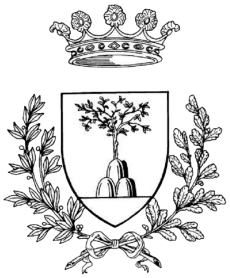
UNIVERSITÀ DEGLI STUDI DI FERRARA



Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Specialistica in Informatica

Relazione progetto di tirocinio per la Laurea Specialistica in Informatica

Simulazione visuale e fisica di tendaggi da interno

Candidato:

Michele Pio Mischitelli

Proponente:

Ing. Marco Barbieri

Tutore Accademico:

Dott. Stefano Marchetti

Tutore Aziendale:

Dott. Roberto Sgolastra

Anno Accademico 2009/10

Desidero dedicare questa tesi alla mia famiglia, per avermi permesso di portare a conclusione il mio percorso di studi. D'ora in avanti sarò io stesso fautore del mio destino, ma è grazie a loro che adesso ho i mezzi e le conoscenze per farcela. Grazie mille mà, papo e stefy :)

Indice

Capitolo 1

Prefazione

La tesi in esame descrive il progetto portato avanti negli uffici della Hypersoft di Ferrara dal sottoscritto, che riguarda la realizzazione di un applicativo in grado di simulare, sia visualmente, che fisicamente tendaggi da interno in maniera interattiva.

1.1 Descrizione del problema

L'applicativo si proponeva, nelle prime fasi di sviluppo, di soddisfare alcuni punti chiave:

- Semplicità di utilizzo: pochi pulsanti e controlli lato utente, lasciando al programma il compito di interpretare le intenzioni dell'utilizzatore.
- Rapidità in fase di creazione di una scena.
- Simulazione visiva dei tessuti di cui si compongono le tende.

- Simulazione fisica dei tendaggi, in modo da suscitare una reazione emotiva nell'utilizzatore, oltre che permettere l'implementazione di un maggior numero di strumenti per l'interazione con la scena.
- La scena deve essere interattiva: questo implica che tutte le operazioni devono avvenire in *tempo reale*.

Aldilà delle considerazioni sulla complessità relativa alla simulazione di tessuti dal punto di vista fisico o da quello visivo, la chiave del problema risiede proprio nell'ultimo punto.

Si pensi ad esempio ai film di fantascienza che, di anno in anno, si avvicinano sempre più alla realtà per quanto concerne la realizzazione di modelli in computer grafica, a volte risultando quasi indistinguibili ad una primo, sommario sguardo. Ma in questo caso, per ottenere un singolo fotogramma della pellicola (che normalmente conta 24 fotogrammi per ogni secondo di filmato), si spendono decine di minuti o addirittura ore di computazione, pur avendo a disposizione render farm composte da centinaia di unità di elaborazione operanti in parallelo. Nel caso in cui ci si accorga che un oggetto sia stato posizionato in maniera errata o che i parametri di illuminazione di una particolare sorgente siano sbagliati, si dovrà provvedere nuovamente a ricalcolare l'intero spezzone di film in cui l'errore è presente. Se da un lato questo approccio garantisce un elevato realismo visivo, dall'altro è stato appena descritto essere molto dispendioso in termini di tempo. Questo tipo di computer grafica è detto *offline rendering*.

Dal momento che l'ultimo punto degli obiettivi del programma richiedeva espressamente la possibilità di interagire in tempo reale con la scena, l'approccio offline era da scartare. Alternativa molto in voga nell'ultimo decen-

nio è il calcolo tramite *GPU*[1], in cui si utilizzano algoritmi che approssimano quelli implementati nei software di offline rendering. L'approssimazione delle equazioni di rendering[2], unitamente alla potenza di calcolo di questi processori altamente paralleli, permette di creare scene dall'elevato impatto visivo in cui è possibile interagire in maniera interattiva con il mondo virtuale.

Parallelamente, anche i calcoli fisici necessari per simulare correttamente il comportamento di una tenda a seguito di una forza esterna richiedono tempi di elaborazione lunghi. Come nel caso precedente, è stato possibile raggiungere l'obiettivo della simulazione in tempo reale tramite accorgimenti algoritmici, semplificazioni del modello fisico e un maggior sfruttamento delle risorse hardware della macchina: *Multi-Threading* ed *Heterogeneous Computing*.

Come spesso accade in fase di sviluppo, i requisiti dell'applicazione difficilmente restano gli stessi. Con il passare del tempo, la maggior diffusione in ambito consumer di tecnologie di input alternative quali ad esempio i *monitor touchscreen*, ha aperto nuove possibilità e nuovi target di mercato che in un primo momento non erano stati presi in considerazione:

- Tramite le tecnologie touchscreen è possibile rendere una maggior idea di interattività con la scena, oltre a poter contare su un maggior impatto emotivo sul cliente.
- L'input touchscreen soffre, inevitabilmente, di problemi di precisione e rapidità con cui si effettuano le operazioni.

[1]Graphics Processing Units - Termine tecnico con cui ci si riferisce alle schede video per computer

[2]Per maggiori chiarimenti si rimanda al Cap.3

- L'introduzione di questo tipo di tecnologia all'interno del software cambia radicalmente il target di mercato: se dapprima ci si focalizzava maggiormente sulle necessità di un venditore di tende, adesso il target si sposta inevitabilmente nell'ambito consumer.

Come intuibile, alcuni dei nuovi requisiti si scontravano con i punti cardine dell'applicazione sino a quel momento sviluppata. Le strade possibili erano quindi due: integrare tutti i comandi e le interfacce in un'unica applicazione con possibilità di selezione della modalità di utilizzo (editing, review, showcase) oppure sviluppare un nuovo front-end grafico che dialogasse in maniera opportuna con l'applicazione già realizzata. Di questa ed altre scelte se ne parlerà esaustivamente nei successivi capitoli.

1.2 Cenni su programmi esistenti

Per introdursi nel mercato con un prodotto, si possono scegliere sostanzialmente due strade: far concorrenza a programmi già esistenti, proponendo le stesse caratteristiche *migliorate*, magari integrando anche nuove funzionalità, oppure identificare un nuovo segmento di mercato in cui non vi è ancora concorrenza.

Delle due, la seconda permette di organizzare meglio le idee e ottenere, se sfruttato bene, un vantaggio concorrenziale sulle altre realtà software che inevitabilmente entreranno in gioco non appena il prodotto verrà commercializzato.

La ricerca di questo nuovo segmento di mercato è avvenuta su due fronti: facendo sondaggi fra le maggiori industrie di tendaggi e, parallelamente, analizzando le proposte software attualmente in commercio.

Il quadro che ne saltò fuori era piuttosto complesso: da un lato, le aziende di tendaggi richiedevano programmi tuttofare, in cui fosse possibile realizzare un intero ambiente di un abitazione in cui poter cambiare tutto fin nei minimi dettagli: tende, bastoni, accessori quali fiocchetti o nastrini, divani.. Dall'altro invece le software house permettevano, sostanzialmente, di creare scene basandosi su un ampio catalogo di oggettistica pre-confezionata con scarso controllo su quello che è il tessuto vero e proprio utilizzato sui tendaggi. Altre tipologie di software permettevano invece di scegliere gli ambienti da una collezione predefinita in cui però era possibile cambiare il tessuto di alcuni elementi chiave della scena: tappeti, tendaggi ed alcuni accessori.

La prima tipologia di programmi, quindi, era molto più simile ad un CAD bidimensionale in cui organizzare gli ambienti come farebbe un architetto per interni; la seconda invece ricordava più una procedura guidata con un medio livello di controllo sui tessuti che caratterizzavano la scena. In entrambi i casi, tuttavia, la scena risultava statica, priva di spessore e soprattutto limitata da un catalogo di oggetti o scene preconfezionati in cui l'utilizzatore finale non aveva libertà di personalizzazione.

Da quanto appena detto si capisce l'importanza dell'ultima clausola delle specifiche iniziali: tutto in *tempo reale*.

1.3 Soluzioni implementate nell'applicazione sviluppata

Come detto in precedenza, il software è in fase di sviluppo dal Maggio 2008 e l'attuale versione commercialmente disponibile è la v2.1.

Molti degli obiettivi sono stati raggiunti, alcuni sono stati addirittura estesi, mentre altri sono ancora in fase embrionale. Di seguito si analizzeranno le principali scelte architetturali e implementative del prodotto sin qui sviluppato.

1.3.1 Architettura generale dell'applicazione

L'applicazione si divide concettualmente in due entità distinte ed indipendenti: l'interfaccia grafica dell'utente e l'applicazione vera e propria. Questa divisione si è resa necessaria per una serie di motivi, sia lato sviluppo che lato utente.

Per quanto riguarda le motivazioni di sviluppo bisognava garantire:

- Un interfaccia grafica facilmente aggiornabile per accomodare tutti i controlli che durante lo sviluppo dell'applicazione verranno continuamente aggiunti.
- Possibilità di definire altre interfacce grafiche in maniera rapida, in modo da riutilizzare l'applicazione sviluppata per altri scopi.
- Semplificazione di alcune meccaniche relative alle librerie grafiche che altrimenti risulterebbero complesse e prone ad errori di programmazione.

L'unica vera motivazione per l'utilizzatore finale, invece, riguardava le performance: un problema tutt'altro che banale, che da solo sarebbe bastato a motivare la scelta.

Scendendo nel dettaglio, la parte d'interfaccia (GUI[3]) si basa sulle tecnologie .NET della Microsoft ed è scritta in linguaggio C#, mentre l'appli-

[3]Graphical User Interface - Interfaccia grafica utente

cazione principale è realizzata come una serie di *DLL*[4] scritte in C++. Se da un lato il C# garantisce semplicità e ridotto TTM (Time-To-Market), dall'altro non è in grado di dialogare a basso livello con l'hardware sottostante, rendendo necessario un meccanismo di *Wrapping* (incapsulamento) di tutte le chiamate alla libreria grafica o fisica. Dato che ciascuna chiamata passante per il *Wrapper*[5] risulta più lenta della sua controparte nativa, si cerca di ridurre il numero di tali chiamate. Si è quindi provveduto ad implementare l'applicazione principale in C++ in modo da dialogare direttamente con l'hardware e garantire ottime performance, mentre per una maggiore semplicità di sviluppo si è scelto di implementare la GUI in linguaggio C#, unendo di fatto i vantaggi di entrambi gli approcci e riducendo al minimo gli svantaggi.

Come accennato in precedenza, la parte di programma scritta in C++ si compone di una serie di DLL, ciascuna rappresentante un blocco logico separato:

- Controller: rappresenta la classe principale dell'applicazione: gestisce gli eventi generati dall'interfaccia utente, istanzia le classi degli oggetti presenti nella scena, implementa il motore di rendering *TEngine* e quello fisico *nVidia*® *PhysX*® . È inoltre in grado di esportare i singoli oggetti presenti nella scena in modo da poterli ricaricare in un momento successivo.

- TEngine: nome interno con il quale si identifica il motore di rendering grafico. Utilizza le specifiche aperte OpenGL per la visualizzazione della scena e per dialogare con l'hardware grafico presente nel sistema.

[4]Dynamic-link Library - Librerie a collegamento dinamico

[5]Coppia di classi che gestiscono lo scambio di dati e l'invocazione di metodi fra le due entità software

Implementa i *GLSL Shaders*, piccoli programmi eseguiti in hardware dalla scheda grafica per ottenere effetti visivi avanzati, ed i *Framebuffer Objects*, una serie di specifiche che consentono di effettuare diverse operazioni fra cui quelle di rendering direttamente sulla memoria video, eliminando di fatto un collo di bottiglia che altrimenti si avrebbe nel trasferire costantemente i dati dalla memoria di sistema a quella video e viceversa.

- ClothPhysX: piccolo e leggero wrapper per alcune chiamate alla libreria esterna nVidia® PhysX® . Tale libreria consente di simulare in tempo reale fluidi, tessuti, deformazioni sui metalli, collisioni e molto altro. Per approfondimenti si rimanda al Cap.3

- SimFramework: framework che contiene la definizione degli oggetti renderizzabili, chiamati *RObjects*, oltre che alcune strutture e costanti riutilizzate in tutto il progetto.

Grazie alla separazione fra GUI e applicazione è stato possibile creare due interfacce utente che pur utilizzando entrambe la stessa applicazione di base, mettono però a disposizione del cliente due ambienti concettualmente distinti: uno chiamato TSim-X Editor o semplicemente TSim-X, l'altro TSim-X Touch o abbreviato TSim-XT.

L'Editor è un ambiente interattivo in cui si realizzano scene composte da tende, accessori e sfondo ed in cui è possibile modificare ogni singolo parametro. Alla fine del processo di creazione è possibile salvare la scena su file per poter essere successivamente ricaricata o importata in una scena già esistente.

TSim-X Touch, come invece suggerisce il nome, è stato pensato specificatamente per i centri commerciali o per i grossi negozi di arredamenti provvisti

di computer con monitor touchscreen, in cui l'utente finale non è un esperto di tendaggi né tantomeno ha interesse a spendere del tempo per creare una scena da zero: l'unica cosa che interessa a questo tipo di utilizzatore è avere un idea del tessuto in un ambiente qualsiasi in cui tuttavia è in grado di intervenire mediante monitor touchscreen per animare le tende presenti nella scena. Parallelamente la stessa immagine verrà proiettata su un secondo monitor, idealmente più grande del primo, a scopo pubblicitario.

Capitolo 2

Tecnologie utilizzate

In questo capitolo vengono analizzate le tecnologie utilizzate nello sviluppo del progetto. La maggior parte di tali tecnologie sono già state utilizzate durante il precedente tirocinio, tenuto nell'anno accademico 2007/08 presso il *Visual Computing Lab* del CNR di Pisa, oltre che durante la mia carriera Universitaria. Altre tecnologie hardware/software, invece, sono state apprese man mano se ne rendeva necessario l'utilizzo.

