *foreground* e di *background* ed il tipo di carattere, ma alternativamente si può definire a priori un **contesto grafico** dotato dei giusti valori ed utilizzarlo nell'ambito delle suddette funzioni.

Nel mondo di **MicroSoft** si lavora con le cosiddette **API** (**Application Programming Interface**), ovverosia con dei programmi esplicitamente finalizzati alla gestione dell'*output* grafico ed all'interrogazione dell'*input* proveniente dalle varie periferiche. Le **API** sono assai varie e ricche di funzionalità perché, a differenza di **X-Window**, **MicroSoft** non si astiene dal fornire un'interfaccia standard: lo si nota assai bene nei vari programmi che girano sotto **Windows**, in cui l'aspetto esteriore e la gestione di vari elementi come pulsanti, barre, menù e caselle è perfettamente uniforme.

2.3 Paradigmi ed ambienti per la programmazione grafica.

Una volta definite le risorse a disposizione per la *Computer Graphics* e deciso il modello con cui agire (**Client/Server** per **X**) onde sapere quali operazioni debbano essere effettuate e su quali entità, diviene possibile categorizzare i vari lavori che nel concreto si possono intraprendere, a seconda delle finalità per le quali risulti necessario operare con la grafica. Seguono alcuni dei consueti paradigmi sull'argomento e gli ambienti in cui essi possono operare.

2.3.1 Grafica accelerata con *Direct Rendering*.

Secondo l'idea più semplice possibile, le funzioni grafiche sono come tutte le altre: quando il *program counter* le raggiunge, il processore le carica e le esegue, con la visualizzazione a schermo degli eventuali risultati. Ciò avviene tramite la corretta compilazione, da parte del programma, del *frame buffer*, ossia la memoria della scheda video che ospita ciò che dev'essere visualizzato in ogni momento (testo, immagine, fotogramma o *mix* di questi), *pixel* per *pixel*. Il *frame buffer* può essere configurato in vari modi, a seconda delle potenzialità della scheda grafica, e permette di commutare tra varie risoluzioni (quantità di *pixel*), cui valori tipici sono:

- *320*x*200*
- *320*x*240*
- *640*x*480*
- *800*x*600*
- *1024*x*768*
- *1280*x*1024*

Consente proporzioni di *4:3* o *16:9* e varie intensità di colore, tra cui:

- monocromatico ad 1 *bit*
- 4 colori a 2 *bit* (tipicamente ciano, magenta, giallo/bianco, nero)
- *palette* 16 colori a 4 *bit*
- *palette* 256 colori ad 8 *bit*
- in scala di grigi ad 8 *bit*
- 65536 colori (*highcolor*) a 16 *bit*
- 16777216 colori (*truecolor*) a 24 *bit*
- 16777216 colori (*truecolor* con supporto della trasparenza) a 32 *bit*

a seconda della memoria a disposizione della scheda (*cfr.* [**FRAMEB**]).

Questa normale politica, che accentra ed omòloga tutte le operazioni e le delega alla **CPU**, generalmente efficace e *standardizzata*, incontra purtroppo delle difficoltà proprio dell'ambito della grafica! È in questa sede, infatti, che le operazioni possono farsi numerose ed assai ripetitive: se ciascuna di esse prevede la scrittura nel *frame buffer*, si crea di fatto un *collo di bottiglia* nell'uso della **CPU**, dal momento che la *bandwidth* **CPU**-*frame buffer* è in genere piccola, quindi prevedendo di scrivere in continuazione sul *frame buffer* (compito tipicamente assai costoso) si hanno problemi di prestazioni.

Per ovviare a queste difficoltà, è nato il cosiddetto *Direct Rendering.* L'idea di base sta nel fatto che molte operazioni grafiche, seppur complesse, sono ripetitive e/o modulari, basti pensare ai calcoli per la determinazione delle prospettive od alla resa di un modello fotorealistico: se, pertanto, si riuscisse a delegare dette operazioni ad *unità di calcolo* dedicate diverse dal processore, quest'ultimo sarebbe lasciato libero di svolgere altre operazioni, lavorando di fatto **in parallelo**. È con quest'idea che è nata la **grafica accelerata**: le operazioni grafiche più pesanti e ricorrenti vengono svolte da *hardware* specifico,

la **GPU** (**Graphic Processing Unit**), lasciando libera la **CPU** di occuparsi di altri compiti di calcolo.

Oltre alla grafica, esistono anche altri usi dell'accelerazione *hardware*, ad esempio:

- la **floating-point acceleration**, in cui una componente *hardware* dedicata si occupa dei calcoli, potenzialmente assai complessi, con numeri in virgola mobile, che vengono demandati ad essa anziché alla **CPU**
- l'utilizzo delle **GPU** come calcolatori paralleli

Un'istanza concerta di quest'ultimo caso si trova in [**FERR10**]: i coprocessori grafici della scheda **NVidia** vengono utilizzati per distribuire il carico di lavoro per il calcolo della visita in ampiezza di un grafo rappresentante gli incroci di una città, nell'ambito della quale devono essere determinati degli specifici percorsi scelti tra i tanti provati (con la necessità, quindi, di una sorgente di calcolo potente e parallelizzabile).

2.3.2 OpenGL.

Figura 2.10: Il *logo* di **OpenGL**.

Il sistema **OpenGL** (*Fig. 2.10*) è una libreria grafica (da cui **GL**) ad altissimo livello, ad alte prestazioni ed indipendente dall'*hardware* che consente la modellazione e la resa di scene grafiche interattive **3D** (ma può essere usata anche per il **2D**). Fu proposta da **Silicon Graphics** e comprende, a tutt'oggi, circa 350 funzioni ([**CASC02**], [**ANGE06**]).

Il sistema grafico è concepito come una **macchina a stati**, ognuno dei quali si occupa d'una fase specifica della modellazione/resa. Il programmatore

definisce i parametri della scena **3D** ed **OpenGL** si dedica alle varie fasi che porteranno alla creazione dell'immagine **2D** relativa. La modellazione avviene tramite **PSP** (*punti*, *segmenti*, *poligoni*) e la resa s'ottiene attraverso le seguenti fasi:

- definizione della geometria degli elementi che compongono la scena attraverso **primitive**
- posizionamento degli oggetti
- posizionamento della telecamera
- definizione di colori, ombre, trasparenze e *textures* degli oggetti
- proiezione in **2D**, colorando opportunamente ogni *pixel*

Come il protocollo di **X-Window**, **OpenGL** segue il modello **client/server**, sebbene non ci sia alcun obbligo che i processi siano nettamente distinti tra **client** e **server**, né che il *rendering* venga effettuato in un processo separato: ciò favorisce la *trasparenza di rete*, perché la codifica in *byte* delle operazioni di **OpenGL** potrà essere inviata attraverso la rete. Inoltre, la libertà dal protocollo consentirà di *bypassarlo* accedendo direttamente all'*hardware* e potendo sfruttare la *grafica accelerata* (*cfr.*).

OpenGL costituisce un'estensione di **X** detta **GLX**, ed essendo *system independent* è stata implementata anche per altri sistemi come **MicroSoft Windows** e **Macintosh**.

Le funzioni di **OpenGL** si dividono tra i seguenti tipi:

- primitive, per la definizione delle entità atomiche (**PSP**, *cfr. Fig. 2.11*)
- attributi (colori, riempimento delle aree, ...)
- trasformazioni (rotazioni, traslazioni, scalature)
- visualizzazione (*viewing*, posizione ed orientamento della telecamera)

- *input*, per l'interazione con le varie periferiche di *input* (tipicamente tastiera e *mouse*)
- controllo (errori, finestre)

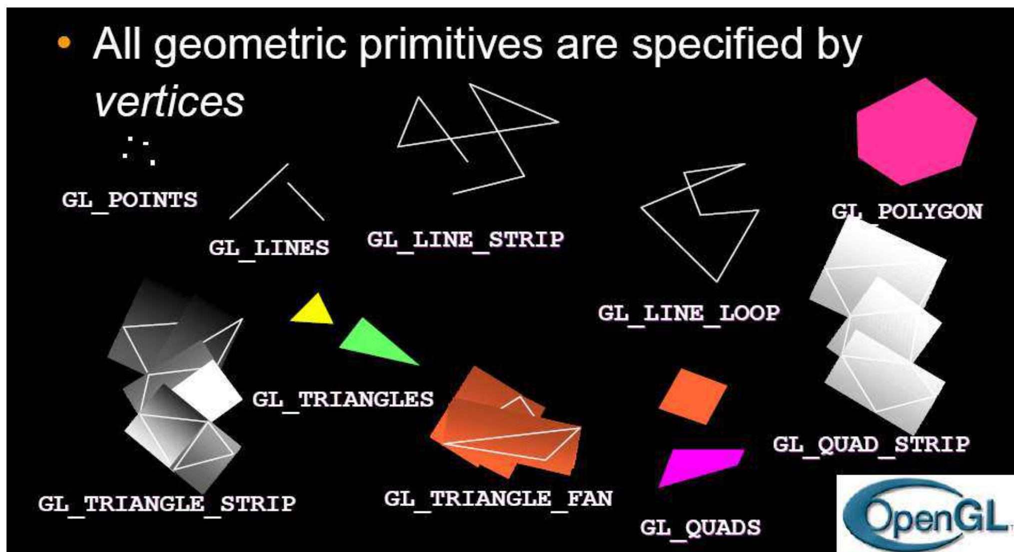

Figura 2.11: Le **primitive geometriche** di **OpenGL**.

Come per **X** e per altre corpose librerie, anche **OpenGL** fornisce funzioni con una sintassi ben precisa e con una formale definizione dei parametri (ordine, quantità, ...). Nello spefico, le funzioni sono suddivise in quattro grandi librerie e distinte dal prefisso corrispondente:

- **GL** (**Graphic Library**, prefisso **gl**), la librerie di base, contiene le funzioni per la definizione di primitive, attributi, trasformazioni, colori ed illuminazione
- **GLU** (**Graphics Library Utility**, prefisso **glu**) contiene codice per oggetti non primitivi d'uso comune (sfere, curve, superfici, *ecc.*) ed altre funzioni di utilità, tutte basate su **GL**, quali ad esempio l'uso di *textures*, la trasformazione di coordinate, la tassellazione di poligoni, la gestione

delle curve e delle superfici **NURBS** (**Non-Uniform Rational B-Spline**) e la descrizione degli errori

- **GLX** (**Graphics Library for X**, o **Graphics Library eXtension**, prefisso **glx**), che come detto è un'estensione di **X**, permette di creare un **contesto grafico** ed associarlo ad una finestra **X** per poter operare a più basso livello, oltre che consentire il *Direct Rendering* ad alte prestazioni

- **GLUT** (**Graphics Library Utility Toolkit**, prefisso **glut**) fornisce le funzionalità necessarie a fare grafica ed un'interfaccia utente in un sistema a finestre, grazie al *parsing* della riga di comando, alla definizione del modo grafico (**RGBA** o **color index**) ed al posizionamento, al dimensionamento ed alla creazione delle finestre

L'inclusione delle librerie richiede particolare attenzione. Infatti...

