STUDIO E SVILUPPO DI UN PROGETTO PER LA PORTABILITÁ MULTIPIATTAFORMA DI UN SISTEMA PER LA VISUALIZZAZIONE DI AMBIENTI VIRTUALI TRIDIMENSIONALI INTERATTIVI

ILARIO SANSEVERINO

Giugno 2014

ABSTRACT

Object of this thesis is to investigate the existing solutions available for cross-platforming, analysing in particular the specific case study of the conversion of a large project, related to a complex multimedia application native to the Microsoft (MS) Windows environment, into a multi-platform project, discussing the problems encountered and the solutions adopted.

SOMMARIO

Lo scopo di questa tesi è di investigare le soluzioni esistenti per la portabilità multipiattaforma, in particolare analizzando lo specifico caso di studio della conversione di un grande progetto, relativo ad una complessa applicazione multimediale nativa per l'ambiente MS Windows, in un progetto multipiattaforma, discutendo dei problemi incontrati e delle soluzioni adottate.

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

ELENCO DEI LISTATI

ACRONIMI

API Application Programming Interface

EAX Environmental Audio eXtensions

EFX Effects eXtensions

FFI Foreign Function Interface

GCC GNU Compiler Collection

GLSL GL Shading Language

GLUT OpenGL Utility Toolkit

GPL GNU General Public License

GTK+ GIMP ToolKit

HMD Head Mounted Display

IDE Integrated Development Environment

JIT Just In Time

LGPL GNU Lesser General Public License

LOD Level Of Detail

MCI Multimedia Control Interface

MinGW Minimal GNU for Windows

MS Microsoft

MSDNAA Microsoft Developer Network Academic Alliance

MSVC Microsoft Visual C++

MSVS Microsoft Visual Studio

POD Plain Old Data

PPC PowerPC

PTS Presentation TimeStamp

RCOD Red Cross Of Death

SCUMM Script Creation Utility for Maniac Mansion

SDK Software Development Kit

SDL Simple DirectMedia Layer

SVN Subversion

VM Virtual Machine

Wine Wine Is Not an Emulator

WSA Windows Socket API

XVR eXtreme Virtual Reality

1 INTRODUZIONE

Per cross-platforming si intende comunemente la capacità di un software di poter essere eseguito su due o più configurazioni diverse di hardware/sistema operativo, come ad esempio un gioco che funziona sia su PC che su console, o un programma in grado di girare su MS Windows, GNU/Linux e Mac OSX senza differenze di comportamento eccessivamente marcate.

Agli albori dell'home computing gli sviluppatori non avevano a che fare con una grande varietà di sistemi. I primi microcomputer, come l'Apple II o il Commodore PET, avevano un hardware ben definito e scarse possibilità di modifica. Inoltre erano oggetti difficili da reperire e parecchio costosi, quindi chi possedeva un calcolatore si concentrava sullo sviluppo per quella macchina specifica, cercando di sfruttare al meglio le componenti a disposizione.

Con la sempre maggiore diffusione dell'elettronica di consumo la situazione ha cominciato a cambiare: le prestazioni dei componenti tecnologici sono migliorate incredibilmente, i prezzi dell'hardware sono scesi, sul mercato sono approdati i PC assemblati; inoltre la produzione di software è diventata un affare molto redditizio, provocando un cambiamento sostanziale dell'approccio produttivo, passando dall'appassionato che programma "artigianalmente" nel suo garage alle aziende che producono per vendere al maggior numero di utenti possibile. Pertanto è diventato imprescindibile avere layer d'astrazione sull'hardware e standard d'interoperabilità.

Negli anni '90 MS Windows ha fatto indiscussamente da padrone sul mercato PC consumer; le esigenze di programmazione multipiattaforma erano per lo più limitate all'ambiente videoludico, in cui distribuire un gioco sia su PC che sulle varie console disponibili poteva essere un affare remunerativo. Al giorno d'oggi invece la situazione è molto più complicata: con l'avvento degli smartphone gli utenti non sono più tanto disposti a tollerare che i loro programmi preferiti siano legati ad un sistema operativo particolare, si aspettano di poter passare da telefono a tablet a PC in maniera rapida ed efficace. Inoltre è più probabile che siano i software ad essere acquistati in base alla disponibilità per il proprio dispositivo che non il viceversa

(come invece accade per le console, in cui esiste il concetto di *killer app*, il software talmente desiderato da spingere le vendite dell'hardware).

Il mobile computing non ha solo portato una grande diffusione di sistemi operativi non Microsoft tra gli utenti, ma anche nuovi paradigmi di utilizzo ed interazione basati su sensori e dispositivi, quali il GPS, i sensori inerziali o gli schermi multitouch, fino ad allora indisponibili o non utilizzati. La programmazione multipiattaforma comporta, dunque, tutta una serie di problemi, tecnici (quali caratteristiche sono riconosciute dal compilatore? Quali librerie posso dare per scontate e quali invece devo fornire insieme al programma? Quali feature non supportate possono essere sostituite, quali escluse e quali invece devono essere considerate vitali?) e d'interfaccia (in quale modo posso efficacemente tradurre una complessa combinazione di tasti per joystick in gesture e accelerazioni?).

In compenso il cross-platforming presenta anche molti vantaggi. In primo luogo la disponibilità per sistemi differenti consente all'utente di utilizzare i programmi che desidera in un unico ambiente, senza la necessità di multi-boot o macchine virtuali dedicate. Inoltre si aggiunge un layer d'astrazione anche sul software, e come affermò una volta David J. Wheeler "all problems in computer science can be solved by another level of indirection" [10]. Infine programmare tenendo presente fin da subito le esigenze legate alla portabilità del codice costringe a tenersi il più possibile aderenti agli standard, rinunciando alle scorciatoie e alle soluzioni che "per ora funzionano, in futuro si vedrà"; cosa che inizialmente porta un po' più di lavoro e fatica, ma nel lungo termine paga parecchio in termini di mantenibilità ed estendibilità. Progetti particolarmente complessi richiedono attenzione, perché hanno requisiti stringenti di timing, funzionalità di rete, multimedialità, ecc. Se il codice non è chiaro e pulito diventa molto difficile trovare e correggere i bug, aggiungere nuove caratteristiche e mantenere il programma al massimo dell'efficienza. Tra i progetti più complessi ci sono sicuramente quelli orientati alla produzione di software multimediale, dal momento che richiedono solitamente una grande quantità di moduli software interconnessi che gestiscono i vari aspetti legati ai diversi media. Tra questi meritano una menzione particolare i progetti per la fruizione di applicazioni di realtà virtuale, che richiedono l'esecuzione di task complessi come la grafica tridimensionale real time, la riproduzione di audio posizionale, il motion tracking, la gestione di periferiche di varia

Quando il sentiero oscuro tu intraprendi, per sempre esso dominerà il tuo destino

natura.

Quando si parla di realtà virtuale non esiste una definizione ben precisa, ma ci sono alcuni concetti chiave validi in ogni possibile definizione. Tanto per cominciare la realtà virtuale è un ambiente generato da un elaboratore, quindi è scorretto usare l'espressione riferendosi a sogni o visioni. L'ambiente deve coinvolgere uno o più sensi dell'utente, perciò anche le avventure testuali non sono da considerarsi realtà virtuale (la lettura non *coinvolge* la vista, si limita ad usarla come mezzo per il trasporto di informazione semantica codificata). L'immersione deve essere tale da indurre l'utente a percepire come reale l'ambiente generato, e questa è la parte più difficile da quantificare in quanto molto soggettiva. Sicuramente un simulatore di volo è da considerarsi realtà virtuale, un videogioco di tipo platform 2D non lo è, uno sparatutto in prima persona potrebbe esserlo.

Cinema e letteratura hanno ampiamente esplorato i concetti legati alla realtà virtuale. Nella serie televisiva Doctor Who ad esempio fu presentato (nel 1976) il *Matrix*, un supercomputer in cui sono registrate coscienze e conoscenze di tutti i Signori del Tempo; nel serial era già presente il topos della morte virtuale che provoca anche la morte nel mondo reale. Nel racconto Burning Chrome lo scrittore William Gibson coniò il termine *cyberspazio*, ormai entrato ufficialmente a far parte del dizionario. In generale la fiction usa il tema della realtà virtuale per questioni più filosofiche che tecnologiche: l'idea dell'essere *intrappolati nel cyberspazio* e la difficoltà nel distinguere tra mondo reale e mondo virtuale sono per lo più ascrivibili al mito platonico della caverna.

Nella pratica nessuno lavora a tecnologie che prevedano il riversamento della propria coscienza in un non-luogo gestito tramite computer. Gli ambienti virtuali possono essere progettati per abbattere costi e tempi di produzione: prima di creare fisicamente un prototipo un'azienda potrebbe fare dei test preliminari su un modello virtuale, che può essere rapidamente modificato o riprogettato a costo quasi nullo. Oppure potrebbero essere utilizzati per dare un senso di compresenza a persone fisicamente lontane, come nel caso di una teleconferenza. Anche nell'insegnamento la virtualità si rivela molto utile: lavorare sotto pressione, come durante un'operazione chirurgica complessa o su un campo di battaglia, richiede l'efficienza e la freddezza che si acquisiscono solo con l'esperienza, ed esercitarsi in ambienti virtuali consente di fare pratica senza mettere a rischio vite umane.

L'obiettivo di questa tesi è quello di portare i vantaggi del cross-platforming nel progetto di XVR, un framework per lo sviluppo e l'esecuzione di applicazioni di realtà virtuale, realizzato dal Laboratorio PERCRO della Scuola Superiore Sant'Anna, che consiste in un vasto e complesso progetto fino ad oggi disponibile solo su piattaforma Microsoft Windows. La scelta di un simile compito è stata fatta sulla base di vari fattori. In primo luogo avevo intenzione fin da subito di lavorare ad una tesi implementativa, perché la programmazione mi affascina fin da quando provai un videogioco per la prima volta (*Radar Rat Race* su Commodore Vic-20). Inoltre volevo imparare il linguaggio C++, perché è una conoscenza imprescindibile per un programmatore professionista. Infine quando mi è capitato nel corso di un esame di dover usare il framework XVR ne sono rimasto entusiasta, quindi ho colto al volo l'occasione di poter contribuire allo sviluppo di un qualcosa che rendesse più facile la vita degli studenti anziché complicargliela inutilmente. Come se non bastasse il programma è moderatamente complesso (nell'ordine del milione di righe di codice), quindi permette di esplorare un'ampia gamma di problematiche relative al linguaggio, e la possibilità di lavorare a stretto contatto con gli sviluppatori originali vuol dire ottenere con facilità chiarimenti su eventuali parti di codice la cui documentazione fosse scarsa o datata.

Ciò che mi è stato richiesto era la possibilità di compilare ed eseguire l'engine di XVR, cioè la versione stand-alone della macchina virtuale e le librerie aggiuntive per la gestione degli ambienti (mesh 3D, audio, fisica, I/O), almeno su un altro sistema operativo desktop oltre a Windows; ho scelto quindi come obiettivo il sistema principale installato sul mio computer, Arch Linux x86_64.

2

STATO DELL'ARTE

Molti sono oggigiorno i software cross-platform. Si va dai videogiochi (l'intramontabile *The secret of Monkey Island* uscì originariamente per Amiga, Atari, MS DOS, Mac OS, Sega CD... In seguito fu distribuita una versione CDROM per MS Windows, e nel 2009 è stata pubblicata una nuova edizione disponibile anche per iOS e Xbox) ai programmi di uso comune (Skype ha versioni per ogni piattaforma moderna, sia desktop che mobile) a quelli professionali (Blender esiste in versione precompilata per tutti i sistemi desktop attuali, esclusi quindi Windows precedente a XP e Mac OSX PPC; MS Office viene venduto per Windows, Mac OSX, smartphone e tablet).

Per ottenere la portabilità delle applicazioni sono però pochi i metodi conosciuti, e quando si parla di motori grafici, specie se 3D, il campo diviene ancora più ristretto. Le tecniche per il cross-platforming dei programmi sono sostanzialmente tre, distinte in base al layer su cui decido di astrarre dalla macchina. La prima possibilità è non astrarre affatto, si utilizza codice diverso su macchine diverse. La seconda è astrarre sul compilatore, l'approccio usato dai linguaggi interpretati. La terza è astrarre sul sistema, quello che avviene con la virtualizzazione; in questo caso però, poiché l'astrazione avviene ad un livello più alto di quello a cui opera il programmatore, il lavoro di portabilità spetta all'utente finale.

Esistono inoltre altre due tecniche di *metaportabilità*: una è la *portabilità per libreria*, l'altra è la *portabilità al contrario* di Wine, in cui anziché rendere il programma compatibile con molti sistemi si preferisce rendere il sistema compatibile con i programmi.

2.1 MACCHINE VIRTUALI

Una delle possibilità per poter utilizzare un programma su una piattaforma diversa da quella per cui è stato progettato consiste nell'utilizzo di Virtual Machine (VM). In questo settore esistono molte soluzioni, sia proprietarie (VMWare, Parallels, Virtual PC) che libere (KVM, QEMU). Le possibilità offerte da una VM sono molteplici, dalla traduzione al volo delle istruzioni della CPU, per poter eseguire programmi su architetture diverse

da quella prevista, alla simulazione di interi sistemi hardware, comprensivi di periferiche, su cui poter installare un sistema operativo senza bisogno di applicarvi modifiche o patch.

L'uso principale delle VM è nell'ambito della sicurezza: se ho diversi utenti che condividono la stessa macchina posso farli lavorare su istanze distinte di VM, ottenendo in tal modo un buon livello di isolamento tra loro, così in caso di compromissione di uno dei sistemi virtuali è meno probabile che l'attaccante riesca a prendere il controllo di tutta la macchina. Questo tipo di approccio è usato nell'analisi di malware dal progetto ZeroWine: un'immagine per QEMU contiene una VM su cui è installata una versione di Debian configurata per l'uso di Wine Is Not an Emulator (Wine) (2.5); il presunto malware viene avviato e, tramite le informazioni di debug fornite da Wine, è possibile vedere quali chiamate di sistema effettua. Un software appositamente studiato per compromettere ZeroWine dovrebbe sfruttare una vulnerabilità di Wine per poter agire come un utente Debian senza privilegi, da lì sfruttare qualche vulnerabilità di Debian o di QEMU per poter uscire dall'ambiente di lavoro virtuale e ritrovarsi ad agire come utente standard sulla macchina host, di tipo sconosciuto e non necessariamente una macchina fisica. Ovviamente un malware simile non è impossibile da realizzare, ma il rapporto costi/benefici suggerisce che è molto improbabile incontrarne uno.

Un altro settore in cui le VM sono particolarmente utili è quello dell'ingegneria inversa. Facendo girare un'applicazione su un layer virtuale si può controllare ogni singola istruzione eseguita, gli accessi all'hardware effettuati, è possibile modificare i risultati delle chiamate di sistema, fermare l'esecuzione dell'intera macchina, salvare tutto lo stato del sistema per controllare come il flusso del programma varia alla modifica di un singolo input... Il tutto senza che (teoricamente) l'applicazione si renda conto dell'esame in corso.

Quando però si usano VM di questo genere per lanciare applicazioni su sistemi diversi da quelli pubblicizzati i vantaggi non sono così evidenti. Un sistema operativo, anche se emulato, ha delle richieste hardware minime che si aggiungono a quelle del sistema ospitante, la più problematica delle quali è probabilmente la memoria a disposizione. Inoltre gli accessi all'hardware sono indiretti, quindi più lenti, perché il driver virtuale deve passare le richieste al controllore sottostante che le tradurrà per il driver fisico. Il supporto per l'accelerazione 3D è abbastanza raro e, quando presente, comunque limitato. Dal

momento che i moderni videogiochi sono molto esigenti sia in fatto di scheda video che di risorse di sistema è impossibile virtualizzarli su sistemi desktop, ma anche altri tipi di applicazioni possono dare problemi. Le periferiche virtuali messe a disposizione del sistema ospite sono di tipo generico, in maniera che siano facilmente mappabili sulle periferiche fisiche presenti; un'applicazione con requisiti specifici, come ad esempio un programma per l'aggiornamento di device esterni (telefoni cellulari, navigatori satellitari) richiede la presenza di driver particolari che di norma non funzionano se installati in sistemi operativi virtualizzati. Infine l'isolamento, che tanti benefici forniva in campo di sicurezza informatica, diventa un fastidio nel caso del multiplatforming. L'utente si aspetta un'esperienza omogenea, con interfacce che rispettano certe linee guida dipendenti dal sistema in uso. Un sistema operativo virtualizzato invece ha un desktop indipendente, segue i propri paradigmi e non si integra minimamente con le applicazioni del sistema ospitante (salvo qualche caso in cui venga supportato il drag&drop tra i due sistemi).

I videogiochi, in quanto applicazioni multimediali ad elevato grado di interattività con requisiti stringenti in fatto di timing e rendering, pongono ai programmatori problematiche molto simili a quelle affrontate nello sviluppo di XVR, pertanto sono casi di studio di particolare rilievo nell'ambito di questa tesi. Vedremo quindi un paio di esempi di successo dell'uso di macchine virtuali per la portabilità in ambito videoludico, e se ciò può sembrare in contraddizione con quanto affermato poc'anzi si tenga presente che ho parlato delle difficoltà nella virtualizzazione di titoli moderni, mentre per il retrogaming si possono ottenere ottimi risultati.

Si tenga presente la legge di Moore

2.1.1 *ScummVM*

L'applicazione ScummVM permette di giocare su praticamente ogni piattaforma possibile (smartphone compresi) con le vecchie avventure grafiche, inizialmente solo quelle prodotte dalla LucasArts (SCUMM è l'acronimo di Script Creation Utility for Maniac Mansion, il motore scritto per la creazione del gioco *Maniac Mansion* e riutilizzato per molti dei titoli successivi), in seguito ampliato ad altri engine e software house (la versione 1.6.0 è compatibile con oltre 200 titoli differenti). ScummVM è un ottimo esempio di portabilità tramite virtualizzazione: le applicazioni funzionano alla grande, anche su sistemi troppo

moderni per gli eseguibili originali, e persino la versione touchscreen è abbastanza usabile. È però obbligatoria una riflessione: tutti i titoli usano grafica 2D senza alcun tipo di accelerazione hardware, se erano fluidi sulle macchine degli anni '90 possono tranquillamente girare sui computer moderni senza perdite di prestazioni, persino su sistemi operativi virtualizzati all'interno di applet JAVA. Inoltre il fatto che quei giochi fossero scriptati ed interpretati già in originale aiuta parecchio, perché è sufficiente programmare solamente l'interprete del linguaggio senza necessità di ricreare l'intero sistema operativo.

2.1.2 *Mupen64+*

Se l'inefficienza delle macchine virtuali vale per i videogiochi moderni, altrettanto non si può dire per quelli un po' datati. Delle vecchie console esistono molti emulatori, in grado di far girare in maniera godevole anche giochi 3D. Mupen64plus è un emulatore di Nintendo64, quindi un programma parecchio più complicato rispetto a ScummVM, in quanto deve emulare un intero hardware (la console originale ha un processore MIPS). Mupen64plus è stato scritto in C/C++, la portabilità è garantita da codice ad hoc (mantenuto principalmente su MS Windows e Linux). I giochi avviati tramite Mupen64 sono molto fluidi, la giocabilità è veramente ottima, ma questo dipende dal fatto che il Nintendo64 aveva 4Mb di memoria RAM e una GPU capace di renderizzare 100.000 poligoni al secondo, un'inezia se paragonati ad esempio ai 275.000.000 di triangoli di una Sony Playstation 3.

2.2 LINGUAGGI INTERPRETATI

Come già accennato a pagina 5 le VM non devono necessariamente simulare un intero hardware, ma possono anche limitarsi alla traduzione delle istruzioni di un singolo programma. Questa facoltà è utilizzata dai linguaggi interpretati: si scrive un programma in un linguaggio di alto livello, eventualmente lo si compila in un bytecode di medio livello, indipendente dall'architettura o dal sistema operativo, ed infine lo si esegue tramite un interprete specifico che traduce le istruzioni del linguaggio in istruzioni macchina. Di fianco alla portabilità tra architetture diverse così ottenuta, i linguaggi interpretati offrono anche una serie di librerie per le funzionalità di base del sistema operativo; in questo modo i programmi accedono all'hardware attraverso

Write once, run anywhere

un'interfaccia standard, mentre l'interprete ha la responsabilità di eseguire tali funzioni per mezzo di chiamate al sistema sottostante.

Poiché astraggono dal sistema fisico, i linguaggi interpretati consentono ai programmatori di concentrarsi solo sulle funzionalità, senza preoccuparsi di quale sia la dimensione di un intero o l'endianness con cui viene rappresentato dal processore. Inoltre la traduzione al volo del codice in istruzioni macchina consente di avere, in caso di errore, molte informazioni per poterne scoprire l'origine. Se l'interprete è privo di difetti, un programma interpretato che funziona bene su di un sistema funzionerà altrettanto bene su tutti i sistemi con requisiti hardware sufficienti, rendendo semplice ed efficace il mantenimento del codice. Come ulteriore punto a favore dei linguaggi interpretati le funzioni di basso livello sono gestite dall'interprete anziché dal programmatore; le librerie standard del linguaggio vengono costantemente migliorate e mantenute, garantendo la massima efficienza ed affidabilità, gli sviluppatori non hanno il peso di doversi assicurare che una risorsa fisica o una porzione di memoria sia stata liberata correttamente.

"ArrayIndexOutOfBoundException:32" è un po' più significativo di "Segmentation fault"

Tutto ciò ovviamente ha un costo. Per prima cosa i linguaggi interpretati sono più lenti, perché le istruzioni devono essere tradotte volta per volta. La generazione di bytecode intermedio e l'utilizzo di compilatori Just In Time (JIT) permette di ridurre parecchio questa latenza, ma non di azzerarla totalmente. Anche il fatto che l'interprete e le librerie standard siano continuamente aggiornate porta a degli aspetti negativi, perché a differenza di un programma compilato per una certa architettura e sistema operativo, che funziona sempre allo stesso modo, l'esecuzione di un programma interpretato dipende dalla versione del runtime utilizzato. Versioni differenti del runtime potrebbero comportarsi in maniera diversa con lo stesso programma, introducendo bug in software già ampiamente testato e consolidato. Per ovviare a ciò gli sviluppatori devono sempre verificare la correttezza del proprio codice sulla versione più aggiornata dell'interprete, dando per scontato che sia quella usata dagli utenti: gli aggiornamenti servono infatti a chiudere falle di sicurezza, non bisogna evitarli a meno che non si abbia un valido motivo per farlo. Per gli utenti la cosa è un po' più problematica: se un aggiornamento dell'interprete rompe la compatibilità con un programma non più mantenuto forse la soluzione migliore è trovare un programma alternativo ancora attivo.

Altro aspetto da prendere in considerazione nei linguaggi interpretati sta nel *look&feel* delle applicazioni grafiche. A pagina 7 si è anticipato questo problema, che ora andremo a vedere più nel dettaglio. Ogni sistema operativo/window decorator ha i propri bottoni predefiniti disposti in una certa maniera. Un linguaggio come JAVA, che fornisce un suo decoratore di finestre, produce interfacce che risultano aliene su *tutti* i sistemi su cui viene fatto girare; ed anche quando viene usato il decoratore di sistema per le finestre, rimangono irrisolti gli aspetti relativi al *feel*: il bottone più significativo di una finestra di dialogo sta a sinistra o a destra? Quale elemento deve avere il focus di default? A quale voce del menu devo accedere per personalizzare l'applicazione?

2.2.1 *Java*

Il più importante dei linguaggi interpretati è sicuramente JAVA. Si tratta di un linguaggio fortemente tipato dalla sintassi dichiaratamente copiata da quella di C++. Il linguaggio si propone infatti come possibile sostituto del C++, forse un po' meno efficiente, ma più sicuro perché effettua controlli migliori sul comportamento dei programmi (l'esecuzione dei quali è circoscritta da un modello di tipo *sandbox*). Per le sue caratteristiche di portabilità, affidabilità e buona efficienza è stato scelto come linguaggio di programmazione principale per le piattaforme Android. Il caso d'uso tipico di JAVA è per la scrittura di applicazioni per uso professionale, dove la stabilità e la sicurezza sono fattori imprescindibili. In ambito di grafica tridimensionale, in cui le prestazioni sono il benchmark principale, stenta un po' a prendere piede, tant'è che solamente con la release 8 del linguaggio le Application Programming Interface (API) 3D sono state integrate ufficialmente (nella libreria JavaFX), in precedenza erano solo un progetto sviluppato dalla comunità. Uno dei software più ambiziosi in ambito di programmazione grafica 3D in JAVA è JMonkey Engine[1].

2.2.1.1 *JMonkey Engine*

JMonkey è una suite completa di programmazione, open source, che usa una versione estesa di NetBeans come Integrated Development Environment (IDE) di sviluppo. Lo scene graph viene creato tramite un editor visuale, ed è composto sia da no-

1 http://jmonkeyengine.org/

di geometrici, collegati a mesh e materiali, che da nodi logici, contenenti solamente trasformazioni ed in grado di raggruppare più nodi insieme. La logica viene gestita con due meccanismi distinti: gli *application state* descrivono comportamenti generali, come la differenza tra il gioco vero e proprio e la schermata di pausa, mentre i *custom controls* possono essere attaccati ai singoli nodi della scena. Per la fisica JMonkey si appoggia a JBullet, implementazione Java di Bullet Physics Library, anche se per il futuro è prevista l'integrazione nativa (con accelerazione hardware) del motore. Il renderer grafico è OpenGL, e per la gestione dei materiali viene usato uno shader monolitico. Per estendere lo shader si scrivono frammenti di codice GL Shading Language (GLSL) indipendenti, con input e output ben definiti, incapsulati in oggetti di tipo *shader node*, i quali vengono assemblati insieme dal sistema per creare lo shader definitivo. Tra le varie utility la suite offre classi built-in per lo shadowing, sia come renderer che come filtro post-renering, nonché un algoritmo di semplificazione automatica delle mesh per la generazione di Level Of Detail (LOD).

2.2.2 *Web apps*

Le applicazioni web sono un tipo particolare di applicazioni interpretate. Anziché richiedere uno specifico interprete installato sul sistema possono essere eseguite tramite un qualunque web browser, rendendole di fatto ubique. Il concetto non è nuovo (sono anni che i router vengono configurati tramite un'interfaccia web), ma la discriminante tra una mera pagina web ed una vera e propria applicazione sta nell'uso di tecnologie quali Ajax e HTML5 che permettono di riprodurre video e audio senza necessità di plugin esterni, usare immagini vettoriali, modificare la pagina senza bisogno di ricaricarla completamente, avvicinando così l'esperienza web a quella delle normali applicazioni stand alone.

Un aspetto cruciale delle applicazioni web è che sono concepite soprattutto in un'ottica di cloud computing: tramite il browser si fornisce all'utente soltanto una *vista*, mentre la validazione e l'elaborazione dei dati avvengono sul server, così come l'accesso e l'archiviazione. È vero che lo standard HTML5 prevede anche delle API per l'uso di web apps in modalità offline, ma si tratta di una caratteristica opzionale progettata più che altro come soluzione tampone tra due distinte connessioni, per permettere di lavorare anche in caso di temporanea man-

canza di rete oppure in mobilità. Il punto di forza delle applicazioni web sta infatti nei continui aggiornamenti del client, che avvengono ogniqualvolta l'utente si connette al server, oltre alla potenza di calcolo e memoria del server, smisurata rispetto ai comuni sistemi desktop (ancor di più se l'utente sta usando l'applicazione su uno smartphone o tablet).

Cloud computing ed elaborazione a livello server sono croce e delizia delle web app: se da un lato forniscono grande capacità computazionale e portabilità ovunque, dall'altro legano l'utente in maniera molto stretta alla compagnia che eroga il servizio. I dati salvati sul server non sempre sono accessibili al di fuori dell'applicazione, quindi una migrazione verso un servizio concorrente potrebbe rivelarsi impraticabile; il client viene scaricato ogni volta che ci si connette, pertanto non è possibile usare una vecchia versione del software; se l'azienda chiude i battenti l'utente non può far altro che trovarsi un nuovo software, non può accontentarsi di utilizzare l'ultima versione stabile disponibile (perché non è disponibile alcuna versione). Per un esempio concreto si pensi alla chiusura di Google Reader, che da un giorno all'altro ha lasciato migliaia di utenti orfani del loro aggregatore di news preferito.

2.2.2.1 *Construct 2*

La suite Construct 2[2] è un programma scritto in C++ che permette di costruire facilmente applicazioni HTML5. È indirizzato a persone con scarse conoscenze di programmazione, permette di creare interi giochi usando drag&drop e descrizioni dei comportamenti basati su un linguaggio di scripting il più semplice possibile, e genera in output codice JAVASCRIPT in grado di usare l'accelerazione 3D tramite WebGL. Le applicazioni create con Construct 2 non solo girano su qualunque browser moderno, ma possono anche essere esportate come applicazioni desktop (usando node-webkit[3]), applicazioni native per Windows 8 oppure Facebook games (tramite appositi plugins).

2.3 SORGENTI AD HOC

L'ultima possibilità per l'ottenimento di codice multipiattaforma è quella di tenere diverse versioni del codice sorgente, una per ogni possibile architettura/sistema/compilatore. Questa è

2 https://www.scirra.com/construct2
3 https://github.com/rogerwang/node-webkit

probabilmente la soluzione più efficace, in quanto fornisce accesso completo alla macchina sottostante e a tutte le sue caratteristiche, ma è anche la più complicata da mantenere.

In realtà non è necessario riscriversi l'intero codice per ogni piattaforma: generalmente solo una piccola parte del codice è *platform specific*, tutto il resto è universale. Nell'ambito della programmazione in C++ gran parte della portabilità può essere ottenuta mantenendosi il più possibile aderenti allo standard. È inutile barcamenarsi tra l'uso di `_stricmp` e `strcasesmp`, specie considerando che è possibile anche che un compilatore non supporti nessuno dei due metodi: molto meglio usare direttamente una combinazione di `toupper` e `strcmp`, che sono garantiti su *ogni* compilatore C++ valido.

L'uso esclusivo di funzionalità standard però può essere limitante. In primo luogo i compilatori o i sistemi operativi potrebbero offrire delle ottimizzazioni specifiche, il mancato uso delle quali renderebbe l'applicazione nativa non più efficiente di una corrispondente applicazione web. Inoltre si potrebbero necessitare funzionalità avanzate ma ricorrenti, riscriversi ogni volta il codice e mantenerlo sarebbe estremamente dispendioso e non funzionale. Per entrambi i problemi la soluzione è una: l'uso di librerie esterne. Le librerie esterne non soltanto implementano funzionalità non banali in maniera il più possibile generica, ma fanno anche da wrapper attorno ai frammenti di codice specifici per una data piattaforma, così da fornire all'utente un'interfaccia unica e indipendente pur dovendo lavorare a stretto contatto con l'architettura sottostante. Il caso più eclatante si ha con le librerie grafiche (Qt, GTK+, wxWidgets, ...), che consentono di creare elementi quali finestre e bottoni astraendo dal sistema grafico di basso livello.

Non reinventare la ruota!