2.1 C++

Il *C++* è un linguaggio di programmazione orientato agli oggetti, con tipizzazione statica. Ideato da Bjarne Stroustrup nel 1983, esso è un'evoluzione del linguaggio *C*: rispetto a quest'ultimo, il C++ offre funzionalità quali le classi, le funzioni virtuali, l'overloading degli operatori, l'ereditarietà multipla, i template e la gestione delle eccezioni.

La grande ricchezza semantica del C++ lo rende un linguaggio estremamente espressivo e potente. Purtroppo questo linguaggio richiede molto tempo per venire appreso e padroneggiato completamente. Il C++ ha una libreria standard. Di particolare importanza in essa è la *Standard Template*

Library o *STL*, la parte che utilizza i template per implementare contenitori generici, come vettori, code, array associativi e altre funzionalità. Per ulteriori informazioni su questo linguaggio è possibile consultare [1, 2].

2.2 C#

Il *C#* (si pronuncia c sharp) è un linguaggio di programmazione object-oriented sviluppato da Microsoft all'interno dell'iniziativa .NET, e successivamente approvato come standard ECMA. La sintassi del C# prende spunto da quella del Delphi (hanno il medesimo autore, ovvero Anders Hejlsberg), del C++, da quella di Java e da Visual Basic per gli strumenti di programmazione visuale e per la sua semplicità (meno simbolismo rispetto a C++, meno elementi decorativi rispetto a Java).

C# è, in un certo senso, il linguaggio che meglio degli altri descrive le linee guida sulle quali ogni programma .NET gira; questo linguaggio è stato infatti creato da Microsoft specificatamente per la programmazione nel Framework .NET. I suoi tipi di dati primitivi hanno una corrispondenza univoca con i tipi .NET e molte delle sue astrazioni, come classi, interfacce, delegati ed eccezioni, sono particolarmente adatte a gestire il .NET framework.

2.3 Framework .NET

La suite di prodotti .NET è un progetto all'interno del quale Microsoft ha creato una piattaforma di sviluppo software, .NET, la quale è una versatile tecnologia di programmazione ad oggetti.

Microsoft ha sviluppato .NET come contrapposizione proprietaria al linguaggio Java (che è open source) e attribuisce un ruolo strategico al lancio di .NET come piattaforma di sviluppo per applicazioni desktop e server nel prossimo decennio per le architetture client/server, internet ed intranet. Ri-

spetto a Java, .Net è uno standard ISO riconosciuto e quindi non è possibile, da parte della casa madre, modificarne la sintassi (a meno di discostarsi dal proprio stesso standard).

La prima versione di .NET è stata rilasciata nel 2002. La sua caratteristica peculiare è di essere indipendente dalla versione operativa di Windows su cui è installata, e di includere molte funzionalità progettate espressamente per integrarsi in ambiente internet e garantire il massimo grado di sicurezza e integrità dei dati. Utilizza in modo esteso il concetto di modularità dei componenti software (*Component Oriented Programming*), proponendosi così come evoluzione dell'esistente modello COM (*Component Object Model*).

La *CLR* (Common Language Runtime) è un insieme di librerie che, insieme alla classe di librerie di base denominata *FCL* (Framework Class Library), è progettata per poter funzionare con qualsiasi sistema operativo. Il compilatore Just In Time esegue un codice assembly denominato CIL (Common Intermediate Language). È inoltre possibile:

- Accedere a componenti scritti in altri linguaggi.
- Quando il sistema operativo sottostante è Microsoft Windows, accedere ai suoi servizi e alle sue API.
- Accedere ai Servizi Web utilizzando il protocollo SOAP.

2.4 GLSL

L'*OpenGL Shading Language* (GLSL) costituisce un linguaggio di programmazione di alto livello con cui scrivere *Shader* che verranno eseguiti dalla *Graphics Processing Unit* al posto della pipeline fissa che caratterizza l'OpenGL standard. Con il termine OpenGL Shading Language si distingue un particolare linguaggio, che una volta compilato e mandato in esecuzione,

viene processato interamente dalla GPU invece che dalla CPU. Dal momento che esistono in OpenGL tre tipi fondamentali di unità programmabili, ci sono a loro volta true diverse tipologie di Shader: i *Vertex Shader*, i *Geometry Shader* ed i *Fragment Shader*.

L'*OpenGL Shading Language* nasce con l'intento di semplificare lo sviluppo di shaders. Prima del GLSL, infatti, gli Shaders venivano scritti direttamente in codice Assembly. Successivamente, l'*OpenGL Architecture Review Board* decise di creare un linguaggio ad alto livello, facile da programmare e che permettesse ai programmatori di effetti grafici di poter sfruttare efficacemente l'hardware: nacque così il GLSL, chiamato anche GLslang. Le caratteristiche principali di questo linguaggio sono:

- Il GLSL è un linguaggio procedurale di alto livello.
- Stessa sintassi e semantica, con alcune piccole eccezioni, per i vertex e fragment shader.
- Basato sulla sintassi del C/C++.
- Supporta in maniera nativa operazioni sui vettori e sulle matrici, in quanto spesso utilizzate nelle operazioni grafiche.
- Più rigoroso del C per quanto concerne i tipi di dati.
- Invece di usare meccanismi di Read/Write, gli input e gli output sono gestiti da qualificatori di tipo.
- Non ci sono particolari restrizioni sulla lunghezza degli Shader.

Per ulteriori informazioni si rimanda a [3].

2.5 GPU

La *Graphics Processing Unit* è un componente hardware dedicato che si trova negli attuali elaboratori sotto forma di una scheda di espansione dedicata o di un chip integrato sulla scheda madre. Le moderne GPU sono particolarmente adatte a compiere elaborazioni legate alla pipeline di rendering e, grazie alla loro natura altamente parallela, sono estremamente efficienti nell'esecuzione di una certa categoria di algoritmi come ad esempio l'elaborazione delle immagini.

Al di là dell'utilizzo nell'ambito dello sviluppo di videogiochi, le GPU stanno ricoprendo un ruolo sempre maggiore nell'ambito del calcolo scientifico, in cui la loro capacità di processare efficacemente una gran quantità (o *stream*) di dati *floating point*, le pone su un'altro livello prestazionale rispetto anche ai più potenti processori attualmente in commercio.

Da poco, infatti, i due maggiori produttori di GPU, *AMD* e *nVidia*, vista la crescente domanda da parte delle compagnie di ricerca, hanno addirittura creato linee di prodotti hardware/software interamente dedicati a questo tipo particolare di applicazioni. In questo caso si parla di soluzioni GPGPU, ossia General-Purpose GPU.

Nonostante la loro specializzazione a svolgere determinate operazioni, le GPU sino a poco fa erano estremamente limitate per una serie di motivi, fra cui l'impossibilità di eseguire correttamente operazioni sugli interi (fondamentali per l'implementazione di un algoritmo *RNG - Random Number Generator*) e la mancanza di una pipeline interna che garantisse operazioni di filtraggio e fusione (blending) tra valori a virgola mobile a singola precisione (*Floating Point a 32bit*). Con le ultime generazioni di GPU (*AMD R600* e *nVidia G80*) gran parte di questi problemi sono stati risolti anche se la mancanza di una pipeline a doppia precisione si continua a far sentire in particolari ambiti: tale limite verrà rimosso probabilmente con le future

versioni delle architetture grafiche.

Per ulteriori informazioni si rimanda a [4].

2.6 OpenGL

OpenGL è un'interfaccia software verso il sistema grafico hardware, sviluppata da Silicon Graphics Inc nel 1992. Tale libreria nasce dalla necessità di visualizzare immagini ad alta qualità che rappresentassero oggetti e ambienti realistici a partire da informazioni di tipo geometrico e ottico.

Questa interfaccia è composta da circa 150 comandi distinti il cui scopo è quello di specificare oggetti geometrici in due o tre dimensioni, definendone le coordinate spaziali e specificandone gli attributi quali le proprietà ottiche del materiale, creare una gerarchia di dipendenze tra di essi e impostare il punto di vista. Queste operazioni sono utilizzate per sviluppare applicazioni interattive che fanno uso di grafica tridimensionale.

OpenGL è concepita come un'interfaccia indipendente dall'hardware. Per raggiungere questa indipendenza, non vengono incluse alcune operazioni come quelle per l'utilizzo di interfacce grafiche a finestra, che sono specifiche del Sistema Operativo, o per il controllo dei dispositivi di input.

In maniera analoga, OpenGL non fornisce comandi ad alto livello per descrivere modelli di oggetti tridimensionali. I comandi forniti permettono di specificare oggetti dall'aspetto relativamente complesso tramite l'uso di un ristretto insieme di primitive geometriche quali punti, linee e poligoni.

Il modo in cui esse vengono visualizzate si basa sull'esistenza di un *framebuffer*, un'area di memoria nella quale è possibile memorizzare l'immagine prodotta. Il suo utilizzo si limita soltanto alla lettura e scrittura di questa porzione di memoria, non esiste alcun'altra periferica hardware con la quale si può interagire.

La seguente lista descrive sommariamente le operazioni grafiche principali che OpenGL compie per disegnare un'immagine sullo schermo:

- Costruzione di forme a partire da primitive geometriche, creando una descrizione matematica degli oggetti[1].
- Disposizione degli oggetti in uno spazio tridimensionale e selezione del punto di vista.
- Calcolo del colore di tutti gli oggetti. Il colore può essere esplicitamente assegnato dall'applicazione, determinato da condizioni di illuminazione specificate, ottenuto applicando una *texture*[2] sugli oggetti, oppure con una qualsiasi combinazione di queste tre azioni.
- Conversione della descrizione matematica degli oggetti e delle informazioni sul colore ad esse associate in pixel sullo schermo. Questo processo è chiamato *rasterizzazione*.

Nel corso di questi stadi, OpenGL può compiere altre operazioni, come l'eliminazione di parti di oggetti che sono nascoste da altri oggetti. La maggior parte delle implementazioni OpenGL seguono lo stesso ordine di operazioni, una serie di stadi di elaborazione chiamata *rendering pipeline*.

Tramite le trasformazioni è possibile traslare, ruotare e applicare operazioni di scalatura ad ogni oggetto tridimensionale. La chiave nell'utilizzo delle trasformazioni risiede non nel trasformare l'oggetto stesso ma nell'agire sul sistema di riferimento in cui questo è stato inizialmente specificato. In OpenGL e, più in generale, in un qualsiasi sistema grafico, esistono diversi tipi di trasformazioni che vengono applicate dal momento della specifica dei vertici alla visualizzazione su schermo:

[1] OpenGL considera punti, linee, poligoni, immagini e bitmap come primitive.

[2] Una texture, o tessitura, è un'immagine bidimensionale applicata alla superficie di un oggetto tridimensionale.

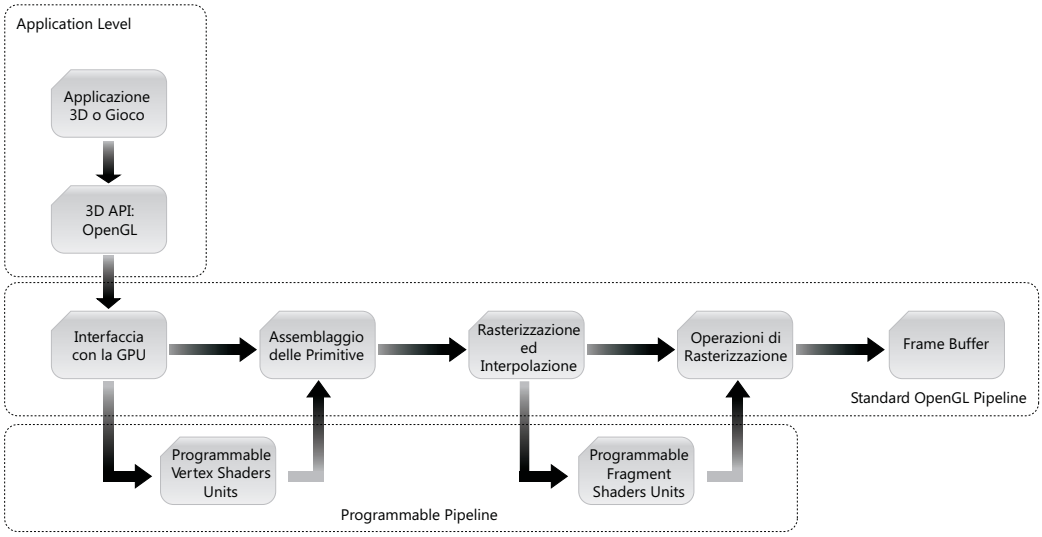


Figura 2.1: *Una rappresentazione schematica della pipeline fissa OpenGL*

- **Viewing**: specifica della posizione e orientamento dell'osservatore.
- **World**: posizionamento dell'oggetto nello spazio.
- **Projection**: definizione del volume di visuale.
- **Viewport**: trasformazione finale per l'output su schermo.

Nel corso degli anni, diverse nuove caratteristiche sono state aggiunte nell'OpenGL nelle versioni 1.1, 1.2, 1.3, 1.4 e 1.5 ma l'architettura base della pipeline è rimasta sostanzialmente invariata. Si usa il termine di *pipeline fissa* o *pipeline standard* perché ogni implementazione delle OpenGL è costretta ad avere la stessa funzionalità e produrre gli stessi risultati, in maniera consistente con le specifiche OpenGL, per una determinata serie di input. In questo caso sia le operazioni che l'ordine con cui vengono eseguite sono definite a priori dalle specifiche.