- Per utilizzare le funzioni base di **GL**, basta includere la libreria relativa:
  ```
  #include <GL/gl.h>
  ```

- Per le funzioni **GLU** servono **sia GL che GLU**:
  ```
  #include <GL/gl.h>
  #include <GL/glu.h>
  ```

- Per le funzioni **GLX** occorre **anche Xlib.h oltre a GL e GLX**:
  ```
  #include <GL/gl.h>
  #include <X11/Xlib.h>
  #include <GL/glx.h>
  ```

- Invece, **GLUT** include automaticamente quant'altro sia necessario:
  ```
  #include <GL/glut.h>
  ```

Si ricordi, infine, che **OpenGL** fa uso della *pipeline grafica* (*cfr.* più avanti).

La *Fig. 2.12* mostra il progetto **Church Virtual Tour**, ricostruzione della Chiesa della Commenda in Rovigo (Parrocchia Cuore Immacolato di

Maria e Sant'Ilario, Santuario Madonna Pellegrina) facente uso incrociato di **OpenGL** e funzioni **X-Window**.

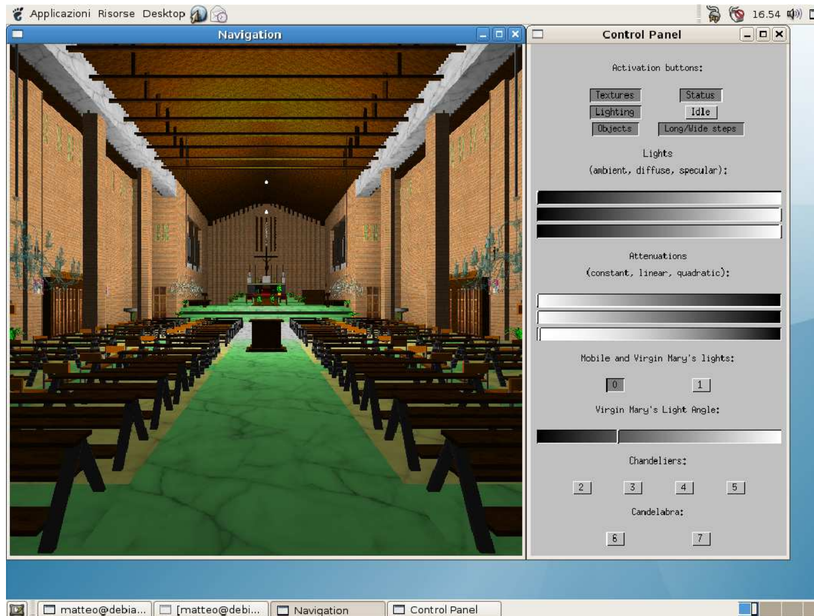

Figura 2.12: **Church Virtual Tour**.

2.4 Le entità fondamentali della grafica interattiva.

Ora che sono stati definiti:

- l'ambiente grafico di lavoro
- le famiglie di paradigmi, sistemi ed applicativi che ne fanno uso

si può procedere analizzando i singoli aspetti della *Computer Graphics* nel suo utilizzo concreto in tali ambienti e paradigmi, studiando le politiche fondamentali che li caratterizzano e l'implementazione delle stesse da parte di **X-Window**.

2.4.1 Il *backing-store* in X-Window ed in ambiente MicroSoft.

Nella sua accezione più semplice, un'immagine resa non è altro che una serie di *pixel* opportunamente colorati sullo schermo. Ciò significa che una volta avvenute le operazioni di disegno da parte di un programma, in linea di principio nessuna entità del sistema grafico avrà compito di tenerne traccia. Pertanto, se in una finestra è visualizzata un'immagine e tale finestra scompare alla vista, perché magari coperta da altri elementi o trascinata fuori schermo, non avrà più alcun contenuto al momento della sua ricomparsa. È ovvio che questo differisce dalla consuetudine durante il lavoro in ambiente grafico: i contenuti della finestra vengono usualmente ripristinati al momento del loro ritorno in primo piano, quindi evidentemente *qualcosa* li avrà memorizzati. Le modalità per raggiungere questi fini variano a seconda del gestore grafico utilizzato.

La competenza di memorizzare i contenuti dello schermo può essere demandata al sistema grafico od ai singoli applicativi. Fin dalle sue prime versioni, **X** ha offerto la possibilità di assegnargli questo compito grazie al *backing-store*: attivando l'omònima opzione, i programmi non hanno l'obbligo di memorizzare i propri contenuti, avendo la garanzia che sia **X** ad occuparsene. Se, da una parte, ciò semplifica notavolmente la vita dei programmatori di applicativi grafici, dall'altra vincola il sistema grafico ad un compito talvolta pesante e computazionalmente oneroso.

Inizialmente, il *backing-store* nacque partendo dal presupposto che tutta la grafica fosse per definizione solo **2D**: essendo gli schemi dei piani, era logico pensare che qualsivoglia operazione, **2D** o **3D** che fosse, avrebbe dovuto produrre, una volta conclusa, una proiezione bidimensionale. Aveva senso, pertanto, immaginare che memorizzare detta proiezione per utilizzarla in caso di eventuali *refresh* fosse sufficiente. La nascita della **grafica 3D** tramite **accelerazione *hardware*** e **pipeline** (*cfr.*) ha così dimostrato che le tre dimensioni possono giungere direttamente ad un passo dal *frame buffer*!

Negli anni recenti, infatti, la politica di **X** è cambiata a favore dell'abbandono del **backing-store**, così come in altri sistemi tipo **Windows**, il che

comporta la necessità, nel lungo termine, di convertire tutti i programmi che ne facciano uso, onde gestire autonomamente i contenuti delle finestre. Nel mondo **MicroSoft** (*cfr. 2.13*), infatti, non s'è mai propeso per offrire ai programmatori una risorsa analoga al *backing-store*. Ne consegue che sin dai primordi degli applicativi grafici **MicroSoft** il ridisegno dei contenuti delle finestre era competenza dei singoli programmi. Se, da un lato, ciò è stato un apparente "disservizio" per i programmatori, dall'altro rende **MicroSoft** più in linea con le intenzioni correnti e modello di *standardizzazione* per altri ambienti grafici.

Figura 2.13: Il *DeskTop* di **MicroSoft Windows Vista**.

Va inoltre detto che nel recente passaggio dalla grafica **3D** alla grafica **3D accelerata**, il **backing-store** non s'è dimostrato particolarmente efficace, in quanto semplice salvataggio di un'immagine bidimensionale: le soluzioni tipiche di **OpenGL**, ossia il passaggio attraverso *pipeline grafica* e *double buffering*, e di **XCModel**, cioè il previo disegno su **PixMap**, hanno permesso di raggiungere risultati maggiormente confacenti alle aspettative.

2.4.2 La *pipeline grafica*.

I più complessi applicativi di grafica tridimensionale necessitano di numerose operazioni tra la specifica del modello e la sua effettiva resa sui *display*.

Dal momento che tali operazioni sono di norma competenza di specifiche entità come le **GPU** (*cfr. Grafica accelerata con Direct Rendering*), è possibile parallelizzarle svolgendo contemporaneamente azioni che necèssitino di una entità diversa, per esempio:

- **modellazione** (vertici, segmenti, poligoni, punti)
- **elaborazione geometriche** (normalizzazione, *clipping*, ombreggiatura)
- **proiezione** (da **3D** a **2D**)
- **rasterizzazione** o **scan conversion** (da insieme di vertici a *pixel*)

Come precedentemente notato, la *pipeline* grafica viene usata da **OpenGL** (*Fig. 2.14*). Tuttavia, **X-Window** non la contempla direttamente, trattandosi di una politica nata, come detto, piuttosto di recente.

2.4.3 Le PixMap.

Come notato parlando delle librerie di **X**, le varie funzioni prevendono un elenco di parametri che a volte può essere molto lungo, dal momento che possono (o devono) essere passati numerosi argomenti come il *display*, lo schermo, la finestra da utilizzare, il *contesto grafico*, colori e liste di coordinate e dimensioni, ed **X-Org** ha stabilito sommariamente l'ordine che ogni funzione dovrà usare. L'elemento che segue *display* e *screen* è solitamente l'*area* su cui deve operare la funzione: tipicamente si tratta d'una finestra, ma non sempre è così! In vari casi si tratta più genericamente di una **drawable**, ossia un'area su cui sia possibile effettuare operazioni di disegno. A tal fine, oltre alle finestre vi sono le **PixMap** (*mappe di* pixel), ovvero delle aree virtuali di memoria trattate come se fossero immagini, in cui ogni *pixel* può esser d'un particolare colore e sulle quali si possono svolgere operazioni di copia, confronto, cancellazione, *ecc.*, ma che di fatto non saranno visibili. L'utilità delle **PixMap** è grande, perché possono fungere da *buffer* temporaneo, da *clipboard*, da aree

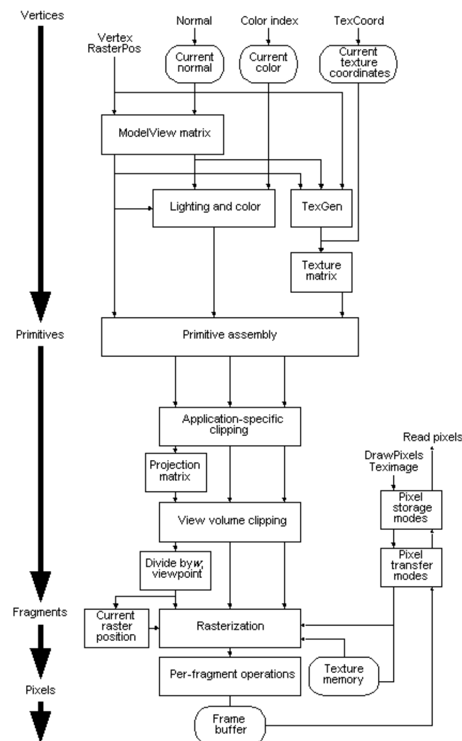

Figura 2.14: La *pipeline* di **OpenGL**.

su cui si deve progressivamente creare un'immagine che non dovrà essere visualizzata prima del completamento, ma senza problemi di cancellazione o sovrapposizione, poiché di fatto avranno delle dimensioni ma non una posizione potenzialmente necessaria a qualcun altro. Inoltre, in ogni momento, sarà possibile vedere a schermo i contenuti d'una *PixMap* copiàndoli semplicemente in una finestra visibile sufficientemente ampia, grazie alle funzioni della famiglia di `XCopyArea`.