Scrivere codice portabile con linguaggi non interpretati introduce un nuovo tipo di difficoltà: la fase di configurazione. Poiché non si conosce a priori l'hardware sottostante, né si sa quali librerie sono disponibili sul sistema e quali invece devono essere aggiunte o ignorate, è necessaria prima della compilazione una fase di *esplorazione* della macchina. La configurazione può limitarsi a generare l'elenco dei percorsi delle librerie esterne, ma potrebbe anche eseguire analisi complesse quali il tentativo di compilazione di piccoli frammenti di codice, generare header sulla base dei risultati ottenuti, definire macro per il preprocessore... GNU Compiler Collection (GCC) si appoggia alla suite di software nota come *autotools* per la generazione di *makefiles*, Microsoft Visual Studio (MSVS) ha i propri file di progetto,

XCode può usare sia progetti specifici che GNU makefiles. Alcune librerie open source forniscono sia makefiles che progetti MSVS e XCode, altre solamente makefiles (e qualche istruzione generalmente troppo datata su come compilare usando MinGW/CygWin), ma sempre più progetti includono anche file per l'utilizzo di CMake[4].

CMake è un software libero (distribuito dietro licenza BSD modificata) che serve per rendere portabile anche la fase di configurazione dei progetti C++. Per ogni cartella che si vuole aggiungere al progetto bisogna creare il file `CMakeLists.txt` in cui sono presenti i comandi necessari: sebbene la sintassi sia molto differente da quella di autotools, molti dei comandi di quest'ultimo hanno un corrispettivo CMake. Sono a disposizione dell'utente moduli built-in per l'esecuzione di comandi complessi o la ricerca delle librerie più conosciute, ma è semplicissimo aggiungere nuove macro o funzioni per supplire a tutto ciò che manca. Quando si processa un progetto di CMake viene chiesto quale generatore utilizzare per l'output. Ne sono supportati a dozzine: Unix Makefile, Ninja[5], MinGW, MSYS, MSVS dal 6 al 12, XCode, più progetti specifici per IDE quali Eclipse, KDevelop3, Code::Blocks...

2.4 PORTABILITÀ PER LIBRERIA

Questo tipo di cross-platforming si avvale di framework dedicati, che da un lato offrono strumenti specifici per certi task o tipologie di programmi, dall'altro sono in grado di cross-compilare il codice per diverse piattaforme di destinazione. La portabilità è limitata ai sistemi per i quali il framework è in grado di generare codice, e le tecniche con cui viene ottenuta possono essere un misto di quelle viste fin'ora, gestite automaticamente dalla libreria che si usa. Nella maggior parte dei casi però, quando si parla di motori grafici 3D, viene creato codice ad hoc, perché le computazioni grafiche sono già di per sé molto onerose, la perdita di efficienza dovuta all'aggiunta di un layer d'astrazione diverrebbe un prezzo troppo alto da pagare. D'altronde la tecnologia attuale è ancora ben lungi dal fotorealismo in tempo reale, qualunque risorsa è meglio spesa nell'aggiunta di poligoni o passate di rendering che non in traduzioni di codice.

4 http://www.cmake.org/
5 http://martine.github.io/ninja/

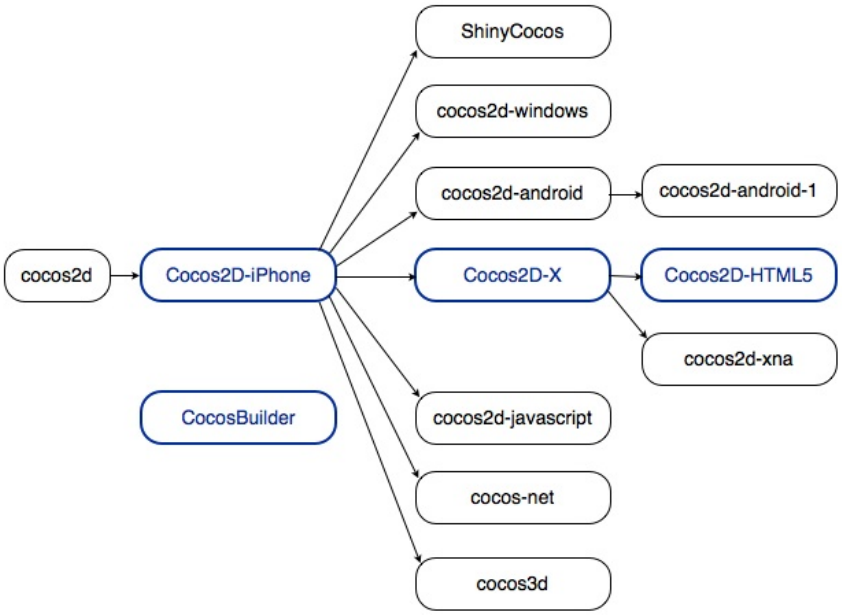


Figura 1: Mappa dei branch di Cocos2d. I riquadri in blu indicano versioni sincronizzate tra loro.

2.4.1 *Cocos2d*

Cocos2d è un framework per lo sviluppo di videogiochi principalmente 2D, open source e disponibile per una discreta varietà di piattaforme. Il progetto originale era quello di un framework per applicazioni desktop, scritto in PYTHON così da ottenere portabilità tramite l'uso di un linguaggio interpretato (2.2). Pochi mesi dopo il rilascio della prima versione la Apple aprì l'App Store ed i programmatori di Cocos2d decisero di rendere disponibile il framework anche per iOS. Purtroppo iOS non era dotato di un interprete PYTHON, perciò l'intero framework dovette essere riscritto in OBJECTIVE-C. In seguito, dal codice originale, nacque un branch per poter portare Cocos2d anche su Android. Altre fork furono originate per aggiungere supporto a vari sistemi e piattaforme, come mostrato in Figura 1. Nel caso di Cocos2d quindi più che una vera compatibilità multipiattaforma si ha una filosofia comune. L'uso di linguaggi diversi per le versioni specializzate infatti limita di molto la portabilità del codice: un'applicazione scritta in OBJECTIVE-C per Cocos2d-iPhone deve essere riscritta in JAVA per poter funzionare con Cocos2d-Android, ma probabilmente avrà bisogno di altre modifiche, perché cambiano le API, i framework di appoggio, le problematiche hardware [7]. Cocos2d-x è una versione

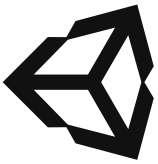
BRANCH	LINGUAGGIO	PIATTAFORMA
cocos2d	Python	Desktop
cocos2d-iphone	Objective-C, JavaScript	iOS
cocos2d-x	C++, Lua, JavaScript	Cross-Platform
cocos2d-html5	JavaScript	Browser HTML5
ShinyCocos	Ruby	iOS
cocos2d-windows	C++	Windows XP/7
cocos2d-android	Java	Android
cocos2d-xna	C#	Piattaforme Microsoft e MonoGame
cocos2d-javascript	JavaScript	Browser HTML5
cocos-net	C#	Piattaforme Mono
cocos3d	Objective-c	iOS

Tabella 1: Branch di Cocos2d e caratteristiche

più recente del framework che cerca di mettere insieme tutti i vari pezzi; è scritto in C++ con parti specifiche per le singole piattaforme e la maggior parte del codice in comune tra tutte.

La disponibilità delle varie versioni di Cocos2d è riassunta in Tabella 1.

2.4.2 *Unity*



Unity è un game engine nato su Mac OS ed in seguito esteso su molte altre piattaforme. Ad oggi supporta i principali sistemi operativi desktop (Windows, Linux, Mac), i più noti sistemi mobili (Android, iOS, Windows Phone 8, BlackBarry), alcune console (Sony PlayStation 3, MS Xbox 360, Nintendo Wii U) ed è disponibile anche come plugin per web browser. Assieme ad una IDE personalizzata, l'obiettivo di Unity è quello di astrarre dalle difficoltà delle specifiche piattaforme, consentendo agli

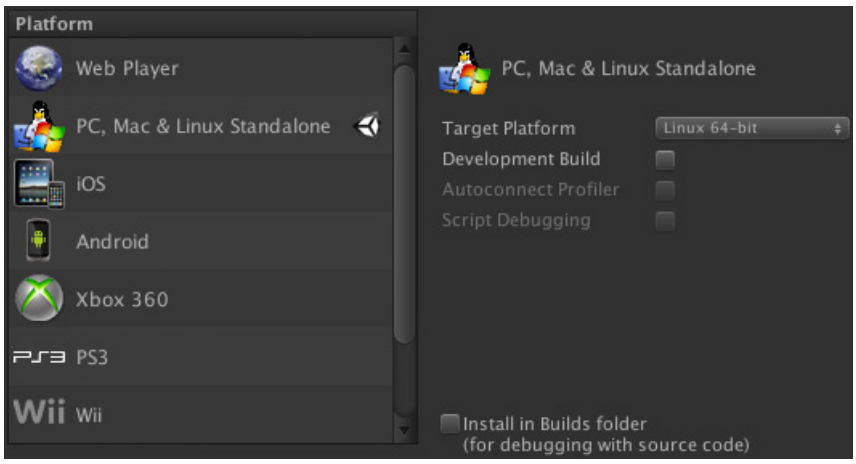


Figura 2: Configurazioni specifiche in Unity

sviluppatori di concentrarsi solamente sullo sviluppo di storie intriganti e grafica accattivante. Gran parte della logica del gioco può essere aggiunta agli oggetti tramite drag&drop attraverso l'editor fornito; è possibile scrivere codice C++ quando si ha bisogno di velocità d'esecuzione ed efficacia, scriptare in C#, JAVASCRIPT o BOO quando la semplicità di programmazione è il requisito fondamentale, ed anche comprare componenti già fatte come script logici, motori particellari, raccolte di mesh...

Unity è stato scritto in C++; parti di codice specifiche consentono di avere le massime prestazioni su ogni piattaforma supportata, mentre altre parti generiche forniscono un'esperienza uniforme su tutti i sistemi. In base alle capacità dell'ambiente in cui viene eseguito, il motore grafico può decidere se usare DIRECT3D (su piattaforme MS), OPENGL (ambienti desktop senza supporto per DIRECT3D), OPENGL ES (ambiente mobile), altre interfacce proprietarie (console). La stessa filosofia viene applicata agli script degli utenti di Unity: gli shader vengono scritti in CG ed automaticamente convertiti in GLSL se eseguiti in ambienti mobili, la fisica è gestita da NVIDIA PHYSX sugli elementi 3D o da BOX2D in ambienti bidimensionali in maniera trasparente all'utente, c'è un'interfaccia dedicata per il fine tuning relativo alle singole piattaforme (vedi Figura 2), così che anche gli utenti possano fare le loro ottimizzazioni specifiche senza bisogno di doversi scrivere file di configurazione appropriati.

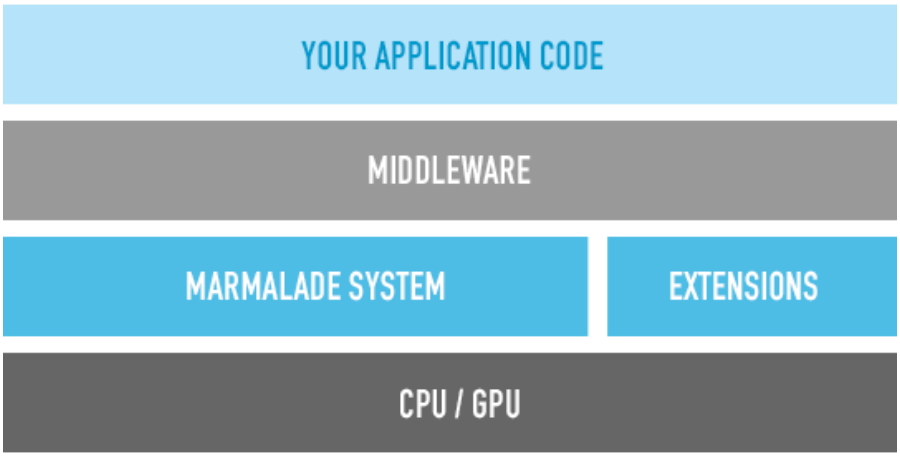


Figura 3: Marmalade c++ stack

2.4.3 *Marmalade*

Il software Marmalade[6] è un Software Development Kit (SDK) per MSVS e XCode che permette di scrivere codice c++ e compilarlo non solo per Windows e MacOS X, ma anche per smartphone. Per ottenere la portabilità fornisce una API multipiattaforma che incapsula le chiamate specifiche dei singoli sistemi (Figura 3); per funzionalità particolari è possibile usare estensioni e API platform-specific. Per la grafica viene messa a disposizione dei programmatori un'interfaccia di rendering, IwGL, che permette di inizializzare un contesto OpenGL ES con una singola chiamata, ma in alternativa si può decidere di utilizzare direttamente le API OpenGL ES. Le mesh Collada sono pienamente compatibili con Marmalade, così come i formati di Maya e 3D Studio Max. Le applicazioni possono essere testate all'interno di un simulatore unico per tutti i device di destinazione, impostando la risoluzione desiderata, inserendo breakpoint ed anche simulando eventi come la ricezione di chiamate per verificare che sospensione e ripristino avvengano senza intoppi.

Al momento della compilazione Marmalade genera un eseguibile per l'architettura bersaglio, al quale aggiunge poi un loader specifico per la piattaforma di destinazione, creando così un package installabile ed eseguibile (come mostrato in Figura 4). Il deploy su una piattaforma differente non richiede la ricompilazione dei sorgenti, è sufficiente unire un loader differente all'eseguibile già compilato.

Per i programmatori iOS è stato sviluppato Marmalade Juice (Figura 5), SDK che offre le stesse funzioni di Marmalade a chi

6 https://www.madewithmarmalade.com/

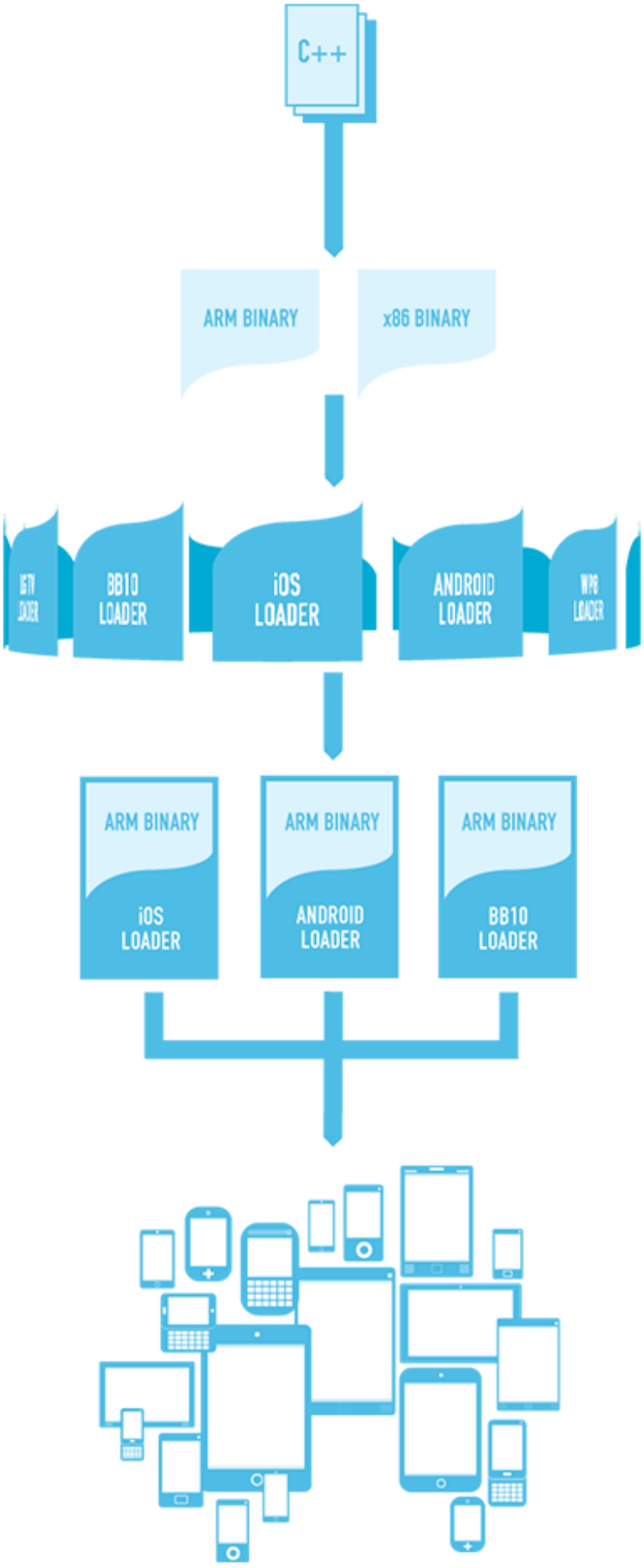


Figura 4: Marmalade workflow

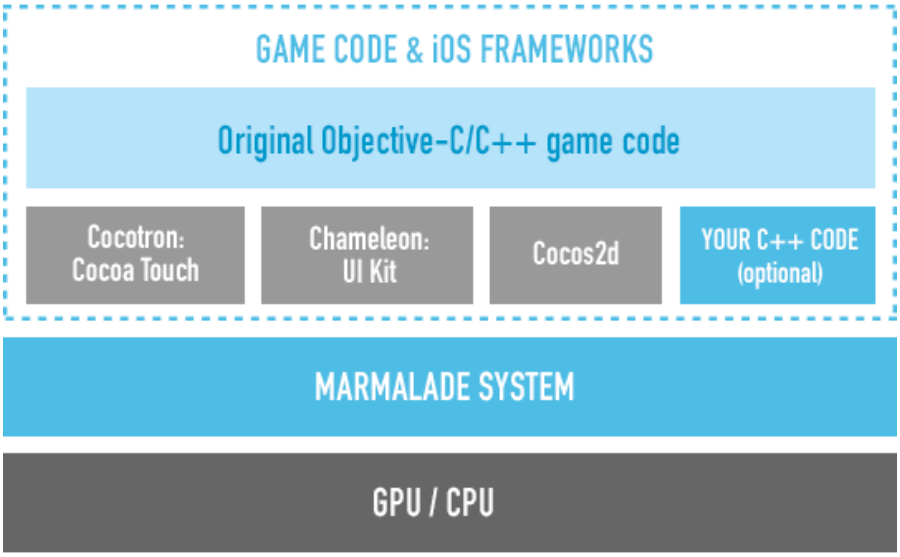


Figura 5: Marmalade Juice stack

parte da sorgenti per iPhone e vuole pubblicare la propria app su Android senza troppe tribolazioni.

2.5 WINE

Wine è stato ideato per poter utilizzare su sistemi Linux i programmi scritti per MS Windows senza necessità di apportarvi modifiche. Il progetto nasce nel 1993, in seguito ad esperienze precedenti quali Wabi e Public Windows Initiative, ma ha visto il rilascio della prima versione stabile soltanto 15 anni più tardi. Molte persone rimangono ancorate a Windows a causa di pochi programmi di cui non possono fare a meno (magari solo per abitudine, o forse per motivi di lavoro) e per i quali non esiste una versione nativa per Linux; d'altro canto i produttori di software non hanno alcun interesse a spendere tempo e denaro per la scrittura di versioni Linux dei propri programmi se la maggior parte degli utenti lavora in ambiente Windows. Il punto di forza di Wine è che permette di usare i programmi Windows direttamente su sistemi Linux (più precisamente su sistemi conformi allo standard POSIX: esistono versioni anche per FreeBSD e Mac OSX), favorendo così la transizione degli utenti da un sistema all'altro ed auspicabilmente rompendo il ciclo "scarsa diffusione tra gli utenti - scarso interesse per gli sviluppatori - scarsa diffusione tra gli utenti" descritto poc'anzi.

Wine funziona da *layer di compatibilità* tra Linux e Windows. Le API di MS Windows sono state completamente (nell'intenzio-

ne, ma ancora solo parzialmente nella pratica) reimplementate a partire dalle specifiche pubbliche e, quando queste fossero irreperibili o lacunose, tramite tecniche di ingegneria inversa di tipo *black-box testing* e *clean room design*, così da evitare eventuali violazioni di copyright. Oltre alla riscrittura delle librerie è stato progettato anche un processo demone, wineserver, che sostituisce le funzionalità fornite dal kernel di Windows (comunicazione tra processi, sincronizzazione dei thread, gestione del registro...). Il demone viene lanciato automaticamente all'avvio di Wine e chiuso quando non ci sono più processi Windows attivi, ed è possibile avere più demoni in esecuzione contemporaneamente se i programmi vengono lanciati da utenti diversi oppure con configurazioni differenti.

I driver di sistema presenti in Wine sono solamente dei proxy per i driver nativi del sistema ospitante, pertanto non è possibile per le applicazioni Windows usare hardware che non sia già supportato in Unix. Installare driver per Windows tramite Wine non è una valida alternativa, in parte perché Wine lavora in *user space*, ma soprattutto perché è assente il supporto di basso livello necessario affinché tali driver possano funzionare.

Oltre all'ambiente di esecuzione esiste una seconda modalità di utilizzo di Wine: se si ha a disposizione il codice sorgente di un programma per MS Windows è possibile compilarlo usando il toolkit WINELIB, che mette a disposizione gli header, le librerie necessarie per il linker e tool quali il compilatore di file di risorse. Usando WINELIB è possibile inserire nel codice anche chiamate alle API Unix. Questo secondo tipo di approccio rientra nella casistica della portabilità per libreria.

La portabilità multipiattaforma ottenuta tramite Wine in teoria non richiede alcuno sforzo: quando il progetto sarà terminato i programmi windows potranno essere avviati sotto un diverso sistema operativo e funzionare perfettamente al 100%. Attualmente però sono state implementate solo le parti più usate delle librerie Windows, quindi la maggior parte delle applicazioni darà problemi o richiederà configurazioni particolari per poter funzionare. Inoltre può essere usato soltanto per il porting da Windows e non verso tutte le piattaforme.

2.6 CONSIDERAZIONI

La scelta della tecnica giusta per il porting di XVR è condizionata soprattutto dal fatto che si tratta di un programma preesistente da centinaia di migliaia di righe di codice. Sarebbe

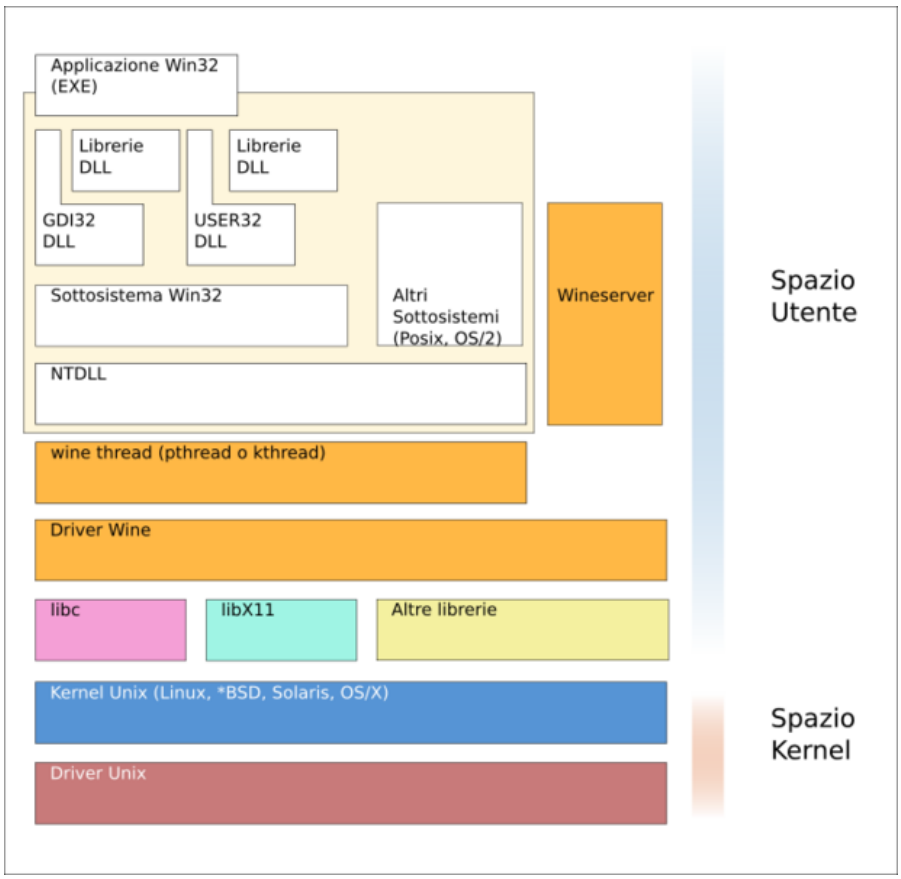


Figura 6: Architettura di Wine (fonte: Wikipedia.it)

meglio evitare tecniche che richiedono la riscrittura da zero, come ad esempio l'adozione di un linguaggio interpretato o di un framework particolare. La prima opzione in particolare andrebbe evitata comunque, perché l'impatto sulle performance renderebbe vano ogni intervento di ottimizzazione teso all'ottenimento di un tool *cutting edge*. L'esecuzione tramite Wine è un'opzione concreta, ma da un punto di vista accademico avrebbe un interesse pressoché nullo. L'unica alternativa valida è quindi quella di riscrivere il codice isolando e riducendo al minimo le parti platform-specific, ed usando Wine come termine di paragone per avere un limite inferiore di ciò che può essere ottenuto.

Ho optato dunque per la scrittura di codice ad hoc, sostituendo le funzioni specifiche di Microsoft Visual C++ (MSVC) con funzioni standard oppure, laddove necessario, introducendo librerie esterne multipiattaforma. Riguardo al criterio di scelta delle librerie uno dei principali fattori era la capacità di risolvere più problemi possibile, così da minimizzarne il numero; il grado di maturità è stato il fattore secondario, perché un progetto che si appoggia ad una vasta comunità offre supporto, esempi d'uso e minore probabilità di venir abbandonato nell'immediato futuro. Data la sua natura di meta-build system e la sua buona diffusione, CMake mi è parsa la scelta ideale per la fase di configurazione del codice sorgente di XVR.

Per la compilazione ho preferito lo standard C++03 al più recente C++11 perché il secondo non è ancora sufficientemente stabile, molti compilatori non hanno ancora implementato tutte le nuove caratteristiche. Il progetto è stato compilato con GCC senza estensioni attive, quindi dovrebbe funzionare su qualunque compilatore standard compliant.

3

ARCHITETTURA

XVR è un ambiente di sviluppo che include una IDE dedicata, un compilatore e un debugger. Lo scopo di XVR è semplificare la creazione e la gestione di realtà virtuali o aumentate. Originariamente era stato sviluppato per il web, permettendo di usare lo stato dell'arte degli algoritmi di rendering all'interno di un browser ben prima che si sentisse parlare di WebGL o anche solo di *Web 2.0*. Gli ambienti virtuali vengono descritti usando s3d, un linguaggio di scripting ideato appositamente, dalla sintassi fortemente ispirata a c++ e corredato di un'ampia libreria standard per le funzioni relative alla realtà virtuale. s3d è un linguaggio semplice: non viene fatto alcun controllo statico sui tipi, non bisogna curarsi della corretta deallocazione di memoria, fornisce funzioni grafiche di alto livello che verranno poi tradotte automaticamente in sequenze di chiamate OpenGL. È però anche molto potente, orientato agli oggetti, con possibilità di uso diretto di OpenGL. Tra le funzioni di base fornite ci sono diversi metodi di controllo della camera, audio 3d posizionale e tutto ciò che riguarda l'uso di mesh, texture e materiali; è possibile creare mesh procedurali (che sono il corrispettivo 3d della grafica vettoriale) oppure lavorare con modelli in formato `AAM`, esportabili tramite gli appositi plugin da 3DStudioMax, Maya o Blender. Il compilatore converte gli script s3d in bytecode binario, applica degli algoritmi di ottimizzazione per ottenere sempre il massimo dell'efficienza di esecuzione ed infine li comprime, perché ai tempi delle connessioni internet 56k anche pochi kilobyte risparmiati potevano essere importanti.

Viste le sue potenzialità, il progetto è stato in seguito esteso per poter supportare una vasta gamma di dispositivi, come i tracker ottici, gli *head mounted display*, gli apparecchi di *force feedback*, e persino i sistemi CAVE, grazie al modulo di network rendering che permette di distribuire e sincronizzare sui vari slave le chiamate OpenGL.

Dal punto di vista dell'utente XVR è suddiviso in *loop*, ognuno con un diverso significato. Il principale è il loop grafico, che ha una frequenza di default di 100 Hz che può essere modificata dall'utente fino a un massimo di 1 kHz, oppure impostata a 0 per non avere limiti; esclusivamente all'interno del loop gra-

fico è possibile chiamare comandi OpenGL. Il loop fisico può variare tra 0,1 Hz e 1kHz ed anche nel suo caso il valore predefinito è di 100 iterazioni al secondo; serve per compiere tutti i calcoli indipendenti dal rendering (calcolo di posizione e velocità degli oggetti, ricezione o invio di dati in rete, ...), ma non ha accesso al contesto grafico, quindi non può effettuare correlazione tra posizione del mouse e ambiente 3D. L'event loop gestisce tutto ciò che non ha una cadenza periodica: segnali di mouse, tastiera o altre periferiche, eventi del sistema operativo, eventi generati dall'utente.

A livello di programmazione tutte le queste funzionalità sono implementate all'interno della libreria VREngine_PI, ma ci sono anche altre librerie di supporto per tutto ciò che riguarda la VM e gli ambienti tridimensionali.