Il trend attuale, comunque, è quello di fornire la possibilitá di rimpiazzare la pipeline fissa con parti programmabili tramite semplici programmi (detti

Shader) che permettono di manipolare i singoli vertici ed i frammenti di una scena OpenGL. Sui vertici, in genere, si applicano tramite Shader trasformazioni e calcoli utili all'illuminazione. I frammenti sono delle strutture create dal rasterizzatore per contenere informazioni sui singoli pixel della scena: in un unico frammento ci sono tutte le informazioni di colore e di profonditá necessarie ad aggiornare una singola locazione nel frame buffer. Per tale motivo, le operazioni effettuate tramite Shader sui singoli frammenti hanno un campo di applicazione così vasto che risulta impossibile immaginare dei limiti entro cui definire tali operazioni.

Per ulteriori informazioni su OpenGL si rimanda a [5, 6].

2.7 RenderMonkey

RenderMonkey è un ambiente di sviluppo realizzato da *AMD* in collaborazione con *3Dlabs* che consente di sviluppare rapidamente Shader sia in linguaggio GLSL che in *HLSL* (utilizzato in DirectX). RenderMonkey è stato creato per venire incontro alle esigenze dei programmatori fornendo strumenti per la creazione di Shader in un ambiente di sviluppo flessibile e potente, ma al tempo stesso facile da utilizzare ed intuitivo. Con l'introduzione di linguaggi di programmazione di alto livello, come ad esempio *DirectX*, *OpenGL* ed *OpenGL ES*, la complessità degli Shader in tempo reale è cresciuta in maniera esponenziale. L'ambiente di sviluppo RenderMonkey non solo permette di prototipizzare e sviluppare facilmente Shader, ma fornisce inoltre un meccanismo integrato per la gestione di tutte le risorse ad essi associate quali modelli 3D, immagini e parametri.

RenderMonkey permette vari livelli di gestione degli Shader, schematizzati come segue.

2.7.1 Sviluppo di Shader

Editor di Shader

Per ogni linguaggio supportato da RenderMonkey, è presente un particolare editor con rispettiva evidenziazione della sintassi. Ciascun editor fornisce al programmatore una integrazione degli errori di compilazione, associando ciascun errore ad una linea ben precisa del codice sorgente.

Modifica dei parametri di uno Shader

Tutti i parametri associati ad uno Shader possono essere modificati tramite una interfaccia utente intuitiva. I colori, ad esempio, possono essere modificati scegliendoli da una palette a forma di ruota; valori scalari e vettoriali possono essere modificati tramite l'ausilio di slider; le texture e gli oggetti che compongono la scena e le relative texture possono essere osservate tramite un'anteprima e molto altro.

2.7.2 Debug degli Shader

Anteprima interattiva con segnalazione di errori

La finestra in cui viene renderizzata la scena risponde immediatamente a qualsiasi cambio di parametri durante il rendering degli effetti. Ciò significa che modificando un qualsiasi parametro, istantaneamente il programmatore avrà un riscontro visuale della modifica, rendendo semplice la messa a punto degli effetti. Nella finestra di rendering, inoltre, ogni errore relativo a risorse mancanti (texture, variabili, et cetera) viene prontamente segnalato tramite scritte in primo piano e rendendo la geometria della scena tutta di colore bianco.

Debug visivo di ciò che l'algoritmo sta effettivamente facendo

Fare il debug degli Shader è un processo molto complesso in quanto non ci sono riscontri numerici ma soltanto visuali. Nel caso di effetti complessi che richiedono più passate di rendering per essere completati, RenderMonkey consente di reindirizzare l'output di ciascuna passata ad una porzione della finestra di rendering: così facendo si è in grado di vedere in ogni istante i singoli output e quindi capire dove è situato l'errore.

2.7.3 Gestione delle risorse XML

RenderMonkey salva tutte le impostazioni e i dati necessari relativi allo Shader sviluppato in un file di lavoro *XML*. Una volta caricato tale file in RenderMonkey, sul lato sinistro dell'interfaccia compare una schematizzazione ad albero di ciascun nodo del file XML, ognuno identificante di una specifica risorsa. Data la natura portabile e chiara del formato XML, è possibile integrare questi file di lavoro in altri programmi, come ad esempio *MeshLab*, rendendo immediato il porting da questo ambiente di sviluppo alle applicazioni grafiche. In altre parole, rendendo quindi gli Shader prototipati immediatamente utilizzabili dalle applicazioni.

Per ulteriori informazioni si rimanda a [7].

2.8 RFID

RFID è una tecnologia per la identificazione automatica di oggetti, animali o persone[3] basata sulla capacità di memorizzare e accedere a distanza a tali dati usando dispositivi elettronici (chiamati TAG o transponder) che sono in grado di rispondere comunicando le informazioni in essi contenute quando interrogati. In un certo senso sono un sistema di lettura senza fili.

[3]AIDC - Automatic Identifying and Data Capture

Il sistema RFID si basa sulla lettura a distanza di informazioni contenute in un tag RFID usando dei lettori RFID: nel dell'applicazione TSim-XT, tale tecnologia è impiegata per il riconoscimento automatico del tessuto in modo da poterlo applicare alle tende presenti nella scena.

2.9 XML

L'*eXtensible Markup Language* come il nome stesso dice, è un linguaggio markup generale. Esso viene definito estensibile, perché permette agli utenti di definire i propri tags. Lo scopo primario dell'XML è facilitare la condivisione dei dati attraverso sistemi diversi, in particolar modo attraverso internet. L'XML è un sottoinsieme semplificato di *SGML*, lo *Standard Generalized Markup Language*, ed è progettato per essere relativamente leggibile dall'uomo.

Aggiungendo vincoli semantici, linguaggi applicativi possono essere implementati in XML; esempi ne sono i linguaggi *XHTML*, *RSS*, *MathML*, *GraphML*, *Scalable Vector Graphics (SVG)*, *MusicXML* e migliaia di altri.

Tale linguaggio è stato impiegato per il salvataggio su file delle scene realizzate: il vantaggio principale di avere un file in formato XML consiste nell'interoperabilità. È infatti possibile aprire facilmente il file con un'altra applicazione per leggerne, anche in parte, il contenuto. Un'altro vantaggio che attualmente non è stato ancora sfruttato è la relativa semplicità con cui sarà possibile memorizzare l'intera scena in un database mediante *LINQ* (Language Integrated Query), un componente del Framework .NET per l'input/output mediante query.

XML è uno standard libero ed aperto, ed è una raccomandazione del *W3C*, il *World Wide Web Consortium*.

Per ulteriori approfondimenti su questa tecnologia si veda [8].

Capitolo 3

Analisi del problema di simulazione

Una coppia di sposini, felicemente rientrata dal viaggio di nozze, entra finalmente nella loro nuova casa: l'odore di vernice fresca li accoglie all'ingresso, mentre il sole entra prepotentemente nella loro abitazione dalle finestre, colpendo il pavimento in parquet ed illuminando la stanza di un chiarore color arancio. Le pareti ancora spoglie attendono di essere rese vive da un bel quadro o una lampada a colonna. La forte luce proveniente dall'esterno invece fa rendere conto ai due giovani ragazzi di come vuote e trasparenti siano le finestre, senza nemmeno un velo a coprire l'intimità della casa.

Decidono così di recarsi presso un negozio di tendaggi. Le vetrine del negozio mettono in bella mostra i tessuti migliori con gli abbinamenti più azzeccati di bastoni ed accessori di vario genere: prima di entrare nel negozio, lei si sofferma un attimo a guardare una splendida composizione costituita da due tende sovrapposte, una coprente e l'altra trasparente che potrebbe star

bene in soggiorno.

Varcato l'ingresso, un distinto uomo sulla trentina dall'aria cordiale e simpatica li accoglie; dopo una breve chiacchierata mostra alla coppia i cataloghi, i tessuti, gli accessori. I due ragazzi, che all'inizio cercavano di immaginare sulle pareti della loro abitazione le composizioni proposte dal commesso, presto si sentono confusi dalla miriade di proposte ed iniziano a non riuscire più a focalizzare con chiarezza gli abbinamenti. Fra sè e sè, la ragazza pensa: "Tutto molto bello.. ma chissà come starebbe *davvero* quella composizione che avevo visto in vetrina, entrando nel negozio...".

Questa breve e simpatica introduzione introduce il problema della *simulazione*. La coppia di sposini, infatti, si domanda "Come starebbe *davvero* quella composizione". Il termine *davvero* rappresenta il nocciolo del problema della simulazione. Per chi non è esperto di grafica computerizzata potrebbe sembrare una semplice enfasi, ma pensandoci un attimo ci si rende subito conto di come quel *davvero* sottintenda qualcosa di concreto in realtà. Chi ha pronunciato la frase riportata poco sopra, infatti, si aspetterebbe di essere in grado in qualche modo di avere un'anteprima il più possibile fedele con la realtà, in modo da poter fare un acquisto certo, senza ripensamenti futuri.

Fin'ora soluzioni che assolvessero a questo desiderio erano tutte molto carenti, da vari punti di vista. Quello che più si avvicinava e tutt'ora rappresenta la scelta migliore, seppur in ambiti molto particolari, è sicuramente il rendering offline di una scena virtuale realizzata da esperti grafici. Nel prossimo capitolo si parlerà di questa e di altre tecniche maggiormente indicate per lo scopo ultimo del programma.

3.1 Simulazione visuale di oggetti

Così come avviene nell'industria cinematografica, in cui il regista si avvale di strumenti informatici per portare sul grande schermo le sue idee, la soluzione migliore al problema della simulazione di tende e ambienti di un appartamento rimane il *Rendering Offline*. Con questo termine ci si riferisce all'output prodotto da software denominati *Computer-Aided Design* (CAD): esempi ne sono Cinema4D, 3D Studio Max o Maya.

Basandosi su complessi modelli fisici di ottica, questi programmi implementano una serie di tecniche volte ad ottenere il massimo realismo possibile in forma digitale. La più importante di queste tecniche prende il nome di Raytracing[9], in cui l'immagine finale è realizzata *sparando* una serie di raggi (da qui il nome *ray*tracing) dalla sorgente luminosa e calcolando di conseguenza riflessione e assorbimento dei fotoni sulle superfici. Altra tecnica molto impiegata è la cosiddetta *Global Illumination* che calcola il contributo di illuminazione globale nella scena[10].

Per illuminazione globale si intendono quell'insieme di algoritmi in cui si tiene conto non solo della luce ricevuta direttamente da una sorgente di luce (illuminazione diretta), ma anche di quella riflessa, diffusa, o rifratta da altre superfici (illuminazione indiretta).

Le immagini renderizzate con l'uso di algoritmi di illuminazione globale, spesso appaiono più fotorealistiche rispetto a quelle che utilizzano solo l'illuminazione diretta. La loro computazione, però, è molto più lenta e costosa. Un approccio comune consiste nel computare l'illuminazione globale di una scena e memorizzare questa informazione in senso geometrico, ad esempio con la Radiosity[11]. I dati così salvati, possono essere usati per creare immagini da differenti punti di vista, generando così dei percorsi animati (walkthroughs) senza dover ricalcolare continuamente l'illuminazione.

3.1.1 Grafica in tempo reale e fotorealismo

Non sempre la scelta migliore, tuttavia, si dimostra anche la più efficace. Se è vero che da un lato la simulazione mediante rendering offline garantisce il più elevato realismo, dall'altro ne limita l'impiego proprio a causa degli elevati tempi di elaborazione richiesti: per una singola immagine statica, infatti, sono necessari su di un personal computer attuale, di fascia alta (CPU Quad Core con Hyperthreading, 6GB di Ram) circa 7 minuti.

Tuttavia, a seguito degli avanzamenti nel campo delle schede grafiche, la sfida si è spostata dalla ricerca di una migliore resa visiva alla possibilità di effettuare un numero sufficientemente elevato di elaborazioni, tale per cui sia possibile riprodurre a schermo una sequenza animata in modo fluido. Per raggiungere tale traguardo è necessario effettuare 30 o più elaborazioni nell'arco di un solo secondo; per rendere meglio l'idea, se con una tecnica raytracing tradizionale si è parlato di circa 7 minuti per calcolare l'intera scena, oggigiorno si punta ad effettuare il tutto in *meno di 30 millisecondi*.

Nonostante le GPU siano diventate particolarmente efficienti e prestanti nel campo dell'elaborazione floating point, è in ogni caso impensabile poter effettuare le stesse identiche operazioni del raytracing in soli 30 millisecondi. Bisogna, quindi, introdurre approssimazioni e semplificazioni che rendano l'intero processo calcolabile in tempo reale.

3.1.2 Una soluzione: OpenGL

Per poter accedere all'elevata potenza elaborazionale parallela delle moderne GPU si rende necessario utilizzare delle librerie che, funzionando da layer intermedio fra l'hardware comandato dai driver e l'applicazione vera e propria, espongono al programmatore una serie di interfacce e metodi con i quali interagire con la scheda grafica.