2.4.4 Il *double buffering.*

Altra politica sfruttata da **OpenGL**, il **double buffering** consiste nel *rendere* un'immagine (magari nell'ultima fase della *pipeline*) non direttamente nel *frame buffer*, ma alternativamente su due *buffer* distinti in modo da non dover

assistere, durante la resa, alla progressiva costruzione delle scene. Disegnare su uno dei due mentre l'altro viene visualizzato risolve il problema.

X-Window non contempla direttamente il **double buffering**, ma consente di ottenere risultati analoghi utilizzando opportunamente una o più **PixMap**.

2.4.5 Grafica interattiva ad eventi.

Tipicamente, un sistema grafico non si limita a mostrare dei contenuti: non appena si raggiunge un certo livello di complessità diviene necessario permettere all'utente di interagire con quel che sta vedendo, al fine di procedere e d'ottenere i risultati per i quali il programma grafico è stato implementato.

Nel caso più semplice, l'*input* e l'*output* sono gestiti distintamente: mentre l'*output* grafico avviene disegnando i contenuti desiderati nelle aree dedicate, l'*input* passa attraverso i consueti canali (tastiera e *mouse*) e catturato dal programma, tramite un controllo ciclico delle periferiche, che in base ai valori letti agirà di conseguenza. È questo il modo in cui, fondamentalmente, lavorano il **BASIC** e il **Pascal** precedentemente citati. **X-Window** sfrutta un metodo che, seppur simile (e generalmente di comportamento analogo), viene strutturato in maniera ben precisa, utilizzando particolari entità dette **eventi**.

Un **evento** è una struttura dati di **X** (peraltro piuttosto complessa, di tipo `XEvent`) che permette di segnalare un particolare avvenimento, come ad esempio la pressione di un tasto sulla tastiera, il movimento e/o il *click* del *mouse* e l'apparizione di un'area precedentemente nascosta di una finestra. Un evento porta con sè numerosi particolari sul suo avvenimento, settando opportunamente i propri campi. Casi tipici sono i seguenti.

- Nel caso in cui il *mouse* venga mosso, viene generato un evento (**MotionNotify**) per ogni istanza del movimento, specificando in ciascuno le coordinate del puntatore e la finestra interessata (puntata), assumendo che quest'ultima sia stata inizializzata per essere sensibile al movimento del *mouse* (**PointerMotionMask**).

- In caso di *click* col *mouse*, si considerano distintamente *due* eventi, uno per la *pressione* (**ButtonPress**) ed uno per il *rilascio* (**ButtonRelease**) del bottone. Per ciascuno vengono specificati nella struttura le coordinate del puntatore, la finestra interessata ed il pulsante utilizzato. Lo *standard* prevede fino a cinque bottoni, ciascuno dei quali può essere premuto o rilasciato. Correntemente, essi sono il pulsante sinistro, la rotellina premuta o rilasciata, il pulsante destro e la rotellina ruotata in su ed in giù. Notare che la gestione distinta di **ButtonPress** e **ButtonRelease** consente operazioni particolari come il trascinamento. Occorre che la finestra che deve segnalare tali eventi sia stata inizializzata con **ButtonPressMask** e/o **ButtonReleaseMask**, a seconda se si voglia essere sensibili ad un tipo di evento, all'altro o ad entrambi.

- Anche qualora venga premuto un tasto sulla tastiera ci saranno i due eventi distinti *pressione* (**KeyPress**) e *rilascio* (**KeyRelease**): la struttura verrà compilata indicando il tipo d'evento, la posizione del puntatore (il che può sembrare insolito, dal momento che nella maggioranza dei casi le operazioni della tastiera sono disgiunte da quelle del *mouse*) ed il codice dell'effettivo tasto premuto o rilasciato. Gestire distintamente i due eventi permette, tra le altre cose, la ripetizione dei tasti. Per rendere una finestra sensibile a questi eventi, la si deve inizializzare con **KeyPressMask** e/o **KeyReleaseMask**.

- Un evento particolarmente interessante è l'**Expose**, che viene lanciato da una finestra (se inizializzata con **ExposureMask**) in tutti i casi in cui una o più parti precedentemente nascoste della stessa ridivèntino visibili (perché magari una finestra viene ingrandita, o trascinata in schermo risultando maggiormente visibile, o ripristinata se dapprima ridotta ad icona, od anche qualora una finestra che la dovesse coprire venisse almeno in parte spostata). È questo l'evento chiave che permette di gestire il ridisegno delle parti scoperte, dal momento che, come osservato prima, nessun componente è responsabile a priori del mante-

nimento dei loro contenuti: o il programma si gestisce da solo o si ricorre al **backing-store** di sistema, sebbene quest'ultimo sia una funzionalità particolare di **X** ed in via di estinzione.

2.4.6 Coda e ciclo degli eventi. Interrogazione e *call-back functions*. Le GUI.

Il particolare fondamentale degli eventi di **X** è la loro gestione **asincrona**: ciò significa che i suddetti avvenimenti non vengono istantaneamente segnalati costringendo il programma ad esser pronto a coglierli, bensì vengono inseriti in una struttura *FIFO* ad uopo dedicata denominata **coda degli eventi**. In tal modo, il programma potrà procedere senza preoccuparsi degli eventi che possano accadere, e contemporaneamente essi succedono senza obbligare alcun processo a coglierli: quando il programma avrà bisogno di conoscere i dettagli di un evento, potrà interrogare la coda (`XNextEvent`) ed agire di conseguenza, peraltro con le certezza del mantenimento dell'ordine con cui sono avvenuti (trattàndosi d'una coda). Ne consegue che **l'*input* non sarà bloccante**, a meno che non si cerchi di prelevare elementi da una coda vuota, ma ci sono comandi che consentono di verificare quanti eventi non ancora letti ci siano (`XEventsQueued`). Volendo, è anche possibile in ogni momento svuotare la coda, in modo da non doversi preoccupare di eventuali sue *entries* non ancora interrogate.

Una delle più interessanti caratteristiche che scaturiscono dalla natura asincrona degli eventi è la gestione di più dispositivi: dal momento che, come osservato, ogni evento mantiene la propria storia (coordinate, situazione scatenante, *ecc.*) e nessuna informazione va sprecata a prescindere dal tempo intercorso tra l'avvenimento e l'utilizzo (sempre che ci sia), ogni dispositivo potrà accodare i propri eventi, senza rischio di bloccare processi o di vedersi ignorare.

Con la consapevolezza di poter contare su queste possibilità, nasce la cosiddetta **programmazione *event-driven***, che scongiura ogni *input* bloccante e consente l'interrogazione contemporanea di vari dispositivi. All'atto prati-

co, ci sarà un segmento del programma denominato **ciclo degli eventi**, in cui avverrà la consultazione della coda (se non vuota) e l'esecuzione d'una funzione per gestire l'evento letto, detta **call-back function**. Tipicamente, alla **call-back function** vengono passati i parametri necessari (coordinate, *ecc.*) letti dalla struttura evento, oppure l'evento stesso di tipo **XEvent** (ciò può essere sconsigliabile in alcuni casi, trattàndosi come detto d'una struttura complessa e quindi pesante da passare come parametro, specialmente in un sistema distribuito). La **call-back function** può essere una funzione vera e propria od uno spezzone di codice nel ciclo degli eventi, solitamente un costrutto `while` con all'interno `if` o `switch`.

Le **GUI** sono librerie per la gestione di un ambiente *event-driven*. Costituiscono la parte appena citata dei programmi grafici, in cui vengono catturati gli eventi e chiamate/usate le **call-back functions** adeguate. Una delle possibilità che nascono cominciando a programmare le **GUI** è utilizzarne più d'una, concentrando ciascuna di esse in un diverso sottoinsieme di azioni possibili (ad esempio, una **GUI** per tutte le azioni della tastiera, un'altra per il *mouse*, un'altra ancora per le finestre, *ecc.*). Questa tecnica s'è rivelata vincente per tanto tempo, distribuendo i controlli delle code degli eventi a più cicli e scongiurando azioni potenzialmente inutili evitando di prevedere tutto in ogni momento. Tuttavia, nel passato recente in cui il *backing-store* ha smesso di essere uno *standard*, l'uso di più **GUI** ha scoperto il proprio fianco, creando il pericolo di trovarsi all'interno del ciclo d'una **GUI** nel momento in cui sia necessaria un'azione di ridisegno: qualora quest'ultimo sia competenza del sistema tramite il **backing-store** non vi saranno problemi, ma in caso contrario la **GUI** insensibile all'**Expose** potrà causare i classici problemi del mancato ridisegno (aree parzialmente o totalmente cancellate), È per questo motivo che, oggigiorno, la politica utilizzata in **X** è quella di evitare il *backing-store* (occupandosi autonomamente del ridisegno) e demandando tutti i controlli **ad un'unica GUI**, intelligentemente programmata per gestire ogni possibile evento senza risultare troppo pesante (idea: controllo dell'**Expose** non continuamente ma ad intervalli fissi). Su quest'aspetto

s'è basata la recente riprogettazione della *suite* **XCModel**, come si vedrà nel capitolo successivo.