3.1 STRUTTURA ORIGINALE

XVR è suddiviso in moduli che assolvono ai vari compiti. Oltre al motore che controlla il flusso del programma c'è una macchina virtuale che interpreta gli script e una serie di librerie di supporto per ambienti virtuali indipendenti dal resto del codice. Solamente i file di codice del progetto originale sono quasi 9000, per un totale di oltre due milioni e mezzo di righe di codice. Secondo i calcoli effettuati con SourceMonitor[1] quasi la metà delle righe (il 48,7%) sono commenti e le 24.251 classi definite contengono in media 33 metodi a testa. Se si considera solamente la cartella relativa all'engine si scende a 2000 file con poco più di mezzo milione di righe di codice, ma anche la percentuale di commenti diminuisce a meno di 1/3. In questo caso il numero di definizioni di classi è 3727 e la media di metodi per classe 20, un numero ancora abbastanza elevato ma comunque molto più basso rispetto al caso precedente. Se non si considerano eventuali librerie esterne, quindi contando solamente i file su cui ho dovuto effettivamente lavorare, si ottiene un totale di 372 documenti contenenti 153.000 linee di codice, con una percentuale di commenti inferiore ad 1/4, in cui vengono definite 412 classi da circa 10 metodi ciascuna. In seguito alle mie modifiche l'engine senza librerie esterne è composto da 365 file sorgenti, con un totale di 157.000 linee. Ciò ovviamente non significa che ho scritto 4.000 righe di codice, perché molte ne ho scritte al posto di istruzioni già esistenti e altre ne ho eliminate

1 http://www.campwoodsw.com/sourcemonitor.html

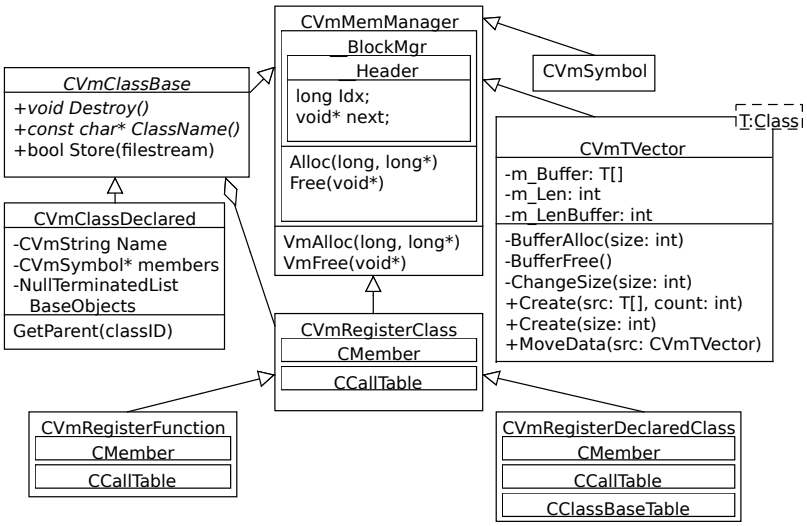
TIPO	FILES	RIGHE	COMMENTI	CLASSI	METODI/ CLASSE
Progetto originale	8970	2567612	48,7%	24251	32,94
Solo engine originale	1956	532285	32,3%	3727	19,34
Engine originale - no librerie	372	153375	24,1%	412	10,33
Engine nuovo - no librerie	365	157080	19,6%	360	9,96
Progetto finale	1621	460333	22,6%	1092	9,52

Tabella 2: Alcune metriche relative al progetto

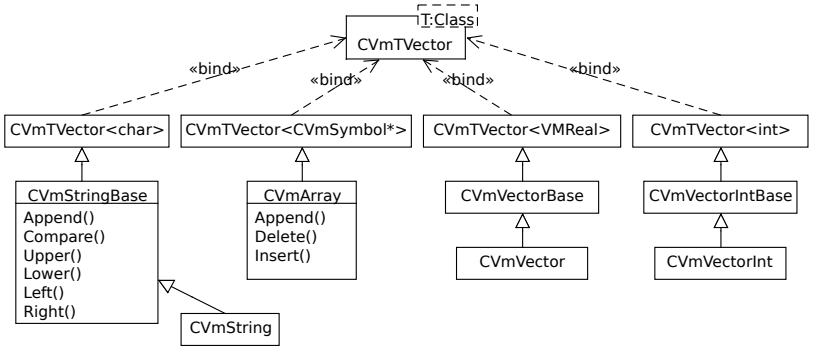
dal progetto originale. Le mie modifiche si mantengono sempre intorno ai 10 metodi per classe, che come valore non indica particolari necessità di riprogettazione degli oggetti, mentre la percentuale di commenti è scesa ad 1/5, non solo perché non sono stato abbastanza diligente nello spiegare cosa ho aggiunto, ma anche perché ho eliminato centinaia di vecchi frammenti di codice commentati da anni. Erano infatti presenti codici scritti a scopo di debug, oppure vecchie versioni rimpiazzate da nuovi algoritmi, che erano state trasformate in commenti e lasciate dov'erano; essendo tutto salvato sul server Subversion (SVN) ho preferito avere codice pulito piuttosto che modifiche facilmente rintracciabili. I dati precisi sono elencati nella Tabella 2.

3.1.1 *Parser*

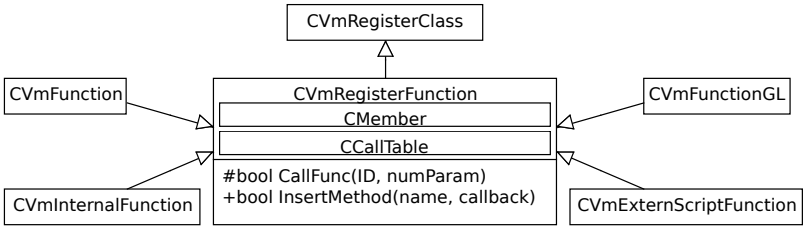
Il cuore pulsante di XVR è sicuramente ParserLib, la libreria in cui si trovano il generatore di bytecode, il debugger, ma soprattutto la VM. Le prime due componenti non fanno parte del player, quindi non sono state prese in esame, mentre l'interpre-



(a) Le classi che allocano memoria virtuale derivano tutte da `CVmMemManager`



(b) Implementazioni concrete di `CVmTVector`



(c) `CVmRegisterFunction` definisce funzioni globali

Figura 7: `CVmMemManager` e derivate

te è essenziale. Al centro della fase di esecuzione degli script c'è la gestione della memoria virtuale della macchina; la classe `CVmMemManager` illustrata in Figura 7a implementa i metodi `_VmAlloc` e `_VmFree`, gli unici utilizzabili all'interno della VM per allocare/deallocare memoria, pertanto chiunque abbia tale necessità deve estendere `CVmMemManager`.

Le fondamenta per la object orientedness del linguaggio S3D sono le classi `CVmClassBase` e `CVmRegisterClass` mostrate in Figura 8. La prima è una classe astratta da cui derivare gli oggetti S3D, la seconda invece serve a fare da ponte tra il linguaggio S3D ed il compilatore C++. `CVmRegisterClass` dichiara due classi interne: `CMember` rappresenta i metodi della classe, identificati dal nome S3D e dalla corrispondente callback C++; `CCallTable` serve per velocizzare le chiamate delle callback, viene usato dalla VM per bypassare la ricerca per nome dei metodi. In `CVmRegisterClass` infatti i `CMember` sono salvati in una tabella di hash ordinata in base al nome del metodo, senza la `CCallTable` ad ogni chiamata dovrebbe corrispondere una ricerca nella tabella seguita dall'invocazione della callback. `CVmClassBase` serve soltanto a garantire che ogni oggetto S3D contenga un riferimento alla relativa `CVmRegisterClass`, ad uso e consumo esclusivo della VM. Sia `CVmClassBase` che `CVmRegisterClass` hanno delle estensioni *Declared*. Tali estensioni servono come punto di partenza per gli oggetti e le classi dichiarati dagli utenti all'interno degli script; aggiungono qualche funzionalità per la gestione delle classi padre. La classe `CVmRegisterFunction` descritta in Figura 7c, anch'essa estensione di `CVmRegisterClass`, serve per la dichiarazione in S3D di funzioni con scoping globale; l'implementazione di `CallMethod`, `GetMember` e `SetMember` che fornisce si limita a lanciare errori a runtime.

Le ultime importanti specializzazioni di `CVmMemManager` sono l'oggetto `CVmSymbol` e la classe template `CVmTVector`. `CVmSymbol` è la rappresentazione S3D delle variabili; contiene una serie di *setters* che permettono di salvare valori tenendo traccia del tipo, così da rendere possibile il binding dinamico. `CVmTVector` è un array di dimensione non fissa. Le sue specializzazioni, elencate in Figura 7b, vanno dalla semplice lista di variabili (`CVmArray`, che consente sia l'inserimento in coda che l'accesso casuale), alle stringhe di testo (`CVmString`, con metodi per la modifica del caso, la concatenazione, la comparazione, la conversione di numeri in testo), fino ai vettori algebrici (`CVmVector` e `CVmVectorInt` hanno metodi per il prodotto scalare e vetto-

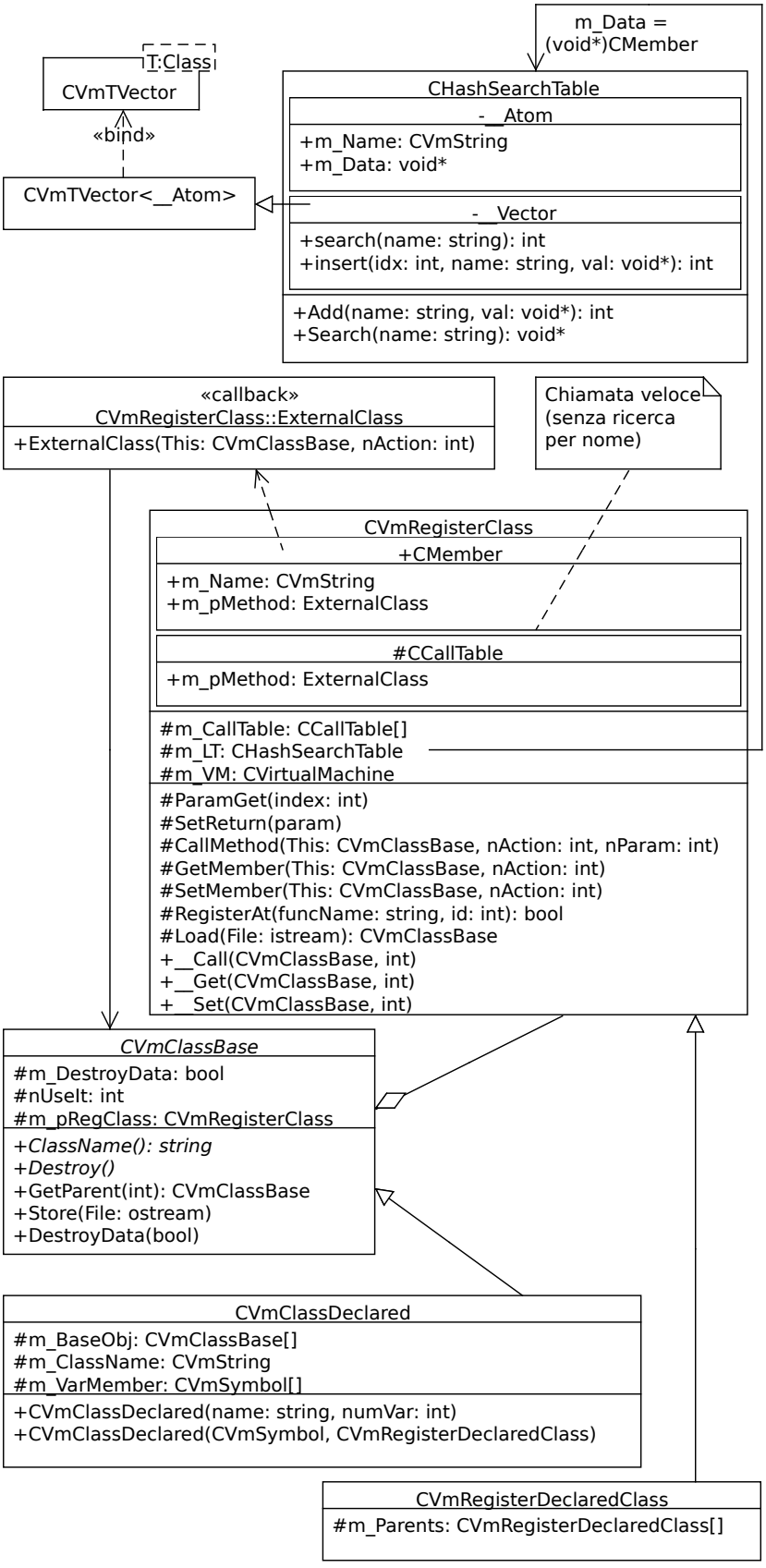


Figura 8: Infrastruttura per le classi S3D

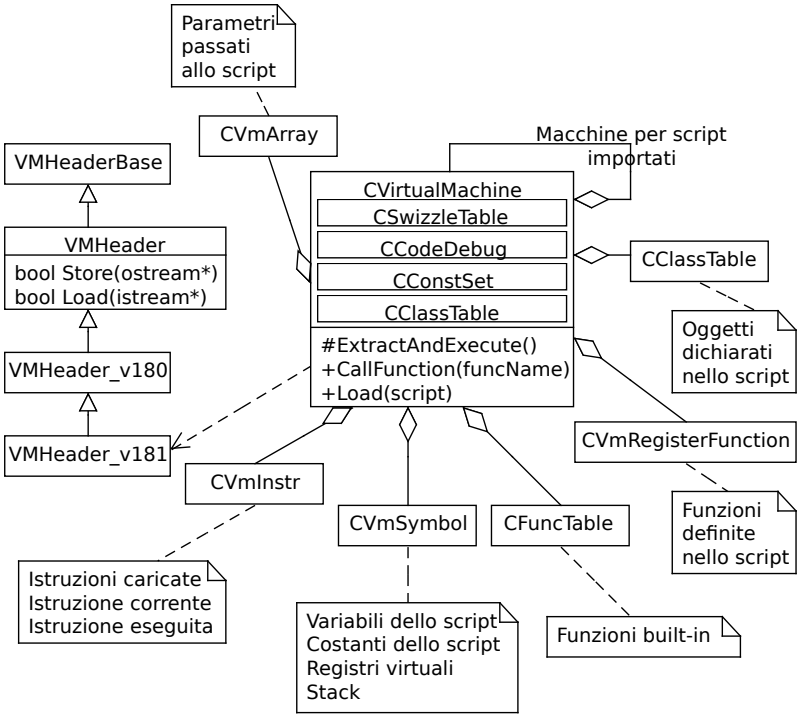


Figura 9: Virtual Machine

riale, il modulo, la scalatura, oltre alla somma, la negazione, la differenza).

Le componenti principali della VM sono riassunte in Figura 9. Il metodo `Load` serve per caricare il bytecode in memoria; per prima cosa viene letto l'header, attraverso il quale si può sia verificare che il bytecode sia compatibile con la macchina usata, sia controllare dove iniziano le varie parti del codice (istruzioni, elenco delle stringhe, elenco delle costanti, ...). Le istruzioni sono salvate in `m_BaseCode`, un array di `CVmInstr` (la classe contenente le informazioni relative ad una singola istruzione assembler virtuale: tipo di operazione, codice dell'operazione, dati su cui eseguire l'istruzione).

Un paio di puntatori, `m_Instr` e `m_IP`, servono a tener traccia all'interno dell'array rispettivamente dell'istruzione che la VM ha iniziato ad eseguire e dell'istruzione attualmente raggiunta in fase di esecuzione. Poiché le istruzioni in assembler virtuale possono avere un massimo di 3 parametri la VM ha 3 puntatori ad altrettanti `CVmSymbol` da utilizzare come registri: quando un'istruzione dev'essere eseguita si leggono i valori dei parametri presenti sullo stack (`VmVoid` se il parametro non è utilizzato), si salvano tali valori nei tre registri (è possibile

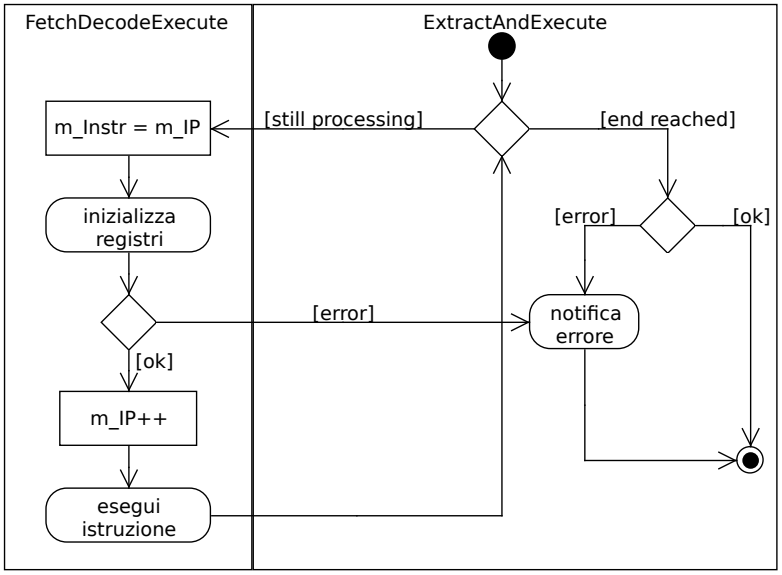


Figura 10: Il metodo ExtractAndExecute, in cui avviene l'effettiva interpretazione del bytecode

modificare anche altri registri d'appoggio, dipende dall'istruzione) ed infine si procede alla chiamata vera e propria (che è un metodo senza parametri né valore di ritorno). La Figura 10 descrive il flusso di esecuzione di una funzione S3D: la parte relativa alla singola istruzione assembler è data dal sottosistema `FetchDecodeExecute`.

Lo stack è un array di `CVmSymbol`, gestito grazie all'ausilio di 4 puntatori: `m_Stack` indica sempre l'inizio dell'array, `m_BP` la cima dello stack all'inizio della chiamata attuale, `m_SP` la cima dello stack in questo momento, `m_StackTop` l'ultima cella allocata dell'array; se ad un certo punto durante l'esecuzione di un comando c'è necessità di ingrandire lo stack oltre `m_StackTop` viene allocato un nuovo array di dimensione doppia rispetto a quella attuale, vi si spostano i `CVmSymbol` contenuti nel vecchio array ed infine si aggiornano i 4 puntatori.

3.1.2 *Engine*

Il collante che tiene insieme tutti gli elementi visti fin'ora è VRENGINE_PI, la libreria che istanzia la VM ogni volta che il programma viene eseguito. All'interno di VRENGINE_PI la prima cosa che colpisce è la presenza di una serie di classi di nome `Vm<Qualcosa>`: si tratta di classi proxy tra gli oggetti S3D `CVm<Qualcosa>` e le implementazioni C++ di VRLIB (3.1.4) con-

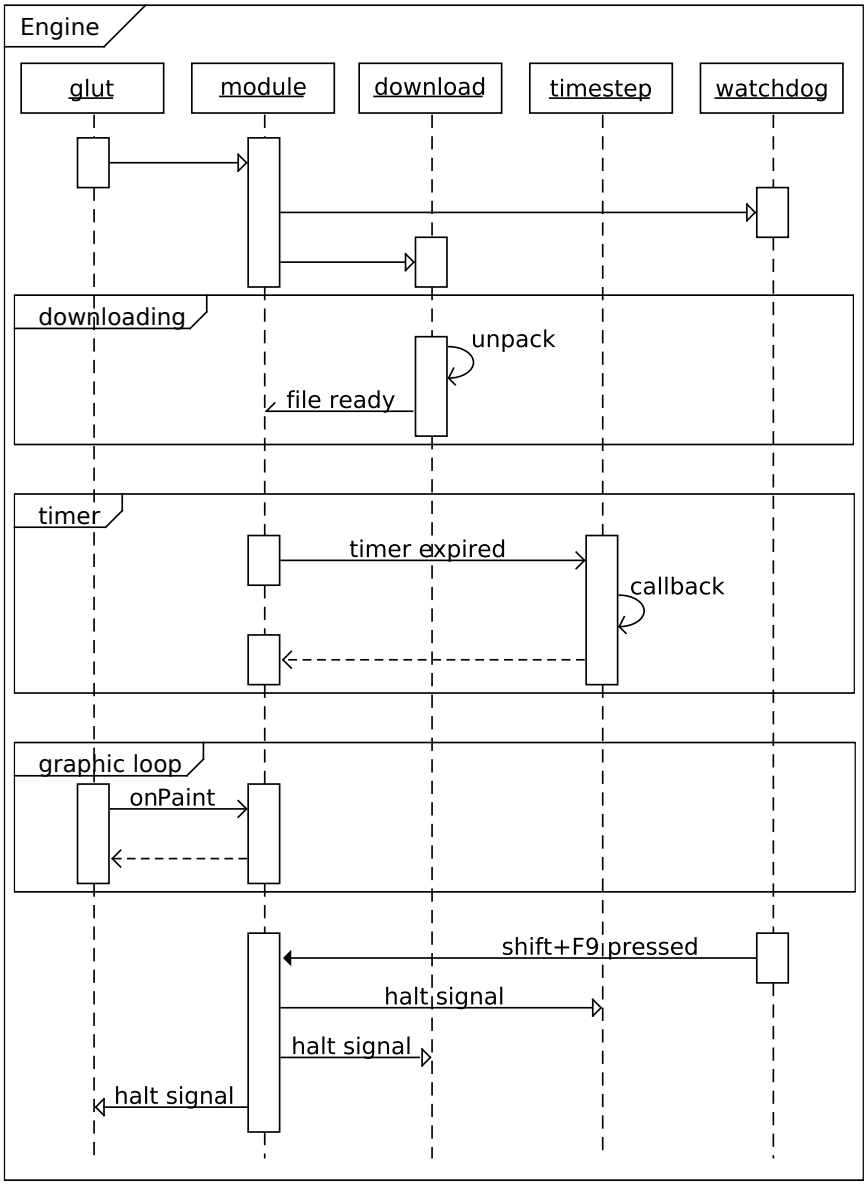


Figura 11: Diagramma di sequenza dell'engine

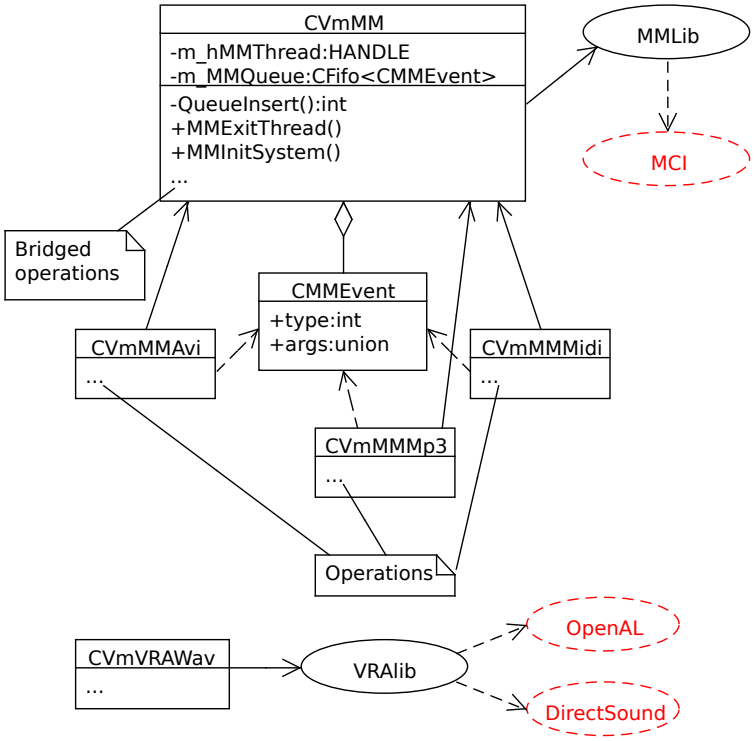


Figura 12: Oggetti multimediali

tenute in `VR<Qualcosa>`. Ma l'oggetto più importante di VREngine_PI è `VRModule`: è il controllore centrale del programma, quello che gestisce i thread e che permette alle varie componenti di comunicare tra loro.

Per poter usare XVR si crea una finestra per l'applicazione, sia essa ottenuta con OpenGL Utility Toolkit (GLUT) oppure tramite controllo ActiveX all'interno di una pagina web, dopodiché se ne passa l'handler al costruttore di un nuovo `VRModule`. Tale modulo crea una `CVRMachine`, una classe che eredita da `CVirtualMachine` (Figura 3.1.1) aggiungendo al padre la creazione degli elementi di default `s3d` (oggetti e una tabella di costanti). `CVirtualMachine` crea gli oggetti multimediali singleton `CVmMMAvi`, `CVmMMMp3`, `CVmMMMidi`, `CVmVRAWav` (Figura 12) ed avvia un nuovo thread. Quando vengono richieste funzionalità agli oggetti questi inseriscono messaggi in una coda, il thread li legge e soddisfa le richieste tramite la Multimedia Control Interface (MCI) di Windows (vedi Sezione 3.1.6).

Il thread *watchdog* serve solamente quando il programma è eseguito come plugin per web browser: se l'utente decide di terminare XVR non è lecito chiudere la finestra del browser ospitante, quindi il watchdog sta in ascolto per la pressione della

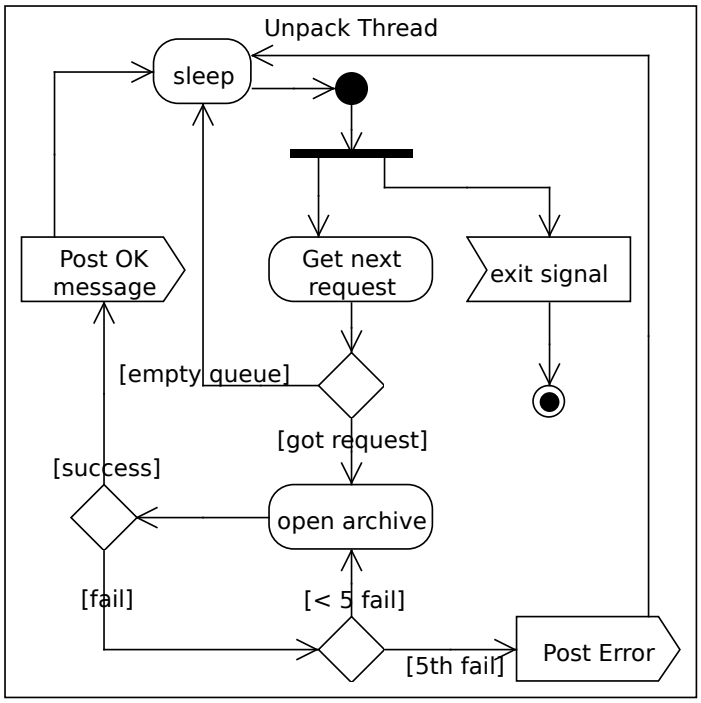
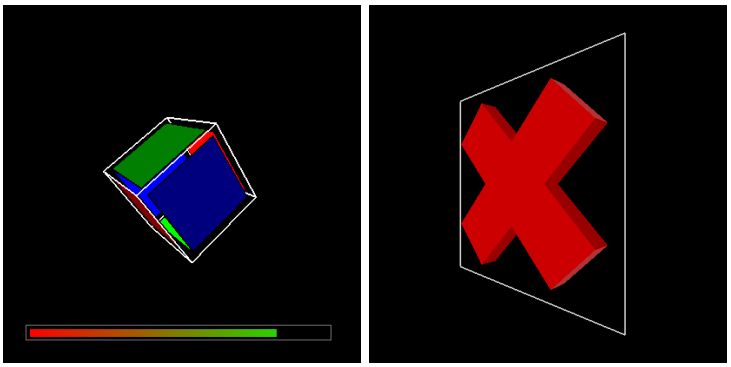


Figura 13: Download ed estrazione

combinazione di tasti `shift+F9` (combinazione ufficiale di arresto di XVR) e quando la riceve manda un messaggio di errore che arresta la VM e avvisa l'utente dell'avvenuta chiusura. L'esecuzione stand alone non ha bisogno del thread, perché tra l'evento keypressed generato dal sistema operativo e l'evento gestito dalla VM ho un layer intermedio, GLUT, che può catturare le combinazioni speciali di tasti, fermarne il bubbling, mandare il segnale di chiusura al modulo, chiamare `std::exit`.

La fase successiva della costruzione del modulo è l'avvio del *download thread* (Figura 13), che crea una directory di lavoro temporanea in cui verranno estratti gli archivi e salvati gli altri file. Se l'esecuzione avviene da una cartella locale è sufficiente copiare i file dalla directory dello script a quella di lavoro, mentre l'esecuzione remota richiede una fase di download vera e propria: il thread soddisfa le richieste ed avvisa il modulo (tramite messaggi di sistema) ogni volta che un file è disponibile, così che lo scaricamento possa avvenire in maniera asincrona rispetto all'esecuzione dello script. Lo script stesso è il primo file che viene scaricato, se necessario decompresso (di default il bytecode viene compresso con algoritmo ZIP) ed usato per l'in-



(a) Loading... (b) Red Cross Of Death (RCOD)

Figura 14: Schermate di XVR

vocazione del metodo `Load` della VM al momento dell'avvenuta ricezione.

Il passo seguente è la partenza del *timestep thread*, quello deputato all'invocazione della funzione `S3D OnTimer`. È basato sul timer multimediale di MS Windows e ad ogni timeout controlla lo stato del `VRModule`: se è *idle* allora viene richiamata la funzione, altrimenti aspetta il prossimo timeout (durante il normale ciclo vita di uno script gli stati possibili sono due, *idle* e *drawing*, e nel secondo caso la `OnFrame` ha la precedenza sulla `OnTimer`. Stati differenti indicano che l'esecuzione non è ancora iniziata oppure c'è stato un errore, perciò non avrebbe senso invocare la funzione).

Finita la creazione del modulo il metodo `OnWinGLPaint` viene impostato come callback grafica; in base allo stato (lo stesso verificato dal timestep thread) può essere disegnata una schermata di caricamento (tipo quella di Figura 14a), la RCOD rotante mostrata in Figura 14b, oppure invocata la `OnFrame`.

Il distruttore di `VRModule` manda i segnali di chiusura ai thread, aspetta che questi terminino il loro ciclo vita ed infine libera tutte le risorse.

3.1.3 *Procedural*

Una delle funzionalità offerte da XVR è la creazione di mesh in maniera procedurale, ovvero il corrispettivo tridimensionale di ciò che è la grafica vettoriale in contesti `2D`. Tale servizio è implementato nella libreria PoceduralLib. Alla base di tutto c'è la classe `Poligoni`, che fornisce alcune funzionalità di base ed il metodo `CopyToMesh`; tutte le altre figure estendono `Poligoni`.

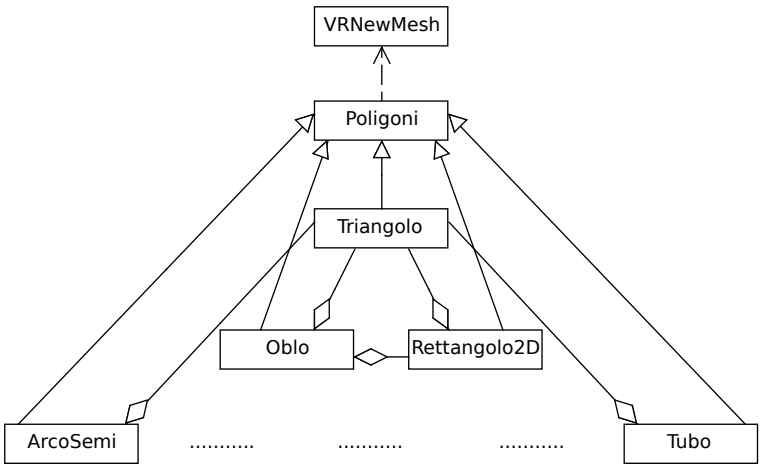


Figura 15: ProceduralLib

La principale delle sottoclassi è `Triangolo`, figura base di tutti gli altri poligoni (dal momento che i triangoli sono le primitive geometriche usate al giorno d'oggi da tutte le pipeline grafiche). `Triangolo` effettua il calcolo delle normali, consente l'aggiunta di vertici, facce e normali ad una figura, permette il salvataggio della figura su file in formato `XML`. Tutti gli altri poligoni non fanno altro che manipolare `Triangolo` per ottenere la forma voluta al livello di dettaglio desiderato.

3.1.4 *VR*

La libreria per la gestione di ambienti virtuali, VRLib, è molto simile in struttura a ProceduralLib (Sezione 3.1.3): ho una classe `VRBaseObj` da cui derivano tutte le altre classi della libreria.

In `VRBaseObj` sono presenti una serie di metodi e campi statici, che forniscono piccole utility (operazioni su vettori, ricerca di token in file testuali) o accesso a campi globali. Tra i vari campi salvati ci sono matrici di trasformazione globali (*model* e *projection*), file e directory di sistema, istanze (uniche per ogni esecuzione) di `VRScene`, `VRTextureManager`, `VRShaderManager`, `VRMeshLoader`.

Come si intuisce già dal nome `VRScene` è il controllore della scena attiva, ovvero gestisce la camera, le luci, e quando nello script `s3d` vengono chiamate le funzioni `SceneBegin` e `SceneEnd` è questo l'oggetto che si occupa del setup e del reset dei parametri del contesto OpenGL. `VRTextureManager` mantie-

ne una coda di `VRtexture` (una mera raccolta di parametri caratteristici di una singola texture); inoltre espone metodi per il caricamento di texture a partire da vari formati e l'importante `SetActiveTexture`, in cui `glBindTexture` viene chiamato.

Il `VRShaderManager` ha lo scopo di evitare il caricamento multiplo dello stesso shader da file; ogni volta che al manager viene chiesto di aprire un file GLSL per prima cosa verifica se è già presente all'interno di una sua collezione privata, dopodiché in caso di risposta negativa (o in caso si sia passata una stringa GLSL anziché il nome di un file) lo shader viene compilato in un `VRShaderProgram` (un'altra classe della libreria), memorizzato nella collezione e ritornato al chiamante.