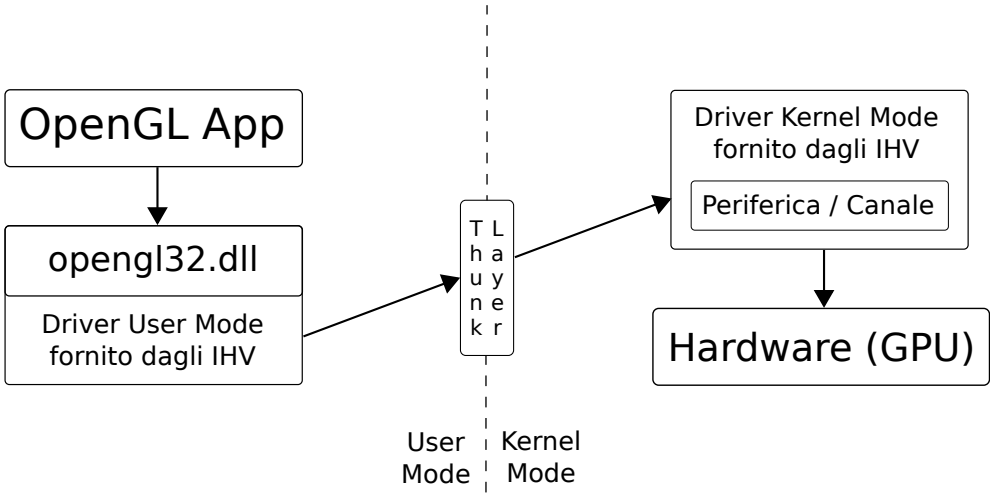


Figura 3.1: *Una rappresentazione schematica di come un applicazione accede all'hardware mediante le librerie OpenGL*

Uno schema rappresentativo delle librerie OpenGL durante la fase di esecuzione di un programma *Win32* è stato riportato in Fig 3.1. Da notare come la libreria a collegamento dinamico "opengl32.dll"sia un semplice layer posto al di sopra dei driver forniti dagli *Independent Hardware Vendor* (IHV), termine tecnico con il quale ci si riferisce ai produttori di hardware. Le chiamate fatte al driver grafico in user mode tramite OpenGL sono quindi instradate attraverso il *Thunk Layer* alla parte di driver residente in modalità Kernel. Nei sistemi operativi Microsoft da Windows Vista in poi, il Thunk Layer si preoccupa di gestire la comunicazione fra l'hardware grafico, comandato dal driver risiedente in modalità Kernel, e lo spazio utente in cui l'applicazione è in esecuzione.

3.1.3 La pipeline di rendering

Grazie allo schema riportato precedentemente, quindi, l'applicazione è in grado di dialogare con l'hardware grafico. A questo punto l'OpenGL potrà

iniziare a processare i comandi forniti dall'applicazione per disegnare su una locazione di memoria denominata *Framebuffer*.

Le operazioni effettuate durante la fase di rendering seguono, tradizionalmente, una logica ben definita e sostanzialmente statica:

- Costruzione di forme a partire da primitive geometriche, creando una descrizione matematica degli oggetti[1].
- Disposizione degli oggetti in uno spazio tridimensionale e selezione del punto di vista.
- Calcolo del colore di tutti gli oggetti. Il colore può essere esplicitamente assegnato dall'applicazione, determinato da condizioni di illuminazione specificate, ottenuto applicando una *texture*[2] sugli oggetti, oppure con una qualsiasi combinazione di queste tre azioni.
- Conversione della descrizione matematica degli oggetti e delle informazioni sul colore ad esse associate in pixel sullo schermo. Questo processo è chiamato *rasterizzazione*.

Nel corso di questi stadi, OpenGL può compiere altre operazioni, come l'eliminazione di parti di oggetti che sono nascoste da altri oggetti. La maggior parte delle implementazioni OpenGL seguono lo stesso ordine di operazioni, una serie di stadi di elaborazione chiamata *rendering pipeline*.

Si usa il termine di *pipeline fissa* o *pipeline standard* perché ogni implementazione delle OpenGL è costretta ad avere la stessa funzionalità e produrre gli stessi risultati, in maniera consistente con le specifiche OpenGL, per una determinata serie di input. In questo caso sia le operazioni che l'ordine con cui vengono eseguite sono definite a priori dalle specifiche.

[1] OpenGL considera punti, linee, poligoni, immagini e bitmap come primitive.

[2] Una texture, o tessitura, è un'immagine bidimensionale applicata alla superficie di un oggetto tridimensionale.

Il trend attuale, comunque, è quello di fornire la possibilità di rimpiazzare la pipeline fissa con parti programmabili tramite semplici programmi (detti *Shader*) che permettono di manipolare i singoli vertici ed i frammenti di una scena OpenGL.

3.1.4 OpenGL Shading Language - GLSL

L'evoluzione della pipeline da fissa a dinamica ha reso necessario introdurre un linguaggio ad alto livello con cui scrivere gli Shader menzionati poc'anzi.

Non appena venne introdotto il concetto di shader all'interno dell'OpenGL, la loro programmazione avveniva tramite linguaggi a basso livello, simili in molti aspetti all'Assembly delle architetture x86. Il problema principale di questo approccio, come naturalmente intuibile, risiedeva nella natura stessa del linguaggio di programmazione: l'Assembly è, per definizione, specifico per ogni singolo produttore di hardware, rendendo particolarmente difficile e lunga la creazione di codice compatibile con un numero elevato di configurazioni differenti.

L'idea, quindi, era quella di creare un linguaggio di programmazione ad alto livello, che si slegasse dalle implementazioni specifiche dei singoli produttori di hardware, ma nello stesso momento che fosse ugualmente flessibile e sufficientemente funzionale da poter divenire uno standard duraturo nel tempo.

La soluzione giunse nel Giugno del 2003, quando l'*OpenGL Architecture Review Board (ARB)* fornì un implementazione, tramite estensioni dell'OpenGL standard, del cosiddetto *OpenGL Shading Language (GLSL)*, successivamente divenendo parte integrante dello standard ed abbandonando la propria natura di estensione.

Il GLSL si basa sulla sintassi del linguaggio di programmazione ANSI C. Ad una prima osservazione, infatti, i programmi scritti in questi due linguag-

gi tendono ad essere molto simili. Ciò è stato intenzionale da parte dell'ARB, proprio per fare in modo che il linguaggio fosse di immediato utilizzo, dato che la quasi totalità dei programmatori di grafica interattiva utilizzano C o C++. Il punto di inizio, da cui parte l'esecuzione del programma stesso, si ha in corrispondenza di un `void main()` ed il corpo del programma è limitato da parentesi graffe. Le strutture condizionali `if-then-else`, i cicli tramite `for`, la dichiarazione delle variabili, *et cetera* sono molto simili al C.

Dal momento che l'OpenGL Shading Language mira, tramite una sintassi conosciuta, ad essere un linguaggio semplice ma nel contempo potente e flessibile, funzioni matematiche utili a codificare algoritmi di natura grafica sono presenti nel linguaggio base: funzioni trigonometriche, moltiplicazioni tra vettori e matrici, prodotti scalari e vettoriali, *et cetera*.

Per quanto riguarda i tipi di dato, nel mondo della grafica tridimensionale spesso si fa riferimento a vettori di tre o quattro elementi, siano essi booleani, interi o floating point. A tale scopo sono stati inseriti diversi tipi di dati specifici per vettori e le matrici. Per dati di tipo floating point, tali vettori sono identificati da `vec2` (due float), `vec3` (tre float) e `vec4` (quattro float); anteponendo una "i" o una "b" si specificano invece vettori rispettivamente di tipo intero e booleano.

Per quanto riguarda la gestione delle matrici si hanno a disposizione i tipi `mat2`, `mat3` e `mat4`. Esistono poi tipi di dati speciali detti *campionatori* che consento di effettuare l'operazione di lettura da textures (`sampler2D` per texture bidimensionali, `sampler3D` per quelle tridimensionali). Ciascun tipo di variabile, se dichiarato al di fuori di un blocco, può presentare un qualificatore di tipo, che ne specifica la semantica di gestione (ad esempio se rimane costante lungo l'invio di una primitiva geometrica, se deve essere interpolato dal rasterizzatore e così via).

Per ulteriori approfondimenti sul linguaggio GLSL si rimanda a [12].

3.1.5 Framebuffer Objects - FBO

I Framebuffer Objects (FBO) costituiscono un'estensione dell'OpenGL standard per effettuare il rendering in un area diversa (off-screen) rispetto al tipico framebuffer utilizzato per visualizzare le immagini su schermo; una delle aree sulle quali è possibile renderizzare con gli FBO può essere ad esempio una texture. Catturare le immagini che normalmente verrebbero visualizzate su schermo permette di implementare vari tipi di filtraggio delle immagini, oltre ad effetti di *post processing*[3]. In OpenGL, i Framebuffer Objects sono largamente utilizzati per via della loro elevata efficenza e facilità di utilizzo; essi hanno di fatto rimpiazzato altri metodi per il rendering off-screen, come ad esempio i p-buffer.

Tipicamente, parlando di framebuffer, in OpenGL ci si riferisce al *color buffer*, ovvero quel particolare buffer in cui si memorizzano i colori da inviare allo schermo. Il color buffer non è tuttavia l'unico tipo di buffer di cui la scheda grafica fa uso, ma ai fini della presente relazione non sono fondamentali e per ulteriori informazioni si rimanda alla lettura di [13].

In generale, quindi, con framebuffer ci si riferisce ad un *insieme* di buffer. Normalmente i buffer su cui è consentito il rendering sono definiti dal sistema; con i framebuffer objects, invece, è possibile definire ulteriori buffer, chiamati *Framebuffer Attachable Images*.

Una Framebuffer Attachable Image può essere costituito da un buffer su cui effettuare il rendering off-screen (chiamato *Renderbuffer*) o da una texture (Fig.3.2).

[3]Il termine viene utilizzato spesso nell'industria cinematografica con riferimento alla manipolazione di un immagine solo dopo che essa sia stata generata

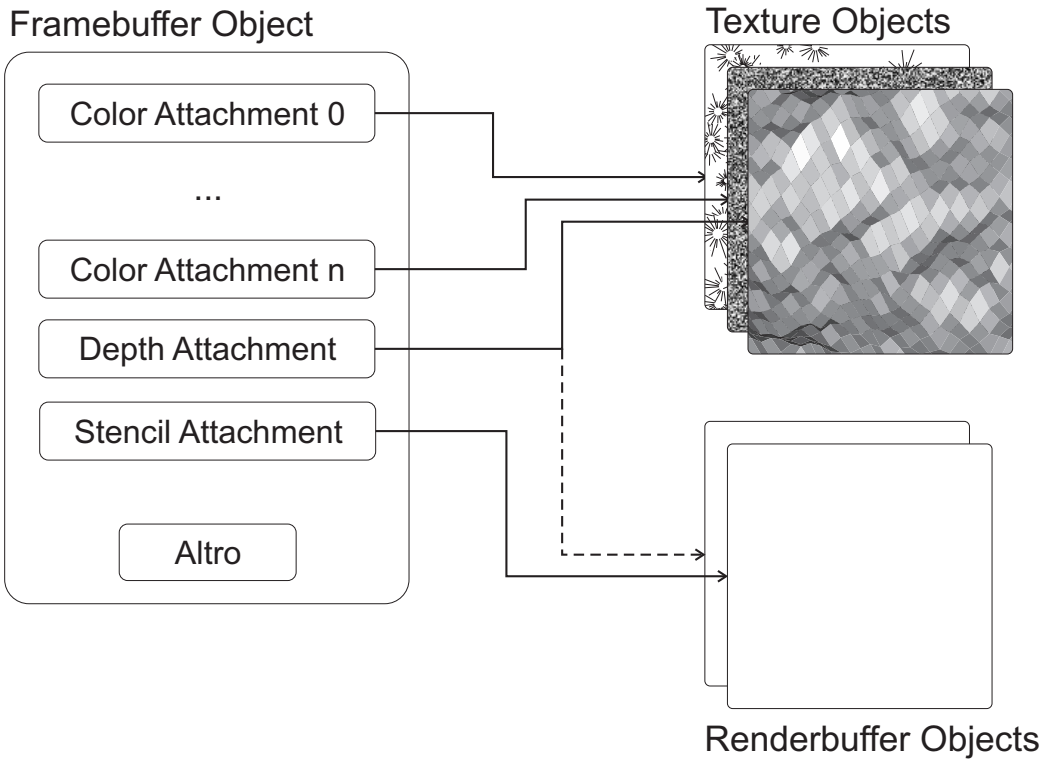


Figura 3.2: *Una rappresentazione schematica del concetto di Framebuffer Object*

Una particolarità dei Framebuffer Objects è che tutte le Framebuffer Attachable Images ad esso associate possono essere utilizzate esclusivamente in lettura o in scrittura. Bisogna pertanto usare più FBO per eseguire più operazioni di lettura/scrittura. Dal momento che alcuni effetti implementati in TSim-X, di cui si parlerà in maniera più approfondita nel prossimo paragrafo, richiedono letture e scritture sulla stessa texture si è reso necessario l'utilizzo alternato di più Framebuffer Objects. Fortunatamente il costo di un operazione di cambio FBO è estremamente contenuto, pertanto le prestazioni non risentono di questo aggràvio.

3.1.6 Le implementazioni: Cloth e Bloom Shaders

Sfruttando Shaders e FBO, talvolta in maniera congiunta, è stato possibile implementare due effetti grafici che hanno consentito di migliorare in maniera decisa la qualità finale del rendering della scena.

Il primo effetto analizzato è il *Cloth Shader*. Esso rappresenta un algoritmo di illuminazione efficace nella rappresentazione di tessuti generici. Prima di scendere nel dettaglio implementativo occorre fare una piccola digressione sui sistemi di illuminazione maggiormente diffusi nell'ambito della computer grafica.

Modello di illuminazione di Phong

Tradizionalmente nel campo della grafica tridimensionale sia in Real-Time che Raytraced le superfici geometriche illuminate da una sorgente mostrano un aspetto piuttosto plasticoso. Il motivo di questo comportamento è da ricercarsi nel modello di illuminazione utilizzato per calcolare il contributo della sorgente luminosa nella scena: esso prende il nome di *modello di riflessione di Phong* [14].