Capitolo 3

XCModel

3.1 La *suite* XCModel.

Come anticipato nell'*Introduzione* e nella *Panoramica*, **XCModel** include una serie di programmi finalizzati alla *modellazione* ed alla *resa* di scene fotorealistiche, dotate di luci, ombre, trasparenza, riflessioni e *textures*, e composte da oggetti **2D** e **3D** a **forma libera** (*cfr.* [**XCMODEL**]). La *suite* **XCModel** si compone dei seguenti programmi (*cfr.* anche *Fig. 3.1*):

- **XCCurv**, un ambiente di lavoro completo per le **curve NURBS 2D**
- **XCSurf**, che consente la creazione e la gestione di **curve** e **superfici NURBS 2D** e **3D**
- **XCBool**, che permette di effetturare **operazioni *booleane*** sugli oggetti
- **XCRayt**, l'ambiente, basato sulla **Libreria *Descriptor***, in cui inserire gli oggetti nella scena, posizionare luci e telecamera, assegnare *textures* e procedere alla *resa fotorealistica*
- **XCView/XMovie**, che consente di serializzare delle *rese* per ottenere una **scena in movimento**, anche esportabile come `.gif` animata, nonché

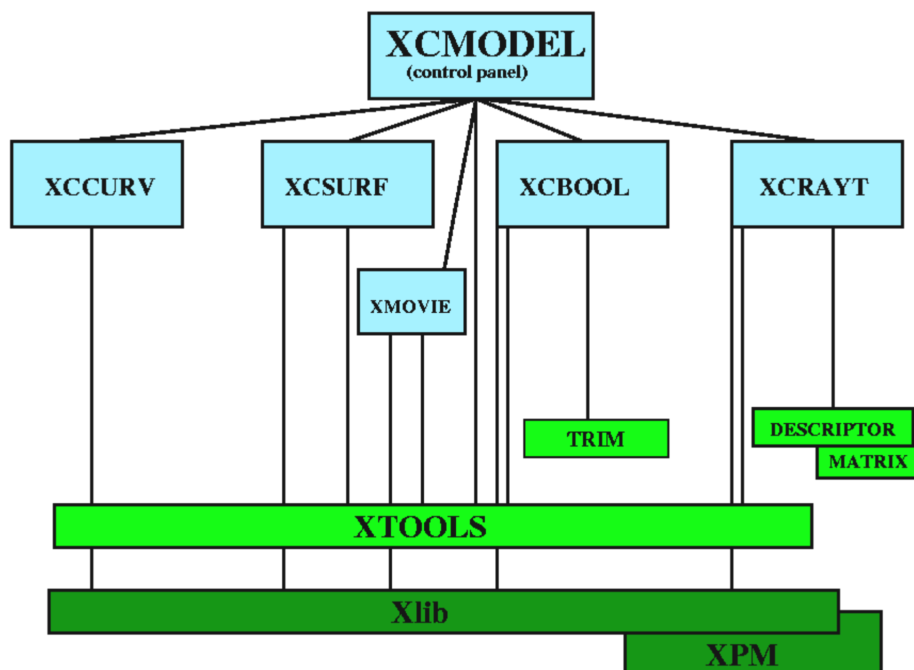

Figura 3.1: I pacchetti della *suite* **XCModel**.

la conversione da formato immagine `.ppm` ad `.hr` per ottenere *textures* da usare nelle *rese*

Tutti i suddetti programmi si basano sulla **Libreria XTools**, che costituisce un'estensione di **X-Window** fornendo strumenti per la creazione d'interfacce e l'utilizzo di risorse grafiche. Si ricòrdino anche le fondamentali **Librerie Trim** e **Matrix**.

XCModel può girare su più sistemi, tra cui:

- **SUN Sparc**, **Solaris**
- **SGI**, **Irix**
- **Intel**, **Linux**

Mentre **XCCurv** ed **XCSurf** non hanno particolari requisiti in termini di risoluzione e colori, **XCBool** ed **XCRayt** richiedono una profondità di colore a 24 *bit* ed uno schermo impostato ad almeno *1024*x*768*

XCModel lavora con *files* di vario tipo. Le estensioni che s'incontrano più spesso lavorando coi pacchetti della *suite* sono:

- `.db` (**Data Base**), **curve NURBS 2D** o **3D** o superfici
- `.dbe` (**Data Base Extended**) e `.tree` (**TREE**), **superfici NURBS trimmate**
- `.obj` (**OBJect**), liste di *files* `.db`, `.dbe` e `.tree`
- `.cp` (**Control Points**) ed `.ip` (**Interpolation Points** od **approxImation Points**), **vettori di punti** per la definizione di curve e superfici
- `.md` (**MoDel**), informazioni su geometria e luce d'una scena **3D**
- `.vw` (**VieW**), informazioni sulla telecamera che inquadra una scena **3D**
- `.arg` (**ARGuments**), parametri per la resa d'una scena tramite *ray-tracing*
- `.sta` (**STAtistics**), informazioni sulla resa d'una scena attraverso *ray-tracing*
- `.hr` (**HRay-tracing**), immagine compressa, corrispondente alla resa d'una scena per mezzo di *ray-tracing*, oppure una *texture*
- `.ppm` (**Portable Pixel Map**), immagine nel formato *standard* per **UNIX**, composto da un'intestazione **ASCII** e dalla codifica binaria o numerica *byte* per *byte* del contenuto
- `.hra` (**HRay-tracing Animation**), animazione, ossia una lista d'immagini `.hr` o `.ppm`

Un progetto realizzato facendo uso dei vari pacchetti di **XCModel** è **Crypt Tabernacle**, che mostra una ricostruzione alternativa della cripta della Chiesa della Commenda di Rovigo: con **XCCurv** sono stati realizzati i profili (sagome) delle figure e le traiettorie *rotazioni*, *tubole* ed altre tecniche

di costruzione, con **XCSurf** le si sono utilizzate per creare le superfici da inserire nella scena, oltre a sfruttare trasformazioni geometriche e superfici *usuali*, e con **XCRayt** e la **Libreria Descriptor** s'è accorpato il tutto in una scena illuminata, colorata e *texturata* (immagini convertite in *textures* grazie ad **XCView**).

Figura 3.2: **Crypt Tabernacle**.

3.2 La Libreria XTools.

Secondo la filosofia di **X-Window**, come visto nel capitolo precedente, di non fornire a priori alcuno strumento per le interfacce grafiche, **XCModel** si basa su una propria libreria, denominata **XTools** ([**XTOOLS**]), che fornisce tutti i mezzi necessari ai programmi del pacchetto, da quelli per la creazione di finestre a quelli per la definizione degli elementi che queste conterranno. Un esempio di programma creato con **XTools** è **Test Bars**, finalizzato proprio alla sperimentazione degli strumenti forniti dalla libreria (*cfr. Fig. 3.3*).

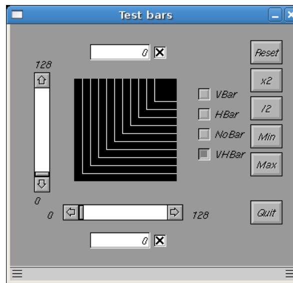

Figura 3.3: Programma **Test Bars**.

Assai interessante è l'uso delle **GUI**. Ogni finestra **XTools**, inizializzata e guarnita degli elementi desiderati, non viene semplicemente passata ad **X-Window** e mostrata, bensì viene aggiunta ad una struttura di tipo `GUI_t`: tale struttura verrà lanciata non appena completata, e le varie finestre potranno apparire/sparire sullo schermo dell'utente a seconda delle necessità e delle specifiche del programma.

Tra i vari tipi di funzioni **XTools**, si possono distinguere quelli per la gestione delle **GUI** e delle finestre, quelli per la creazione degli oggetti e quelli per il loro utilizzo (ossia, per estrapolarne dati utilizzabili dal programma, ad esempio la pressione di un bottone, la posizione di una barra, il valore contenuto in una *text box*, *ecc.*).

3.2.1 Funzioni XTools.

Innanzitutto, vanno ricordate le funzioni necessarie all'inizializzazione ed alla cessazione della connessione con **X** (secondo il **modello Client/Server** precedentemente analizzato). Esse sono:

- `Initgraph`
- `Closegraph`

Poi, ci sono le funzioni per creare le **GUI**, inserirvi le finestre, lanciarle e fermarle:

- `InitGUI`
- `AddWindowToGUI`
- `RunGUI`
- `StopGUI`

Le finestre da aggiungere alla **GUI** appartengono al tipo proprio di **XTools**, ossia `Win_t`. Le funzioni per trattarle (creare, mostrare, nascondere) sono:

- `CreateWindow`
- `ShowWindow`
- `HideWindow`

Una volta fornita la possibilità di creare e gestire finestre e **GUI**, si possono definire le varie funzioni per l'inserimento e l'utilizzo degli elementi dell'interfaccia grafica. Tali funzioni sono assai numerose, ed accomunate da una sintassi similare e da un'analoga disposizione dei parametri formali. Ognuna appartiene ad una determinata *classe*, a seconda della definizione e delle finalità che la caratterizzano.

- Creazione (`Add...`), specificando la finestra in cui aggiungere l'elemento ed i parametri che lo caratterizzano, quali ad esempio posizione e dimensioni), per l'allocazione in memoria dell'elemento desiderato:
  `AddBar`
  `AddBox`
  `AddButton`
  `AddCheckBox`
  `AddCheckPoint`
  `AddCursor`
  `AddLabel`
  `AddTextBox`

- Esecuzione (`Set...Script`), associando al verificarsi di un determinato evento la chiamata all *call-back function* adeguata:
  `SetBarBPScript`
  `SetBoxBPScript`
  `SetCheckBoxBPScript`
  `SetCheckPointBPScript`
  `SetTextBPScript`
  ...
  `SetButtonBRScript`
  ...
  `SetTextLFScript`
  ...
  `SetBarMNScript`
  ...
  `SetWindowEXScript`
  `SetWindowCNScript`
- Abilitazione/Disabilitazione (`Set...Enable`, passando come parametri la finestra, l'indice dell'elemento ed un *booleano* per specificare se attivarlo o meno), per concedere o negare l'accesso ad un elemento:
  `SetBarEnable`
  `SetBoxEnable`
  `SetButtonEnable`
  `SetCheckBoxEnable`
  `SetCursorEnable`
  `SetLabelEnable`
  `SetTextEnable`
  `SetCheckPointEnable`
  `SetCheckPointsEnable`

Per quanto riguarda l'aspetto dell'esecuzione, si tratta di funzioni che ne chiamano altre, le **call-back functions**, al verificarsi di determinati elementi, con l'effetto di modificare un elemento, prelevarne i valori o reagire in altro

modo. I parametri di ogni funzione di esecuzione sono tipicamente la finestra ospite, l'(eventuale) indice dell'elemento interessato ed il nome della **call-back function**, mentre l'identificativo stesso della funzione specificherà l'elemento e l'evento interessati. Quest'ultimo, indicato da due caratteri in maiuscolo (come visibile nei sovraccitati esempi), appartiene ad una delle seguenti famiglie:

- **Button Press** (**BP**, soprattutto con barre, caselle di testo, *Check Box* e *Check Points*)
- **Button Release** (**BR**, principalmente nei pulsanti, che tipicamente si attivano con un *click* completo, quindi al momento del rilascio)
- **Lost Focus** (**LF**, l'uscita del puntatore del *mouse* dalla superficie dell'elemento, per confermare l'inserimento di valori in una casella di testo)
- **Motion Notify** (**MN**, soprattutto nelle barre, per rilevare l'effettivo scorrimento)
- **Expose** (**EX**, **importantissimo**, per rilevare la comparsa di una zona precedentemente nascosta)
- **Configure Notify** (**CN**, in caso di modifica di impostazioni, come ad esempio il ridimensionamento di una finestra)

Vi sono poi varie altre funzioni, per la definizione delle etichette (*label*) e delle finestre di *Help*, per l'inizializzazione, la lettura ed il settaggio dei valori associati ai vari elementi (posizione di una barra, valore inserito in una casella di testo, spunta o meno di una *Check Box*, *ecc.*), per il disegno di cornici e per l'uso delle **PixMap**.