La classe `VRMeshLoader` non ha campi, solo funzioni per il caricamento di mesh da file e per il calcolo delle normali. Tutte le funzioni fornite possono tranquillamente essere definite statiche, e magari potrebbe diventare una sottoclasse di `VRBaseObj`.

Oltre agli oggetti statici accessibili tramite `VRBaseObj` la libreria ha anche classi che corrispondono agli oggetti di default S3D. Gli elementi relativi agli ambienti 3D sono divisi in entità fisiche ed entità logiche. Le prime sono mesh, billboard, testi 3D, terreni... ovvero tutto ciò che viene disegnato. Le seconde servono a gestire meglio le trasformazioni, come gli oggetti (rappresentati dalla classe `VRObj`), che possono modificare le mesh in maniera non distruttiva (cambiando la matrice di trasformazione, mentre una scalatura o rotazione applicata ad una `VRMesh` agisce direttamente sui vertici), oppure i `VRCharacter`, che manipolano complesse gerarchie di oggetti.

3.1.5 *Avatar*

la libreria AVATARLIB (Figura 16) permette di gestire con facilità l'utilizzo di avatar umanoidi all'interno dell'applicazione 3D. Un avatar è sostanzialmente un character a cui è associata una skin; la skin è un semplice materiale, e può essere caricato da texture oppure definito tramite script, mentre il character deve necessariamente essere caricato a partire da un file `AAM` esplicitamente creato all'uopo. La classe `CVmAvatar` è aderente ad *H-Anim*, lo standard ISO/IEC FCD 19774 per la definizione di umanoidi animati ideata per favorire l'interoperabilità tra modelli 3D e hardware per il motion capture. È possibile dunque animare gli avatar riproducendo i movimenti dell'utente, usando animazioni precomputate (già presenti nel file `AAM` o caricabili esternamente da un file `BVH`), oppure in maniera

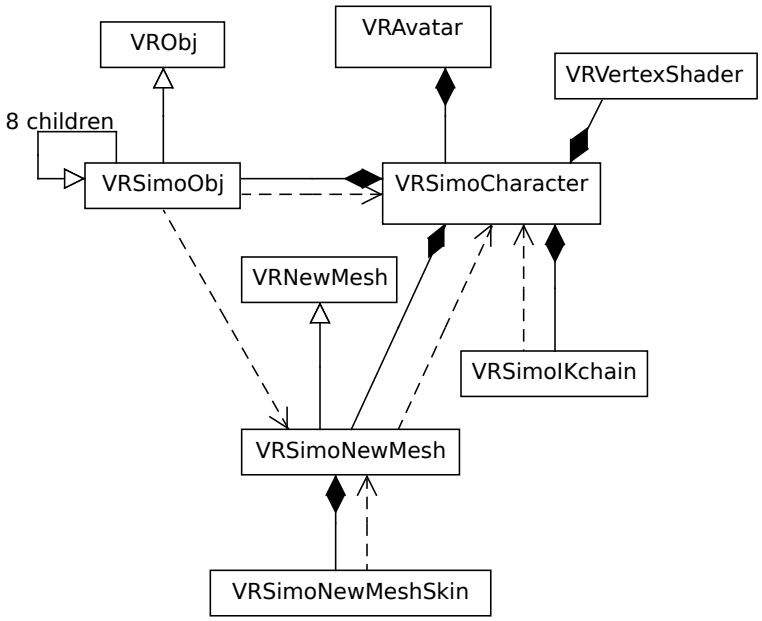


Figura 16: AvatarLib

programmatica con tecniche di cinematica inversa.

3.1.6 *Multimedia*

La MMLib serve come ponte tra gli oggetti multimediali di XVR e la MCI, la vecchia interfaccia multimediale di Windows deprecata a partire dall'avvento di Vista. Usata su versioni di Windows più recenti, anziché dare errori o avvisi, si ottengono comportamenti bizzarri (file audio muti, messaggi di codec mancante quando si riproducono file video). Essendo il comportamento così instabile ho preferito eliminarla del tutto piuttosto che mantenerla per casi specifici.

La Figura 17 mostra le interazioni che avvengono quando un utente esegue un'operazione su un oggetto multimediale: si inserisce un evento in una coda, dopodiché un thread svuota la coda e richiama la MMLib, la quale a sua volta tradurrà l'operazione in una sequenza di funzioni MCI. Sarà il thread ad elaborare il risultato, senza necessità di restituire il controllo all'utente.

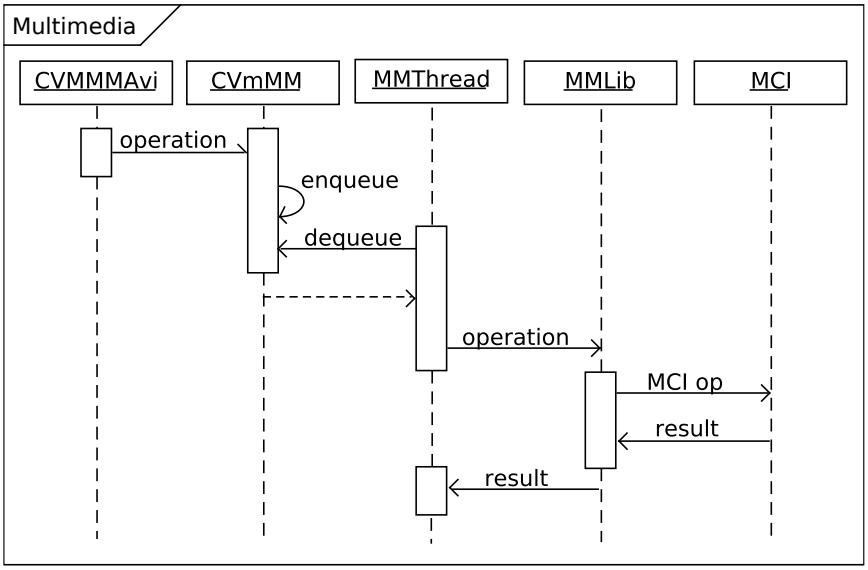


Figura 17: Interazione con la libreria multimediale

3.1.7 *Audio*

La libreria VRALIB serve come supporto per l'audio posizionale. L'interfaccia d'accesso alla libreria è contenuta nel file `VRAlib.h`, ma dei metodi esposti sono fornite due distinte implementazioni. Il comportamento predefinito della libreria è infatti quello di usare DIRECTSOUND per gestire l'audio 3D, ma per chi lo desiderasse viene data la possibilità di utilizzare OPENAL. Originariamente la scelta andava fatta in fase di compilazione, ma versioni più recenti della libreria hanno aggiunto la facoltà di cambiare approccio a runtime.

Uno degli utenti di spicco di VRALIB è l'oggetto `CVmVRAWav`, come si vede in Figura 12; a differenza degli altri oggetti multimediali, questo non ha richiesto modifiche implementative, perché anche dopo la rimozione di DIRECTX dal progetto l'interfaccia della libreria audio è rimasta invariata.

3.2 LIBRERIE UTILIZZATE

Non reinventare la ruota!

L'uso di librerie esterne è uno dei fondamenti cardine della buona programmazione. Permette di ridurre la complessità dei programmi, perché tutte le funzionalità fornite sono codice che non richiede di essere mantenuto da chi le usa; inoltre se sono librerie multipiattaforma consente di astrarre dai dettagli implementativi e concentrarsi solo sulle funzionalità di alto li-

vello che si desidera svilupparci sopra. Aumenta la sicurezza, in quanto a cercare eventuale vulnerabilità non c'è solo lo sviluppatore originale, ma buona parte dei programmatori che utilizzano attivamente la libreria. Semplifica la vita di chi dovrà estendere il programma dopo di te, perché generalmente associati alle librerie ci sono forum, mailing list e comunità a cui chiedere chiarimenti, nonché tutorial e documentazioni su cui basarsi, mentre tutto il codice scritto ex novo e non ben commentato è un inferno da maneggiare.

Quando si sceglie di introdurre una nuova libreria nel proprio progetto però non basta considerare le funzionalità fornite e la disponibilità di documentazione adeguata, ma bisogna considerare anche le questioni inerenti la licenza. XVR è un software proposto sia in versione freeware che commerciale, ma non open source, pertanto librerie distribuite esclusivamente dietro licenza GNU General Public License (GPL) non possono essere utilizzate, così come altre eventuali librerie coperte da licenze copyleft forti.

3.2.1 *Boost*

La prima libreria introdotta in XVR è Boost[2]. Boost nacque nel 1998 [4] come repository di librerie C++ open source, portabili, peer reviewed che implementassero funzionalità generiche di uso comune. Le linee guida rigorose, che obbligano gli sviluppatori di componenti di Boost ad adeguarsi alle buone abitudini di programmazione in C++, l'utilità delle funzionalità implementate e la licenza adottata, che garantisce la massima flessibilità anche in programmi chiusi o commerciali, hanno fatto sì che una decina delle librerie di Boost entrassero a far parte dello standard C++11. Usare Boost laddove possibile significa appoggiarsi ad una libreria disponibile su una vasta gamma di sistemi, attivamente mantenuta, ben documentata ma soprattutto parecchio diffusa, aspetto da non sottovalutare quando si lavora ad un programma elaborato "a più mani". Una delle più grosse difficoltà nel modificare XVR sta infatti nel capire a cosa servono le singole componenti e come interagiscono tra loro; l'uso di librerie note permette al programmatore entrante di capire all'incirca cosa viene fatto e di seguire più facilmente il flusso del programma. Per fare un piccolo esempio la presenza di un `boost::filepath` chiarisce da subito che non si

2 http://www.boost.org/

sta manipolando una stringa qualunque, bensì un nome di file o cartella; la complessità del programma viene altresì ridotta, perché si possono eliminare tutti i controlli sul separatore dei percorsi, si può accedere all'estensione del file senza cercare a mano il carattere '.' a partire dal fondo della stringa, concatenare nomi di file a percorsi predefiniti, verificare se un elemento del filesystem esiste e se si tratta di un file o una cartella.

Oltre alla componente FILESYSTEM di BOOST ce ne sono altre che sono risultate molto utili. La libreria THREAD non solo permette di astrarre dall'implementazione fornita dal sistema, ma costringe pure a uscire dai thread in maniera pulita. Non esiste infatti nell'interfaccia un metodo *kill* o *stop* che ne forzi la chiusura, ma solo la definizione di punti di interruzione predefiniti. Certo è possibile accedere all'handle nativo del thread ed arrestarlo, ma la documentazione afferma chiaramente che non è il modo giusto di operare e sarebbe meglio fermarsi prima a considerare di non aver commesso errori di progettazione. Per la sincronizzazione e la comunicazione tra thread viene usata la libreria INTERPROCESS, mentre CHRONO consente di lavorare sulle temporizzazioni, fornendo rappresentazioni del tempo che vanno dal minuto al nanosecondo e permettendo di definire orari e durate a piacimento (fermo restando che in fase di esecuzione la granularità del sistema, spesso nell'ordine dei millisecondi, pone un limite inferiore alla precisione raggiungibile).

3.2.2 *SDL*

Simple DirectMedia Layer (SDL) è nata come libreria videoludica multipiattaforma. Scritta in C e rilasciata inizialmente nel 1998 da Sam Lantinga, a partire dalla versione 2 viene distribuita dietro licenza ZLib [8], quindi può essere usata all'interno di XVR senza alcun problema di natura legale. La libreria supporta ufficialmente Windows, Linux, Mac OSX, iOS e Android, ma data la natura open source del progetto è possibile adattarla anche ad altri sistemi secondo le proprie necessità. Lo scopo principale di SDL è fornire accesso di basso livello alle periferiche (device di input, audio, hardware grafico), ma esistono anche delle librerie aggiuntive ufficialmente supportate per funzionalità extra di più alto livello (ed anche molte altre non ufficiali ma ugualmente valide).

VIDEO Il sottosistema video di SDL è in grado di creare e gestire molteplici finestre. Per la grafica tridimensionale si appog-

gia alle API OpenGL oppure, solo su macchine MS Windows, a quelle Direct3D. L'accelerazione 2D è ottenuta via hardware tramite le stesse API usate per il 3D, oppure via software come opzione di fallback per sistemi datati. In XVR viene usato solo in congiunzione con la libreria ufficiale SDL_ttf per il rendering del testo della console su texture, perché usarlo come gestore delle finestre avrebbe richiesto riprogettare tutta la parte di codice relativa a GLUT; un lavoro troppo oneroso se si considera che SDL è stato inserito nel progetto solo in fase molto avanzata per rimpiazzare le funzioni multimediali e di controllo delle periferiche considerate secondarie.

AUDIO Il sottosistema audio accetta stream a 8, 16 o 32 bit. I canali di uscita possono essere mono, stereo o Dolby 5.1, ed i formati non supportati dall'hardware vengono automaticamente convertiti in formati accettabili. L'uso diretto del sistema audio non è però consigliabile se non si hanno esigenze specifiche: è stato ideato per essere utilizzato in congiunzione con un mixer, e ne viene anche fornito uno ufficiale.

SDL_mixer [3] è anch'esso coperto da licenza ZLib, e permette di gestire con facilità sorgenti audio distinte. Una delle caratteristiche che saltano subito all'occhio di SDL_mixer è la distinzione che viene fatta tra musica e suoni, evidente retaggio della natura videoludica di SDL. Nella libreria infatti i suoni sono file di piccole dimensioni, possibilmente in formato `WAV`, da usare per il feedback in risposta alle azioni del giocatore o per gli avvisi; ne possono essere riprodotti anche molti contemporaneamente, specie durante fasi di gioco particolarmente frenetiche. La musica invece è quella di sottofondo, serve solamente a dare *colore* al gioco. I file `WAV` vengono caricati in maniera nativa, mentre gli altri formati richiedono l'attivazione esplicita al momento della compilazione. Per i formati aperti `OGG` e `FLAC` bisogna linkare le librerie Vorbis, gli `Mp3` si appoggiano a SMPEG, i `MOD` a MikMod. I file `MIDI` possono essere sequenzializzati usando Timidity++, ma sono richieste anche delle patch aggiuntive al sistema per poter funzionare.

EVENTI SDL fornisce un'infrastruttura per la gestione degli eventi, sia generati dal sistema operativo che creati dagli utenti. Gli eventi sono il metodo preferenziale per la gestione delle periferiche di input, ma richiedono che si lavori all'interno di una finestra SDL per poter essere generati. Fortunatamente SDL permette anche l'accesso alle periferiche tramite polling, perché

in origine la libreria era stata aggiunta ad XVR proprio come sostituto multipiattaforma di DIRECTINPUT, e questo secondo metodo consente di usarlo efficacemente senza eccessivi sforzi di integrazione col vecchio codice.

SDL_TTF La libreria ufficiale per il rendering di stringhe di testo è solamente un wrapper per FreeType[3]. È in grado di disegnare il testo in tre modi differenti: *solid*, la trasformazione più veloce ma anche meno gradevole, adatta per testi che cambiano frequentemente, *shaded*, esteticamente più piacevole anche se più lenta da calcolare, antialiased ma renderizzata insieme al rettangolo di background, *blended*, che prevede antialising, hinting e genera texture RGBA.

THREADS Un semplice meccanismo di creazione e sincronizzazione di thread viene fornito, ottimo per separare la grafica dalla logica in un videogioco, ma troppo semplicistico rispetto alle funzionalità fornite da BOOST.

TIMERS Tra le funzionalità di SDL è previsto anche un meccanismo di temporizzazione delle callback, che vengono richiamate da un apposito thread una volta scattato il timeout definito, dopodiché sono riaccodate se il loro valore di ritorno è non nullo. Dato che le callback sono invocate da un thread diverso da quello principale non è possibile usarle per interagire con OPENGL, ma si può comunque generare un evento personalizzato da elaborare nel main loop.

3.2.3 *FFmpeg*

FFMPEG è un progetto lanciato nel 2000 da Fabrice Bellard, proprietario del nome e del dominio www.ffmpeg.org, e tutt'oggi molto attivo nonostante la scissione all'interno della comunità di sviluppatori. È una libreria che supporta moltissimi formati di file multimediale, basti pensare che solamente per il video ci sono quasi 170 codec che possono essere decodificati! Comprende varie librerie, che sono state utilizzate in XVR per rimpiazzare le funzionalità video di MCI; `libavcodec` per la decodifica dei video (anche per la codifica; potrebbe essere interessante aggiungere un'opzione per registrare ciò che il player esegue, ma il mio lavoro si è concentrato solo sulla portabilità

3 http://www.freetype.org/

del già esistente), `libavformat` per la gestione dei vari contenitori multimediali, `libswscale` e `libswresample` per la modifica di immagini e stream audio. Le ultime due sono necessarie per convertire l'output della decodifica, dipendente dal video riprodotto, in un formato definito a priori.

FFmpeg è distribuito con licenza GNU Lesser General Public License (LGPL)2.1 [2] e ciò pone invero qualche difficoltà. La licenza *lesser* permette sì di fare distinzione tra "lavoro derivato" e "lavoro che utilizza il software", ma per essere sicuri di rientrare nella seconda categoria (e quindi poter distribuire XVR senza dover per forza adottare la GPL) è opportuno che la libreria sia linkata dinamicamente. La parte problematica sta nella divisione della comunità a cui accennavo poc'anzi. La storia di come una parte degli sviluppatori tentò senza successo di estromettere dal progetto il principale mantainer per poi generarne la fork LibAV è abbastanza interessante, ma questa non è la sede opportuna per discuterne[4]. Il punto è che le distribuzioni Linux derivate da Debian usano LibAV al posto di FFmpeg, perché uno dei programmatori "ribelli" è anche uno dei packager di Debian, ma LibAV è solo un sottoinsieme del progetto originale, quindi ci sono problemi di compatibilità (nella fattispecie `libswresample` è rimpiazzato da `avresample`, che ha un'interfaccia differente).

3.2.4 *LibFFI*

In origine il passaggio di parametri tra funzioni esterne e macchina virtuale avveniva tramite inline assembly: libFFI espone un'interfaccia C alle istruzioni assembler necessarie per tale operazione. Nonostante il nome non si tratta di una vera e propria Foreign Function Interface (FFI), ma solamente del suo layer di più basso livello. In XVR la FFI è la classe `CVmExternDll`, in cui i `CVmSymbol` vengono trasformati in tipi primitivi prima di essere passati come parametri al metodo esterno e il viceversa avviene per il valore di ritorno; l'uso di libFFI per la chiamata vera e propria consente di rendere la classe portabile su decine di architetture differenti (vedi Tabella 3). La licenza estremamente liberale di libFFI[5] ne consente l'integrazione in programmi commerciali senza problemi legali.

4 Per chi fosse interessato ad approfondire la questione consiglio la lettura di http://blog.pkh.me/p/13-the-ffmpeg-libav-situation.html

5 https://github.com/atgreen/libffi/blob/master/LICENSE

ARCHITETTURA	SISTEMA
AArch64	Linux
Alpha	Linux
	Tru64
ARM	Linux
	iOS
AVR32	Linux
Blackfin	uClinux
HPPA	HPUX
IA-64	Linux
M68K	FreeMiNT
	Linux
	RTEMS
Meta	Linux
MicroBlaze	Linux
MIPS	IRIX
	Linux
	RTEMS
MIPS64	Linux
PowerPC	AMIGA
	AIX
	FreeBSD
	Mac OSX
	Linux
S390	Linux
S390X	Linux
SPARC	Linux
	Solaris
SPARC64	FreeBSD
	Linux
	Solaris
TILE-Gx/TILEPro	Linux
X86	FreeBSD
	GNU HURD
	Interix
	kFreeBSD
	Linux
	Mac OSX
	OpenBSD
	OS/2
	Solaris
	Windows/Cygwin
	Windows/MingW
X86-64	FreeBSD
	Linux
	Linux/x32
	OpenBSD
	Solaris
	Windows/MingW
Xtensa	Linux

Tabella 3: Piattaforme supportate da LIBFFI

3.2.5 *Libarchive*

LIBARCHIVE è una libreria, licenziata FreeBSD [1], per la lettura e creazione di archivi con un buon supporto ai formati più comuni, riconoscimento automatico del tipo di file anche in caso di archivi compressi, possibilità di espansione ai formati più esotici tramite registrazione di callback di input/output personalizzate. È stata utilizzata per il caricamento degli script (che sono compressi in formato ZIP), e all'interno del download thread di XVR, in congiunzione con ZLIB[6]; in precedenza le stesse funzionalità erano fornite in `pkunzip.cpp`, un file di origine poco chiara che faceva troppe assunzioni sul sistema sottostante (endianness, dimensione degli interi) per poter essere riutilizzato efficacemente.

3.3 MODIFICHE APPORTATE

A livello di architettura quasi tutte le modifiche si sono concentrate su VRENGINE_PI, perché è lì che vengono usati messaggi di Windows e thread. Le altre librerie hanno avuto bisogno di qualche modifica a livello di implementazione, ma come struttura erano già indipendenti dal sistema sottostante; l'unica eccezione è il caricamento di file multimediali, che come vedremo nella Sezione 3.3.3 è stato completamente stravolto.

Per cominciare la versione multipiattaforma di XVR non prevede (ancora) l'avvio come plugin. Il vecchio codice era basato su un controllo ActiveX, in grado di funzionare esclusivamente su piattaforme MS Windows all'interno del browser Internet Explorer. Una versione portabile del player web dovrebbe essere implementata come plugin per browser multipiattaforma o meglio ancora sfruttando le potenzialità di HTML5, tentare di lavorare partendo dal codice già presente sarebbe più una perdita di tempo che un vantaggio[7]. All'interno dei miei sorgenti quindi sono dati per scontati l'avvio tramite GLUT e l'esecuzione stand alone.

Per la sincronizzazione dei thread il `VRModule` dichiarava una serie di handler su cui effettuare le chiamate di sistema `SetEvent` e `WaitForSingleObject`; l'analisi del codice ha però mostrato che tutti gli handler venivano usati da un singolo thread per la segnalazione e da un singolo thread per l'attesa. Non c'è stato

6 http://www.zlib.net/

7 Attualmente è in via di sviluppo presso il Laboratorio PERCRO un plugin cross-browser basato su FireBreath

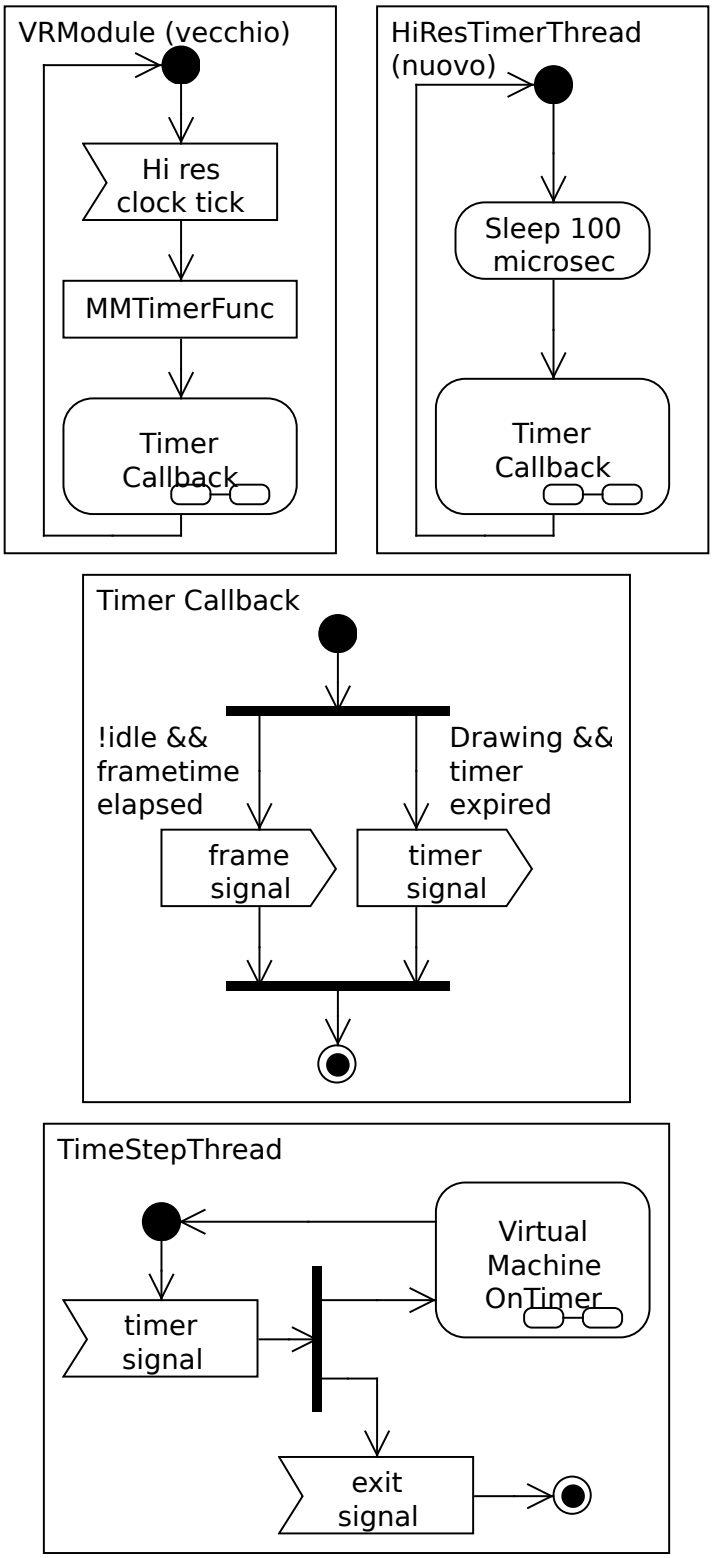


Figura 18: Architettura per la `OnTimer`

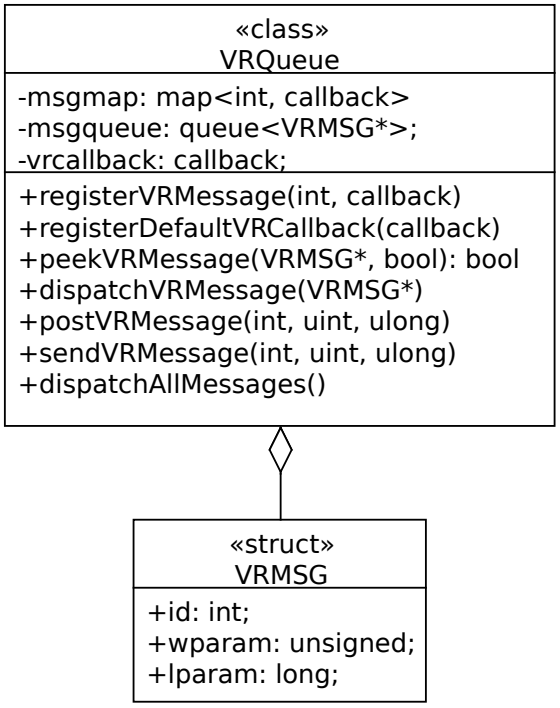


Figura 19: Message queue

quindi bisogno di sfruttare le potenzialità della componente INTERPROCESS di BOOST per costruire semafori condivisi, è stato sufficiente sostituire gli handler con dei `volatile bool`, in cui `SetEvent` diventa `bool = true` mentre `WaitForSingleObject` diviene `while(!bool){}`. La Figura 18 mostra come in tal modo si sia riusciti a mantenere quasi inalterata la vecchia architettura. Come si vede infatti l'unico cambiamento strutturale è stato nel conteggio del tempo: mentre in precedenza si faceva affidamento sull'orologio multimediale di Windows per richiamare la `TimerCallback`, adesso si è aggiunto un nuovo thread che richiama la funzione ogni 100 microsecondi.

100 microsecondi compatibilmente con la granularità del sistema, che difficilmente sarà così fine

3.3.1 *Message passing*

Se per la sincronizzazione tra thread è stato possibile cavarsela senza troppe tribolazioni altrettanto non si può dire riguardo alle comunicazioni. È vero che la maggior parte delle volte l'invio di messaggi era usato dal web player per la comunicazione tra il controllo ActiveX e la `DLL` di XVR, ma ci sono anche casi in cui le funzioni `::PostMessage` e `::PeekMessage` vengono usate pure dal programma stand alone, come ad esempio le segna-

lazioni *post ok message* e *post error* già incontrate in Figura 13. Poiché lo scambio di messaggi avviene sempre all'interno dello stesso programma ho potuto emulare il meccanismo di comunicazione con una semplice coda, mostrata in Figura 19. Nonostante i nomi ed i tipi della `struct VRMSG` siano ricopiati dall'omologa struttura `MSG` di Windows, il funzionamento del nuovo sistema di messaggistica è abbastanza differente. In Windows infatti si dice a quale finestra il messaggio è diretto, e per ogni finestra si può definire la procedura di default da richiamare se nessuna *window procedure* particolare viene selezionata. Non avendo a disposizione gli handler la mia coda si basa solo sugli ID dei messaggi: si definisce una chiamata di default tramite `RegisterDefaultVRCallback`, dopodiché è possibile registrare callback specifiche per singoli ID. Quando un messaggio deve essere processato viene verificata la mappa delle associazioni ID/Callback: se esiste una funzione particolare per l'ID del messaggio viene invocata quella, altrimenti si procede a chiamare la funzione di default.

È possibile effettuare il dispatch di tutti i messaggi presenti nella coda (cosa che peraltro accade nella callback video di GLUT, subito dopo aver eseguito `glutSwapBuffers`) oppure quello di un singolo messaggio. Nel caso in cui, durante l'esecuzione di `dispatchAllMessages`, una delle callback chiamate usi `postVRMessage` per accodare una nuova segnalazione, non ci sono rischi di ricorsione infinita: vengono serviti solamente i messaggi che erano già presenti in coda al momento della chiamata. Effettuare il dispatch di un singolo messaggio appena creato permette di processare il messaggio senza passare dalla coda, ma lo stesso comportamento lo si può ottenere in maniera più pulita utilizzando `sendVRMessage`.

La funzione `peekVRMessage` è stata un po' semplificata rispetto all'omologa `PeekMessage`, prendendo in input solamente il puntatore del messaggio da riempire e una variabile booleana che indica se rimuovere o no la struttura dalla coda.

3.3.2 *Embedded Resources*

Una delle funzioni molto comode offerte da MSVS è il compilatore di risorse, un programma che permette di incapsulare elementi quali cursori, piccole immagini, menu e fonts all'interno di un file eseguibile. XVR faceva uso di tale compilatore per i messaggi di errore della macchina virtuale e per il logo "*Powered by XVR*" mostrato dalle versioni gratuite del player.