Il modello di riflessione di Phong può essere visto come una semplificazione della più generica *equazione di rendering* [15]; con il vantaggio di semplificare il calcolo del colore di un punto della superficie:

- È un modello di riflessione locale, ovvero non considera riflessioni di secondo ordine, a differenza di quanto fatto dal ray tracing e dalla radiosità. Al fine di compensare la perdita di parte della luce riflessa, un'ulteriore luce ambiente viene aggiunta alla scena.
- Divide la riflessione in tre fattori, riflessione speculare, riflessione diffusiva e riflessione d'ambiente.

Definiamo per prima cosa, per ogni sorgente luminosa, i componenti i_s e i_d, che sono rispettivamente le intensità (spesso misurate in RGB) dei componenti speculare e diffusivo della luce. Una costante i_a controlla la luce ambiente, e viene a volte calcolata come somma dei contributi forniti dalle varie sorgenti.

Se definiamo, per ogni materiale (che solitamente sono assegnati ai materiali con una relazione 1 a 1):

- k_s: costante di riflessione speculare, la percentuale di luce entrante che viene riflessa
- k_d: costante di riflessione diffusiva, la percentuale di luce entrante che viene diffusa (riflettenza Lambertiana)
- k_a: costante di riflessione d'ambiente, percentuale di riflessione della luce ambiente presente in ogni punto della scena
- α: costante di lucentezza del materiale in questione, che decide in quale modo la luce viene riflessa.

Definiamo inoltre *lights* come insieme di tutte le sorgenti di luce, N è la normale in questo punto, R è la direzione di un raggio perfettamente riflesso (rappresentato con un vettore), e V è la direzione verso chi guarda (ad esempio una camera virtuale).

Il valore di shading per ogni punto della superficie I_p viene calcolato usando l'equazione 3.1:

$$I_p = k_a i_a + \sum_{lights} (k_d(L \cdot N) i_d + k_s(R \cdot V)^{\alpha} i_s) \tag{3.1}$$

Quello di Phong è un modello di riflessione empirico, non basato sulla descrizione fisica dell'interazione di luce, ma piuttosto sull'osservazione informale. Phong osservò che le superfici altamente riflettenti creavano un

riflesso piccolo, mentre l'intensità di quelle opache diminuiva in maniera più graduata.

Il Cloth Shader

Cotone, lino e seta non sono caratterizzati da zone di forte illuminazione speculare ma, piuttosto, tendono a distribuire la luce ricevuta su tutta la superfice. Questo perché il processo di tessitura fa in modo che la superfice sia composta da dettagli su piccola e larga scala che influenzano il modo in cui la luce viene riflessa. Al fine di ottenere un risultato realistico, queste interazioni fra la luce ed i tessuti devono essere prese in considerazione.

In base al tipo di lavorazione, i tessuti possono essere divisi in due macro-categorie: lavorati a maglia ed intrecciati. Molti studi sono stati portati avanti nel corso degli anni sulla simulazione di tessuti lavorati a maglia [16] [17]. Tuttavia, i tendaggi da interno sono realizzati mediante intreccio, il che obbliga a trovare alternative che garantiscano:

- La rappresentazione dei fili intrecciati fra loro (trama)
- Una corretta modellazione delle interazioni fra luce e fili intrecciati

Il problema della rappresentazione di una trama è già stato affrontato in passato. Westin et al. [18] considerano un particolare tipo di trama ed ottengono un rendering realistico dei tessuti intrecciati. Nella loro pubblicazione considerano una semplice trama costituita da fili intrecciati in modo alterno e modellano l'illuminazione su scala millesimale facendo un integrazione sulla superfice. Yasuda et al. [19] descrivono invece un modello di visualizzazione che enfatizza l'interazione della luce con le singole fibre che costituiscono il tessuto.

Oltre alle pubblicazioni che affrontano direttamente il problema del rendering dei tessuti, esiste una classe di lavori che si basa sull'interazione della

luce con le superfici caratterizzate da micro-rilievi che può essere adattata al caso dei tessuti. Fra questi, Ashikhmin et al. [20] presenta una tecnica basata sui micro-rilievi e fornisce un modello che simula efficacemente le interazioni fra la luce e la superfice, in maniera molto simile a quanto visibile su velluto e raso. Anche le tecniche di illuminazione presentate da Heidrich et al. [21] e Sloan et al. [22] trovano diretta applicazione alla visualizzazione di tessuti dato che tengono conto dei micro-dettagli della superfice in esame.

Come evidenziato sin'ora, pertanto, per un efficiente rappresentazione dei tessuti bisogna tenere conto della distribuzione dei micro-rilievi. Nel modello di illuminazione qui analizzato e documentato in dettaglio in [23], per simulare i micro-rilievi ci si avvale della seguente funzione di distribuzione che risulta essere *fisicamente plausibile*:

$$q(\mathbf{h}) = e^{-tan^2\theta_h\left(\frac{cos^2\phi_h}{m_x^2}+\frac{sin^2\phi_h}{m_y^2}\right)} \tag{3.2}$$

$$D(\mathbf{h}) = \frac{1}{\pi m_x m_y cos^4\theta_h} q(\mathbf{h}) \tag{3.3}$$

La funzione 3.3 è la versione *anisotropica* della funzione di distribuzione di Beckmann utilizzata in [24, Cook and Torrance 1981]. L'anisotropia è la proprietà per la quale un determinato oggetto ha caratteristiche che dipendono dalla direzione lungo la quale esse sono considerate. Seta, raso e molti altri tipi di tessuto sono infatti caratterizzati da tale proprietà e bisogna tenerne conto in fase di computazione per un rendering realistico.

La funzione di distribuzione dei micro-rilievi *normalizzata* obbedisce alla seguente equazione:

$$\int_0^{2\pi}\int_0^{\pi/2} D(\mathbf{h}) cos\theta_h sin\theta_h d\theta_h d\phi_h = 1 \tag{3.4}$$

In pratica questa equazione evidenzia come D(h) sia una distribuzione di altezze, ovvero l'area totale proiettata di tutte le facce dei micro-rilievi deve essere uguale all'area base.

Per quanto concerne l'illuminazione vera e propria, la formula impiegata è la seguente:

$$f(\omega_i, \omega_o) = \frac{k_d(1 - F(\omega_o \cdot \mathbf{h}))}{\pi} + \frac{k_s F(\omega_o \cdot \mathbf{h}) D(\mathbf{h})}{4(\omega_o \cdot \mathbf{h})((\omega_i \cdot \mathbf{n})(\omega_o \cdot \mathbf{n}))^{\alpha}} \tag{3.5}$$

dove k_d è l'albedo diffusivo del materiale, k_s la riflettività speculare e $F(\omega_o \cdot \mathbf{h})$ è la funzione di Fresnel del materiale che può essere approssimata velocemente con la formula di Schlick[25]:

$$F(\omega_o \cdot \mathbf{h}) = f_0 + (1 - f_0)(1 - (\omega_o \cdot \mathbf{h}))^5 \tag{3.6}$$

I risultati ottenuti sono riportati di seguito. Nella Fig.3.3 l'equazione 3.5 è stata calcolata con i parametri k_d, k_s, f_0, m_x, m_y e α impostati secondo quanto suggerito in Matusik et al. [26] al fine di ottenere la rappresentazione del velluto. A dimostrazione della versatilità del modello utilizzato, impostando i parametri sopra elencati in maniera diversa (sempre seguendo il lavoro di Matusik et al.) è stato possibile ottenere in Fig.3.4 l'effetto del raso.

Il Bloom Shader

> In una fresca e limpida giornata primaverile, Paolo decide di uscire di casa per prendere una boccata d'aria. Non appena apre il portone che da nel cortile della sua villetta, Paolo quasi d'istinto solleva il braccio per coprirsi gli occhi dalla forte luce del sole che splende alto in cielo ed abbaglia per un attimo la sua vista.

Grazie a questo breve esempio si introduce un altro effetto implementato in TSim-X al fine di migliorare la qualità visiva complessiva della scena, il cosiddetto *Bloom Shader*.

Figura 3.3: *Simulazione visuale del velluto*

Figura 3.4: *Simulazione visuale del raso*

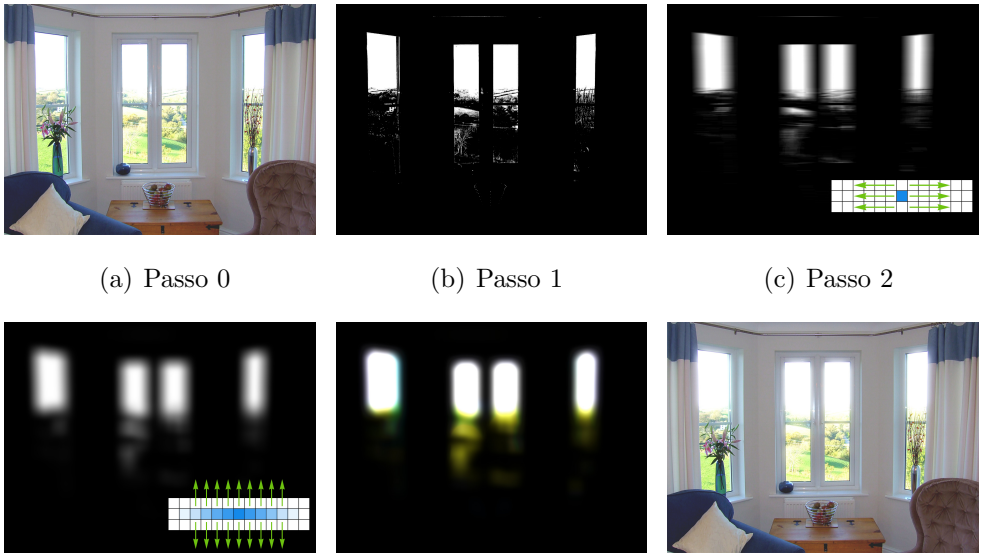
Normalmente, quando siamo colpiti da una forte luce, per un attimo veniamo abbagliati da essa ma non appena la pupilla si adatta alla nuova luminosità ambientale, ricominciamo a vedere normalmente. Lo stesso effetto, seppur in maniera opposta, lo si ha passando da zone illuminate a zone scure. In questo modo siamo in grado di vedere, più o meno bene, in un *range dinamico* di luminosità molto ampio.

Nella computer grafica si parla di *high dynamic range rendering* quando il calcolo dell'illuminazione della scena avviene usando un range dinamico più ampio del normale. In questo modo si preservano i dettagli che potrebbero altrimenti essere persi a causa del contrasto limitato in condizioni di forte o scarsa illuminazione.

Oltre a mettere in risalto dettagli che altrimenti andrebbero persi, tramite l'HDR rendering è possibile aggiungere effetti percettivi che aumentano la luminosità apparente. A titolo di esempio, una forte sorgente luminosa sullo sfondo di un inquadratura sembrerà avere un alone che in parte copre gli oggetti presenti in primo piano. Questo effetto prende il nome di *Light Blooming* o semplicemente Bloom e può essere utilizzato per creare l'illusione che zone chiare siano più luminose di quanto in realtà non lo siano. Ed è proprio questo l'effetto implementato in TSim-X.

Lo schema di funzionamento è piuttosto semplice ed è schematizzato nella Fig.3.5. E' bene precisare come questo effetto, a differenza del Cloth Shader analizzato poco prima, richieda più passate di rendering per essere completato. Questo implica che la scena viene calcolata più volte, ogni volta con un effetto diverso, per ottenere il risultato finale: tuttavia, computazionalmente parlando, il *multi-pass rendering* non è sempre così pesante come potrebbe sembrare, specie se la geometria della scena risulta particolarmente semplice, come in questo caso.

La motivazione a quanto detto è piuttosto intuitiva: se in un'unica passata di rendering le elaborazioni sui singoli pixel implicano calcoli onerosi (*acos*,



(a) Passo 0 (b) Passo 1 (c) Passo 2

(d) Passo 3 (e) Passo 4 (f) Passo 4

Figura 3.5: *Schema esemplificativo dell'algoritmo di Light Blooming:3.5(a) lo sfondo viene caricato sulla GPU; 3.5(b) viene convertito in bianco-nero e si escludono le zone non di luce tramite thresholding; 3.5(c) si applica il filtro gaussiano separabile orizzontalmente; 3.5(d) si applica il filtro gaussiano separabile verticalmente; 3.5(e) si aumenta il contrasto, si regola l'esposizione e si restituiscono i colori al bloom; 3.5(f) l'immagine finale*

asin, elevazioni a potenza), allora uno shader *single-pass* potrebbe risultare più pesante di uno multi-pass.

Tornando ad analizzare il bloom shader, nella prima passata le zone più luminose dello sfondo sono estrapolate mediante tecniche di elaborazione delle immagini direttamente sulla memoria della scheda video: una volta convertita l'immagine in bianco/nero, si applica un *threshold* in modo da separare le zone più chiare dal resto. Tale valore soglia è modificabile a piacimento in fase di compilazione del programma anche se in futuro potrà essere regolato manualmente o automaticamente dal software stesso, in base

alla quantità di luce presente nello sfondo. Attualmente, tuttavia, questo parametro è impostato al 90% : ciò vuol dire che, una volta convertita l'immagine in bianco e nero, saranno tenute in considerazione al fine del calcolo del bloom, solo le zone che presentano un livello del grigio del 10% o inferiore.

Nella passata di rendering successiva, si renderizza normalmente la scena, aggiungendo però al canale *alpha* l'immagine calcolata nel passo precedente. Normalmente nel canale alpha di un immagine si memorizzano le informazioni sulla trasparenza, basti pensare ad esempio alle immagini PNG trasparenti, ampiamente utilizzate sul web. Nel caso della semplice visualizzazione su schermo il canale alpha non è utilizzato, pertanto le informazioni calcolate nel primo passo sono state integrate direttamente nell'immagine da visualizzare: questa scelta permetterà nei passaggi successivi di risparmiare un accesso alla memoria dato che sarà sufficiente leggere una sola texture invece che due (una per il colore ed una per il bloom).