3.2.2 Il progressivo sviluppo di XTools.

In linea con le scelte progettuali di **XCModel**, di cui costituisce la base, e di **X-Window**, su cui si applica e come pacchetto di espansione del quale

s'incarna, **XTools** ha mantenuto fede all'ideale di sviluppo continuo e di progressivo adattamento agli aggiornamenti *hardware* e *software* nel corso degli anni. Se ciò ha comportato il positivo aspetto di mantenere sempre aggiornato **XCModel** rispetto ad **X-Window**, senza dubbio ha risentito dei cambiamenti progettuali di quest'ultimo, obbligando periodicamente a massicci adeguamenti al fine di non restare indietro e consentire all'utente finale d'utilizzare macchine e Sistemi Operativi moderni (non obbligandolo, quindi, ad un *downgrade* per mantenere la compatibilità e l'utilizzabilità di **XCModel**).

Quando cominciò la programmazione di **XCModel** e dei suoi vari pacchetti, si fece la scelta più logica: progettarli in linea col contesto dell'epoca, sfruttando le risorse a disposizione e seguendo le linee proposte da **X-Org**. La scelta fondamentale in quest'ambito, nonché quella che ha causato maggiori problemi nelle fasi successive, è stata l'avvalersi del *backing-store* a carico di **X-Window**, ossia dare per scontato che una volta attivato tale servizio (specificàndone all'utente la necessità) nessuno dei programmi avrebbe più dovuto preoccuparsi dell'occultamento delle finestre di **XCModel** da parte di altre, né di dover gestire la loro ricomparsa: il *backing-store* si sarebbe occupato di tutto. Con questa certezza, si sono potute fare delle scelte implementative utili per snellire il lavoro e velocizzare l'esecuzione dei programmi a servizio dell'utente. Tra queste scelte vanno ricordate le seguenti.

- L'utilizzo di più **GUI**, chiamando varie volte la funzione `RunGUI` dopo avervi associato diverse finestre, ed assegnàndone una differente ad ogni *subroutine* del programma. Ciò permette di snellire l'utilizzo della **CPU** demandando ad un numero maggiore di **GUI** più semplici (e lanciate in diversi momenti) il controllo dell'interfaccia utente.

- La presenza di più **cicli degli eventi**, ossia lanciando vari processi contemporaneamente sensibili al manifestarsi di eventi, in modo da ottimizzare l'uso della memoria e suddividere i controlli in *famiglie di eventi* (del *mouse*, della tastiera, dello schermo, ...), ognuno affidato ad un ciclo particolare.

Le suddette linee guida si sono rivelate efficaci fintantoché s'è fatto affidamento sul *backing-store* di sistema, ma quando quest'ultimo ha smesso di essere uno *standard* ed, anzi, ha cominciato ad essere sconsigliato (delegando il *refresh* ad ogni applicativo), sono nati svariati problemi nella visualizzazione dei contenuti delle finestre di **XCModel**. La situazione è stata temporaneamente tamponata grazie ad un *Live CD* di **Linux** opportunamente configurato per il lavoro con **XCModel**: di norma, esso veniva fornito agli utenti (tipicamente gli studenti del corso di **Grafica**) per consentire l'uso di **XCModel** su *computer* in cui non fosse installato **Linux** (per esempio, a casa). Inoltre, consentiva di ovviare ai problemi presenti qualora **Linux** usasse i propri *driver* per le schede grafiche, non i binari forniti dalle case produttrici, causando quindi possibili malfunzionamenti di visualizzazione (ad esempio, etichette assenti o non correttamente rinfrescate sui bottoni). La configurazione del *Live CD* poté quindi esser già conforme alle necessità di *backing-store*, permettendo agli utenti una visualizzazione corretta. Naturalmente, una tale soluzione non avrebbe potuto che essere provvisoria, e quanto prima si sarebbe dovuto procedere all'aggiormamento della *suite* secondo i rinnovati *standard*.

3.2.3 Le fasi risolutive.

Le modifiche che portano all'aggiornamento di **XCModel** secondo le nuove necessità riguardano sia la libreria **XTools** che i programmi che su di essa si basano: **XTools** cura l'interfaccia e fornisce gli strumenti (tra cui le funzioni per l'interrogazione della coda degli eventi contemplando la possibilità di eventuali *refresh*: *cfr.* più avanti), mentre i contenuti ed il loro mantenimento in caso di **Expose** sono di competenza dei singoli applicativi.

I vari interventi necessari alla risoluzione dei problemi possono essere classificati per sommi capi, come segue:

- eliminazione della dipendenza dal *backing-store* di **X-Window**
- creazione d'un proprio metodo di *refresh*

- utilizzo di un'unica **GUI** per captare gli eventi desiderati e per gestire eventuali *refresh* delle finestre

Per quanto riguarda il terzo punto, l'uso di più **GUI** e cicli degli eventi aveva portato alla nidificazione dei controlli, facendo sì che qualora fosse avvenuto un evento di **Expose** nel momento in cui il programma si fosse trovato in un ciclo non sensibile ad esso (praticamente tutti, vista la pregressa dipendenza dal *backing-store*), non sarebbe avvenuto il *refresh* dei contenuti, con gli effetti tipici del caso (finestre parzialmente o completamente illeggibili, composte dal solo colore di sfondo definito dal **contesto grafico** corrente). Invece, una sola **GUI** sensibile **anche** al *refresh* avrebbe risolto il problema.

Quanto al secondo punto, la possibilità di associare una *call-back function* all'evento di **Expose** poteva essere contemplata, grazie alla possibilità data da **XTools** di reagire al manifestarsi di eventi con funzioni dedicate: nacque così `SetWindowEXScript` (*cfr.* **Funzioni XTools**)! Pertanto, sarebbe bastato chiamare una funzione dedicata al *refresh* al momento dell'**Expose**. Da dove prelevare i contenuti da copiare nelle finestre? Se le funzioni di disegno agiscono direttamente su una finestra, tale *output* risulta l'unica copia, e non c'è modo di preservarlo per le operazioni di *refresh*. D'altro canto, non si può imporre al programma di rifare i calcoli che hanno portato ad essi, dal momento che potrebbero essere assai complessi e numerosi ed impedire in ogni caso il ridisegno in tempo reale (oltre che appesantire ingiustificabilmente il sistema). La soluzione sta nelle **PixMap**: effettuando tutte le operazioni di disegno su di esse, s'otterrà una *copia* di ogni *output* (e certamente non occultabile da alcunché, data la natura *virtuale* delle **PixMap**), utilizzabile per ripristinare in tempo reale i contenuti delle finestre all'atto del *refresh*. Per applicare ciò ad **XCModel**, **XTools** ha previsto l'uso di una funzione dedicata: `ProcessGUIEventsUntil`. Essa ha sostituito tutte le occorrenze di `XNextEvent`, che per sua natura potrebbe bloccarsi in un ciclo insensibile alle necessità di *refresh*, ed è pensata per occuparsi di tutti gli eventi d'interfaccia (tra cui i decisivi **Expose**, chiamando la funzione di ridisegno specificata da `SetWindowEXScript` per ogni finestra) **fintantoché non accada uno degli**

eventi elencati nella funzione di *event test* passata come parametro, nel qual caso potrà procedere con la *call-back function* associata all'evento occorso. Inoltre, si sono dovute (re-)impostare le finestre per essere sensibili agli eventi di **Expose** e **ConfigureNotify**, grazie ad `XSelectInput` settato con le maschere di eventi **ExposureMask** e **StructureNotifyMask**. Questo metodo incarna l'idea, emersa nel capitolo precedente, di gestione contemporanea dell'interfaccia grafica e delle periferiche di *input*, con un ciclo degli eventi sensibile all'utente ma anche, periodicamente (dal punto di vista della macchina, il che significa praticamente in continuazione per l'utilizzatore *umano*), ad eventuali necessità di *refresh*. Tra l'altro, queste occasionali interrogazioni sulla presenza di **Expose** si rivela particolarmente utile durante le operazioni pesanti, come ad esempio la resa di una scena con tante sorgenti luminose e riflessioni, oppure il ***knot insertion*** in una curva **NURBS** complessa: tipicamente, una tale operazione monopolizzerebbe la **CPU** per un certo tempo, compromettendo la possibilità di effettuare dei *refresh* in caso, magari, di spostamento di finestre durante i calcoli. Invece, controllando ad intervalli regolari la coda degli eventi alla ricerca di **Expose**, si può sopperire al problema e garantire una corretta visualizzazione, anche a costo di un leggero prolungamento di un'operazione comunque costosa. `ProcessGUIEventsUntil` ha rimpiazzato le funzioni precedentemente adibite alla cattura degli eventi (cioè, dotate al loro interno di chiamate ad `XNextEvent`), le quali davano per scontato il mantenimento dei contenuti da parte del *backing-store* e che quindi processavano la coda degli eventi (con `XSync`) eliminando indiscriminatamente tutti gli eventi accodati ad eccezione di quelli d'interesse, ovverosia del *mouse* e della tastiera, e perdendo quindi quelli di **X11** (**Expose**, **ConfigureNotify**, **MouseMove**, **ReparentNotify**, ...): ciò avrebbe impedito il *refresh* delle finestre ad opera dell'applicativo. Non si dimentichi, poi, che la *de-standardizzazione* del *backing-store* ha comportato un sottile ma estremamente significativo cambiamento: tra i messaggi di **X11** che venivano intercettati ed eliminati non c'erano solo quelli relativi all'**Expose**, bensì anche quelli collegati ad eventuali problemi nella realizzazione dei contenuti

delle finestre, prima ancora che gli stessi potessero essere memorizzati dal *backing-store*, compromettendo di fatto una resa corretta!