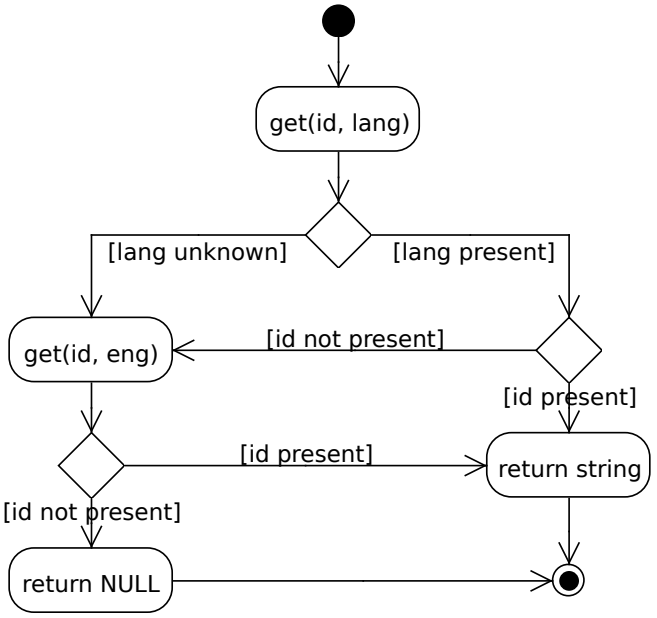


Figura 20: Supporto all'internazionalizzazione

La questione del logo è importante: aggiungerlo esternamente rispetto al file eseguibile renderebbe molto facile sostituirlo con un'immagine trasparente, eliminando di fatto il watermark. Uno dei primi tentativi provati per rendere il logo binario ed includerlo direttamente nell'eseguibile è stato quello di convertirlo con *imagemagik*, che tra i vari formati di output supporta anche gli header C. Il problema è che l'immagine trasformata in header viene anche convertita in `PNM`, un formato grafico (poco efficiente ma facile da manipolare) non supportato da PaintLib (il decoder di immagini usato da XVR). Nonostante ciò l'idea di ricavare un array di byte dal file originale non era male, quindi mi sono scritto un semplice programma che converte un file qualsiasi in `unsigned char[]` (si veda la Sezione 4.3 per maggiori approfondimenti), ed ho aggiunto il file `StaticResources.h` contenente il logo, le stringhe di testo ed anche un font usato come carattere di default dalla console.

Per le stringhe non mi sono limitato alla copia nel file, ma ho progettato un supporto completo all'internazionalizzazione (descritto in Figura 20). Quando si cerca una stringa per ID si può opzionalmente specificare un linguaggio: se esiste viene restituita la stringa tradotta, altrimenti la stringa in inglese (oppure `NULL` se l'ID è sconosciuto). Attualmente sono presenti

stringhe tradotte in italiano e francese[8], attivabili all'avvio di XVR con la nuova opzione `-lang`.

3.3.3 *Video*

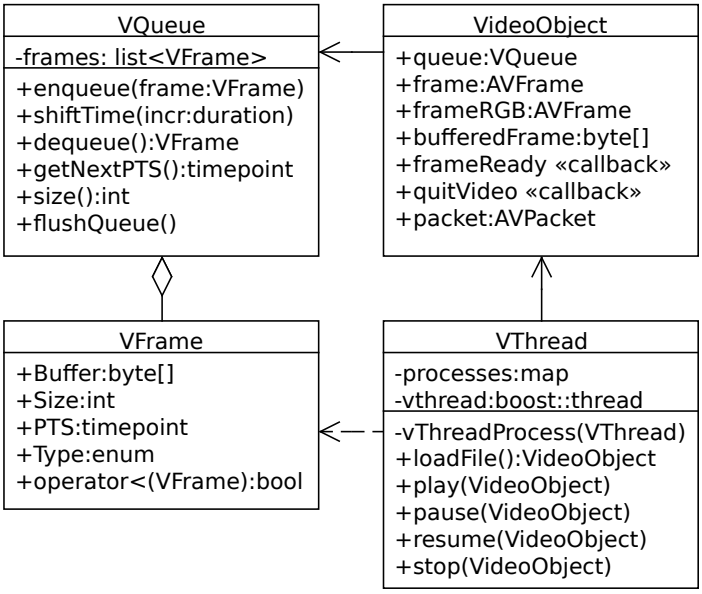
Per la riproduzione di video ho introdotto la libreria VRVLIB basata su FFMPEG (vedi Sezione 3.2.3) anziché MCI. Come si vede dalla Figura 21a ci sono solamente quattro oggetti nella libreria. `VFrame` è una struct contenente le informazioni per l'utente finale: tipo di frame (video, audio, sottotitoli, altro), buffer, PTS. Il buffer assume significati diversi a seconda del tipo di frame: per i fotogrammi video è un'immagine `RGBA` a 32 bit, le cui misure sono le stesse del video originale; per i pacchetti audio è un flusso `PCM` stereo a 16 bit, 22050 Hz; altri tipi di frame non sono al momento supportati. Il PTS è salvato come time point assoluto in cui il frame deve essere mostrato all'utente, così è sufficiente paragonarlo all'ora attuale di sistema per capire se è già scaduto e da quanto tempo.

`VQueue` è la coda di fotogrammi di una singola sessione di riproduzione multimediale: i `VFrame` definiscono gli operatori di paragone come confronto tra PTS appositamente, per poter tenere ordinata la coda così che il frame di testa sia sempre il primo da riprodurre. Non è vietato né sintatticamente scorretto usare la stessa coda per due o più file distinti, ma siccome non c'è modo di risalire al file di origine di un frame il risultato sarebbe quello di due flussi multimediali interlacciati tra loro in maniera impredicibile, quasi sicuramente non un comportamento auspicabile. La coda può essere interrogata per conoscere il numero di frame presenti, il tipo e il PTS del frame di testa. È anche presente un metodo, `shiftTime`, che consente di aggiornare il PTS alla fine di una pausa.

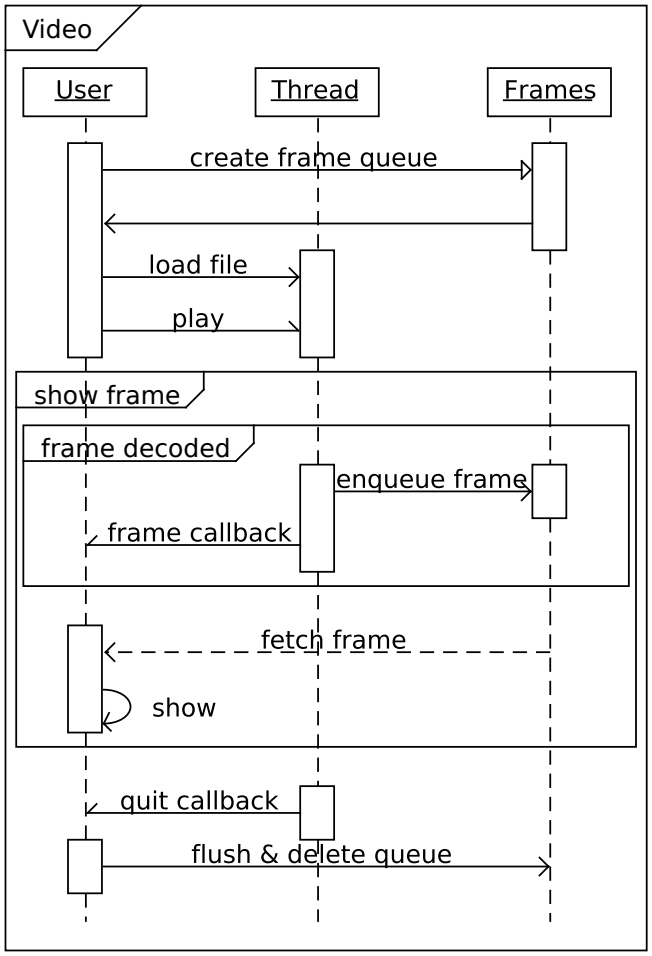
Il `VideoObject` è lo snodo di comunicazione tra tutte le componenti: contiene le informazioni di decodifica necessarie per poter lavorare con FFMPEG (contesti, codec, strutture di supporto, ...), un puntatore alla coda in cui devono essere inseriti i frame codificati, le callback da richiamare quando avvengono eventi video (frame da mostrare oppure riproduzione terminata). Il thread video discrimina i vari processi in corso in base al `VideoObject` ad essi associato.

L'oggetto `VThread` viene istanziato in XVR come oggetto globale; avvia il thread vero e proprio quando viene caricato il

8 Grazie ad Ottavia Sanseverino per l'ottimo lavoro di traduzione



(a) Diagramma delle classi della libreria video



(b) Interazione tra utente, thread video e coda di frame

Figura 21: Video Library

primo video e lo ferma quando non ci sono più operazioni in corso. Questo mi è parso un comportamento ragionevole, perché è difficile che avvengano riavvii ad una frequenza tanto elevata da rendere problematico l'overhead causato dall'operazione. Il thread, per ogni `VideoObject` in coda, decodifica il frame successivo (se la coda non è già piena), controlla se ci sono frame da mostrare e in caso affermativo richiama la callback `frameReady`. Se anche ci dovessero essere più pacchetti pronti la callback viene richiamata una sola volta, è responsabilità dell'utente finale decidere quali e quanti fotogrammi debbano essere scartati. Per garantire la massima flessibilità di utilizzo della libreria la callback prende in input un puntatore opaco definito dall'utente.

Una volta giunto alla fine dello stream, il thread può ricominciare dall'inizio (se il file era stato caricato con l'opzione `loop` attiva) oppure notificare l'evento richiamando `quitVideo`, che prende in input il medesimo puntatore opaco di `frameReady`.

4

IMPLEMENTAZIONE

La realizzazione dei cambiamenti descritti, così come i test effettuati, sono avvenuti tutti sulla stessa macchina, un laptop Acer Aspire con processore Intel® Core™2 Duo P7450 e scheda video Nvidia GeForce GT 240M. I sistemi operativi su cui ho lavorato sono stati principalmente due: inizialmente Windows 7 Professional per la programmazione con MSVC/MinGW, in seguito Arch Linux, che è una distribuzione *rolling release* tenuta costantemente aggiornata, per lo sviluppo Linux e i test con Wine. Entrambi i sistemi operativi sono a 64bit, ma su Windows ho usato versioni a 32bit del compilatore: nel caso di MSVC per necessità, perché il progetto originale era a 32bit, mentre per Minimal GNU for Windows (MinGW) si è trattato di opportunità, perché la versione a 64bit non è ancora sufficientemente testata. Il lavoro relativo a MSVC è stato fatto su MSVS 12, e per questo devo ringraziare Microsoft per aver lanciato il programma MSDNAA e l'Università degli Studi di Pisa per avervi aderito. La parte riguardante GCC (versione 4.8.1 nel caso di MinGW, 4.8.2 su Linux) è stata svolta con QtCreator, IDE del QtProject[1] leggera ma completa.

4.1 BUILDING...

Ciò che avevo a disposizione per iniziare a lavorare, una volta eseguito il primo checkout SVN, era (oltre ai sorgenti C++) una serie di file di progetto per MSVS 8. Il primo passo da compiere è stato di aggiornare tali file di progetto alla versione MSVS 12, dal momento che è questa la versione dell'IDE che mi è stata messa a disposizione. Questa operazione era semplice, perché la conversione è quasi automatica, gli unici interventi manuali sono stati la disabilitazione dell'opzione `SAFESEH`, la selezione del runtime multithread statico, la sistemazione di alcuni percorsi di librerie esterne.

Ottenuto un progetto che compila ed esegue correttamente ho potuto cominciare ad esaminare la struttura di XVR: capire come le varie librerie interagiscano tra loro, quali non servono per il player, quali file dipendano da librerie esterne ed in che

1 https://qt-project.org/

misura. Tutto questo perché, come descritto nella Sezione 2.3, il mio obiettivo è quello di giungere ad un metaprogetto CMake compilabile ovunque. L'approccio è stato quello dell'ingegneria inversa clean room design: creo il file `CMakeLists.txt`, genero un progetto per MSVS 12, lo apro e verifico che sia uguale all'originale. Alla fine del processo ero pronto per iniziare la compilazione con MinGW, così da trovare le prime criticità legate all'uso di operazioni non standard.

Durante questa fase ho sostituito i file `DLL` presenti nel progetto con i relativi codici sorgenti, laddove disponibili, perché naturalmente librerie compilate con MSVC non possono essere linkate ad eseguibili compilati con GCC. È qui che è stata aggiunta la compilazione di PaintLib, Zlib, VRPN, Tokamak, freeGLUT. Quando non c'erano sorgenti disponibili la libreria è stata momentaneamente rimossa, assieme a tutte le componenti di XVR da essa dipendenti.

"This is not the end, it is not even the beginning of the end, but it is perhaps the end of the beginning"

Arrivati alla compilazione ed esecuzione con MinGW, con un player un po' ridotto ma ancora abbastanza fedele all'originale, ho potuto procedere con le operazioni preliminari in vista della futura compilazione sotto Linux: eliminazione del maggior numero possibile di `#include <windows.h>` e relative funzioni. Le operazioni svolte sono state le seguenti:

- Sostituzione di `::CreateThread` con `boost::thread`
- Rimozione di tutti i vari `::PostMessage`, `::PeekMessage`, `::GetMessage`
- Ricerca di variabili di tipo `HANDLER`, `HWND`, `HINSTANCE` per capire a cosa servissero (la maggior parte sono state semplicemente rimosse, tanto erano usate solo per la messaggistica, e per lo più solo quando si compilava come plugin)
- Rimpiazzamento di tutte le operazioni su file e directory con le funzioni di `boost::filesystem`
- Correzione di tutte le volte scovate in cui una variabile di tipo `const char*` era stata dichiarata senza il modificatore `const`
- Eliminazione dei cast stile C in cui incappavo in favore dei corretti cast C++

Finiti gli adattamenti più evidenti ero pronto a testare il progetto in Linux; in questo periodo il codice è stato "ridotto ai minimi termini": tutto ciò che non poteva essere compilato e non era

considerato fondamentale veniva abolito senza remore. Durante tale compilazione l'unica cosa effettivamente riadattata sono state le chiamate OpenGL. Poiché il setup di XGL richiede qualche accortezza particolare (bisogna definire alcune macro per l'uso delle estensioni, ma soprattutto bisogna eliminare la definizione di macro che confliggono con nomi e typedef usati in XVR) ho creato un header unico, `XVROpenGL.h` (Listato 1), da usare sempre quando si necessita di includere `gl`, `glu` o `glext`.

Giunti alla compilazione senza errori si può pensare di avviarsi verso l'esecuzione del programma. Per prima cosa era necessario reintrodurre un sostituto della messaggistica, perché altrimenti non si sarebbero ricevute notifiche quando lo script fosse stato pronto per il caricamento. Per questo è stata creata l'infrastruttura descritta in Sezione 3.3.1. Dopodiché è stato necessario sistemare la decompressione degli archivi, perché gli script compressi non venivano riconosciuti. Il passo successivo è stato l'uso di interi a dimensione fissa; durante il caricamento da file avevo infatti notato che alcuni campi venivano saltati. Il problema era che venivano scritti su file, ed in seguito letti, valori di tipo `long`, che su Windows a 32 bit occupano 4 byte, mentre su Linux a 64 bit ne usano 8. Per ovviare a questa differenza ho cercato nella VM tutti i pezzi di codice che si aspettano variabili di tipo `long` e li ho modificati per utilizzare `int32_t`, gli interi a dimensione fissa definiti in `inttypes.h`.

Sistemato questo aspetto (l'endianness era già stata presa in considerazione nei vecchi sorgenti) XVR era ormai in grado di eseguire gli script S3D anche in ambiente Linux, rimanevano da risolvere gli eventuali bug a runtime e reintegrare il maggior numero possibile delle feature rimosse.

4.1.1 *Piccoli bug insidiosi*

Durante il lavoro sono stati identificati e corretti alcuni piccoli difetti del codice che erano rimasti latenti per anni senza causare problemi. Quattro di questi sono casi interessanti, quindi ho ritenuto opportuno illustrarli.

TEXTURE TYPE Passando da MSVC a MinGW ho riscontrato un comportamento assai strano nel disegnare il logo *Powered by XVR*, ovvero durante la maggior parte delle esecuzioni al posto del logo veniva usata una delle altre texture caricate dallo script. L'errore si verificava solo all'inizio dell'esecuzione, se XVR partiva col logo giusto allora continuava a disegnarlo esat-

```
#ifndef XVR_OGL_INCLUDED
#define XVR_OGL_INCLUDED

#ifndef FREEGLUT_STATIC
#   define FREEGLUT_STATIC
#endif
#define FREEGLUT_LIB_PRAGMAS 0

#ifndef WIN32
#   ifndef GLX_EXT_PROTOTYPES
#       define GLX_EXT_PROTOTYPES
#   endif
#   ifndef GL_EXT_PROTOTYPES
#       define GL_EXT_PROTOTYPES
#   endif
#   include <GL/glx.h>
#   include <GL/glxext.h>
#   undef Status
#   undef CurrentTime
#   undef Bool
#   undef True
#   undef False
#   undef Button1
#   undef Button2
#   undef Button3
#   undef Button4
#   undef Button5
#else // Win32
#   include <GL/gl.h>
#   include <GL/wglext.h>
#   include <GL/glext.h>
#endif

#include <GL/glu.h>
#include <GL/freeglut.h>

#endif
```

Listato 1: Inclusione delle funzioni OpenGL

to fino alla chiusura. Il problema stava in `VRTextureManager`: quando viene chiamata la funzione `SetActiveTexture` si procede con `glBindTexture` passando flags a seconda del campo `type` della `VRTexture`; solo che il campo non veniva inizializzato al caricamento di texture 2D, quindi se non venivano letti valori validi la bind non avveniva e il logo era disegnato usando l'ultima texture in memoria. Su MSVC non c'erano mai stati problemi perché il `type` di una texture 2D corrisponde all'intero 0, ed evidentemente il compilatore effettuava l'azzeramento della memoria allocata; GCC invece no, quindi talvolta si otteneva il comportamento corretto solo perché lo zero è un valore particolare, ma spesso il campo conteneva dati pattume, ed in quei casi avveniva l'errore.

MEMCPY Lo standard C++ [6] è chiaro: quando si usa `memcpy` i buffer sorgente e destinazione non devono essere sovrapposti, altrimenti il risultato è un *undefined behaviour*. In `CVmArray`, quando si rimuoveva un elemento, tutti quelli successivi venivano spostati in blocco con un `memcpy`. La cosa non dava problemi né con MSVC né con MinGW, ma GCC su Linux causava crash dell'applicazione a runtime. Riuscire a scoprire la causa della terminazione improvvisa non è stato banale, ma una volta scovata l'istruzione incriminata è stato sufficiente sostituirla con `memmove`, che è meno efficiente (perché usa un buffer intermedio), ma può essere usata sempre in maniera sicura.

DELETE VOID* In `CVmRegisterClass` è presente un oggetto di tipo `CHashSearchTable` usato per costruire la `CCallTable` e per le chiamate per nome dei metodi S3D. Internamente la classe `CHashSearchTable` ha un `CVmTVector<__Atom>` (descritto nella Sezione 3.1.1 ed illustrato in Figura 7b), dove `__Atom` conteneva un'associazione tra una `CVmString` (la chiave della tabella) ed un puntatore `void*` (il valore ricercato). I puntatori erano sempre ottenuti a partire da oggetti di tipo `CMember`, una sottoclasse di `CVmRegisterClass`. Compilando il progetto con tutte le warning attivate mi veniva segnalato che il distruttore di `CHashSearchTable` richiamava al suo interno quello di `__Atom`, che a sua volta chiamava `delete` sul puntatore `void*`. Anche in questo caso siamo di fronte ad un *undefined behaviour*, che se non ha mai causato problemi probabilmente è perché veniva richiamato esclusivamente alla chiusura del programma. Per trasformarlo in codice corretto è stato sufficiente rendere template la classe `CHashSearchTable` ed il tipo di dato contenuto in

`__Atom`.

CVMEXTERNDLL L'ultimo caso è solamente una svista in fase di scrittura del codice, che non causa problemi perché facente parte di un blocco di istruzioni assolutamente ininfluente. In `CVmExternDll` viene creato un array di `CDllTable*` della dimensione `NumFuncReg` (che corrisponde al numero di funzioni caricate). A quel punto veniva eseguito il comando `memset (m_FuncTable, NumFuncReg, sizeof( CDllTable* ))`, che non ha molto senso (riempio la dimensione di un puntatore con il contenuto del byte meno significativo del numero di funzioni presenti). Poiché l'array viene effettivamente popolato solo in una fase successiva (al di fuori del costruttore) è probabile che l'intenzione del programmatore fosse quella di inizializzare la memoria per evitare errori, perciò l'ho modificato in `memset( m_FuncTable, 0, sizeof(CDllTable*) * NumFuncReg)`. Comunque sia il programma fa un uso corretto della classe, inserendo un elemento in ogni posizione dell'array prima di accedervi, quindi il comando non ha mai causato errori di alcun genere.

4.2 RESTYLING

Come già accennato il primo player eseguito su Linux era corretto ma minimale: funzionavano solo le chiamate OpenGL e il logo watermark. Un pezzo alla volta sono state reimplementate molte delle componenti presenti, procedendo in ordine di importanza dell'elemento e di semplicità di riscrittura.

4.2.1 *Networking*

Per prima cosa sono state ricostruite le funzionalità di rete. La complessità era minima: sia Winsock che Pthread sono basati su Berkley Socket, le piccole differenze possono essere efficacemente sistemate in un header unico, mostrato nel Listato 2; l'unica accortezza è che Windows richiederà qualche pezzo di codice in più per il setup, cleanup e gestione degli errori di Windows Socket API (WSA).

In `VRNetManager.cpp` è possibile trovare sia frammenti di codice resi compatibili dall'inclusione dell'header sia le operazioni relative a WSA, come si vede nel Listato 3 e nel Listato 4.

```
#ifndef XVR_NETWORKING_INCLUDED
#define XVR_NETWORKING_INCLUDED

#ifndef WIN32
#   include <arpa/inet.h>
#   include <inttypes.h>
#   include <netdb.h>
#   include <netinet/in.h>
#   include <sys/socket.h>
#   include <sys/types.h>

#   define INVALID_SOCKET -1
#   define SOCKET_ERROR -1
#   define closesocket(something) close( (something) )
#   define ioctlsocket ioctl

typedef int SOCKET;
typedef int HANDLE;
typedef unsigned int socketAddrSize;
typedef hostent HOSTENT;
typedef enum  {
    BINDF_ASYNCHRONOUS              = 0x00000001,
    BINDF_ASYNCSTORAGE              = 0x00000002,
    BINDF_NOPROGRESSIVERENDERING    = 0x00000004,
    BINDF_OFFLINEOPERATION          = 0x00000008,
    BINDF_GETNEWESTVERSION          = 0x00000010,
    BINDF_NOWRITECACHE              = 0x00000020,
    BINDF_NEEDFILE                  = 0x00000040,
    BINDF_PULLDATA                  = 0x00000080,
    BINDF_IGNORESECURITYPROBLEM     = 0x00000100,
    BINDF_RESYNCHRONIZE             = 0x00000200,
    BINDF_HYPERLINK                 = 0x00000400,
    BINDF_NO_UI                     = 0x00000800,
    BINDF_SILENTOPERATION           = 0x00001000,
    BINDF_PRAGMA_NO_CACHE           = 0x00002000,
    BINDF_GETCLASSOBJECT            = 0x00004000,
    BINDF_RESERVED_1                = 0x00008000,
    BINDF_FREE_THREADED             = 0x00010000,
    BINDF_DIRECT_READ               = 0x00020000,
    BINDF_FORMS_SUBMIT              = 0x00040000,
    BINDF_GETFROMCACHE_IF_NET_FAIL  = 0x00080000,
    BINDF_FROMURLMON                = 0x00100000,
    BINDF_FWD_BACK                  = 0x00200000,
    BINDF_PREFERDEFAULTHANDLER      = 0x00400000,
    BINDF_ENFORCERESTRICTED         = 0x00800000
} BINDF;

#else // WIN32
#   include <winsock2.h>
#   include <tchar.h>
typedef int socketAddrSize;
#endif

#endif
```

Listato 2: Header per le funzioni di rete

```
VRNetManager::VRNetManager(){
#ifdef WIN32
   m_output_channel = INVALID_SOCKET;

   WORD   wVersionRequested;
   WSADATA wsadata;
   wVersionRequested = 0x0101;
   if(WSAStartup(wVersionRequested,
     &wsadata))
      VRNotifyMessage("Network
     startup error");
   else {
#endif
      m_output_channel = socket(
     AF_INET, SOCK_DGRAM, 0);
      //... codice comune
```

Listato 3: WSA Setup

```
VRNetManager::~VRNetManager(){
   if(m_output_channel !=
     INVALID_SOCKET)
      closesocket(m_output_channel);
   //...
#ifdef WIN32
   WSACleanup();
#endif
}
```

Listato 4: Close socket and WSA cleanup

4.2.2 *Audio*

Nella Sezione 3.1.6 si è parlato degli oggetti multimediali e di come si interfacciassero col sistema tramite MCI. `CVmMMMp3` e `CVmMMMidi` sono stati riscritti per usare SDL. Al momento della creazione degli oggetti viene inizializzato SDL_MIXER con `Mix_OpenAudio(MIX_DEFAULT_FREQUENCY, MIX_DEFAULT_FORMAT, 2, 4096)`, che specifica come formato di uscita uno stream a 22050 Hz, con campioni di 16 bit con segno, stereo, in pacchetti di 4096 byte. I due oggetti assumono però significati differenti rispetto a quelli che avevano in origine: se prima il nome della classe specificava il formato audio supportato, adesso `CVmMMMidi` serve per riprodurre i suoni, mentre `CVmMMMp3` viene usato per la musica (secondo i concetti descritti nel Sezione 3.2.2). Inoltre può sembrare ironico ma i formati `Midi` e `Mp3` non sono più utilizzabili: il primo non è integrato bene in SDL_MIXER, richiede l'applicazione di patch al sistema per poter funzionare, mentre il secondo è stato abbandonato per questioni legali. In compenso è possibile riprodurre (con entrambi gli oggetti indifferentemente) file di tipo `FLAC`, `OGG`, `WAV`, `VOC`. Ad ogni operazione utente corrisponde una chiamata diretta ad SDL, l'oggetto `CVmMM` e la libreria MMLIB sono stati rimossi dal progetto finale.

"Accidentalmente" lo stesso formato del buffer prodotto dal thread video

Non c'è stato invece bisogno di alcun lavoro diretto sulla classe `CVmVRAWav`, l'oggetto che gestisce l'audio posizionale; in compenso però la VRALIB a cui si appoggiava ha dovuto subire alcune modifiche. Il supporto ad Environmental Audio eXtensions (EAX) per esempio è stato rimosso, perché fornito tramite un file precompilato per MSVC, mentre OPENAL, anch'esso for-

nito come libreria già compilata, è stato sostituito con OpenAL Soft, da ricompilare. La sostituzione è stata una buona occasione per passare ad una versione più aggiornata della libreria, che però non fornisce più il metodo per il caricamento dei file `WAV`: fortunatamente un rapido studio del formato è stato sufficiente per poter sostituire la funzione deprecata, come si evince dal Listato 5.

4.2.3 *Device di input*

L'input da mouse o joystick può avvenire in due modi: tramite le funzioni esposte dalla VM oppure usando gli oggetti deprecati `CVmMouse` e `CVmJoystick`. Il primo approccio usa una struttura dati interna per gestire gli stati delle periferiche, che viene aggiornata catturando gli eventi del sistema operativo; GLUT è capace di notificare eventi simili a quelli catturati, perciò è stato sufficiente generare pseudo-eventi di Windows all'interno di callback GLUT per avere la portabilità delle periferiche. Nel secondo caso, mentre `CVmMouse` è solo un wrapper per le funzioni della VM, il joystick deprecato offre ancora qualche funzionalità extra basata direttamente su DirectInput. Poiché avevo già introdotto SDL (che tra le varie cose è anche un sostituto di DirectInput) ho preferito mantenere l'oggetto piuttosto che eliminarlo, tanto si è trattato di fare poco più che una mera traduzione 1:1 tra vecchie e nuove chiamate; nel confronto tra il Listato 8 e il Listato 9 è evidente tale operazione (al di là della presenza della macro, che serve per evitare il copia-incolla dello stesso metodo per tutti e 16 i bottoni), mentre dai Listati 6 e 7 si può capire come la struttura interna sia stata nel complesso semplificata.

```
typedef struct {
    char RIFF[4]; // string "RIFF"
    int32_t filesize; // file size -8
    char WAVE[4]; // string "WAVE"
    char fmt_[4]; // string "fmt "
    int32_t chunklenght; // should be 16
    int16_t format; // 1 = PCM
    int16_t channels; // number of channes
    int32_t frequency; // sample rate
    int32_t bytePerSec;
    int16_t blockAlign;
    int16_t bitsPerSample;
    char dataTxt[4]; // string "data"
    int32_t dataSize; // size of actual data
} WavHeader;

void OAslLoadWavFile(const char* fileName, ALenum* format, void** data, ALsizei*
     size, ALsizei* frequency) {
    WavHeader head;
    FILE* wavfile = fopen(fileName, "rb");
    if(fread(&head, 1, sizeof(WavHeader), wavfile) != 44)
        return;
    if(strncmp(head.RIFF, "RIFF", 4) != 0)
        return;
    if(strncmp(head.WAVE, "WAVE", 4) != 0)
        return;
    if(strncmp(head.fmt_, "fmt ", 4) != 0)
        return;
    if(strncmp(head.dataTxt, "data", 4) != 0)
        return;
    *size = head.dataSize;
    *data = malloc(*size);
    if(fread(*data, 1, *size, wavfile) != *size){
        free(*data);
        *data = NULL;
        return;
    }
    *frequency = head.frequency;
    if(head.channels == 1)
        *format = head.bitsPerSample == 8?
            AL_FORMAT_MONO8 : AL_FORMAT_MONO16;
    else
        *format = head.bitsPerSample == 8?
            AL_FORMAT_STEREO8 : AL_FORMAT_STEREO16;
}
```

Listato 5: Caricamento di file `Wav` in OpenAL

```
CVmJoystick::CVmJoystick(long JoyNo){
  JOYINFO joyinfo;
  UINT wNumDevs;
  m_JoyID = -1;
  if(JoyNo < 0){
    if((wNumDevs = joyGetNumDevs()) !=
     0){
      for(int i = 0; i < wNumDevs && i
     < 16; i++)
         if(joyGetPos(i, &joyinfo) ==
     JOYERR_NOERROR){
           m_JoyID = i;
           break;
      }
    }
  }
  else{
    if(joyGetPos(JoyNo, &joyinfo) ==
     JOYERR_NOERROR)
      m_JoyID = JoyNo;
  }

  if(m_JoyID >=0){
    joyGetDevCaps(m_JoyID, &m_JoyCaps,
     sizeof(m_JoyCaps));
    Update();
    s_Counter[m_JoyID]++;
  }
}
```

Listato 6: `CVmJoystick`: vecchio costruttore

```
CVmJoystick::CVmJoystick(int JoyNo){
  vrjoy = NULL;
  if(sdlparams.sdlJoystickON){
    int wNumDevs;
    if(JoyNo < 0){
      if((wNumDevs = SDL_NumJoysticks()
     ) != 0){
        for(int i = 0; i < wNumDevs &&
     i < 16; i++){
          if(vrjoy = SDL_JoystickOpen(i
     ))
            break;
        }
      }
    }
    else
      vrjoy = SDL_JoystickOpen(JoyNo);
  }
}
```

Listato 7: `CVmJoystick`: nuovo costruttore

```
VM_ATTRIBUTE(CVmJoystick, Button1)
   CVRMachine *VM  = GetVRMachine();
   switch(nAction){
   case VM_CALL_METHOD:
   case VM_GET_MEMBER:
      SetReturn((bool)((This->m_JoyInfo
     [This->m_JoyID].dwButtons &
     JOY_BUTTON1) > 0));
      break;
   default:
      Error();
   }
}
```

Listato 8: `CVmJoystick`: vecchia query

```
#define XVR_JOYSTICK_BUTTON(n)\
VM_ATTRIBUTE(CVmJoystick, Button ## n)\
   SDL_JoystickUpdate();\
   switch(nAction){\
   case VM_CALL_METHOD:\
   case VM_GET_MEMBER:\
      SetReturn(static_cast<bool>(
     SDL_JoystickGetButton(This->vrjoy,
      n)));\
   break;\
   default:\
      Error();\
   }\
}
```

Listato 9: `CVmJoystick`: nuova query

La tastiera non è stata così diretta da portare, e il comportamento finale è leggermente diverso da quello originale. Il problema è in GLUT, che traduce gli eventi del sistema operativo in caratteri `ASCII` prima di trasmetterli all'utente, perciò non c'è alcun modo di distinguere il tastierino numerico dal resto dei tasti. Al di là di questa piccola perdita c'era un altro problema:

GLUT non prevede il polling attivo della tastiera (cosa che invece XVR consente), ma solamente la ricezione passiva di eventi. Ciò che ho fatto in questo caso è stato aggiungere un paio di variabili in cui salvare lo stato ad ogni modifica, così da poterlo ricostruire in caso di interrogazione da parte del `VRModule`. Nel Listato 10 si possono vedere gli elementi usati: una mappa di associazione tra codici GLUT e codici Windows, una variabile per salvare lo stato dei tasti speciali (`glutGetModifiers` non può essere chiamato al di fuori di una callback), una lista di tasti premuti.