Nel terzo passo di rendering si riprende l'output del secondo passo e lo si rimpicciolisce, quindi si applica un filtro gaussiano per sfuocare l'immagine. Il rimpicciolimento dell'immagine è un operazione completamente gratuita, computazionalmente parlando, in quanto il rendering avviene su di un *framebuffer object* di dimensioni inferiori rispetto alla dimensione reale dello schermo. Per maggiori approfondimenti sui framebuffer object si rimanda alla sezione precedente di questo capitolo. Il rimpicciolimento dell'immagine permette da un lato di calcolare il filtro gaussiano su un numero inferiore di punti e dall'altro consente di ottenere *gratuitamente* un'ulteriore sfuocatura, seppur non di tipo gaussiano, nel momento in cui si chiede all'hardware grafico di ingrandire l'immagine. Questo è dovuto al modo in cui le GPU odierne sono realizzate: una delle operazioni che più di frequente i programmatori richiedono all'hardware grafico è l'interpolazione bilineare sulle texture. I

produttori di hardware hanno quindi ottimizzato la logica interna delle schede grafiche per far si che tali operazioni siano estremamente veloci. Chiedendo quindi alla GPU di ingrandire una texture utilizzando l'interpolazione bilineare sarà, computazionalmente parlando, molto simile a richiedere un interpolazione lineare. Questa peculiarità consente quindi di calcolare il filtro gaussiano solamente due volte in totale: una volta per le ascisse ed una per le ordinate, risultando pertanto leggero ed alla portata di qualunque hardware grafico, compreso quello dei portatili, tradizionalmente più lento della controparte desktop. Il filtro gaussiano implementato è di tipo separabile a sette punti. A differenza del filtro media, i pesi della maschera sono distribuiti in maniera significativa verso il centro e in maniera meno significativa verso la periferia. Al fine di ottimizzare il calcolo, i pesi della maschera sono stati pre-calcolati e salvati in due vettori, riducendo il numero totale di operazioni effettuate. In questo passo, quindi, il filtro applicato lavora orizzontalmente rispetto all'immagine, mentre nel quarto lavora verticalmente.

Nel quinto ed ultimo passo, infine, si regola l'esposizione ed il contrasto dell'effetto di bloom, in modo da non risultare eccessivo. Una comparazione della stessa scena con e senza l'effetto di bloom può essere osservata nella Fig.3.6.

3.2 Simulazione fisica di tessuti

Un'ottima simulazione visuale della scena tuttavia non basta. Per ottenere un risultato che sia *verosimile* alla realtà ci dev'essere anche un qualche tipo di interazione con la scena, per avere l'idea che quanto osservato non è una semplice immagine statica ma qualcosa di concreto. Per questo motivo è necessario simulare anche la fisica degli oggetti che compongono la scena.

Figura 3.6: *A sinistra un immagine standard; a destra la stessa immagine con l'effetto di Light Blooming*

La simulazione dei tessuti è caratterizzata da un forte parallelismo sui dati: una porzione di tessuto può infatti essere modellata come una rete bidimensionale costituita da particelle con la stessa dinamica. Ciò rende l'elaborazione dei calcoli tramite GPU una strada non solo percorribile, ma preferenziale.

Tuttavia, motivazioni di tempo e necessità di avere una robusta architettura con cui lavorare, hanno costretto ad utilizzare una libreria già compilata prodotta dalla nVidia® , che prende il nome di PhysX® .

Nonostante questa scelta, il problema della simulazione fisica verrà analizzato ugualmente. Prima di iniziare l'implementazione in TSim-X della libreria citata, infatti, erano stati svolti degli studi su come implementare

un motore fisico proprietario, avvalendosi anche dell'aiuto del centro *Math4Tech*. Inoltre, come accennato in precedenza, data la natura altamente parallela del problema, si è da subito iniziato a pensare ad un ottimizzazione che sfruttasse le risorse computazionali fornite dalle GPU attuali.

L'implementazione pensata era orientata più verso le prestazioni e il realismo visivo piuttosto che verso l'accuratezza fisica. Questo proprio in virtù del fatto che il tutto sarebbe dovuto essere facilmente calcolabile in tempo reale da qualsiasi computer, in particolare i portatili. A tal proposito tutta l'elaborazione sarà demandata alla scheda grafica che nel contempo sgrava il processore dai calcoli.

3.2.1 Il modello Spring-Mass

Una tenda, o più in generale un oggetto fatto di tessuto, può essere modellato come un insieme di particelle. Ogni particella è soggetta a:

- Forze esterne, come la gravità, il vento ed il trascinamento;
- Una serie di vincoli:
  - Mantenere la forma generica dell'oggetto (vincoli di elasticità)
  - Evitare la compenetrazione con l'ambiente circostante (vincoli di collisione)

L'equazione delle particelle in movimento risultante dall'applicazione delle forze esterne è calcolata usando l'integrazione di Verlet.

I vincoli esplicitati poco sopra creano un sistema di equazioni che legano fra loro le particelle nello spazio. Il sistema è risolto ad ogni step di simulazione per rilassamento; ciò avviene applicando i vincoli uno dopo l'altro per un certo numero di iterazioni.

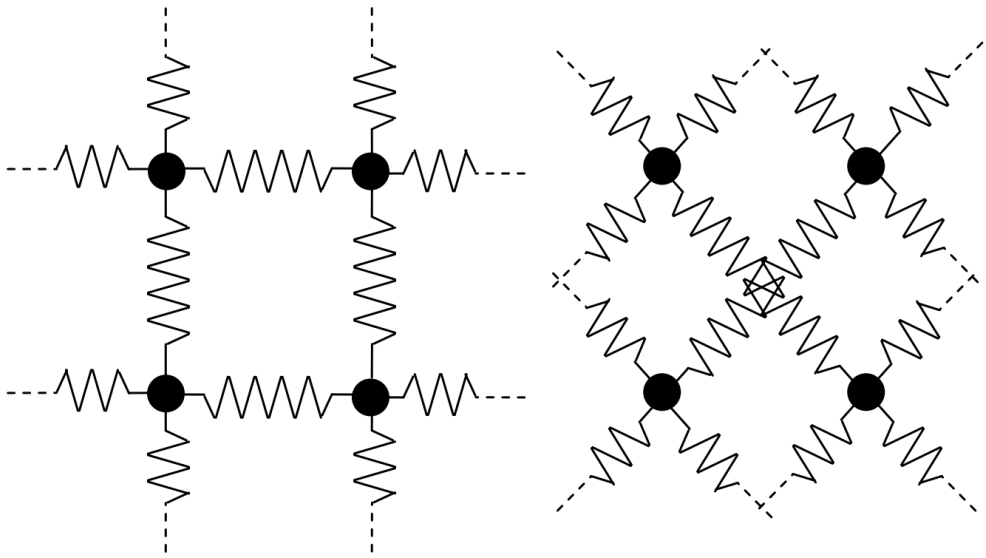


Figura 3.7: *Vincoli di elasticità fra particelle vicine*

I vincoli di elasticità fra particelle vicine possono essere categorizzati in due tipi, come illustrato in Fig.3.7.

Un vincolo di elasticità fra due particelle, **P** e **Q**, è definito come un vincolo sulla distanza fra **P** e **Q**:

$$\text{Dist}(P, Q) = \text{distanza a riposo}$$

ed è raggiunto allontanando o avvicinando tra loro **P** e **Q** come illustrato in Fig.3.8.

L'ambiente circostante è definito invece come un insieme di oggetti con i quali è possibile avere una collisione, caratterizzati da varie forme geometriche: piani, sfere, capsule, ellissoidi. Un vincolo di collisione fra una particella e un oggetto è ottenuto verificando che la particella sia all'interno o meno dell'oggetto. Se la particella risulta inclusa nella geometria dell'oggetto, per far si che il vincolo sia rispettato si sposta la particella in un punto sulla superficie esterna dell'oggetto, che di solito è anche il punto più vicino alla posizione originaria della particella.

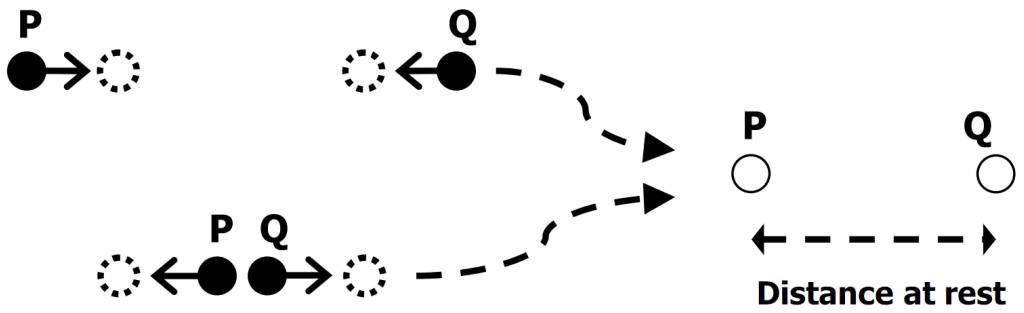


Figura 3.8: *Vincoli di distanza fra le particelle P e Q*

Riassumendo quanto detto fin'ora, per ogni step di simulazione si ha:

- **Step 1**: Su ogni particella non ancorata si applicano le forze attraverso l'equazione di moto.
- **Step 2**: Per ogni particella ancorata si aggiorna la loro posizione
- **Step 3**: Per ogni step di rilassamento:
  - **Step 3a**: Si applicano i vincoli di elasticità
  - **Step 3b**: Per ogni particella si verifica che siano rispettati i vincoli di compenetrazione con tutti gli oggetti simulati nella scena.

3.2.2 Possibile implementazione sulla GPU

Le particelle vengono memorizzate in un buffer e, per ogni step di simulazione, processate attraverso una serie di passate di rendering.

Le forze possono essere applicate a ciascuna particella in parallelo. In maniera similare, i vincoli sulle collisioni possono essere verificati per ciascuna particella in maniera parallela. Pertanto, gli **Step 1** e **3b** possono essere implementati come un'unica passata di rendering durante la fase di *Vertex Shading*.

Anche l'aggiornamento della posizione delle ancore può essere effettuato in maniera parallela, ma dato che i punti ancorati, solitamente, sono una piccola percentuale rispetto al totale dei punti, lo **Step 2** può essere implementato come un unica passata di rendering nella fase di *Fragment Shading* in modo da far coincidere ogni punto ancorato ad un texel sul framebuffer: in pratica, le componenti RGB del framebuffer si utilizzano per memorizzare le componenti XYZ che definiscono la posizione nello spazio dell'ancora dopo essere stata traslata.

Un vincolo di elasticità coinvolge due vertici; pertanto viene processato durante la fase di *Geometry Shading* come una linea. L'unità di Geometry Shading è stata introdotta con lo Shader Model 4.0 delle DirectX 10. Successivamente il supporto è stato ufficializzato anche per le OpenGL in versione 3.2 senza il bisogno di usare alcuna estensione. La sua presenza all'interno della pipeline di rendering permette di generare nuove primitive come ad esempio punti, linee e triangoli partendo da quelle fornite all'inizio della pipeline grafica.

Data la possibilità di fornire in input più vertici contemporaneamente (fino a sei considerando una primitiva triangolare con adiacenze) sarebbe inoltre possibile processare più vincoli in un'unica chiamata. Non esiste, tuttavia, una scelta migliore in questo senso: da un lato, se si processano pochi vertici in una passata del geometry shader, saranno necessarie più passate di rendering per applicare tutti i vincoli ai vertici; d'altro canto, le performance del geometry shader degradano piuttosto velocemente aumentando il numero di vertici forniti in ingresso. La scelta preferibile, come sempre accade in questi casi, risiede nel mezzo: processare quattro vertici la volta, attraverso una primitiva lineare con adiacenza, dovrebbe garantire un buon compromesso.