Va detto che l'idea di progettare un'evoluzione del codice e delle funzionalità di **XTools**, dotata di ulteriori strumenti e funzioni meglio strutturate, e che gestisse le risorse in maniera meno invasiva, è antecedente alla necessità di manutenzione causata dalle specifiche della nuova versione di **X-Org**: di fatto, questo cambiamento ha accelerato la tabella di marcia!

Allo stato attuale, si sta lavorando con una versione di **XTools** a cui è stata applicata una *patch* che la modifica secondo quanto indicato, ma è in corso d'opera la realizzazione di una nuova *release* completa e corretta che contempli tutti i miglioramenti. Come si vedrà immediatamente, i vari pacchetti di **XCModel** sono stati adattati a tale *patch*, concretizzando gli interventi necessari precedentemente citati, e realizzati nell'ottica di rendere l'adeguamento all'imminente nuova versione un lavoro estremamente rapido ed efficace: ciò sottolinea come alla base ci siano un ragionamento ed una progettazione assai ben strutturati.

3.2.4 Le PixMap nei pacchetti di XCModel: il *disallineamento* di XCSurf.

Una volta fornito **XTools** della funzione `ProcessGUIEventsUntil` ed associata ad ogni finestra la funzione per il ridisegno, è sorto il problema della *standardizzazione* dell'*outout*, assicuràndosi che tutte le operazione di quel tipo avvenissero su **PixMap** in modo da permetterne la copia su finestra (con `XCopyArea` in caso di **Expose**). Questa fase del lavoro s'è rivelata piuttosto facile con **XCModel**, **XCCurv**, **XCBool**, **XCRayt** ed **XMovie/XCView**, dal momento che già di *default* effettuavano ogni disegno su **PixMap**, quindi la *call-back function* dell'evento di **Expose** per esse è consistita essenzialmente nella copia da **PixMap** a finestra. S'è cercato, a riguardo, di minimizzare il numero di tali copie, limitàndolo se possibile ad una sola per *gruppo* di operazioni di disegno e per il *refresh*: tale propensione è scaturita dalla con-

sapevolezza che **XCModel** lavora esclusivamente via *software* senza disporre di *accelerazione grafica*, puntando quindi a favorire il *real-time*.

Rispetto agli altri pacchetti, invece, **XCSurf** s'è rivelato notevolmente non uniformato, dal momento che le operazioni di disegno non avvenivano in differenti aree di un'unica finestra monolitica, bensì erano suddivise su varie finestre (*cfr. Fig. 3.4*):

- tre per le proiezioni ortogonali

- una per la proiezione prospettica

- una per la resa del modello secondo le disponibilità date dalla libreria **Trim** (*Wire Frame*, *Hidden Line*, *Depth Cueing*, *Flat Shading*, *Gourand Shading* e *Phong Shading*, *Cfr. Fig. 3.5*) e per le operazioni di **Free Form Deformation**

- una per la *control net* del modello e per la modifica dello stesso tramite *punti di controllo*

A queste va aggiunta la finestra principale del programma, il **pannello di controllo**. Peraltro, quest'ultimo è l'unica finestra **XTools** (cioè di tipo `Win_t`) del programma: le altre sei sono normali finestre di **X-Window** (di tipo `Win`), create con versioni alternative alla `CreateWindow` di **XTools** (`CreateWindow1` e `CreateWindow2`), al fine di sfruttare possibilità non contemplate da **XTools**, come ad esempio il posizionamento dinamico (necessario perché le finestre di **XCSurf** dovevano mantenere una posizione reciproca ben precisa). Inoltre, l'*output* grafico di **XCSurf** non era tutto indirizzato a **PixMap**, bensì solo in certi casi, mentre in altri veniva spedito direttamente alle finestre.

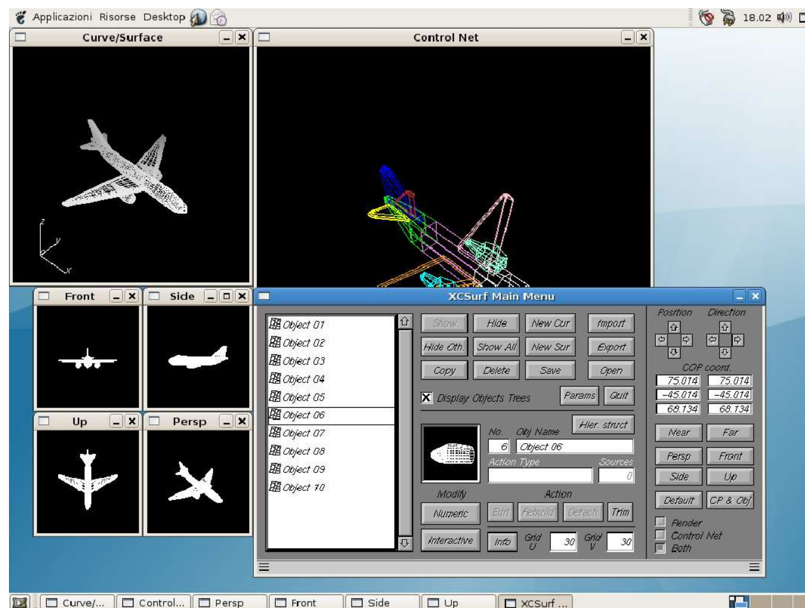

Figura 3.4: Versione precedente di **XCSurf**, a più finestre.

3.3 La riprogettazione dell'interfaccia utente in XCSurf.

Il desiderio di eliminare le difformità tra **XCSurf** e gli altri pacchetti di **XCModel**, nonché di risolvere anche per esso i problemi derivanti dalla scomparsa del *backing-store*, ha dato il via al presente progetto, durato per la maggior parte dell'anno 2010 e sfociato in un pacchetto rinnovato, potenziato, uniformato ed autosufficiente in termini di *refresh*. S'è colta l'occasione per raccogliere le varie difficoltà emerse nel corso degli anni, anche e soprattutto quelle segnalate dagli studenti durante la realizzazione delle superfici necessarie alla *scena fotorealistica* per il secondo progetto del corso di Grafica, al fine d'inserire tutto in un programma ampiamente ragionato.

Quando si operano modifiche ad un codice esistente, infatti, è essenziale garantire il mantenimento della correttezza e della funzionalità, ed è necessario interagire con codice scritto da altri, il che richiede che esso sia ben strutturato ed alterabile. **XCSurf** ha soddisfatto queste esigenze, facendo parte di una

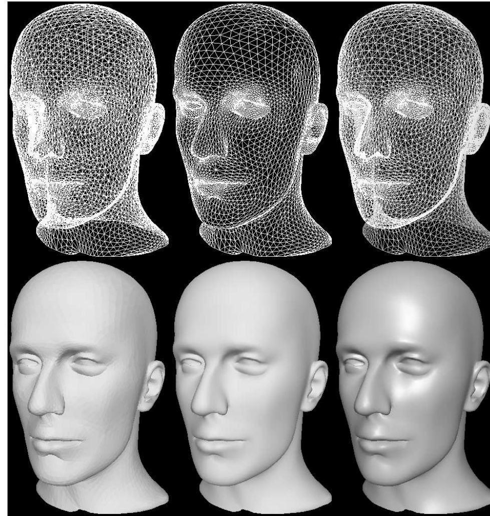

Figura 3.5: Modalità di **Rendering**: *Wire Frame*, *Hidden Line*, *Depth Cueing*, *Flat Shading*, *Gourand Shading* e *Phong Shading*.

suite accademica che per definizione dovrà essere mantenuta nel corso degli anni da diverse persone.

I campi principali in cui s'è operato, suddivisi in ambiti più circoscritti in corso d'opera, sono stati i seguenti:

- trasporto dei contenuti precedentemente suddivisi in più finestre di vario tipo in un'unica finestra monolitica del tipo `Win_t` di **XTools**, sullo stile degli altri pacchetti di **XCModel** (*cfr. Fig. 3.6*)

- teorizzazione e ragionamento sull'essenzialità e sull'organizzazione delle singole aree (oggetti d'interfaccia, zone di resa, ...) della finestra monolitica

- correzione di tutti i problemi di ripristino dei contenuti in caso di **Expose**, demandando il *refresh* al pacchetto stesso, grazie all'utilizzo degli oggetti forniti dalla nuova versione di **XTools** e della resa su **PixMap** di tutti i contenuti grafici

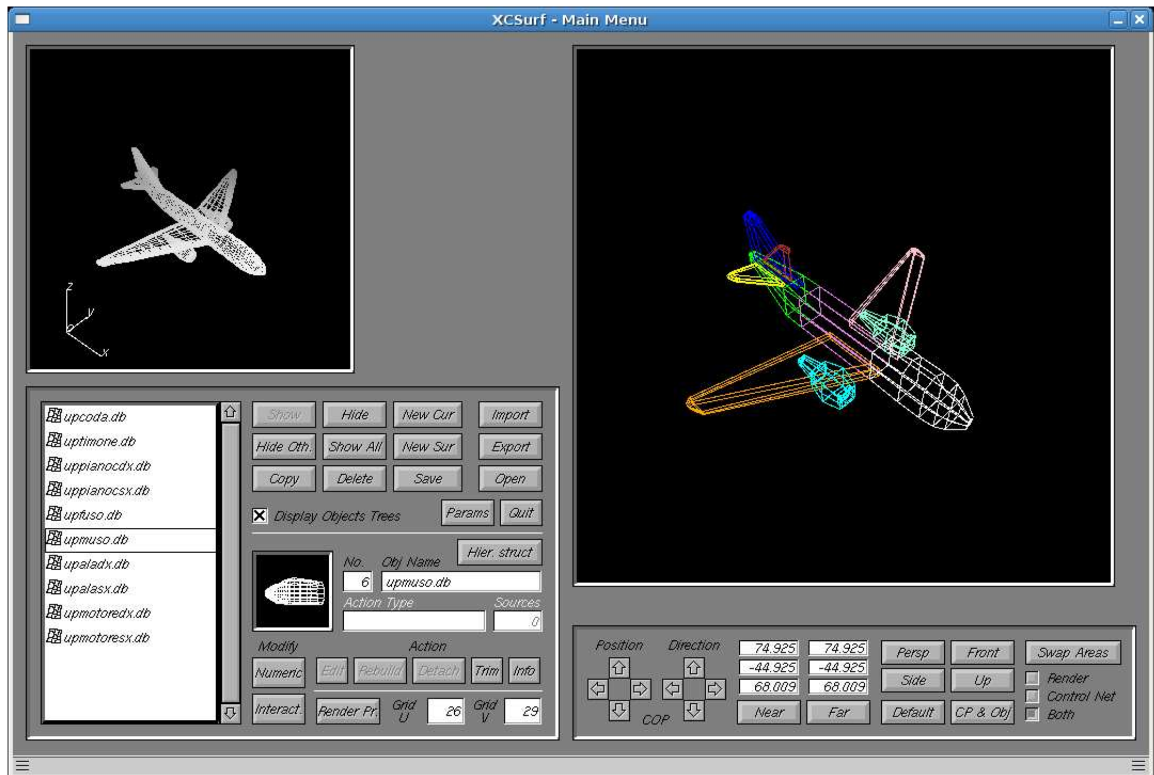

Figura 3.6: Nuova versione di **XCSurf**, in un'unica finestra monolitica.