4.2.4 *Console*

L'ultima componente reintegrata grazie all'ausilio di SDL è stata la classe `CVmConsole`, che al pari del joystick ha subito una profonda semplificazione. Il file originale apriva un font TrueType tramite chiamate di Windows, lo convertiva in bitmap font ed infine generava una display list per il rendering; lo stesso font aperto con differente dimensione naturalmente causava la generazione di una nuova display list. In seguito alle mie modifiche i problemi relativi al rendering vengono demandati ad SDL_TTF, che restituisce, a partire da un `TTF_Font*` e una stringa di testo, una texture 2D pronta per essere disegnata. La nota dolente però è nel caricamento dei glifi: non sono riuscito a trovare una soluzione cross-platform per la ricerca e l'apertura dei caratteri nelle directory di sistema, quindi adesso i file `TTF` devono essere allegati assieme alle risorse dello script; in caso di font assente viene usata l'opzione di fallback, il carattere DejaVu, che è incluso in forma binaria.

4.2.5 *DLL esterne*

XVR consente l'inclusione negli script di librerie esterne. Si tratta di una caratteristica molto importante, che permette la massima flessibilità: si possono per esempio collegare al proprio PC periferiche di input non standard (come la MS Kinect) ed interfacciarle coi propri file caricandovi la `DLL` per gestirle. Un player privo di tale capacità sarebbe stato un esercizio di stile, non certo un programma di cui andare fieri.

In Sezione 3.2.4 è stato accennato il problema (inline assembly) ben visibile nel Listato 11, nonché le soluzione adottata. La libreria ivi presentata non solo fornisce i frammenti di assembler necessari, ma è anche concettualmente molto vicina a ciò che

```
#ifdef WIN32
#   define INVIO '\n'
#else
#   define INVIO '\r'
#endif

std::map<int, int> const keymapping = boost::assign::map_list_of('\b', 0x08)('\t',
     0x09)(INVIO, 0x0D)(27, 0x1B)(' ', 0x20)(GLUT_KEY_PAGE_UP, 0x21)(
     GLUT_KEY_PAGE_DOWN, 0x22)(GLUT_KEY_END, 0x23)(GLUT_KEY_HOME, 0x24)(
     GLUT_KEY_LEFT, 0x25)(GLUT_KEY_UP, 0x26)(GLUT_KEY_RIGHT, 0x27)(GLUT_KEY_DOWN,
     0x28)(GLUT_KEY_INSERT, 0x2D)(127, 0x2E)('*', 0x6A)('+', 0xBB)('-', 0xBC)('/', 0
     x6F)(GLUT_KEY_F1, 0x70)(GLUT_KEY_F2, 0x71)(GLUT_KEY_F3, 0x72)(GLUT_KEY_F4, 0
     x73)(GLUT_KEY_F5, 0x74)(GLUT_KEY_F6, 0x75)(GLUT_KEY_F7, 0x76)(GLUT_KEY_F8, 0
     x77)(GLUT_KEY_F9, 0x78)(GLUT_KEY_F10, 0x79)(GLUT_KEY_F11, 0x7A)(GLUT_KEY_F12,
      0x7B)(',', 0xBC)('.', 0xBE);

static std::list<int> pressedkeys;
unsigned int glmods = 0;

int glutToWinVK(int key){
   if(key >= 'A' && key <= 'Z')
      key -= 'A' - 'a'; //oppure key = std::tolower(key);
   std::map<int, int>::const_iterator mit = keymapping.find(key);
   if(mit != keymapping.end())
      key = mit->second;
   return key;
}

inline void registerKey(int key, bool pressed = true){
   if(pressed){
      if(std::find(pressedkeys.begin(), pressedkeys.end(), key) == pressedkeys.end
     ())
         pressedkeys.push_front(key);
   }
   else
      pressedkeys.remove(key);
}

inline void processKey(int key){
   registerKey(key, true);
   g_Module3D->m_VR->PutKeyboardChar(key);
   xvrqueue->postVRMessage(0x0100, key, 0);
}

void keyboardSpecialEvent(int key, int x, int y){
   if(!g_Module3D->m_VR)
      return;
   glmods = glutGetModifiers();
   if((key == GLUT_KEY_F9 && (glmods & GLUT_ACTIVE_SHIFT))
      || key == GLUT_KEY_F4 && (glmods & GLUT_ACTIVE_ALT)){
      g_Module3D->quit();
   }
   processKey(glutToWinVK(key));
}

bool getKeyState(int winVK){
   switch(winVK){
   case 0x10: // Shift
      return glmods & GLUT_ACTIVE_SHIFT;
   case 0x11: // Ctrl
      return glmods & GLUT_ACTIVE_CTRL;
   case 0x12: // Alt
      return glmods & GLUT_ACTIVE_ALT;
   default:
      return std::find(pressedkeys.begin(), pressedkeys.end(), winVK) !=
     pressedkeys.end();
   }
}
```

Listato 10: Gestione della tastiera

```
int CVmExternDLL::CRegister::CallFunction(void *pFunc, CU_Val *vParams, int
     nParams, CU_Val *uRetVal){
        CVRMachine *pVR = GetVRMachine();
        int OldSP = 0;
        int iRetval;
        double dRetval;
#if defined _M_IX86
        __asm mov OldSP, esp;
        for(int i=nParams-1; i>=0; i--){
                if(vParams[i].m_type == C_DOUBLE){
                        double val = vParams[i].m_Data.d;
                        __asm{
                                fld qword ptr val;
                                sub esp, 8;
                                fstp qword ptr [esp];
                        }
                }
                else{
                        int val = vParams[i].m_Data.i;
                        __asm push val;
                }
        }
        __try{
                __asm{
                        call pFunc;
                        mov esp, OldSP;
                        fstp qword ptr dRetval
                        mov iRetval, eax;
                }

                if( uRetVal->m_type == C_DOUBLE )
                        uRetVal->m_Data.d = dRetval;
                else if( uRetVal->m_type == C_FLOAT )
                        uRetVal->m_Data.f = (float) dRetval;
                else
                        uRetVal->m_Data.i = iRetval;
        }
        __except(EvalException(GetExceptionInformation())){
                pVR->DumpCallStack();
        }
#endif// M_I_X86
        return 0;
}
```

Listato 11: `CVmExternDLL`: vecchia invocazione

```
ffi_type* CVmExternDLL::typeFromCUVal(CU_Val* val){
   switch(val->m_type) {
   case C_VOID: return &ffi_type_void;
   case C_INT: return &ffi_type_sint32;
   case C_FLOAT: return &ffi_type_float;
   case C_DOUBLE: return &ffi_type_double;
   default: return &ffi_type_pointer;
   }
}

void* CVmExternDLL::actualVarFromCUVal(CU_Val* val){
   switch(val->m_type){
   case C_VOID: return NULL;
   case C_INT: return &val->m_Data.i;
   case C_FLOAT: return &val->m_Data.f;
   case C_DOUBLE: return &val->m_Data.d;
   default: return &val->m_Data.pt;
   }
}

int CVmExternDLL::CRegister::CallFunction(void* pFunc, CU_Val* vParams, int
     nParams, CU_Val* uRetVal){
   ffi_cif* funcCif = static_cast<ffi_cif*>(malloc(sizeof(ffi_cif)));
   ffi_type* retType = typeFromCUVal(uRetVal);
   ffi_type* argTypes[nParams];
   for(int i = 0; i < nParams; ++i)
      argTypes[i] = typeFromCUVal(&vParams[i]);
   ffi_status cifStat = ffi_prep_cif(funcCif, FFI_DEFAULT_ABI, nParams, retType,
     argTypes);
   if(cifStat == FFI_OK){
      void* retVar = actualVarFromCUVal(uRetVal);
      void* argVars[nParams];
      for(int i = 0; i < nParams; ++i)
         argVars[i] = actualVarFromCUVal(&vParams[i]);
      ffi_call(funcCif, FFI_FN(pFunc), retVar, argVars);
   }
   free(funcCif);
   return cifStat;
}
```

Listato 12: `CVmExternDLL`: nuova invocazione

avveniva già in `CVmExternDLL`, quindi le modifiche da apportare alla classe sono state minime (quasi tutte concentrate nei frammenti di codice mostrati nel Listato 12)

4.2.6 *Video Library*

Nella Sezione 3.3.3 è stata descritta l'architettura della libreria video, ma nulla è stato detto su come le classi siano state effettivamente realizzate. Dalla Figura 21a si capisce che alla base della libreria c'è l'oggetto `VFrame`, l'unico indipendente dagli altri. In teoria si tratterebbe di una classe Plain Old Data (POD), ma nella fase iniziale di progettazione ci aggiunsi anche il costruttore di copia, quindi ho scelto di implementarlo come `class`

```
/** Header */

class VFrame {
public:
    VFrame(const uint8_t* buffer, int size, millitime PTS, frameType type);
#if __cplusplus >= 201103L
    VFrame(const VFrame& fr):
        VFrame(fr.myBuffer, fr.mySize, fr.myPTS, fr.myType){}
#else
    VFrame(const VFrame& fr);
private:
    void init(const uint8_t* buffer, int size, millitime PTS, frameType type);
public:
#endif
//...

/** Implementazione */

VFrame::VFrame(const uint8_t* buffer, int size, millitime PTS, frameType type) {
#if __cplusplus < 201103L
    init(buffer, size, PTS, type);
}

VFrame::VFrame(const VFrame& fr) {
    init(fr.myBuffer, fr.mySize, fr.myPTS, fr.myType);
}

void VFrame::init(const uint8_t* buffer, int size, millitime PTS, frameType type){
#endif
    mySize = size;
    myBuffer = new uint8_t[mySize];
    memcpy(myBuffer, buffer, mySize);
    myPTS = PTS;
    myType = type;
}
```

Listato 13: costruttore copia di `VFrame`

anziché come `struct`. Durante un successivo refatctoring della libreria il thread ha abbandonato l'uso della clonabilità dell'oggetto, ma il costruttore era una buona occasione per provare i *delegating constructors* di C++11 e nel contempo scrivere codice portabile anche su C++03 il più compatto possibile, perciò non ho rimosso le parti di codice visibili nel Listato 13 e di conseguenza non ho riconvertito l'oggetto a `struct`. Per gli operatori di confronto ho scelto di implementarli come non-member, secondo le regole suggerite in [11], riscrivendo non solo l'operatore '<' richiesto per l'ordinamento delle liste, ma anche tutti gli altri operatori di confronto in funzione del primo (vedi Listato 14).

In `VQueue` i `VFrame` sono salvati in una `std::list`. Si tratta di un contenitore leggero, con pochissimo overhead e che può essere facilmente tenuto ordinato. I fotogrammi vengono per lo più decodificati in ordine di presentazione, e comunque anche

```
inline bool operator<(const VFrame& lhs, const VFrame& rhs) {
    return lhs.myPTS < rhs.myPTS;
}
inline bool operator<=(const VFrame& lhs, const VFrame& rhs) {
    return !(rhs < lhs);
}
inline bool operator>(const VFrame& lhs, const VFrame& rhs) {
    return !(lhs <= rhs);
}
inline bool operator>=(const VFrame& lhs, const VFrame& rhs) {
    return !(lhs < rhs);
}
inline bool operator==(const VFrame& lhs, const VFrame& rhs) {
    return lhs.myPTS == rhs.myPTS;
}
inline bool operator!=(const VFrame& lhs, const VFrame& rhs) {
    return !(lhs == rhs);
}
```

Listato 14: Operatori di confronto

i B-frame sono di poco precedenti all'ultimo frame accodato, perciò quasi sempre la lista ordinata richiederà un inserimento in coda. Se si fa una ricerca della posizione corretta usando un *reverse iterator* si avrà spesso il caso ottimo di successo della ricerca alla prima iterazione, e tutte le altre volte casi subottimi in cui occorre esaminare 2-3 elementi (a prescindere dalla dimensione della coda), quindi si può tranquillamente affermare che il costo di inserimento è costante.

```
void VQueue::enqueue(VFrame* frame) {
    std::list<VFrame*>::reverse_iterator it;
    for(it = frames.rbegin(); it != frames.rend() && **it > *frame; ++it);
    frames.insert(it.base(), frame);
}
```

Listato 15: Iteratore inverso in `enqueue`

L'iteratore inverso viene usato anche nel metodo `shiftTime`, ma in questo caso per ridurre le possibilità di errore. La funzione serve per quando si esce da una pausa nella riproduzione, e durante il periodo di pausa è lecito continuare la decodifica dello stream. Nel caso in cui la coda non sia satura mi trovo ad avere un thread che incrementa il PTS dei frame accodati ed un altro che aggiunge nuovi frame col PTS già aggiornato. Usando un forward iterator si otterrebbe una sorta di gara tra `shiftTime` e il thread video, in cui il primo cerca di raggiungere la fine della lista mentre il secondo continua ad allontanare il traguardo, ed ovviamente tutti i fotogrammi aggiunti dopo la fine della pausa si ritroverebbero con un timestamp errato.

```
void VQueue::shiftTime(millisecs& increment) {
```

```
millitime VQueue::getNextPTS() {
    if(frames.empty())
        throw std::logic_error("Queue is empty");
    return frames.front()->myPTS;
}

frameType VQueue::getNextType() {
    if(frames.empty())
        throw std::logic_error("Queue is empty");
    return frames.front()->myType;
}

VFrame* VQueue::dequeue() {
    if(frames.empty())
        return NULL;
    VFrame* frame = frames.front();
    frames.pop_front();
    return frame;
}
```

Listato 17: Query con coda vuota

```
    for(std::list<VFrame*>::reverse_iterator rit = frames.rbegin();
            rit != frames.rend(); ++rit)
        (*rit)->myPTS += increment;
}
```

Listato 16: Iteratore inverso in `shiftTime`

Per capire esattamente come funziona la pausa ci si riferisca al Listato 23. Iterando a partire dal fondo invece gli unici frame sbagliati sarebbero quelli calcolati tra quando viene calcolato il nuovo orario e la chiamata di `rbegin()`.

Dal momento che accedere al primo elemento di una lista vuota è un *undefined behaviour* è lecito trattare come errori le query sul primo elemento quando non ci sono frame in coda, quindi i metodi `getNextPTS` e `getNextTime` lanciano uno `std::logic_error` anziché definire arbitrariamente valori di errore. Il metodo `dequeue` invece ritorna un puntatore, elemento per il quale esiste già un valore universale di errore, perciò in quel caso una chiamata sulla lista vuota si limita a restituire `NULL`. Nel Listato 17 sono mostrati entrambi i casi.

Il `VideoObject` fa da collante tra FFmpeg, `VQueue` e `VThread`. Infatti oltre al puntatore alla coda in cui inserire i frame ha anche tutta una serie di dati e strutture di appoggio per la decodifica (Listato 18) e per il thread (Listato 19). Nonostante le apparenze non si tratta affatto di un POD, in questa classe il costruttore è fondamentale perché si preoccupa di inizializzare correttamente tutti i codec e i vari dati extra richiesti da FFmpeg:

```
AVFormatContext* formatCtx;
AVCodecContext *vCodecCtx, *aCodecCtx;
SwsContext* vSwCtx;
SwrContext* aSwCtx;
AVCodec *vCodec, *aCodec;
AVFrame* frame; // frame decodificato
AVFrame* frameRGB; // frame video riconvertito
uint8_t* bufferedFrame; // frameRGB salva qui i dati effettivi
AVPacket avPacket;
int videoStream, audioStream;
int picSize, picWidth, picHeight; // misure del frame video, servono anche agli
    utenti della VRVlib
```

Listato 18: Dati per FFmpeg

```
char* fileName; // Lo salvo così posso ricaricarlo a piacimento
bool looping;
millitime startTime, pauseTime; //per il calcolo del PTS
reproductionState repState; // play, pause, stop, quit
void(*frameReady)(void*);
void(*quitVideo)(void*);
```

Listato 19: Dati per `VThread`

```
VideoObject::VideoObject(const char* myFileName, VQueue* myQueue, bool loop, void
    (*frameCallback)(void*), void(*quitCallback)(void*)) {
    if(!myFileName)
        throw std::invalid_argument("File name can't be NULL");
    if(!myQueue)
        throw std::invalid_argument("An existing queue to fill is required");

    fileName = new char[(std::strlen(myFileName) + 1)];
    strcpy(fileName, myFileName);
    // ...

    av_init_packet(&avPacket);
    formatCtx = avformat_alloc_context();

    if(avformat_open_input(&formatCtx, fileName, NULL, NULL) != 0)
        reset(std::string("can't open vieo file ") + fileName);
    if(avformat_find_stream_info(formatCtx, NULL) < 0)
        reset("Can't file video stream information");

    for(unsigned int i = 0; i < formatCtx->nb_streams; i++) {
        switch(formatCtx->streams[i]->codec->codec_type) {
        case AVMEDIA_TYPE_VIDEO:
            videoStream = i;
            break;
        case AVMEDIA_TYPE_AUDIO:
            audioStream = i;
        }
    }

    initCodec(&videoStream, &vCodecCtx, &vCodec, "Video");
    if(videoStream >= 0) {
        picWidth = vCodecCtx->width;
        picHeight = vCodecCtx->height;
        picSize = avpicture_get_size(AV_PIX_FMT_RGBA, picWidth, picHeight); //
    calcola il numero di byte per ogni frame video
```

```
        bufferedFrame = static_cast<uint8_t*>(av_malloc(picSize*sizeof(uint8_t)));
        frame = avcodec_alloc_frame();
        frameRGB = avcodec_alloc_frame();
        avpicture_fill(reinterpret_cast<AVPicture*>(frameRGB), bufferedFrame,
            AV_PIX_FMT_RGBA, picWidth, picHeight); // collego il buffer al frame.
     Il cast è brutto ma inevitabile...
        vSwCtx = sws_getContext(picWidth, picHeight, vCodecCtx->pix_fmt, picWidth,
            picHeight, AV_PIX_FMT_RGBA, 0, NULL, NULL, NULL);
    }

    initCodec(&audioStream, &aCodecCtx, &aCodec, "Audio");
    if(audioStream >= 0) {
        aSwCtx = swr_alloc_set_opts(aSwCtx, AV_CH_LAYOUT_STEREO,
            AV_SAMPLE_FMT_S16, MIX_DEFAULT_FREQUENCY, aCodecCtx->channel_layout,
            aCodecCtx->sample_fmt, aCodecCtx->sample_rate, 0, NULL);
        swr_init(aSwCtx);
    }
}

void VideoObject::initCodec(int* stream, AVCodecContext** ctx, AVCodec** codec,
     const char* type) {
    if(*stream >= 0) {
        *ctx = formatCtx->streams[*stream]->codec;
        if(*ctx != NULL) {
            *codec = avcodec_find_decoder((*ctx)->codec_id);
            if(*codec != NULL && (avcodec_open2(*ctx, *codec, NULL) >= 0))
                return;
        }
    }
    // cleanup in caso di errore
    cout << type << " codec not supported" << endl;
    *codec = NULL;
    *ctx = NULL;
    *stream = -1;
}

inline void reset(std::string error) {
    delete[] fileName;
    fileName = NULL;
    av_free_packet(&avPacket);
    avformat_free_context(formatCtx);
    throw std::runtime_error(error);
}
```

L'ultimo oggetto della libreria è `VThread`, che si occupa di gestire il `boost::thread` vero e proprio. Quest'ultimo è suddiviso in due parti: nella prima decodifica ed accoda il frame successivo:

```
std::map<VideoObject*, void*>::iterator it;
for(it = manager->processes.begin(); it != manager->processes.end(); ++it) {
    VideoObject* obj = it->first;
    if(obj->repState != stopped
            && obj->queue->size() < MAX_QUEUE_SIZE
            && av_read_frame(obj->formatCtx, &(obj->avPacket)) >= 0) {
        if(obj->videoStream >= 0
                && obj->avPacket.stream_index == obj->videoStream)
            manager->decodeVideo(obj);
        else if(obj->audioStream >= 0
                && obj->avPacket.stream_index == obj->audioStream)
            manager->decodeAudio(obj);
    }
    // ...
```

mentre nella seconda verifica se ci sono fotogrammi da mostrare ed eventualmente lo notifica:

```
    // ...
    try {
        if(obj->repState == playing
                && obj->queue->getNextPTS() <= chr::high_resolution_clock::now())
            obj->frameReady(it->second);
    }
    catch(const std::logic_error&) { // non ho più fotogrammmi, segno che il video
      è finito
        if(obj->looping) { // il video è in loop, lo riavvolgo e ricomincio da
     capo
            manager->stop(obj);
            manager->play(obj);
        }
        else { // rimuovo il processo dalla coda di quelli attivi
            if(obj->quitVideo)
                obj->quitVideo(it->second);
            ++it;
            manager->removeProcess(obj);
            delete obj;
            if(manager->processes.empty())
                break;
            --it;
        }
    }
    boost::this_thread::interruption_point();
}
```

Per mantenere il codice pulito e leggibile le istruzioni relative al decoding vero e proprio sono state spostate in metodi separati (Listato 20 per il video e Listato 21 per l'audio). Purtroppo le interfacce di SwScale e SwResample non sono omogenee, quindi i due metodi sono molto diversi; se per il video il codice è abbastanza lineare e pulito, per l'audio non esiste una tecnica analoga, sono richieste tutta una serie di operazioni ricavate leggendo tutorial trovati su internet oppure frammenti di codice di esempio. Il `const_cast` è stato necessario per una svista nella firma del metodo `swr_convert` (per essere corretto il penultimo parametro dovrebbe essere di tipo `const uint8_t * const *`). Per il calcolo del PTS inizialmente ho provato a guardare come

```
int frameReady = 0;
AVFrame* frame = obj->frame;
while(frameReady == 0)
    avcodec_decode_video2(obj->vCodecCtx, frame, &frameReady,
        &(obj->avPacket));
sws_scale(obj->vSwCtx, frame->data, frame->linesize, 0,
    obj->picHeight, obj->frameRGB->data, obj->frameRGB->linesize);
millitime pts = computePTS(obj, obj->vCodecCtx->time_base);
VFrame* frm = new VFrame(obj->bufferedFrame, obj->picSize, pts, frameVideo);
obj->queue->enqueue(frm);
```

Listato 20: Decodifica di un frame video

```
AVFrame* frame = obj->frame;
int packetSize = obj->avPacket.size;
while(packetSize > 0) {
    int frameReady = 0;
    while(frameReady == 0) {
        int decodedLen = avcodec_decode_audio4(
            obj->aCodecCtx, frame, &frameReady, &obj->avPacket);
        if(decodedLen < 0)
            return;
        packetSize -= decodedLen;
    }
    int resampleSize = av_rescale_rnd(
        swr_get_delay(obj->aSwCtx, obj->aCodecCtx->sample_rate)
        + frame->nb_samples, MIX_DEFAULT_FREQUENCY,
        obj->aCodecCtx->sample_rate, AV_ROUND_UP);
    uint8_t* audioBuffer;
    av_samples_alloc(&audioBuffer, NULL, 2, resampleSize,
        AV_SAMPLE_FMT_S16, 1);
    resampleSize = swr_convert(obj->aSwCtx, &audioBuffer, resampleSize,
        const_cast<const uint8_t**>(frame->data), frame->nb_samples);
    int bufSize = av_samples_get_buffer_size(NULL, 2, resampleSize,
        AV_SAMPLE_FMT_S16, 1);
    VFrame* vframe = new VFrame(audioBuffer, bufSize,
        computePTS(obj, obj->aCodecCtx->pkt_timebase), frameAudio);
    obj->queue->enqueue(vframe);
    av_free(audioBuffer);
}
```

Listato 21: Decodifica di un frame audio

avviene in [5], poi ho esaminato la documentazione ufficiale, scoprendo che nel frattempo altri campi erano stati aggiunti per facilitare il calcolo. Il codice mostrato nel Listato 22 deriva dalle varie osservazioni fatte, ed è stato testato stampando a schermo i risultati per verificarne la correttezza.

```
millitime VThread::computePTS(VideoObject* obj, AVRational ratio) {
    int64_t realPTS = obj->avPacket.pts;
    if(realPTS == AV_NOPTS_VALUE) {
        realPTS = obj->avPacket.dts;
        if(realPTS == AV_NOPTS_VALUE)
            realPTS = 0;
    }
    double timeBase = av_q2d(ratio) * 1000;
    double nextFrame = realPTS * timeBase;
    millisecs delay = millisecs(static_cast<int64_t>(nextFrame));
    return obj->startTime + delay;
}
```

Listato 22: Calcolo del PTS

```
void VThread::pause(VideoObject* obj) {
    if(obj->repState != playing)
        return;
    obj->pauseTime =
        chr::time_point_cast<millisecs>(chr::high_resolution_clock::now());
    obj->repState = paused;
}

void VThread::resume(VideoObject* obj) {
    if(obj->repState != paused)
        return;
    millisecs delay =
        chr::time_point_cast<millisecs>(chr::high_resolution_clock::now())
        - obj->pauseTime;
    obj->startTime += delay;
    obj->queue->shiftTime(delay);
    obj->repState = playing;
}
```

Listato 23: Pausa

4.2.6.1 *Uso della libreria*

In `CVmMMAvi`, al caricamento di un file, viene inizializzato il `VideoObject` e creata una finestra di dimensioni opportune in cui riprodurre il video:

```
This->obj = xvrvthread->loadFile(completeFileName, This->vqueue, false, This,
    This->frameReadyCallback, This->videoQuitCallback);
This->parentWin = glutGetWindow();
This->subWin = glutCreateSubWindow(This->parentWin, 0, 0, This->obj->picWidth,
    This->obj->picHeight);
CVmMMAvi::screenBuffer = new uint8_t[This->obj->picSize];
glutDisplayFunc(displayVideo);
glutSetWindow(This->parentWin);
```

L'array statico che viene allocato serve per poter disegnare nella nuova finestra creata. Poiché l'oggetto `CVmMMAvi` è un singleton e gestisce un solo video alla volta non ci sono problemi nel fatto che il buffer sia statico, ma in compenso possiamo usarlo all'interno della *display function* di GLUT, che non può avere parametri in input. Avendo settato le dimensioni della finestra uguali a quelle dell'immagine, e usando un formato noto a priori, è banale generare una texture 2D contenente il fotogramma da mostrare, quindi è sufficiente assegnarla ad un quadrilatero che occupa l'intero campo visivo per avere l'effetto di un riproduttore multimediale:

```
void CVmMMAvi::displayVideo() {
    glEnable(GL_TEXTURE_2D);
    GLuint tex2d;
    glGenTextures(1, &tex2d);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
    glBindTexture(GL_TEXTURE_2D, tex2d);
    glTexImage2D(
        GL_TEXTURE_2D, 0, GL_RGBA, glutGet(GLUT_WINDOW_WIDTH),
        glutGet(GLUT_WINDOW_HEIGHT), 0 , GL_RGBA,
        GL_UNSIGNED_BYTE, CVmMMAvi::screenBuffer
    );
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glBegin(GL_QUADS);
        glTexCoord2f(0, 0);
        glVertex2f(-1, 1);
        glTexCoord2f(1, 0);
        glVertex2f(1, 1);
        glTexCoord2f(1, 1);
        glVertex2f(1, -1);
        glTexCoord2f(0, 1);
        glVertex2f(-1, -1);
    glEnd();
    glDisable(GL_TEXTURE_2D);
}
```

La callback della VRVLIB non deve far altro che copiare il contenuto del frame nel buffer statico e mandare un messaggio di segnalazione, così che al momento del dispatch dei messaggi accodati la finestra venga invalidata. Poiché il buffer viene conti-

nuamente sovrascritto tutti i frame che non sono stati disegnati in tempo vengono automaticamente scartati:

```
void CVmMMAvi::frameReadyCallback(void* opaque) {
    int chn = -1;
    CVmMMAvi* opq = static_cast<CVmMMAvi*>(opaque);
    VFrame* fr = opq->vqueue->dequeue();

    switch(fr->myType) {
    case frameVideo:
        memcpy(opq->screenBuffer, fr->myBuffer, fr->mySize);
        xvrqueue->postVRMessage(VM_VIDEO_REFRESH, opq->parentWin, opq->subWin);
    break;
    // ...
```

Nel caso di pacchetti audio lavoro su un anello di buffer, perché tra i vari tentativi fatti per capire quale fosse il problema ho voluto anche verificare che non si trattasse del tempo perso per fermare il vecchio frammento e caricare quello nuovo. Il campo `AUDIO_SAMPLES_NUM` naturalmente non è una macro ma una variabile di tipo `const int`:

```
    // ...
    case frameAudio:
        chn = (opq->currentChannel+1) % AUDIO_SAMPLES_NUM;
        if(opq->audioChannel[chn] >= 0 && !Mix_Playing(opq->audioChannel[chn]))
            Mix_FreeChunk(opq->audioChunk[chn]);

        if(opq->audioChannel[chn] < 0 || !Mix_Playing(opq->audioChannel[chn])) {
            delete[] opq->audioBuffer[chn];
            opq->audioBuffer[chn] = new uint8_t[fr->mySize];
            memcpy(opq->audioBuffer[chn], fr->myBuffer, fr->mySize);
            opq->audioChunk[chn] = Mix_QuickLoad_RAW(opq->audioBuffer[chn],
                fr->mySize);
            opq->audioChannel[chn] = Mix_PlayChannel(opq->audioChannel[chn],
                opq->audioChunk[chn], 0);
            opq->currentChannel = chn;
        }
    break;
```

Il risultato di tutto il lavoro fatto lo si può ammirare con un paragone tra la Figura 22, in cui è visibile l'output di un CVmMMAvi appena descritto, la figura Figura 23, che mostra come il player originale sia problematico anche sotto MS Windows, e la Figura 24 da cui si capisce che Wine fallisce completamente nel task.