Due vincoli di elasticità possono essere applicati in parallelo se sono tra loro indipendenti; ciò significa che se due coppie di particelle non condividono alcun vertice allora possono essere processate contemporaneamente. Lo **Step**

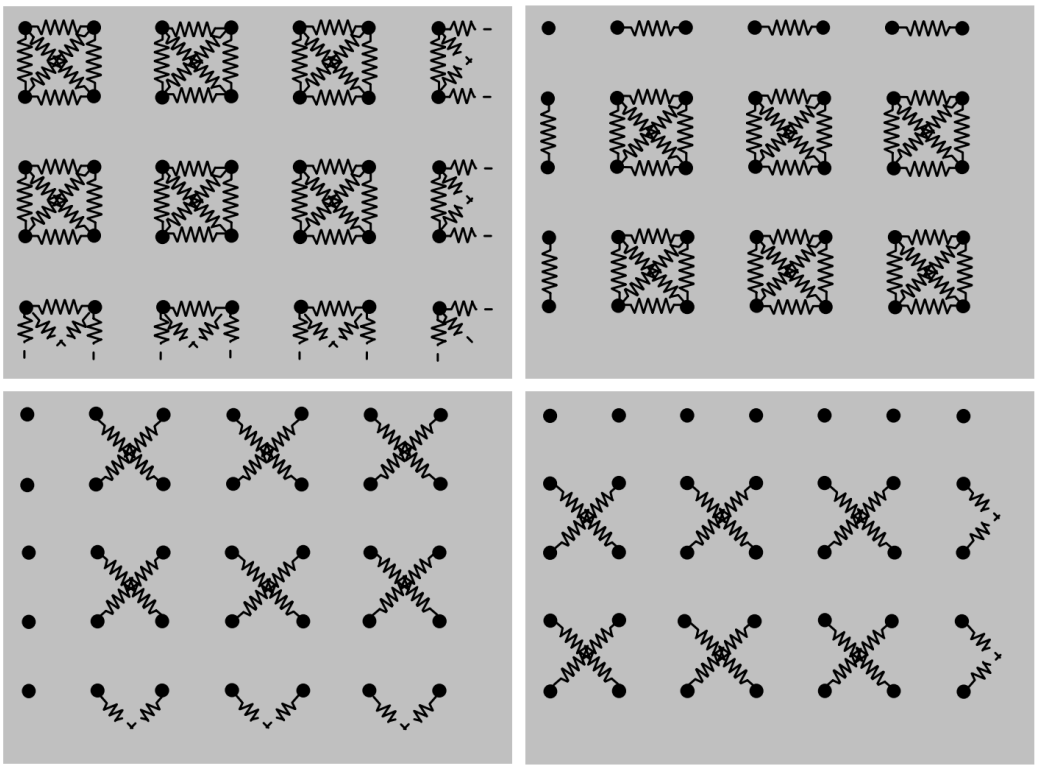


Figura 3.9: *Partizionamento totale dell'insieme dei vincoli di elasticità in quattro gruppi indipendenti*

3a pertanto consiste di una serie di passate di rendering, in cui per ognuna si processano gruppi indipendenti di particelle. Questi gruppi devono essere una partizione del totale dei vincoli da calcolare e, volendo minimizzare il numero di passate di rendering necessarie, bisogna assicurarsi che siano composti dal maggior numero possibile di particelle. In Fig.3.9 si mostra il partizionamento ideale in quattro gruppi di vincoli: lo **Step 3a** avviene pertanto in quattro passate di rendering.

3.2.3 La libreria nVidia PhysX

Come accennato in precedenza, vincoli di tempo e robustezza dell'algoritmo di simulazione hanno portato a scegliere un motore fisico sviluppato da terze parti. Gli obiettivi da tenere in mente durante la scelta del motore fisico erano:

- Possibilità di simulare tessuti: sul mercato sono disponibili diversi motori di fisica per la simulazione delle interazioni tra forze ed oggetti solidi, ma non tutti sono in grado di simulare i tessuti

- Velocità di esecuzione: non è necessario che gli algoritmi di simulazione siano fisicamente accurati, quanto piuttosto devono risultare *fisicamente plausibili* all'occhio di un non esperto. Pertanto è richiesta velocità di esecuzione e non precisione: il candidato ideale avrebbe dovuto supportare le architetture multi-core dei moderni processori e/o un supporto per l'accelerazione tramite hardware grafico.

- Facilità di implementazione: dato che l'obiettivo è quello di creare un software che successivamente sarebbe stato introdotto sul mercato per la vendita, ridurre il cosiddetto *Time-To-Market* è un beneficio sempre ben accetto.

- Minor costo: al fine di massimizzare i profitti, il costo della libreria sarebbe dovuto essere minimo, se non proprio gratuito.

Dopo una serie di ricerche, si è giunti alla conclusione che l'opzione migliore per la tipologia di prodotto software che sarebbe stata realizzata era la nVidia® PhysX® .

La PhysX® è un middleware proprietario che implementa un motore fisico real-time sottoforma di un SDK, attualmente di proprietà nVidia ma che ha visto nel corso degli anni diversi proprietari. La libreria è completamente

gratuita anche per scopi commerciali ma priva di codice sorgente. Inoltre, è ottimizzata per architetture multi-core ed implementa tecniche di GPGPU[4] per l'accelerazione della computazione. Supporta svariate casistiche di simulazione fra cui:

- Collisioni fra primitive quali sfere, cubi, capsule, piani ed altri
- Varie tipologie di giunzioni: sferiche, prismatiche, a cilindro ed altre
- Fisica ragdoll, usata principalmente nei videogame, consente di simulare le reazioni del corpo umano in condizioni di rilassamento muscolare
- Materiali e frizione quando avviene una collisione
- Gestione delle collisioni in maniera continua
- Creazione e simulazione di fluidi volumetrici
- Creazione e simulazione di tessuti, prevenzione dell'auto-compenetrazione e strappamento a seguito di forti tensioni
- Corpi morbidi per la simulazione di oggetti volumetrici deformabili

Di tutte queste caratteristiche, tuttavia, nel corso dello sviluppo del programma è stata implementata esclusivamente la parte relativa ai tessuti, con l'idea futura di generare oggetti simulabili tridimensionali con cui far interagire i tessuti disegnati (un tavolo o una sedia per esempio).

Il lavoro di integrazione è stato minimo e nell'arco di poche settimane è stato possibile integrare quanto fino ad allora sviluppato con le API fornite dalla libreria PhysX® . Per dettagli implementativi si rimanda al successivo capitolo, in cui si analizza la struttura vera e propria dell'applicazione TSimX.

[4]General-Purpose GPU, tramite appositi linguaggi si può utilizzare la GPU per eseguire codice generico[4]

Capitolo 4

L'applicazione realizzata

Analizzate le problematiche di simulazione e presentate le possibili soluzioni, in questo capitolo si discuterà esclusivamente di quanto realizzato praticamente, si affronteranno le motivazioni alla base delle scelte effettuate e si evidenzieranno eventuali carenze dell'applicativo per come è stato strutturato. Infine, le ultime sezioni di questo capitolo tratteranno le periferiche di input alternativo utilizzabili in TSim-X: i tag RFID e i monitor Touchscreen.

L'analisi dell'applicazione si basa sul codice sorgente relativo alla versione 2.1 del software.

4.1 Analisi generale della struttura

L'applicazione è suddivisa in due macro entità: l'interfaccia grafica ed il motore di rendering denominato *TEngine*. Le due entità sono tra loro indipendenti; questa scelta progettuale permette allo sviluppatore di portare avanti il lavoro di *tuning* delle prestazioni e debugging indipendentemente dall'interfaccia che verrà utilizzata in base alle esigenze di mercato. Inoltre, in un team di sviluppatori, questa scelta permette maggior velocità di sviluppo dato che più persone possono lavorare in totale autonomia.

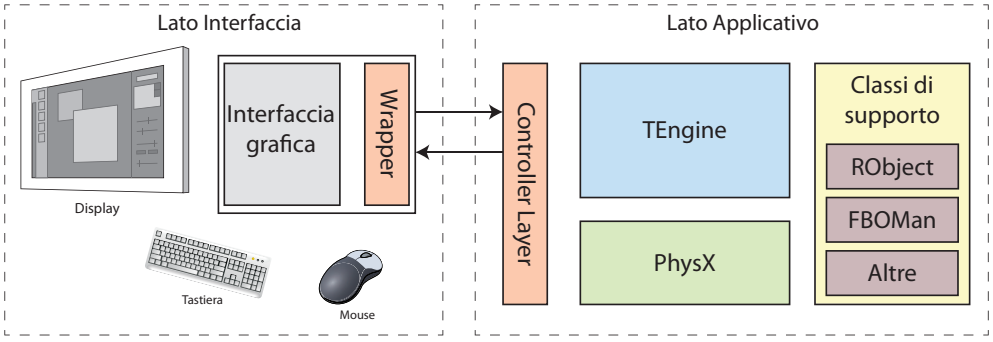


Figura 4.1: *Una rappresentazione schematica a grandi linee della struttura di TSim-X*

Il motore di rendering TEngine, data la complessità dell'applicativo e della necessità di dialogare sia con l'interfaccia che con altre classi deputate a compiti più specifici (come è il caso della fisica, ad esempio), si avvale di un layer intermedio che gestisce e coordina le operazioni del motore stesso, oltre a fornire i metodi necessari all'interfaccia per dialogare con esso. Tale layer è incluso nel macro-blocco applicativo e prende il nome di *ControllerLayer*.

La libreria nVidia® PhysX® è richiamata direttamente dalle classi che ne richiedono l'utilizzo, senza bisogno di layer intermedi. Tuttavia, al fine di rendere più semplice ed immediata la programmazione da parte degli sviluppatori, è stata creata una classe apposita chiamata banalmente *Fisica* che implementa una serie di metodi richiamati spesso all'interno dell'applicazione. Lo scopo di tali metodi è soltanto quello di racchiudere al proprio interno una serie di operazioni svolte di frequente, in modo da avere un codice più pulito e leggibile.

Le classi di supporto invece sono entità che sarebbero dovute essere richiamate in svariati punti del codice. Invece di creare una struttura software per la gestione e la condivisione di queste entità all'interno del programma, data anche la loro reciproca diversità, si è deciso di tenerle separate singolarmente. Se da un punto di vista stilistico e progettuale potrebbe sembrare una cattiva scelta, l'esperienza dimostra come sia in realtà la migliore. Fra

queste classi, infatti, sono presenti gli oggetti renderizzabili RObject, il gestore dei Framebuffer Objects, un generatore di geometria per gli RObjects ed una serie di metodi di debug, enumeratori e strutture globali: molte di queste entità hanno visto sconvolgimenti importanti nella propria struttura, altre sono state aggiunte soltanto successivamente. Se ci fosse stato un gestore per tali entità, molto tempo sarebbe stato speso per aggiungere metodi *getter / setter* o test sullo stato degli oggetti piuttosto che focalizzare l'attenzione su problemi più importanti.

Passando invece dal lato utente e quindi dell'interfaccia, la struttura dipende esclusivamente dall'implementazione voluta. Come si è detto diverse volte, TSim-X non è legato a nessuna interfaccia utente predefinita tant'è che ne sono state realizzate due, caratterizzate per livelli di complessità e funzionalità offerte completamente differenti. L'unica cosa che accomuna tali interfacce è la presenza di una classe chiamata *ControllerWrapper*: essa importa e rende disponibili all'interfaccia i metodi pubblici del ControllerLayer e definisce le strutture utilizzate nello scambio di dati da e verso l'applicativo.

4.2 Le due interfacce grafiche utente

Perché due interfacce? I requisiti dell'applicativo tendono a cambiare durante le fasi dello sviluppo: per questo motivo un buon programmatore deve cercare di scrivere codice quanto più generico possibile in modo da poterlo riadattare nel caso in cui cambino alcuni requisiti, o essere in grado in futuro di aggiungere nuove funzionalità al programma. Ma quando subentra un nuovo cliente che richiede un programma *simile ma diverso* la situazione diventa più complessa.

4.2.1 TSim-X Editor

La prima delle due interfacce analizzate è anche la più complessa. Normalmente si fa riferimento a questa interfaccia con l'acronimo di *GUI v2* o per semplicità *Editor*: il motivo di questa nomenclatura è da ricercarsi, nel primo caso, nella volontà di separare in maniera netta la vecchia interfaccia dalla nuova; con il secondo nome invece si pone l'accento sull'insieme di funzionalità che caratterizzano questa interfaccia da quella chiamata *Touch*, analizzata nella prossima sezione.

L'antenata: GUI v1

All'inizio dello sviluppo di TSim-X, gli obiettivi che avrebbero dovuto caratterizzare l'evoluzione dell'interfaccia utente erano pochi e ben definiti:

- Semplicità ed immediatezza
- Look allineato al resto del sistema operativo
- Pochi ed intuitivi comandi per interagire con la scena: il resto lo avrebbe fatto il programma in maniera intelligente
- Interfaccia ottimizzata per gli schermi widescreen 16:9

Lo scopo quindi non era di creare un *CAD 3D* semplificato, ma una sorta di wizard interattivo in cui proporre all'utilizzatore una serie di scelte durante la fase di creazione della scena, oltre che una serie di strumenti per poter interagire con quanto creato.

L'area di lavoro era divisa principalmente in tre sezioni principali, disposte orizzontalmente in modo da occupare lo schermo in ampiezza e potendo quindi fruire appieno dell'area dei moderni schermi LCD, sempre più *widescreen* (Fig.4.2).

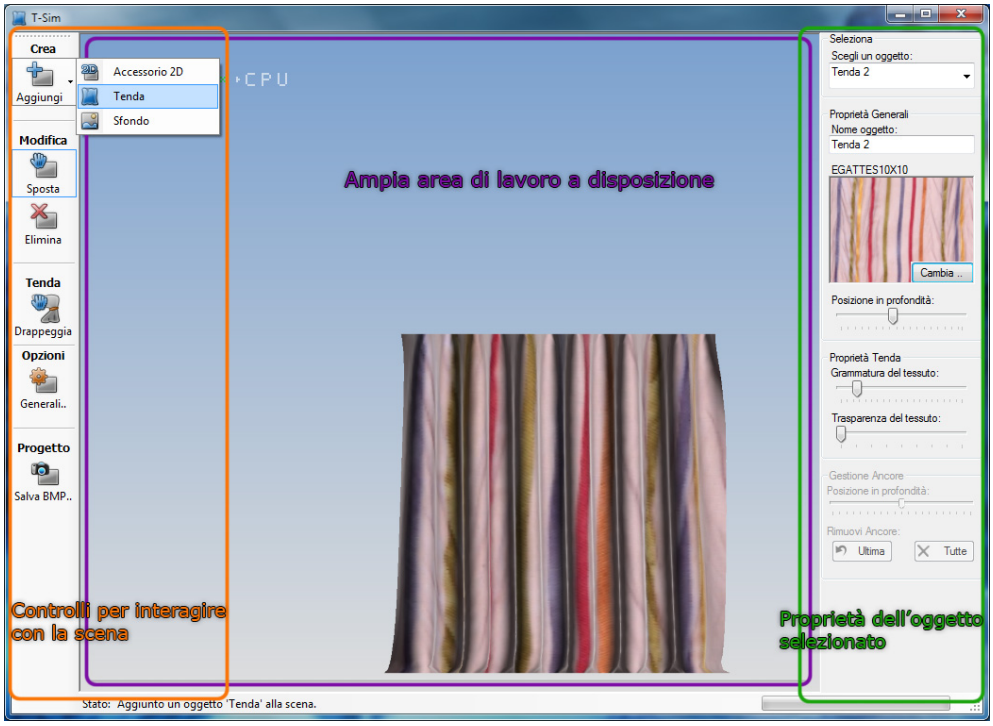


Figura 4.2: *La schermata principale di TSim-X in una delle sue prime versioni funzionanti*

Sulla sinistra erano disposti i controlli preposti all'interazione con la scena: creazione di accessori bidimensionali, tende o sostituzione dello sfondo; spostamento ed eliminazione di oggetti; drappeggio di tende, ovvero la possibilità di alterare la geometria di una tenda già disegnata in modo da creare composizioni complesse, tutto simulato in tempo reale dal motore PhysX® . Era inoltre disponibile una schermata di opzioni in cui modificare alcuni parametri prestazionali del programma, rimasta ad oggi sostanzialmente invariata. Era infine possibile esportare la scena realizzata in TSim-X come immagine Bitmap, per la condivisione online o per essere stampata.