Prima di tutto, si sono considerate le funzioni presenti in **XCSurf**, categorizzàndole e capendo quali di esse andassero modificate, e quanto.

- Le funzioni grafiche sono state radicalmente manomesse, in modo da indirizzare l'*output* su finestra monolitica e/o su **PixMap** e gestendo il *refresh* secondo le nuove possibilità offerte da **XTools**.

- Le funzioni di calcolo, al contrario, sono state alterate il meno possibile, in modo che le strutture dati predisposte ad ospitare i dati sulle curve e superfici **NURBS** e su tutti gli aspetti geometrici potessero essere mantenute, conservando coerenza tra loro. Le modifiche più consistenti sono avvenute in concomitanza di funzioni contenenti calcoli dipendenti da parametri d'interfaccia grafica (ampiezza dell'area di *rendering*,

ecc.). Al limite, ci si occuperà nella fase di resa (ossia, nelle sovraccitate funzioni grafiche) di interpretare correttamente tali dati geometrici, tenendo conto del mutato aspetto della finestra di lavoro.

- A loro volta, le funzioni di *input/output* sono state ampiamente modificate. Nel primo caso, s'è contemplata la nuova geometria della finestra, tenendo conto per esempio che le coordinate ad essa relative restituite da un evento di tipo **ButtonPress** del *mouse* sarebbero state un *offset* dal punto in alto a sinistra della finestra monolitica, e quindi si sarebbero dovute normalizzare all'effettiva area interessata dall'evento (zona di *rendering*, *control net* o quant'altro). Prima, invece, un *click* su una delle finestre esterne, ad esempio quella di *render*, avrebbe interessato solo l'aspetto del *rendering* ivi contenuto, senza coinvolgere altre finestre e potendo peraltro usufruire delle coordinate *pure* dell'evento di **ButtonPress**. Il secondo caso si rifà alle funzioni geometriche: ogni *output* va posto nella posizione corretta della finestra monolitica.

Successivamente, si sono fatte delle scelte operative sugli elementi da tenere inalterati, quelli da modificare e quelli da eliminare. Per quanto riguarda le sei finestre esterne, s'è deciso di includere nelle finestra monolitica solo quelle del *rendering* e della *control net*: le altre quattro (proiezioni ortogonali e prospettica) si sono escluse poiché si possono ottenere immagini identiche, ma più in grande ed in doppia versione (resa e *control net*), grazie ai pulsanti **Front**, **Side**, **Up** e **Persp** del *pannello di controllo* di **XCSurf**. La resa prospettica, peraltro, è il contenuto di *default* dell'area *render*.

Un altro aspetto caratteristico nella riprogettazione dell'interfaccia è stata la suddivisione della finestra monolitica in quattro aree: due per ospitare le *ex-finestre* di *render* e *control net* e due contenenti gli oggetti **XTools**, distinti tra costanti e non. **XCSurf**, infatti, prevede di poter commutare tra due visualizzazioni (sempre nella medesima finestra, l'unica che anche precedentemente fosse del tipo `Win_t` di **XTools**):

- **principale**, contenente il modello realizzato e la lista delle superfici che

lo compongono, con la possibilità di mostrare/nascondere una o più di esse, oltre ai pulsanti per crearne di nuove, per modificare le preferenze di resa e per interagire col *file system*

- **hierarchical**, per trattare una singola *superficie gerarchica* ed effettuare operazioni di *overlay* sulla stessa

Dividere la finestra in quattro parti indipendenti ha permesso la *parametrizzazione*: la posizione e, nel caso delle zone di disegno *render* e *control net*, le dimensioni di ogni area possono essere lette da un *file* di configurazione (chiamato `.xcsurfrc` e posto, a scelta, nella directory degli eseguibili di **XCModel**, in quella da cui l'utente lancia il programma od in `/etc`, in tal caso senza punto iniziale) ottimizzato secondo le esigenze dell'utente. Sostituisce il precedente file che memorizzava posizioni e dimensioni delle varie finestre, ora che non sono più presenti.

Un'altra caratteristica aggiunta è stata la possibilità, grazie ad un pulsante dedicato, di scambiare i contenuti delle due aree di disegno: ciò permette, tra le altre cose, di visualizzare nell'area più grande ciò che si desidera osservare più in dettaglio.

Una delle modifiche più interessanti riguarda la corrispondenza tra la *lista delle superfici* e gli identificativi dei *files* che le contengono. Precedentemente, **XCSurf** rinominava tutte le superfici lette da *file* in **Object *nn***, ove ***nn*** era un numero di due cifre (con eventuale *leading zero*) che aumentava progressivamente all'aggiunta di nuove superfici, a prescindere dalla provenienza di queste (da *file* o create *in loco* da **XCSurf**), pur mantenendo informazioni sul percorso di ognuna. Ciò comportava che all'atto del salvataggio di un oggetto, ogni superficie modificata (o creata) dovesse essere salvata, facendo apparire una finestra di `FileReq` per ciascuna di esse oltre che per l'oggetto che costituivano. Modificare tante superfici, quindi, avrebbe comportato numerosi salvataggi. Ora, invece, le superfici in lista mantengono l'identificativo del *file* da cui provengono, completo di estensione per distinguerne il tipo (`.db`, `.dbe`, *ecc.*), e quindi il salvataggio potrà avvenire tramite finestre di `FileReq` *precompilate* e di facile e rapido utilizzo.

3.3.1 Interventi.

Segue ora un *excursus* sulle numerose modifiche apportate ad **XCSurf**: esse riguardano sia operazioni necessarie o convenienti per l'allineamento agli altri pacchetti o ad **XCModel**, sia aggiustamenti di *bug* od imperfezioni preesistenti.

- La definizione delle **preferenze** per la *resa* degli oggetti è ora un pulsante a sè stante, accessibile direttamente dal pannello di controllo (*cfr. Fig. 3.7*). Ivi è possibile stabilire se il *rendering* debba avvenire per *Wire Frame*, *Hidden Line*, *Depth Cueing* oppure *Shading* (*Flat*, *Gourand* o *Phong*), oltre alla posizione della sorgente luminosa ed ai valori di riflessione della luce ambiente, diffusa e speculare.

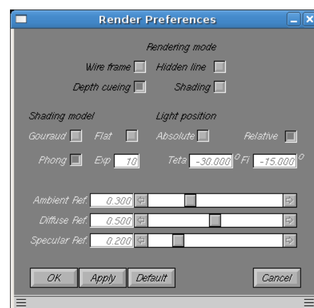

Figura 3.7: La finestra **Render Preferences**.

- Sono stati corretti alcuni errori nella visualizzazione e nella selezione dei punti di controllo durante le modifiche *numeriche* od *interattive*.

- Anche gli errori che avvenivano all'atto dell'apertura di certi formati di *file* (come `.ip`) sono stati eliminati

- Alcune imprecisioni nei parametri di **Free Form Deformation** (grado dei polinomi e suddivisioni, *cfr. Fig. 3.8*) sono state sistemate.

  Vi è infatti un vincolo secondo cui i valori di *subdivision* per ogni dimensione devono essere strettamente maggiori del corrispettivo *grado*

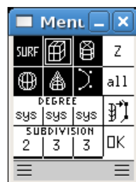

Figura 3.8: **Free Form Deformation**.

(*degree*). Infatti la modifica in *subdivision*, i cui numeri aumentano cliccando col *pulsante sinistro* del *mouse* e diminuiscono col *pulsante destro*, è implementata in modo che tali valori non possano mai crescere oltre a *5* o decrescere fino a diventare minori od uguali al *grado* omologo. I *gradi*, invece, si modificano solo col *pulsante sinistro* e còmmutano ciclicamente tra *sys*, *1* e *2*, e prima della correzione non v'era alcun controllo dei valori di *subdivision* sottostanti: ciò avrebbe comportato la possibilità di raggiungerli e superarli, ed un'incoerenza nell'invariante

- Sempre in **Free Form Deformation** c'era un errore per cui qualunque grado diverso da *sys* avrebbe sortito effetti indesiderati. È stato corretto con un semplice, chirurgico intervento direttamente nel codice, specificando che un elemento del vettore che eredita i dati immessi dall'utente nella finestra di **Free Form Deformation** venisse posto esattamente a quel valore (e non a *0*, com'era erroneamente).

- L'uso di un'unica finestra causava problemi con l'`XSelectInput`: prima, ogni finestra poteva avere la propria maschera di eventi da cogliere mentre, dopo, avrebbe fatto testo solo quella della finestra monolitica. La difficoltà principale s'è avuta con **ButtonMotionMask**, posseduta dalle precedenti finestre di *render* e di *control net*, dal momento che in esse ogni evento di **MotionNotify** avrebbe avuto effetti esclusivamente col pulsante del *mouse* premuto: tale condizione, tuttavia, non si ripete per la finestra monolitica. Il problema è stato risolto facendo ricorso alle nuove funzioni di **XTools** `StartLazyMotionNotify` e

`StopLazyMotionNotify`, che modificano a *run-time* la sensibilità della finestra al **MotionNotify** normale od a quello con tasto premuto.

- Alcune funzioni deducevano che valori utilizzare per i calcoli, tra quelli delle variabili a disposizione, a seconda del parametro *finestra* ricevuto. Ora che tale parametro è diventato costante (ossia, ha smesso d'esser necessario), è sufficiente comunicare alle suddette funzioni, modificandole leggermente nei contenuti e nei parametri (anche i prototipi), se si tratti di un'operazione su *render* o *control net*.