Figura 22: Video riprodotto con la nuova libreria

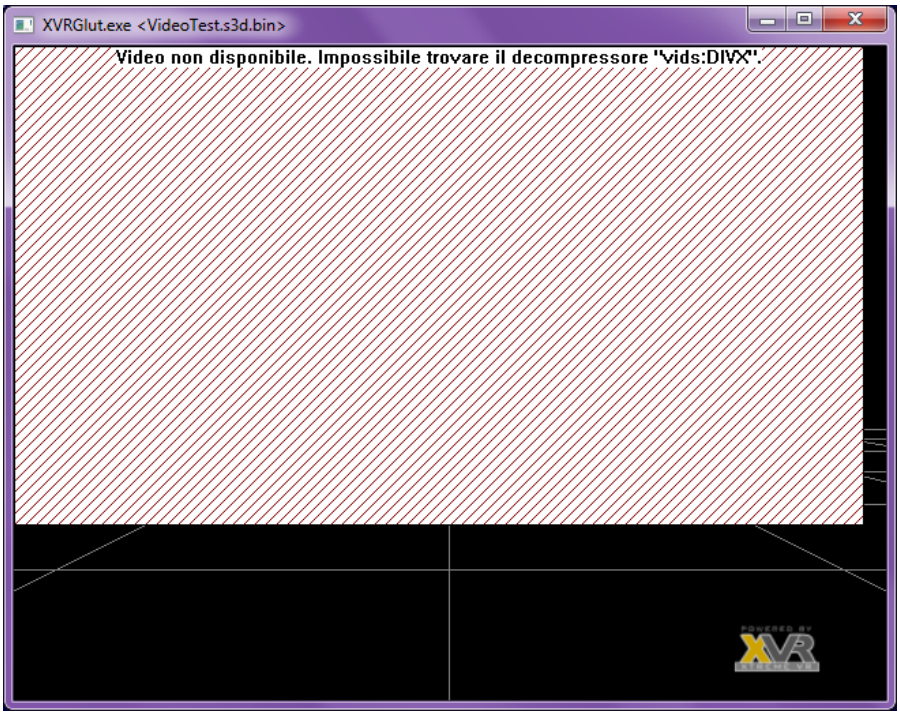


Figura 23: Tentativo di riproduzione col player originale

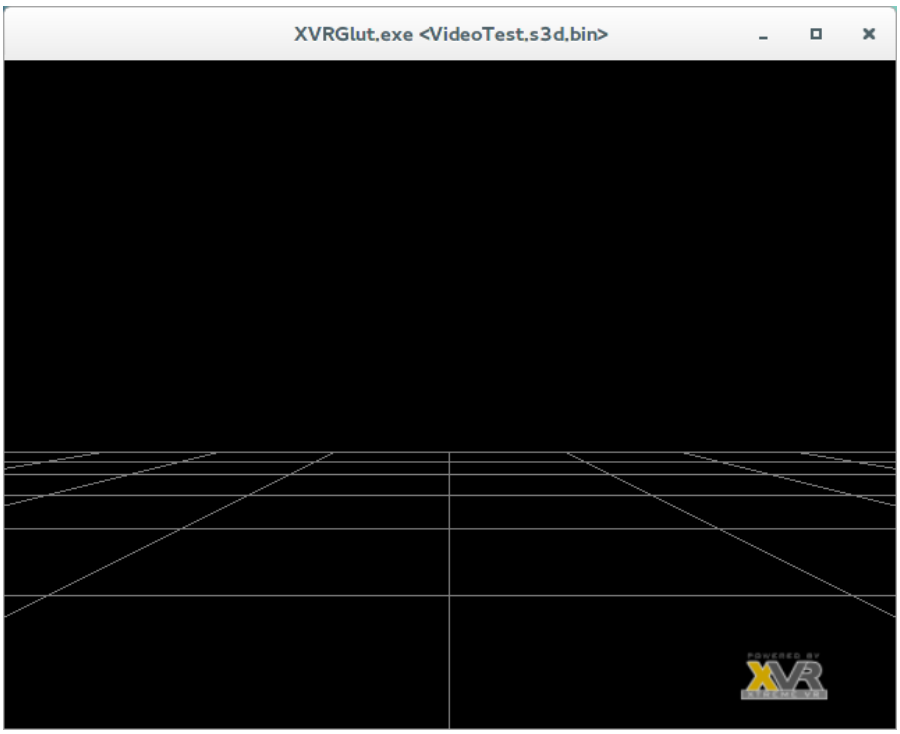


Figura 24: "Riproduzione" di video con wine

4.2.7 *Shared Memory*

Gli oggetti per la gestione di memorie condivise sono un ottimo esempio di come molto del lavoro sia avvenuto *under the hood*. Nel file `SharedMemory.h` sono presenti due oggetti: un mutex per gestire gli accessi condivisi e la memoria vera e propria. La costruzione del mutex avveniva sfruttando funzionalità di windows:

```
class PbmMemoryMutex {
    HANDLE m_hMutex;
    boolean existing;
public:
    PbmMemoryMutex(char *mutex_name) {
        m_hMutex = ::CreateMutex(NULL, FALSE, mutex_name);
        if (GetLastError() ==  ERROR_ALREADY_EXISTS)
            existing = true;
        else
            existing = false;
        return;
    }
```

La libreria INTERPROCESS di BOOST fornisce degli oggetti analoghi, i `named_mutex`, che essendo stati scritti in C++ usano le eccezioni invece degli error code:

```
class PbmMemoryMutex {
    itp::named_mutex* m_hMutex = NULL;
    bool existing;
```

```
public:
    PbmMemoryMutex(char* mutex_name) {
        try {
            m_hMutex = new itp::named_mutex(itp::create_only, mutex_name);
            existing = false;
        }
        catch(itp::interprocess_exception& e) {
            m_hMutex = new itp::named_mutex(itp::open_or_create, mutex_name);
            existing = true;
        }
    }
```

L'ingresso nella sezione critica è stato modificato per adeguarsi alla gestione del tempo di Boost, da:

```
bool Enter(DWORD timer=INFINITE) {
    return (::WaitForSingleObject( m_hMutex, timer ) == WAIT_OBJECT_0 );
}
```

a:

```
bool Enter(unsigned int timer = UINT_MAX) {
    pt::ptime timerposix;
    if(timer == UINT_MAX)
        timerposix = pt::pos_infin;
    else
        timerposix = pt::ptime(pt::second_clock::local_time()+pt::millisec(100));

    try {
        return m_hMutex->timed_lock(timerposix);
    }
    catch(itp::interprocess_exception& e) {
        printf("Mutex error: %s", e.what());
        return false;
    }
}
```

Il blocco `try-catch` era necessario, perché il vecchio codice non lanciava alcuna eccezione, quindi gli utenti non sarebbero pronti a gestirle.

Nell'oggetto `sharedMemory` venivano utilizzati oggetti mappabili e viste, di cui Boost fornisce gli ottimi sostituti multipiattaforma `shared_memory_object` e `mapped_region`. Per prima cosa il vecchio codice verificava se veniva creato un `FileMapping` ex novo oppure aperto uno già esistente:

```
hFile = CreateFileMapping(
    INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, size, name);
if (hFile == NULL) {
    printf(" Unable to create a shared memory file.\n");
    return;
}
else
    if (GetLastError() == ERROR_ALREADY_EXISTS)
        existing = true;
```

Gli stessi controlli vengono fatti gestendo le eccezioni, e ovviamente il tipo di `hFile` non è più un `HANDLER` bensì un puntatore a `boost::interprocess::shared_memory_object`:

```
try {
    hFile = new itp::shared_memory_object(
        itp::create_only, name, itp::read_write);
}
catch(itp::interprocess_exception&) {
    try {
        hFile = new itp::shared_memory_object(
            itp::open_only, name, itp::read_write);
        existing = true;
    }
    catch(itp::interprocess_exception& e) {
        printf(" Unable to create a shared memory file: %s.\n", e.what());
        return;
    }
}
hFile->truncate(size);
```

Per la successiva mappatura della memoria condivisa Windows ritorna un indirizzo, mentre con Boost si crea un oggetto che tra le altre cose può anche fornire l'indirizzo di partenza della regione di memoria:

```
hView = (char*) MapViewOfFile(hFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
if (hView == NULL) {
     printf("Unable to create a view on shared memory.\n");
     return;
}
m_buffer = hView;
```

```
hView = new itp::mapped_region(*hFile, itp::read_write);
m_buffer = static_cast<char*>(hView->get_address());
if (!m_buffer) {
     printf("Unable to create a view on shared memory.\n");
     return;
}
```

Al posto dei messaggi di errore e dei return sarebbe stato più corretto lanciare un'eccezione, così da essere certi che nessuno utilizzi oggetti non completamente inizializzati, ma il mio obiettivo era quello di ricalcare il più fedelmente possibile il comportamento del codice esistente.

Il resto della classe non fa altro che bloccare/rilasciare il mutex e scrivere in `m_buffer`: poiché né l'interfaccia del primo né la semantica del secondo sono cambiate non sono richieste altre modifiche perché il codice funzioni.

4.3 FILE EMBEDDER

È stato spiegato in Sezione 3.3.2 che per l'inserimento di risorse binarie ho utilizzato un semplice programmino scritto appositamente da me. Il file non fa parte di XVR, ma ne può essere considerato uno *spin off*, quindi per completezza ho voluto pubblicarlo in questa sede. Nel Listato 24 è visibile l'intero file,

come si può capire dal codice l'output è un array di `unsigned char` scritti in forma `0xNM`; è possibile decidere dopo quanti byte inserire un fine linea e se non viene specificato il nome di un file di output il risultato viene stampato a schermo.

```
/**
* FileEmbedder: converts files to C char arrays.
* Copyright (C) 2013  Ilario Sanseverino
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as
* published by the Free Software Foundation, either version 3 of the
* License, or (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU Affero General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#include <stdio.h>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string.h>
#include <stdlib.h>

using std::cout;
using std::endl;

char* inputfile = NULL;
char* outputfile = NULL;
int bytesPerRow = 12;
bool infoDisplayed = false;

void usage() {
   cout << endl << "Convert the input file to an unsigned char array." << endl;
   cout << "Usage: embedder -i inputfile [-o outputfile] [-n N] [-h]" << endl;
   cout << "-o filename: save output to file. If omitted displays result to the standard
     output." << endl;
   cout << "-n N: prints N bytes before breaking to new line. Default: 12." << endl;
   cout << "-h: shows this info and continues execution" << endl << endl;

   infoDisplayed = true;
}

void errorUsage(const char* errorMessage) {
   std::cerr << endl << errorMessage << endl;
   usage();
}

void parseParams(int argc, char* argv[]) {
   for(int i = 1; i < argc; ++i) {
      const char *p = argv[i];
      if(p[0] == '-') {
         switch(*(++p)) {
         case 'i':
            inputfile = argv[++i]; break;
         case 'o':
            outputfile = argv[++i]; break;
         case 'n':
```

```
            bytesPerRow = atoi(argv[++i]); break;
         case 'h':
            usage(); break;
         default: {
               const char* errmsg = "unknow option: %c";
               char msg[strlen(errmsg)];
               sprintf(msg, errmsg, *p);
               errorUsage(msg);
               exit(-1);
            }
         }
      }
      else {
         errorUsage("Wrong parameter");
         exit(-1);
      }
   }
}

int main(int argc, char* argv[]) {
   parseParams(argc, argv);
   if(!inputfile) {
      if(!infoDisplayed)
         errorUsage("Missing input file. Please try 'embedder -h' for help.");
      exit(-1);
   }

   FILE* fl = fopen(inputfile, "r");
   std::ofstream ofstr;
   std::ostream* ostr = &cout;
   if(outputfile) {
      ofstr.open(outputfile);
      ostr = &ofstr;
   }

   fseek(fl, 0, SEEK_END);
   size_t fsize = ftell(fl);
   fseek(fl, 0, SEEK_SET);
   unsigned char buffer[fsize];
   int byteread = fread(buffer, 1, fsize, fl);

   *ostr << "static unsigned char myfile[] = {";
   for(int i = 0; i < fsize; ++i) {
      if(i%bytesPerRow == 0)
         *ostr << endl << "\t";
      *ostr << "0x" << std::setw(2) << std::setfill('0') << std::hex << (int)buffer
     [i] << ", ";
   }
   *ostr << endl << "};" << endl;

   if(ofstr.is_open())
      ofstr.close();

   return byteread;
}
```

Listato 24: Binary file embedder

5 CONCLUSIONI

5.1 OBIETTIVI RAGGIUNTI

Lo scopo iniziale del lavoro era quello di ottenere un player di base funzionante almeno in Linux, ed è stato ampiamente centrato. Gli script con operazioni di rendering adeguatamente leggere girano attorno ai 100 frame al secondo (se non diversamente specificato dall'utente), le mesh `AAM` e le texture da immagini vengono correttamente caricate, il mouse è gestito a dovere. Inoltre, cosa molto importante, non è stata rotta la retrocompatibilità: se una componente funziona allora funziona come prima, se invece è stata rimossa si preferisce usare warning oppure ritornare (se previsti) valori che indicano fallimenti, piuttosto che lanciare errori a runtime. Ma comunque gli script non devono essere ricompilati, l'header `Script3d.h` (che viene incluso automaticamente nei file `S3D`) è rimasto inalterato.

Sono inoltre arrivato ad un buon punto con l'obiettivo secondario, che è quello di mantenere anche le caratteristiche minori. È possibile caricare librerie esterne, quindi se uno script necessita di `DLL` per le quali esiste anche una versione `SO` per Linux lo si può utilizzare tranquillamente. Anche il supporto per il networking è presente e completo. La console è considerata un elemento secondario solamente perché non funziona nel CAVE, ma è molto utile, perché permette di mostrare tooltip all'utente; anch'essa è stata reimplementata ed ora funziona quasi come in origine, l'unica differenza è che bisogna includere tra le risorse dello script i propri file `TTF` se si vogliono usare caratteri diversi da quello di default, non è più previsto il caricamento del font di sistema per nome.

Per i file audio e video sono andato addirittura oltre l'originale: su Windows 7 i formati multimediali non funzionavano, mentre grazie al mio lavoro si possono nuovamente utilizzare file sonori e filmati. È stato pure aggiunto il supporto a tutti i formati audio liberi, mentre il vecchio XVR accettava solamente `Mp3`.

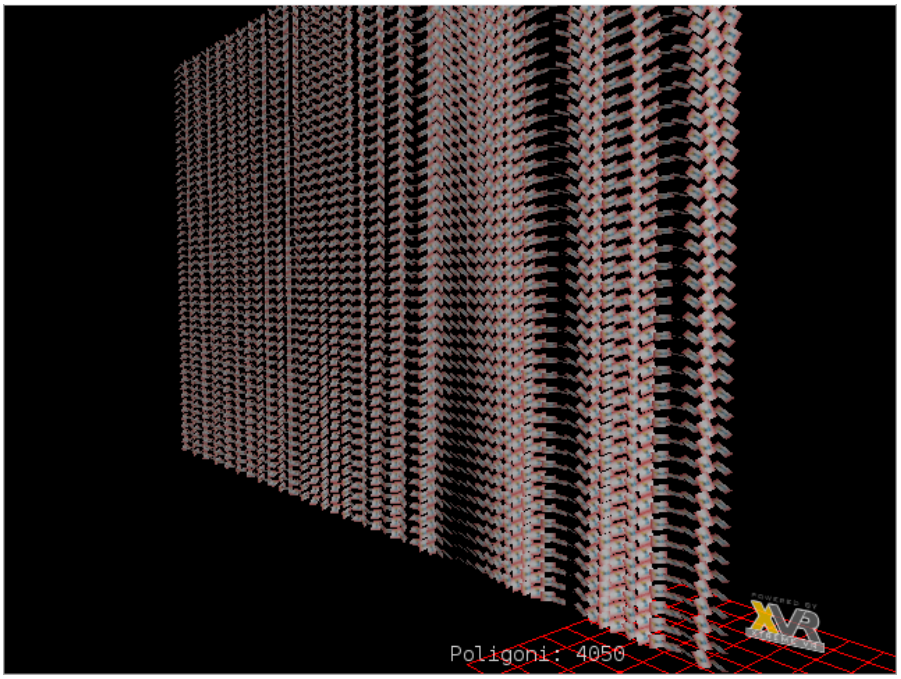


Figura 25: Test sul numero di poligoni in azione

5.2 CONFRONTI

Il lavoro svolto sicuramente non è stato inutile. Nei confronti fatti tra la versione originale di XVR e quella da me riprogettata si può vedere che non c'è alcuna perdita di performance. Se anche è vero che risultati simili si ottengono lanciando il software tramite Wine bisogna però considerare che il mio player può essere eseguito direttamente, senza richiedere l'installazione di qualche centinaio di megabyte di file aggiuntivi; il codice riscritto potrebbe essere ricompilato per girare su Mac OSX PPC, mentre Wine funziona solo sulle versioni x86; l'aver riesaminato il player ha permesso di riorganizzare il codice sorgente, scoprire quali funzioni sono obsolete, trovare eventuali errori, aggiornarlo.

5.2.1 *Capacità del sistema*

La prima prova riguarda il numero di poligoni che il player è in grado di disegnare senza sensibili degradazioni del framerate. Per fare il test ho usato lo script mostrato nel Listato 25, che genera mesh procedurali in continuazione, le ruota, le texturizza e le disegna; un esempio dello script in esecuzione è mostrato in Figura 25. I risultati sono abbastanza incoraggianti: tutte

e tre le versioni iniziano a rallentare al di sopra dei 1000 oggetti, arrivati a 2000 sono tutti intorno ai 40 frame al secondo, aggiungendo altre mesh le prestazioni degradano in maniera molto simile. Nel primo test effettuato, in cui su Linux usavo la versione *debug* del player, il framerate maggiore (seppure di poco) l'ho ottenuto con Wine, mentre nel test con la versione *release* il player Linux nativo ha ottenuto il risultato migliore, come si vede in Figura 26 e Figura 27. La differenza comunque è molto bassa, probabilmente dipende più dalla temperatura raggiunta dall'hardware che non da più efficienti implementazioni delle chiamate OpenGL. Una performance sensibilmente più scarsa invece la si ottiene su Linux se si usano i driver Nouveau anziché quelli proprietari Nvidia, ma d'altronde il sito ufficiale[1] lo dice chiaramente: *Expect low 3D performance on laptops using a Tesla GPU*.

```
#include <Script3d.h>
#include <Camera.s3d.h>

var angles = {}, meshes = {}, curtime, texture, material, Light0;
var itemsPerStep = 50, timestep = 500;

function OnInit(params){
   texture = CVmTexture("BlackMage-80x80.png");
   material = CVmMaterial();
   material.SetTexture(texture);
   curtime = GetTime();
   Light0 = CVmLight();
   Light0.SetPosition([10, 10, -1]);
   Light0.SetDiffuse(1, 1, 1);
   Light0.SetAmbient(0.5, 0.5, 0.5);
   Light0.SetSpecular(0.7, 0.7, 0.7);
   Light0.Enable();
   CameraGetCurrent().SetPosition(5, 5, -20);
}

function OnFrame(){
   CameraMoveMouse_MBL_LC();
   Light0.SetPosition(CameraGetCurrent().GetPosition());
   increaseAngle();
   SceneBegin(VR_SHOW_STATS);
      DrawGrid(20, 20);
      for(var v = 0; v < len(angles); ++v)
         drawProceduralRotatingPlane(v);
   SceneEnd();
   ConsoleText(0.5, 0.01, "Poligoni: "+str(len(angles)*itemsPerStep));
}

function increaseAngle(){
   var s = len(angles);
   var actualtime = getTime();
   if(curtime + timestep <= actualtime) {
      curtime = actualtime;
      aadd(angles, 1.0);
      var msh = CVmNewMesh(VRP_PLANE, 1, [0, 0, 0], [1, 1, 0], 2, 2);
      var o = CVmObj();
```

1 http://nouveau.freedesktop.org/wiki/

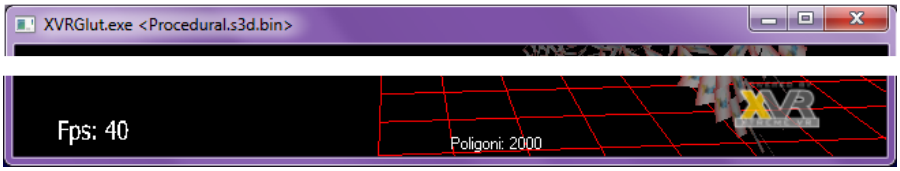
```
      o.SetPivotPoint(0.5, 0.5, 0);
      o.LinkToMesh(msh);
      o.ForceMaterial(material);
      aadd(meshes, o);
      ++s;
   }
   for(var v = 0; v < s; ++v)
      angles[v] += 0.003;
}

function drawProceduralRotatingPlane(index){
   var mesh = meshes[index];
   var angle = angles[index];
   mesh.Rotate(angle, 0, 1, 0);
   for(var c = 0; c < itemsPerStep; ++c) {
      mesh.setPosition(index, c, 0);
      mesh.Rotate(0.1, 1, 0, 0);
      mesh.Draw();
   }
}

function DrawGrid(col, size){
   glPushAttrib(GL_LIGHTING_BIT | GL_LINE_BIT | GL_CURRENT_BIT);
   glLineWidth(1);
   glDisable(GL_LIGHTING);
   glColor(col);
   var max = size / 2.0;
   glBegin(GL_LINES);
      for(var i = -max; i <= max; i += size/10.0){
         glVertex(i, 0, max);
         glVertex(i, 0, -max);
         glVertex(max, 0, i);
         glVertex(-max, 0, i);
      }
   glEnd();
   glPopAttrib();
}
```

Listato 25: Test sul numero di poligoni supportati

5.2.2 *Lego Editor*

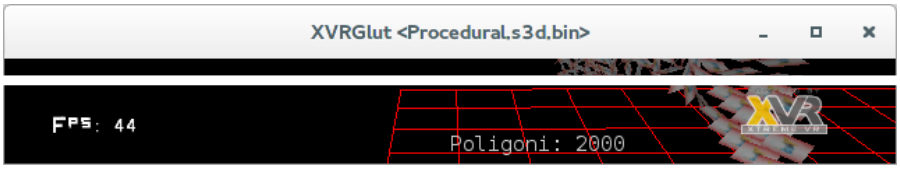
Per assicurarmi che il programma funzionasse a dovere, man mano che apportavo modifiche al codice, ho usato come test di riferimento l'editor di mattoncini Lego®. Si tratta di una serie di script S3D che sviluppai in occasione di un esame, rivelatisi ottimi come punto di riferimento:

- usano sia mesh procedurali che modelli caricati da file AAM
- includono materiali generati sia da texture che computazionalmente
- permettono di fare rapidamente prove con molti poligoni, in quanto su ogni maniglia la scritta LEGO è stata modellata in 3D (Figura 29)

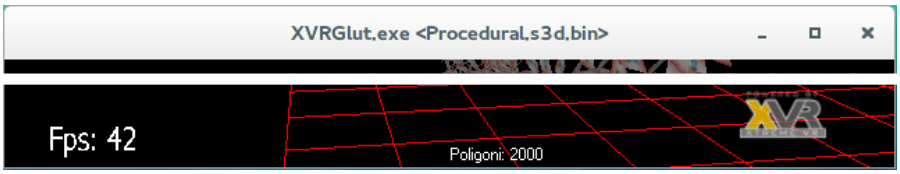


(a) Player originale

(b) Player Linux (Debug)



(c) Player Linux (Release)



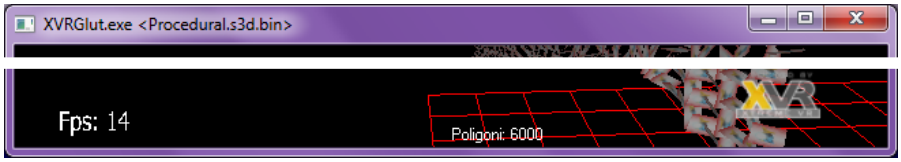
(d) Player emulato con Wine

Figura 26: Esecuzione con 2000 oggetti

- sfruttano sia il mouse, rotella compresa, che la tastiera
- utilizzano la rete
- fanno uso delle funzioni `OnEvent` (per la gestione dell'input) e `OnTimer` (per la connettività)
- mostrano testo tramite la console

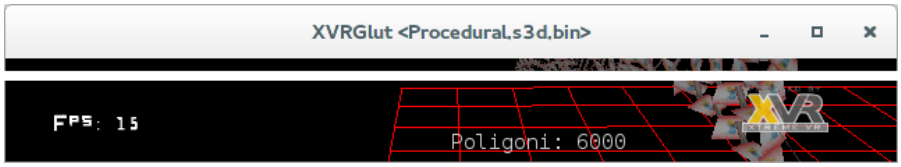
Oltre a consentire la verifica di una così vasta gamma di componenti in un colpo solo c'è anche una buona indipendenza dei vari elementi: se la rete non funziona il server continua ad essere utilizzabile come applicativo stand alone, se la console non è implementata si perde la chat ma si possono fare ancora le prove coi modelli 3D, se il mouse non risponde ci sono i comandi alternativi da tastiera.

La prova sul rendering non lascia alcun dubbio: come mostrato in Figura 30 il player Windows originale e la mia versione Linux sono perfettamente sovrapponibili. L'immagine proposta

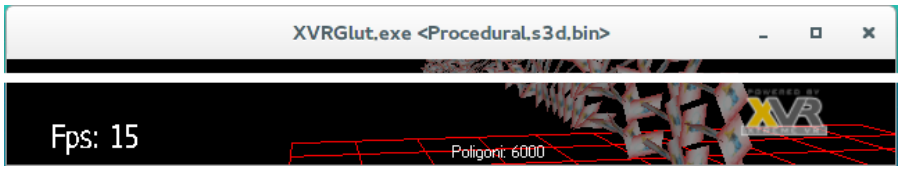


(a) Player originale

(b) Player Linux (Debug)



(c) Player Linux (Release)



(d) Player emulato con Wine

Figura 27: Esecuzione con 6000 oggetti

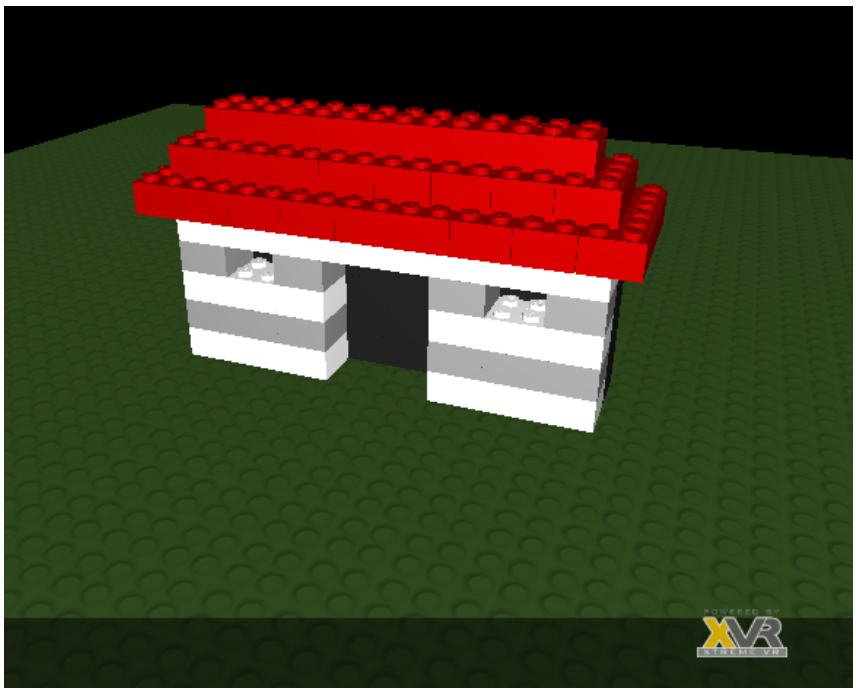


Figura 28: Una semplice costruzione ottenuta con l'editor

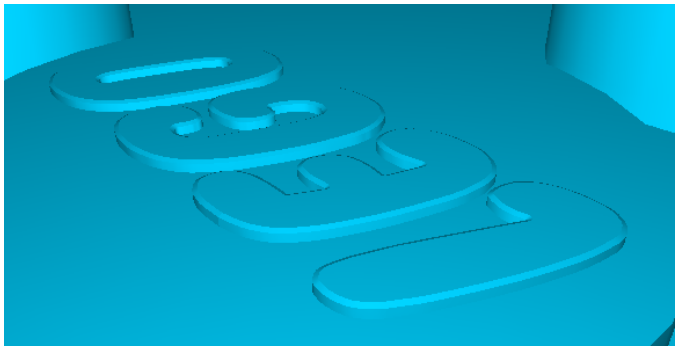


Figura 29: La scritta LEGO modellata in tutto il suo splendore

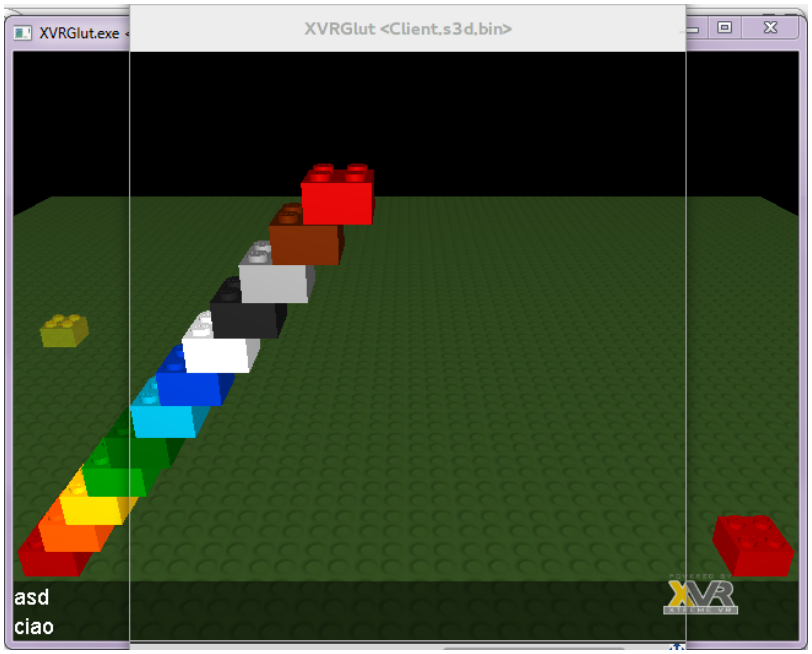


Figura 30: Confronto tra player originale e versione Linux

N° PEZZI	LINUX	WINE	WINDOWS
7	70	71	73
9	52	55	58
11	45	46	48
15	36	37	38
19	30	31	31
23	24	25	25

Tabella 4: Confronto sul framerate

non è un montaggio, bensì lo screenshot di due screenshot sovrapposti (la linea bianca che separa i due player è il bordo della finestra di GTHUMB).

Per fare osservazioni sul framerate ho posizionato una serie di mattoncini di lunghezza 15 (ovvero l'equivalente di un rettangolo di 15 mattoni quadrati da $2 * 2$ maniglie) ed ho guardato come le performance diminuiscono all'aumentare del numero di pezzi. Nella Tabella 4 sono riportati i risultati in frame al secondo relativi a varie combinazioni di numero di mattoncini e tipo di player, e quello che salta all'occhio è che viene completamente ribaltato il verdetto ottenuto nel caso precedente. In questo caso infatti è il player Windows a dominare, seguito da Wine che riesce sempre ad ottenere almeno 1 fps in più rispetto a Linux. La discrepanza tra i due risultati potrebbe essere attribuita a fattori ambientali, oppure dipende dal fatto che i due test erano sostanzialmente diversi: nel primo si disegnano texture caricate da file, nel secondo si usa `glMaterialfv` in maniera estensiva; nel primo gli oggetti vengono ruotati e spostati ad ogni frame, nel secondo restano sempre fermi.

Le prove di connettività segnano un punto decisamente a favore della mia versione del player. Non ci sono problemi durante l'esecuzione: client e server comunicano correttamente, sia sulla stessa macchina che quando eseguiti su macchine distinte, senza problemi di integrazione tra Windows, Wine o Linux. Le magagne nascono alla chiusura del client, come si può constatare dalla Tabella 5: nella versione originale il server dà errore su Windows e si blocca con Wine, mentre il nuovo player non ha alcun problema, il server continua a funzionare regolarmente.