- Tutte le finestre di *Help*, dal comportamento simile (mostrare il messaggio ed attendere la pressione del pulsante destro del *mouse*, **Button3**), sono state accomunate da un semplice ciclo che, con `ProcessGUIEventsUntil`, attende la pressione del pulsante e provvede a nascondere la finestra.

- Le dimensioni complessive della finestra monolitica vengono determinate in base alla posizione ed alla superficie dei quattro elementi contenuti: si tratta, in pratica, della minima *bounding-box* in grado di ospitarli.

3.4 Conclusioni.

Come detto, tutto il lavoro relativo all'aggiornamento di **XCSurf** si può raccogliere in due *macro-famiglie*:

- aggiornamento alle nuove specifiche di **X-Org**

- allineamento agli altri pacchetti di **XCModel**

S'è trattato di un impegno decisamente lungo e complesso, ben più di quanto l'analisi iniziale del problema avesse dato ad intendere. Una volta modificate tutte le operazioni di *output* affinché operassero su **PixMap** e controllata la correttezza dell'aggiornamento e del *refresh*, infatti, s'aveva la sensazione che il lavoro fosse in discesa: non sembrava che l'accorpamento di tutto in un'unica finestra e la conseguente alterazione dell'*input* sarebbe stata

più di quel tanto complessa. In effetti, l'unificazione monolitica, compresa la parametrizzazione della finestra, non fu particolarmente difficile, ma la cattura dell'*input* da essa, ed in particolare la dipendenza dalla zona *cliccata* si rivelò un lavoro lungo e certosino, a maggior ragione visto il *debugging* di problemi preesistenti che s'operò nel suo ambito.

Innegabile, comunque, che sia stato un lavoro interessante e stimolante, che ha permesso l'interazione diretta con codice delicato e potente ed ha regalato la soddisfazione di toccare con mano gli effetti di ogni intervento, di scovare e correggere *bug* inaspettati e di partecipare ad un progetto accademico d'indiscusso valore e pregio.

Come risultato, s'è ottenuto un pacchetto completo e fruibile, coerente con lo stile della *suite* in merito ad apparenza e comportamento (e numero) delle finestre. Nonostante i difetti che certamente ancora contiene (sarà compito di studenti e tesisti scovarli e correggerli), permette un'interazione più semplice, immediata e gratificante per l'utente.

3.4.1 Gli sviluppi futuri.

È pleonastico sottolineare che **XCModel**, **XCSurf** e tutti gli altri pacchetti, più della maggior parte dell'altro *software* esistente, saranno sempre soggetti a costanti aggiornamenti e modifiche, data la già enfatizzata natura dinamica ed accademica del prodotto. Quel che si può fare è cercare d'ipotizzare quali strade possano prendere gli sviluppi futuri dei programmi, in base all'orientamento mostrato dalle moderne tecnologie informatiche e del settore grafico in particolare. Il primo, ovvio cambiamento che subirà l'intera *suite* nell'immediato futuro sarà il passaggio dall'attuale versione *patch* di **XTools** alla nuova versione corretta, completa ed in linea col progresso dettato da **X-Org**. Come precedentemente osservato, l'impegno per adattare i pacchetti alla nuova libreria sarà irrisorio, dell'ordine di pochi giorni di lavoro, se non ore: ciò è frutto del ragionamento e della teorizzazione che sono stati posti alla base dei recenti aggiornamenti del programma, che hanno ad uopo orientato l'elaborazione della *patch*.

Ragionando più ad ampio spettro, si possono immaginare vari orizzonti per **XCModel**, a seconda di quali paradigmi sia necessario (od accettabile) ammettere nella filosofia del suo sviluppo. Due grandi istanze di ciò possono essere le seguenti.

- **XCModel** potrebbe essere riprogrammato in **OpenGL**, avvaléndosi delle sue funzioni di *input*, di *output* e di *refresh*: così facendo, non soltanto vi si demanderebbero tutti gli oneri di interazione col sistema grafico e le periferiche, ma se ne sfrutterebbe la **portabilità** permettendo alla *suite* di girare su vari sistemi e di adattarsi ad un più ampio ventaglio di risorse *hardware*. Va da sè che un progetto del genere andrebbe pianificato in maniera più che mai ragionata e ponderata, data la delicatezza degli interventi da operare e la considerevole quantità degli stessi.
- Se **XCModel** abbandonasse (o, quantomeno, diventasse più elastico su) gli obiettivi primordiali che lo caratterizzano, vale a dire l'indipendenza da *hardware* specifico e l'orientamento all'**algoritmo**, potrebbe adattarsi ai principi dell'**accelerazione grafica**: le linee guida che da sempre lo caratterizzano sarebbero indebolite od annullate, e ciò non sembra affatto auspicabile né utile o didattico per i futuri sviluppatori, ma porterebbe all'istantanea (si fa per dire, viste le pregresse operazioni necessarie...) risoluzione di qualsivoglia problema di lentezza, *refresh* o carico di lavoro durante le operazioni pesanti.

Comunque vadano le cose, il *team* di sviluppo sa come pianificare ogni possibile lavoro, soppesarne il carico e valutarne l'efficacia, quindi si può star certi che qualunque strada verrà imboccata da **XCModel** varrà lo sforzo intrapreso per percorrerla.

Elenco delle figure

1.1 XCModel. 7
1.2 XCCurv. 8
1.3 XCSurf. 8
1.4 XCBool. 9
1.5 XCRayt. 10
1.6 XCView. 11
1.7 Una finestra ne copre un'altra: è necessario preservare le parti nascoste per ridisegnarle quando ridiverranno visibili. 11
1.8 Schema operativo della *pipeline*. 14
1.9 Il *double-buffering*. 15

2.1 Il *logo* di **X-Window**. 17
2.2 **OpenOffice.org Writer**. 18
2.3 The **GIMP**. 18
2.4 Il **Window Manager TWM**: come tutti, esso implementa autonomamente gli strumenti per la creazione di un'interfaccia. 20
2.5 **GNOME DeskTop** per **Debian Linux Etch**. 21
2.6 Il *logo* di **X-Org**. 22
2.7 Il **modello Client/Server**. 23
2.8 **SuperPipeline II** (**Tubature**), **Lazy Jones** (**Sala Giochi**) e **Jet Set Willy II** (**Ponzie**). 25
2.9 Esempi di finestre e relazioni: **1** è la radice, **2** e **3** sono sue figlie e sorelle tra loro (**2** è in parte non visibile), **4** e **5**, sorelle, sono **sottofinestre** di **3** (**4** esce parzialmente). 29

2.10 Il *logo* di **OpenGL**. 32
2.11 Le **primitive geometriche** di **OpenGL**. 34
2.12 **Church Virtual Tour**. 36
2.13 Il *DeskTop* di **MicroSoft Windows Vista**. 38
2.14 La *pipeline* di **OpenGL**. 40

3.1 I pacchetti della *suite* **XCModel**. 48
3.2 **Crypt Tabernacle**. 50
3.3 Programma **Test Bars**. 51
3.4 Versione precedente di **XCSurf**, a più finestre. 61
3.5 Modalità di **Rendering**: *Wire Frame*, *Hidden Line*, *Depth Cueing*, *Flat Shading*, *Gourand Shading* e *Phong Shading*. . . . 62
3.6 Nuova versione di **XCSurf**, in un'unica finestra monolitica. . . 63
3.7 La finestra **Render Preferences**. 66
3.8 **Free Form Deformation**. 67

Bibliografia

[JON89] O. Jones. *Introduction to the X-Window system*, Prentice Hall (1989).

[NYE92] A. Nye. *XLib programming manual*, O'Reilly and associates, vol. I (1992).

[XWINDOW] X Window System,
`http://it.wikipedia.org/wiki/X_Window_System`

[XORG] X.Org,
`http://it.wikipedia.org/wiki/X.Org`

[XWINDPA] X Window System protocolli e architettura,
`http://it.wikipedia.org/wiki/X_Window_System_protocolli_e_architettura`

[XWINDCP] X Window System core protocol,
uffer `http://it.wikipedia.org/wiki/X_Window_System_core_protocol`

[FRAMEB] FrameBuffer,
`http://en.wikipedia.org/wiki/Frame_buffer`

[SCHEIF] R. W. Scheifler. *X Window System Protocol*, X Version 11, Release 6.3, X Consortium Standard.

[GETSCH] J. Gettys, R. W. Scheifler. *Xlib - C Language X Interface*, X Version 11, Release 6.3, X Consortium Standard.

[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

[XCSURF] G. Casciola. *XCSurf: the 3D modeller* User's guide (1999),
http://www.dm.unibo.it/~casciola/html/xcmodel.html

[XTOOLS] G. Casciola, S. Bonetti. *XTools library: Programming Guide*, Version 2.0 (2001).

[CAMO00] G. Casciola, S. Morigi. *The trimmed NURBS age: advances in theory of computational mathematics: recent trends in numerical analysis* vol. III, Ed. D. Trigiante, Nova Science Publishers, Inc. (2000).

[CASC02] G. Casciola. *La Libreria Grafica OpenGL*, Argomenti del Corso di Grafica (2001/2002).

[BOTT02] M. Botton. *Modellazione dinamica di curve e superfici mediante albero di costruzione*, Tesi di Laurea, Università di Bologna (2002).

[ORLA03] L. Orlandini. *Modellazione gerarchica con curve e superfici*, Tesi di Laurea, Università di Bologna (2003).

[PANT04] A. Pantaloni. *Modellazione gerarchica con curve e superfici NURBS*, Tesi di Laurea, Università di Bologna (2004).

[ANGH04] M. Anghileri. *Modellazione gerarchica in XCSurf: progettazione e sviluppo di un ambiente dedicato*, Tesi di Laurea, Università di Bologna (2004).

[ANGE06] E. Angel. *Interactive Computer Graphics*, Fourth Edition, Pearson - Addison - Wesley (2006).

[TRON07] C. Tronche. *Graphics and User Interfaces - The X-Window System - Xlib Programming Manual* (2007),
http://tronche.com

[CLEM08] G. Clemens. *lists.x.org Mailing Lists* (2008), `http://lists.x.org/mailman/listinfo`

[PREM10] G. Premi: *XTools: refactoring della Graphical User Interface di XCModel*, Tesi di Laurea, Università di Bologna (2010).

[FERR10] A. Ferracin. *Algoritmo di best path in tempo reale, confronto tra elaborazioni su CPU e GPU*, Tesi di Laurea, Università di Bologna (2010).