SERVER	CLIENT	DISCONNESSIONE DEL CLIENT
Originale	Originale	RCOD
Wine	Wine	Freeze
Linux	Linux	Nessun problema
Wine	Originale	Freeze
Linux	Wine	Nessun errore
Originale	Linux	RCOD
Linux	Originale	Tutto OK
Originale	Wine	RCOD
Wine	Linux	Freeze

Tabella 5: Comportamento della rete

5.2.3 *Novità*

I pezzi di codice riscritti ex novo mi hanno dato molte soddisfazioni in fase di test. Il più importante è sicuramente il thread video, il cui risultato è senza appello: sia su Windows 7 che con Wine la riproduzione di filmati fallisce silenziosamente, ovvero nessun video viene riprodotto e non viene nemmeno fornito alcun feedback all'utente; su Linux invece viene mostrato il video con immagini ben definite e fluide (dei problemi audio ho già parlato nella Sezione 5.3).

I file audio su Windows si comportano esattamente come i video: niente sonoro e niente messaggi d'errore. Wine invece è differente: spesso fallisce nel creare l'oggetto `VRAWAV` (ma quando ci riesce riproduce correttamente i file `WAV`), non riproduce i file `MIDI` (mandando in output il messaggio d'errore: `fixme: mcimidi:MIDI_player NIY: MIDI port=0, track=1`), ma crasha miseramente quando si prova a caricare un file `Mp3`. Linux avvisa (con un messaggio sullo standard output) del fallimento quando si prova a caricare un `MIDI` o un `Mp3`, ma riproduce `FLAC` e `WAV` senza alcun problema.

La console è stata un passo avanti non indifferente. La Figura 31 evidenzia in maniera netta la differenza tra la vecchia versione e la nuova: il player attuale esegue il dithering con lo sfondo, quindi i glifi sono più stondati e gradevoli anche a

(a) Windows/Wine (b) Linux

Figura 31: Esempi di output tramite console

grandi dimensioni.

5.2.4 *Retrocompatibilità*

Per verificare la correttezza dell'implementazione ho provato ad eseguire alcuni script abbastanza pesanti relativi a vecchi progetti e dimostrazioni. La prima demo testata è stata una ricostruzione virtuale della città di Livorno. Si tratta di 386 Mb di materiale, di cui 133 sono modelli tridimensionali, il resto quasi totalmente texture ad alta definizione. La Figura 32 mostra il risultato, e l'esecuzione procede senza intoppi fino a quando non viene richiamata una funzione presente in versioni dell'engine più recenti di quella su cui ho lavorato io.

La seconda demo permette di muoversi nella ricostruzione di un complesso industriale. Sono due mesh da 120 e 126 Mb, e sebbene i tempi di caricamento siano considerevoli, una volta avviata la simulazione si può navigare nell'ambiente in maniera fluida. Alcuni dettagli sono mostrati nella Figura 33.

Una terza prova effettuata è stata l'esecuzione di una demo relativa all'uso degli shader. Vengono caricati ben 14 diversi shader per un totale di 1034 righe di codice GLSL, divisi tra *vertex* e *fragment* shader come riepilogato nella Tabella 6. Ciò che viene mostrato dallo script è una stanza quadrata, con al centro una grossa sfera luminosa fissa attorno alla quale ruota una seconda sfera luminosa più piccola. Sulle pareti sono disposti alcuni gargoyle fatti di materiali differenti: il modello 3D usato è sempre lo stesso, ma ognuno viene renderizzato con un differente shader. La scena non è interattiva, la camera si muove automaticamente orbitando attorno alla sfera centrale. In questo caso i tempi di caricamento sono molto contenuti, e i risultati dei diversi effetti sono mostrati nella Figura 34.

FILE	VERTEX	FRAGMENT	TOTALE
bumping	62	68	130
bump	61	39	100
envmap	27	62	89
erosion	18	43	61
flame	33	56	89
fur	15	15	30
hatch	63	31	94
lattice	41	32	73
metal	17	25	42
multitex	11	20	31
renderman	25	80	105
toon1	18	34	52
toon	18	28	46
wood	25	67	92

Tabella 6: Righe di codice degli shader

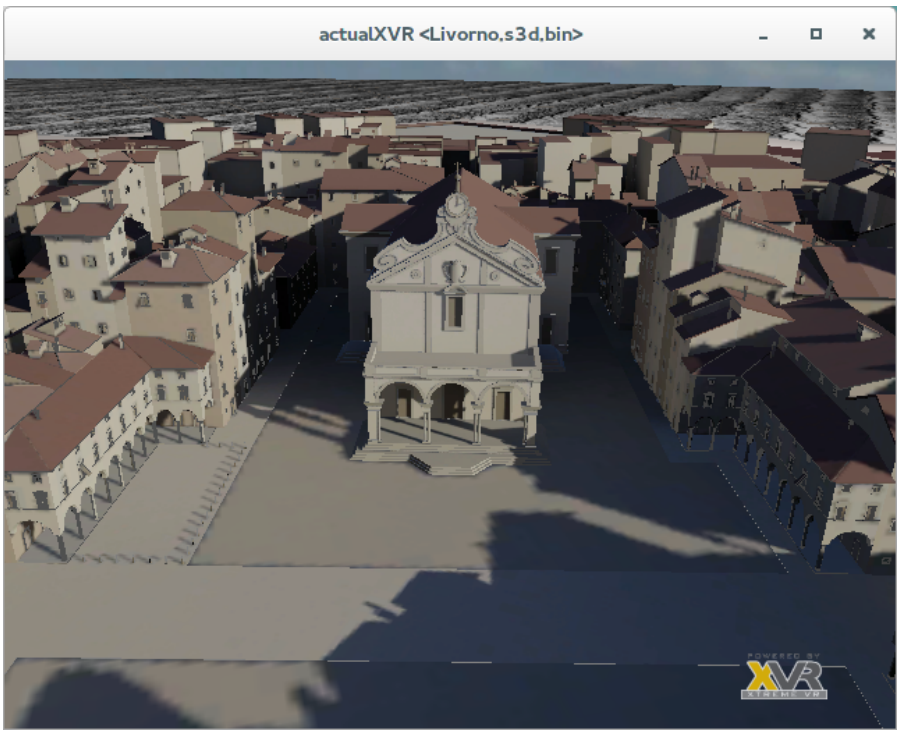


Figura 32: Livorno Virtuale

5.2.5 *Oculus VR*

Nella Sezione 4.2.5 ho parlato della capacità di utilizzo di periferiche di qualunque tipo in XVR grazie alla possibilità di integrazione di librerie esterne negli script. Come prova che non si tratta solo di una eventualità teorica bensì di un elemento concreto ho eseguito uno script che si interfaccia al device Oculus Rift[2]. L'Oculus è un Head Mounted Display (HMD) stereoscopico dotato di sensori per il tracciamento della posizione della testa a bassissima latenza, che permette dunque una sensazione di immersione totale per quanto riguarda la vista. Si tratta di una periferica molto vantaggiosa nell'interazione con ambienti virtuali ad elevato grado di complessità, pertanto nel laboratorio PERCRO è stata già studiata e utilizzata, rendendomi quindi possibile l'accesso con facilità sia all'hardware che a script S3D non banali per le prove. Il device è ufficialmente supportato anche su Linux, quindi sono relativamente bassi i rischi di incorrere in falsi negativi dovuti ad un'implementazione incompleta dei driver anziché a mancanze di XVR. Per poter interfacciare XVR con l'SDK dell'Oculus è necessario utilizzare una libreria che faccia da tramite tra i due, quella che viene ca-

2 http://www.oculusvr.com/rift/

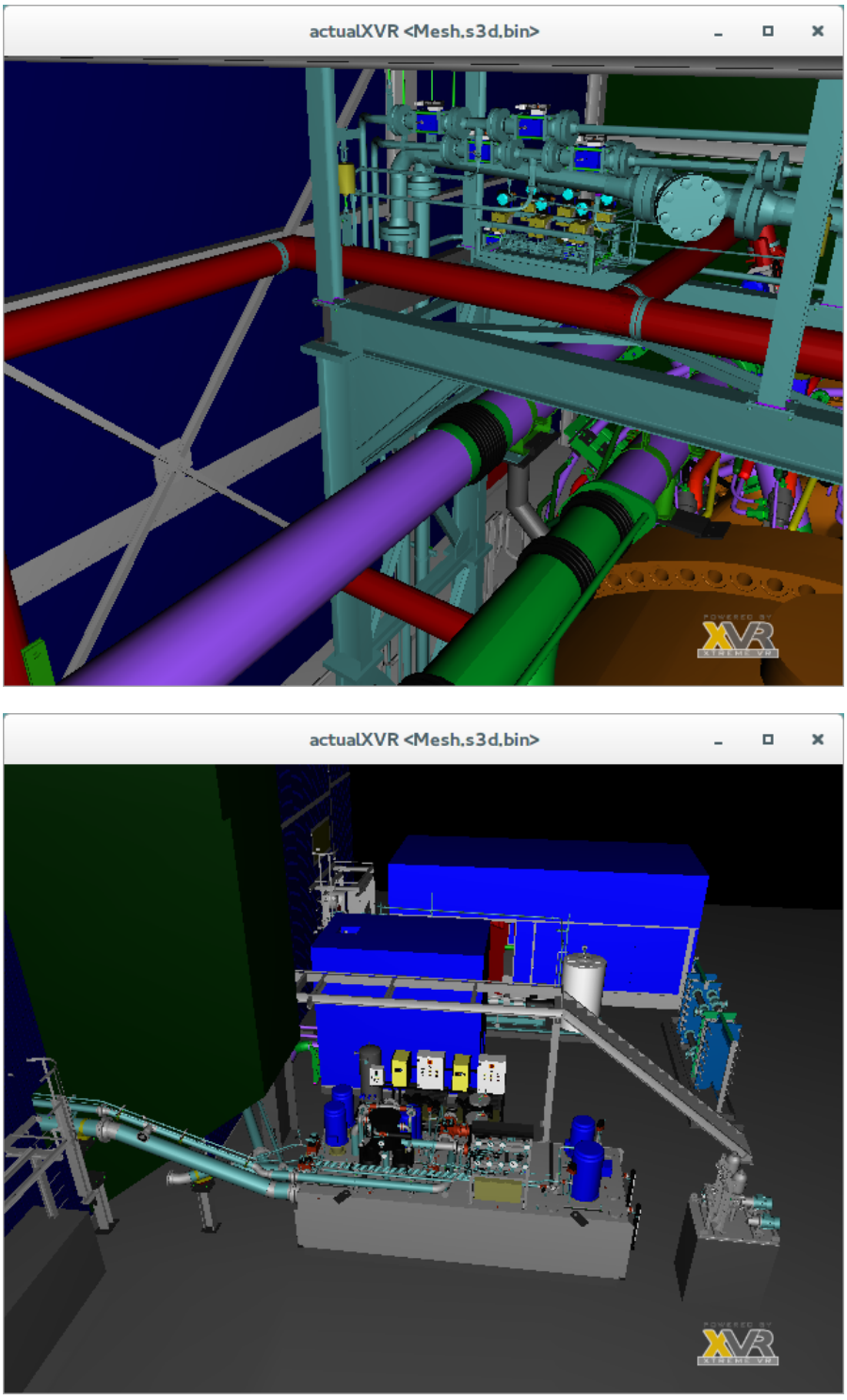


Figura 33: Ambiente tridimensionale vasto e dettagliato

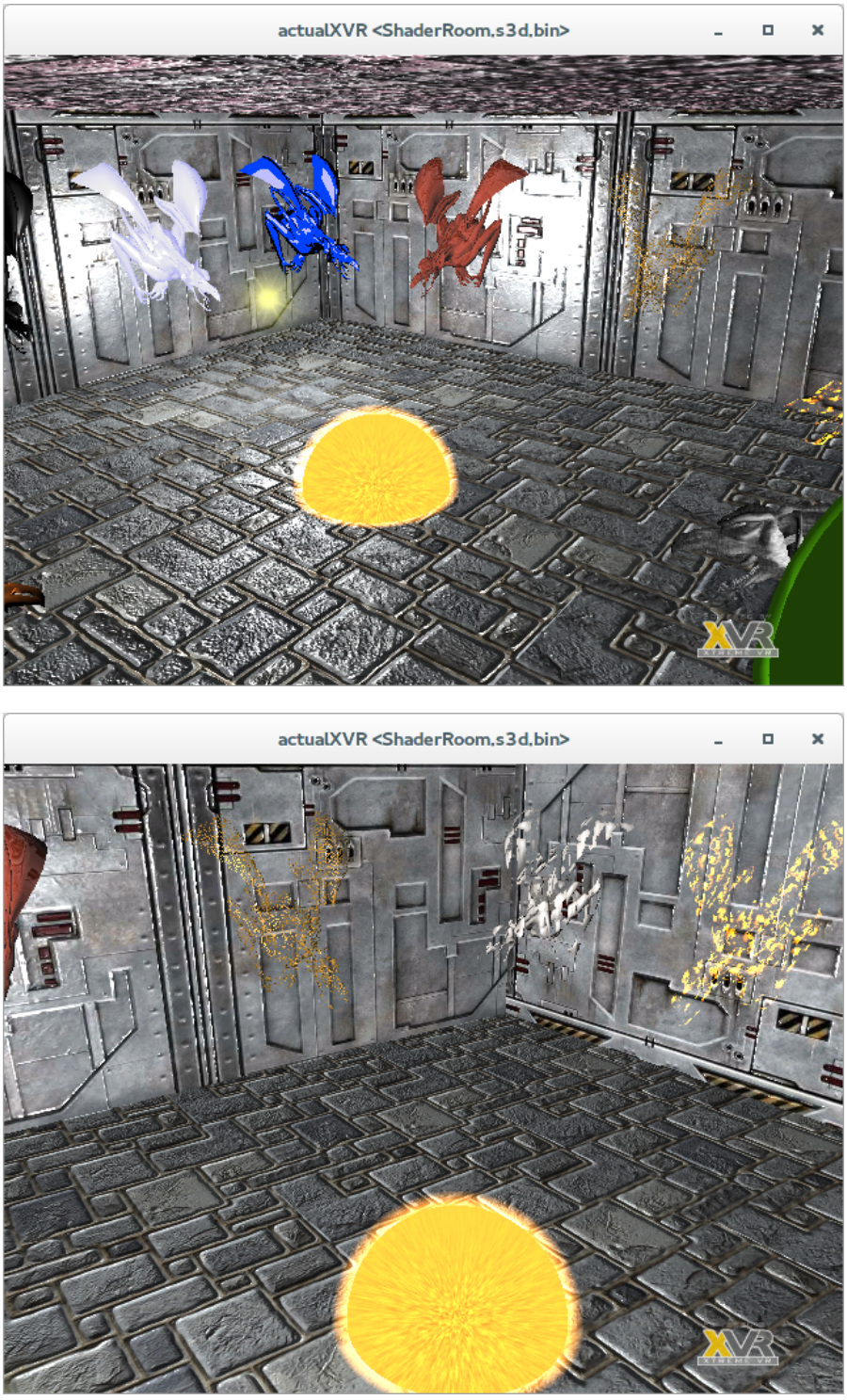


Figura 34: Esempi di uso di shader GLSL

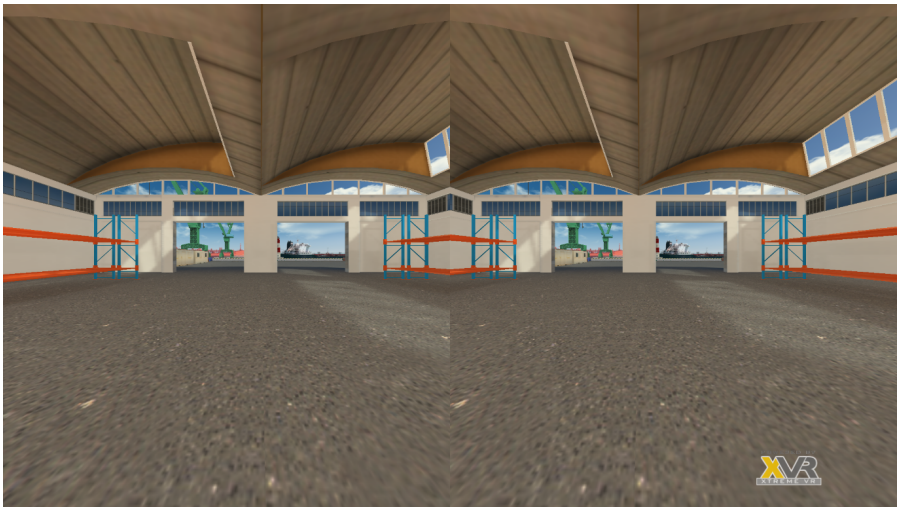
ricata dallo script; tale libreria per Windows è già stata scritta, pertanto è stato sufficiente ricompilarla con GCC per poterla utilizzare sotto Linux, perché naturalmente un file `.DLL` compilato con MSVC può essere linkato solo in applicazioni MSVC.

Quando l'Oculus viene collegato ad un computer funziona semplicemente come un secondo display, quindi basta fargli visualizzare un'applicazione a tutto schermo per iniziare a utilizzarlo. Una volta indossato l'Oculus l'utente si ritrova ad avere due monitor distinti, uno per occhio, ed è gestendoli opportunamente che si ottiene un'immagine stereoscopica. La discriminazione delle immagini da mandare sui due monitor avviene dividendo a metà lo schermo, perciò gli script per l'utilizzo della periferica renderizzano due scene distinte (una per ogni occhio) e le mostrano affiancate in un unico fotogramma, come mostrato in Figura 35a. Per riuscire a coprire l'intero campo visivo i monitor dell'Oculus vengono guardati attraverso delle lenti, ma ciò introduce una pesante deformazione all'immagine visualizzata dall'utente; è quindi necessario compensarla tramite un apposito shader, ma XVR supporta senza problemi GLSL. L'immagine finale da mostrare sull'Oculus è pertanto quella della Figura 35b.

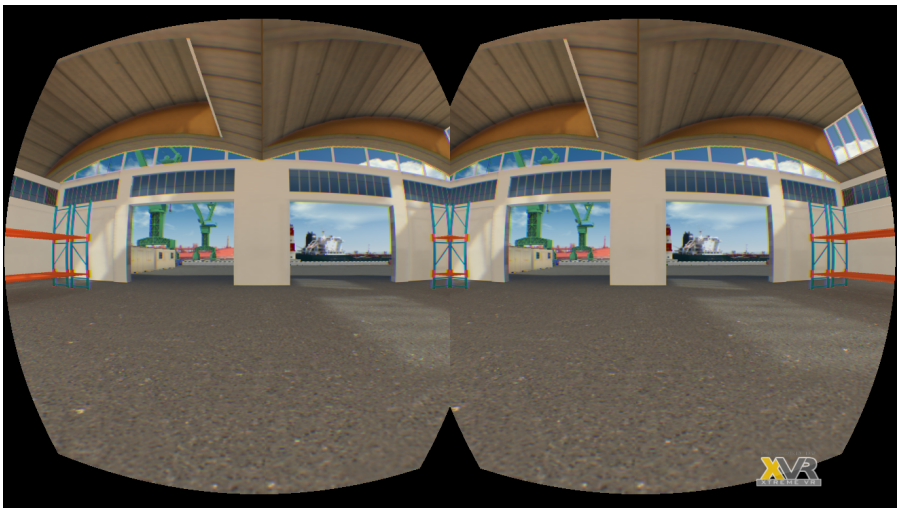
A parte la ricompilazione della libreria non è stata necessaria nessun'altra modifica per poter fare la prova, sia lo script binario già esistente che lo shader allegato hanno funzionato come previsto; nella Figura 36 lo si può vedere in azione, notando come ad un piccolo spostamento della testa corrisponda una piccola variazione della posizione della camera. Il display del computer è infatti clonato sull'Oculus, quindi ciò che si vede nella foto è esattamente ciò che viene visualizzato sul device.

5.3 SVILUPPI FUTURI

Nonostante il grande lavoro svolto fin'ora il margine di miglioramento è ancora parecchio ampio. Intanto il codice sorgente è quasi privo di documentazione, per qualsiasi modifica bisogna seguire le chiamate dei metodi e capire cosa fanno, oppure chiedere a qualcuno che ci ha lavorato sopra in precedenza; è prioritario scrivere dei commenti adeguati se si vuole poter mantenere il codice con facilità. Inoltre si dovrebbe lavorare sul build system per renderlo più omogeneo. Inizialmente ho soltanto provato a ricostruire un progetto simile a quello di partenza, poi ho pian piano aggiunto librerie esterne, usando i loro file CMake originali quando presenti. Ciò significa che



(a) Immagine divisa per l'Oculus ma non ancora deformata



(b) Lo stesso ambiente, dopo l'applicazione della deformazione

Figura 35: Script per Oculus, prima e dopo lo shading

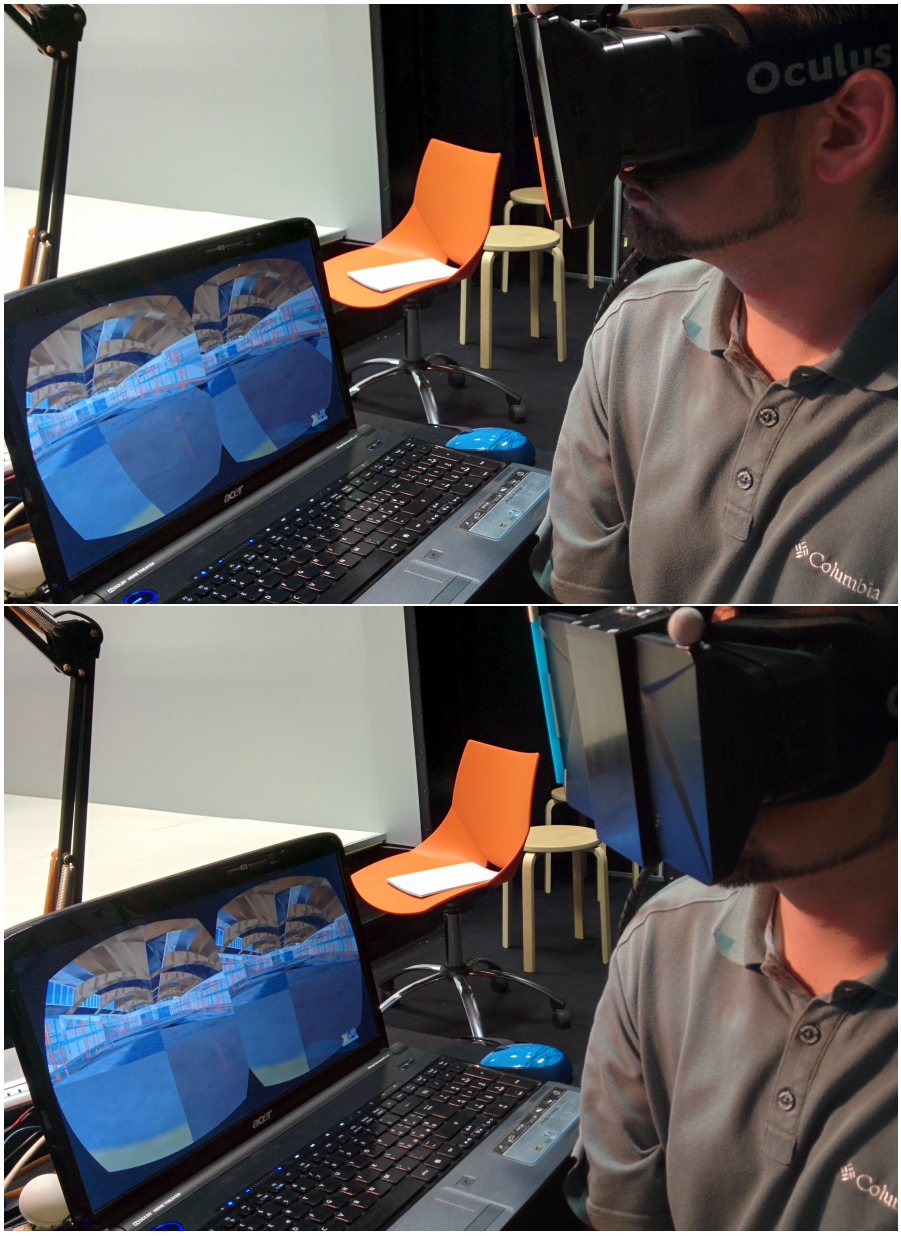


Figura 36: Interazione tramite Oculus Rift

alcuni controlli potrebbero essere ripetuti più volte, sprecando moltissimo tempo. Configurare il progetto in maniera uniforme significa leggersi attentamente tutti i file `CMakeLists.txt` forniti dalle varie librerie, capire di quali parametri hanno bisogno, spostare il possibile nella root directory; un lavoro lungo e tedioso con un impatto minimo sul programma finale, per questo considerato a priorità molto bassa.

La libreria video presenta ancora qualche criticità in fase di utilizzo. Non so se il timestamp calcolato da FFmpeg non è abbastanza preciso oppure se SDL_mixer inizia la riproduzione del suono con un ritardo non insignificante, fatto sta che l'audio dei filmati viene riprodotto con scatti e artefatti ben udibili, talvolta distorcendo completamente la sorgente originale. Provando a riempire un unico buffer con molti pacchetti consecutivi prima di avviare la riproduzione si ottiene un netto miglioramento, ma il passaggio tra due buffer distinti continua ad essere percepibile.

Rimangono alcuni elementi secondari da reintegrare nella versione multipiattaforma di XVR. Sono spariti i menu, perché la gestione consentita da GLUT è molto rudimentale, e sono talmente poco usati che non valgono il tempo necessario ad essere riscritti. È stato tolto anche il file manager, ma potrebbe essere utile avvalersi di un framework tipo Qt per implementare sia la navigazione nel filesystem che i menu personalizzabili. Il web browser integrabile non c'è più, perché richiamava Internet Explorer per funzionare; renderlo portabile significa passare a Gecko o WebKit (quest'ultima opzione verrebbe facilitata dall'ipotetica inclusione di Qt appena paventata).

L'audio ha bisogno di un po' di revisioni. L'estensione ambientale EAX è ormai deprecata, dovrebbe essere sostituita con EFX; bisogna inoltre riscrivere il supporto per i file `MIDI`, magari con una libreria di sequencing multiplatform. I file `Mp3` per ora sono esclusi, ma per riabilitarli basta aggiungere un flag in fase di compilazione di SDL_mixer.

Il parsing di modelli Collada viene fatto tramite una libreria binaria esterna, quindi al momento funziona solamente su Windows. Trattandosi di un formato aperto non è difficile riscrivere il parser oppure trovarne uno cross-platform.

Ultimo, ma non per importanza, è certamente il porting su altre piattaforme oltre a Linux. Sicuramente bisogna aggiungere qualche riga nell'header di inclusione di OpenGL e probabilmente bisogneranno fare piccoli aggiustamenti qua e là, ma tutto il codice è stato scritto (o riscritto) nella maniera meno

platform-dependent possibile, quindi mi aspetto meno di mezza giornata di lavoro per ottenere una versione funzionante per Mac OSX. La compilazione per mobile invece richiede qualche lavoro di adattamento più incisivo. In primo luogo bisogna tener presente che il supporto per il multitasking negli attuali dispositivi è solamente parziale, perciò è fondamentale gestire la sospensione e la ripresa dell'esecuzione in risposta alle chiamate del sistema operativo. Dopodiché è necessario prevedere un meccanismo per attivare (e nascondere) la tastiera virtuale, così che gli script esistenti continuino a funzionare senza modifiche. Si devono inoltre uniformare i nuovi tipi di input generati dagli smartphone alle vecchie modalità di trasmissione: è vero che posso sempre gestire tutto nella `OnEvent`, ma se per joystick e tastiera esistono strutture dati per il mantenimento dello stato allora mi aspetto che anche per il monitor multitouch ci sia qualcosa di analogo. Infine i dispositivi portatili usano OpenGL ES, un sottoinsieme di OpenGL, perciò una versione mobile di XVR richiede non solo che si verifichi la compatibilità delle chiamate grafiche effettuate, ma anche che al caricamento degli shader GLSL vengano fatte le opportune verifiche.

Parte I

APPENDICE

Quando ho iniziato questo percorso conoscevo molto bene il linguaggio JAVA e avevo avuto occasione di approfondire OBJECTIVE C, ma del C++ avevo solo un'infarinatura superficiale, nonché un'infondata convinzione che si trattasse di un linguaggio ormai datato; avevo deciso di impararlo comunque solo perché mi pareva utile in ambito lavorativo. Cominciando a studiarlo per bene però mi si è aperto un nuovo mondo: ho scoperto un linguaggio ancora fresco e vitale, con funzioni di alto livello ma in grado di eseguire operazioni di bassissimo livello se necessario, con una sintassi sufficientemente pulita ed elegante, imperativo, orientato agli oggetti ma anche utilizzabile con uno stile funzionale [9] a seconda delle proprie esigenze. L'aver imparato il C++ è stata decisamente la più grande conquista ottenuta con questo lavoro.

BIBLIOGRAFIA

[1] The BSD 2-Clause License. http://opensource.org/licenses/bsd-license.php, 1998. Last retrieved 2014-04-23. (Citato a pagina 46.)

[2] GNU Lesser General Public License, version 2.1. http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html, February 1999. Last retrieved 2014-04-23. (Citato a pagina 44.)

[3] Jonathan Atkins. SDL_mixer. http://www.jonatkins.org/SDL_mixer, November 2009. Last retrieved 2014-04-22. (Citato a pagina 42.)

[4] Beman G. Dawes. Proposal for a C++ Library Repository Web Site. http://www.boost.org/users/proposal.pdf, May 1998. (Citato a pagina 40.)

[5] Stephen Dranger. An FFmpeg and SDL Tutorial or How to Write a Video Player in Less Than 1000 Lines. http://dranger.com/ffmpeg/ffmpeg.html, May 2007. Last retrieved 2014-05-07. (Citato a pagina 75.)

[6] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012. URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372. (Citato a pagina 58.)

[7] Steffen Itterheim. *Learn iPhone and iPad cocos2d Game Development*. Apress, 1st edition, 2010. (Citato a pagina 15.)

[8] Jean loup Gailly and Mark Adler. Zlib License. http://www.gzip.org/zlib/zlib_license.html, October 2004. Last retrieved 2014-04-22. (Citato a pagina 41.)

[9] Brian McNamara and Yannis Smaragdakis. Functional Programming in C++. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 118–129, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: 10.1145/351240.351251. URL

http://doi.acm.org/10.1145/351240.351251. (Citato a pagina 105.)

[10] Diomidis Spinellis. Another Level of Indirection. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 17, pages 279–291. O'Reilly and Associates, Sebastopol, CA, 2007. ISBN 0-596-51004-7. URL http://www.dmst.aueb.gr/dds/pubs/inbook/beautiful_code/html/Spi07g.html. (Citato a pagina 2.)

[11] H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. C++ In-Depth Series. Pearson Education, 2004. ISBN 9780132654425. URL http://books.google.it/books?id=mmjVIC6WolgC. (Citato a pagina 69.)

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured at:

http://postcards.miede.de/

Stampato presso i tipi di Romolo Piacentini, siti in Pievepelago (MO). La Tipografia Galli si trova in Via Sagradino, 8 ed è raggiungibile telefonicamente (0536 71883), oppure via email (grafica@appenninostampa.it), o altrimenti accedendo al sito web http://www.appenninostampa.com.

Final Version as of June 19, 2014 (`classicthesis`).