Parallelamente, sul lato destro, trovavano posto le proprietà dell'oggetto attualmente selezionato, oltre che un comodo menù a tendina da cui poter

selezionare gli oggetti presenti nella scena. Le proprietà che era possibile cambiare riguardavano il nome dell'oggetto, la texture ad esso associata e la posizione in profondità dell'oggetto rispetto alla scena. Esclusivamente per le tende era infine possibile modificare la grammatura (pesantezza) del tessuto, la trasparenza dello stesso e la posizione in profondità dei singoli vertici durante la fase di drappeggio.

Sull'area di lavoro trovava posto un ultimo controllo che faceva parte della scena OpenGL renderizzata, il pavimento. Variando la posizione del pavimento è possibile impostare un altezza massima oltre la quale i tendaggi non possono spostarsi in fase di simulazione.

L'evoluzione: GUI v2

Arrivati ad una versione stabile e funzionante del programma, si è partiti con l'aggiunta di nuove funzionalità che per motivi di tempo non avevano trovato spazio nel primo ciclo produttivo. Tuttavia nuove funzionalità richiedono nuovi controlli da visualizzare su schermo e da subito questa necessità si scontrava con l'angusto spazio messo a disposizione dall'interfaccia grafica. Varie strade erano state pensate per arginare il problema: aggiungere più menù a tendina come per il caso del comando *Aggiungi*; creare un selettore di modalità (creazione, modifica e così via) nella parte superiore dello schermo, in modo da filtrare i comandi visualizzati nella parte sinistra dell'interfaccia. Altre strade erano state pensate ma sono state subito scartate in quanto avrebbero complicato troppo un'interfaccia nata per essere semplice ed intuitiva.

L'idea del selettore di modalità era valida, perché permette facilmente di moltiplicare lo spazio a disposizione per i controlli anche se risulta poco *user-friendly* in quanto scompaiono alcuni controlli per dar posto ad altri: come alcuni test effettuati in azienda hanno dimostrato, questa scelta tende a

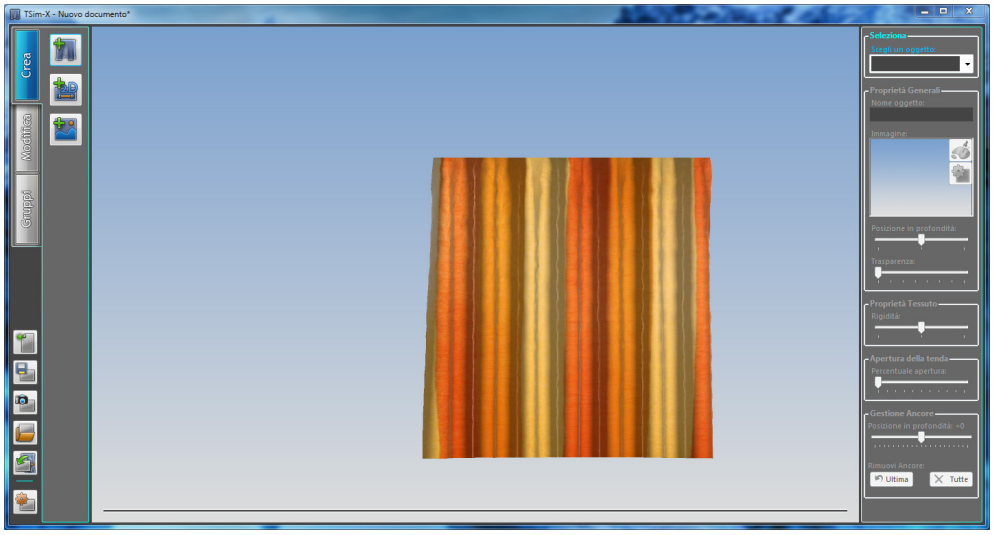


Figura 4.3: *La schermata principale di TSim-X nell'attuale versione 2.1*

confondere le idee di un utilizzatore alle prese con il programma per la prima volta. Per questa ragione, sarebbe stato preferibile utilizzare molti menu a tendina, come praticamente qualsiasi software continua a fare dai tempi di Windows 3.1 in poi. Tuttavia, con l'aumentare dei controlli nei menu l'interfaccia diventa pesante alla vista e spesso si perdono di vista alcuni controlli[27].

La soluzione implementata nella seconda versione dell'interfaccia di TSim-X è rappresentata in Fig.4.3. Si è fatto ricorso ad uno strumento ampiamente utilizzato per strutturare in maniera ordinata le applicazioni: i tab. Ciascun tab di TSim-X trova spazio nella parte sinistra dello schermo ed è disposto verticalmente, questo per dare l'idea di un selettore di modalità, come detto prima, ma con un vantaggio in termini di usabilità. Invece di far sparire i controlli al click di un mouse, grazie al meccanismo dei tab è possibile istruire l'utente in maniera estremamente naturale sulla disposizione dei componenti: al pari dei cassetti di un comodino, in cui ciascuno contiene (si spera) la biancheria divisa per tipologia, i tab in TSim-X contengono controlli rag-

gruppati per modalità di utilizzo. Ciascun tab inoltre è etichettato in modo da rendere il tutto più ordinato:

- **Crea**: il primo tab ad essere visualizzato di default è anche il più semplice; ciò permette di instaurare da subito con l'utente un legame di leggerezza e di fiducia, evitando lo spavento che si ha con i programmi costituiti da tanti piccoli menu ed icone. Permette di creare le tre tipologie di oggetti che attualmente è possibile disegnare con TSim-X: accessori, tende e sfondi.

- **Modifica**: racchiude i controlli per manipolare la scena. Rispetto alla prima versione dell'interfaccia, molti altri controlli sono stati aggiunti: duplica oggetto, sostituisci la texture a tutte le tende, ridimensiona un accessorio, ribalta tenda lungo l'asse verticale.

- **Gruppi**: una nuova categoria di controlli prima assenti; consente di raggruppare più oggetti fra loro in modo da poterli considerare come un unico oggetto durante le operazioni di spostamento. Questa feature si dimostra molto utile in scene complesse, in cui una tenda è in realtà composta da più elementi uniti fra loro. È inoltre possibile ridimensionare tutti gli oggetti dello stesso gruppo di una percentuale definita dall'utente.

- Esiste un quarto gruppo, seppur diverso dagli altri, ed è costituito dalle icone disposte in verticale nello spazio venutosi a creare sotto le etichette dei tab. In questa sezione trovano luogo tutti quei controlli che sono indipendenti dal tipo di modalità in cui il programma opera e riguardano più in generale l'intero documento di lavoro: troviamo infatti i pulsanti nuovo, salva, carica ed importa documento; salva come immagine ed infine il pulsante per accedere alle opzioni di TSim-X.

Anche l'aspetto generale del programma ha subito modifiche. Abbandonando il *look'n'feel* di sistema, TSim-X ha maturato un'interfaccia più personale, in modo da caratterizzarlo meglio e creare un'impronta distintiva che lo identificasse univocamente.

Il colore predominante è il grigio scuro, non a caso: avere come colore di sfondo il grigio scuro acromatico permette di evitare distrazioni ed impedisce alla scena visualizzata di subire alterazioni di colore a livello di retina nel bulbo oculare. Inoltre, uno sfondo grigio è riposante per i coni dell'occhio, oltre ad aumentare l'abilità di discernere i colori senza avere immagini persistenti sulla retina. Infine, avere uno sfondo grigio piuttosto che nero evita, in presenza di immagini molto luminose, il fenomeno detto *glare*, in cui il forte contrasto con lo sfondo altera la percezione delle sfumature.

La parte destra del programma è rimasta sostanzialmente invariata, con l'unica aggiunta costituita da uno slider per la regolazione dell'apertura di alcuni tipi di tende, analizzate in seguito.

Analisi strutturale dell'interfaccia

Dal punto di vista informatico, l'interfaccia grafica di TSim-X è stata realizzata sfruttando le tecnologie .NET di Microsoft attraverso l'uso del linguaggio C♯ ed importa attraverso la classe *CtrlWrapper* i metodi della parte applicativa del programma. Il meccanismo dell'importazione dei metodi da una DLL in codice nativo all'interno di codice managed è ottenuta attraverso le *PInvoke*:

```
[DllImport("<nome_della_libreria.dll>")]
static private extern <tipo_ritornato> <firma_del_metodo>
```

Il metodo sarà quindi messo a disposizione della classe che lo importa.

La logica applicativa che interessa in maniera esclusiva l'interfaccia viene gestita invece in una classe separata chiamata *MainApp*. Il motivo per cui si

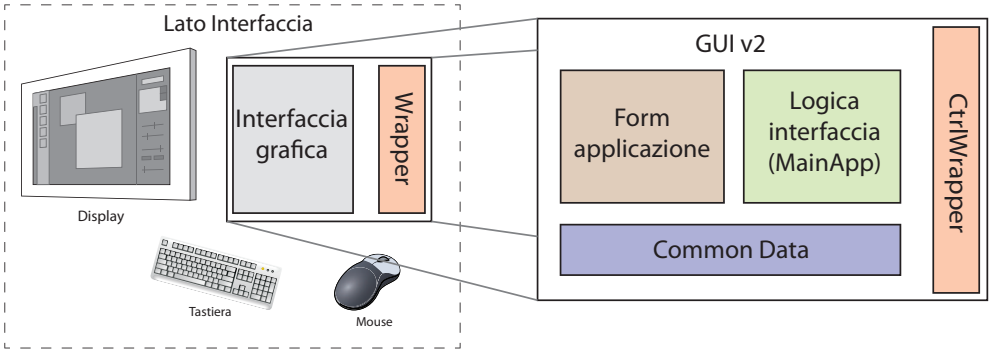


Figura 4.4: *Uno schema rappresentativo della struttura logica delle classi che compongono l'interfaccia grafica di TSim. Da notare come lo schema rimanga identico volendo cambiare interfaccia: basta infatti sostituire il blocco dei form e nient'altro.*

è scelto di separare la logica applicativa dall'interfaccia vera e propria è duale: prima che subentrasse la necessità di sviluppare una seconda interfaccia utente, tutto ciò che riguardava la GUI veniva implementato direttamente nella classe del form visualizzato su schermo. Questo approccio benché semplice ed immediato si è rivelato essere altamente inefficiente nel momento in cui sono iniziati i lavori sull'interfaccia Touch: molti metodi infatti risultavano praticametne identici nell'implementazione fra le due interfacce e sarebbe stato un grosso problema dover ricopiare interamente il codice già scritto in una nuova classe. Non solo avrebbe richiesto del tempo, ma l'onere di tenere costantemente aggiornate e sincronizzate le due interfacce a livello di metodi sarebbe stato troppo impegnativo. Per questa ragione, tutto ciò che riguardava la logica applicativa dell'interfaccia, fra cui:

- creazione di oggetti
- duplicazione e ridimensionamento di oggetti
- salvataggio e ricaricamento di una scena

- gestione delle opzioni del programma

e più in generale, tutto ciò che non riguardasse esclusivamente la rappresentazione visiva dei controlli su schermo, venne spostata nella classe MainApp. Altro vantaggio di questa scelta fu l'elevata chiarezza del codice presente nella classe del form principale, il quale attualmente gestisce esclusivamente la logica di selezione degli oggetti tramite menu a tendina ed il controllo degli slider, oltre che mostrare all'utente le schermate di apertura e salvataggio di files: tutte cose di esclusivo appannaggio dell'interfaccia, per l'appunto.

Sempre nell'ottica di una maggior separazione dei ruoli è stata creata la classe *CommonData* che mette a disposizione metodi e valori costanti condivisi fra le classi. Fra questi sono da menzionare:

- FileIO: classe che permette di salvare su file il contenuto della scena. Si preoccupa di tradurre gli oggetti presenti in memoria in una stringa in formato XML che verrà quindi salvata su file. Viceversa per quanto riguarda i caricamenti.

- Enum: gran parte del programma funziona sulla base delle costanti e delle enumerazioni definite in questa classe.

A questo breve elenco si sommano anche una serie di piccoli metodi di utility riutilizzati in svariate parti del codice.

Funzionamento dell'interfaccia

Creazione di oggetti Per creare un oggetto, il programma deve trovarsi in modalità Crea. Quindi cliccando su uno dei pulsanti è possibile scegliere che tipologia di oggetto creare: tenda, accessorio o sfondo. Nel caso si voglia creare una tenda, apparirà la schermata mostrata in Fig.4.5.

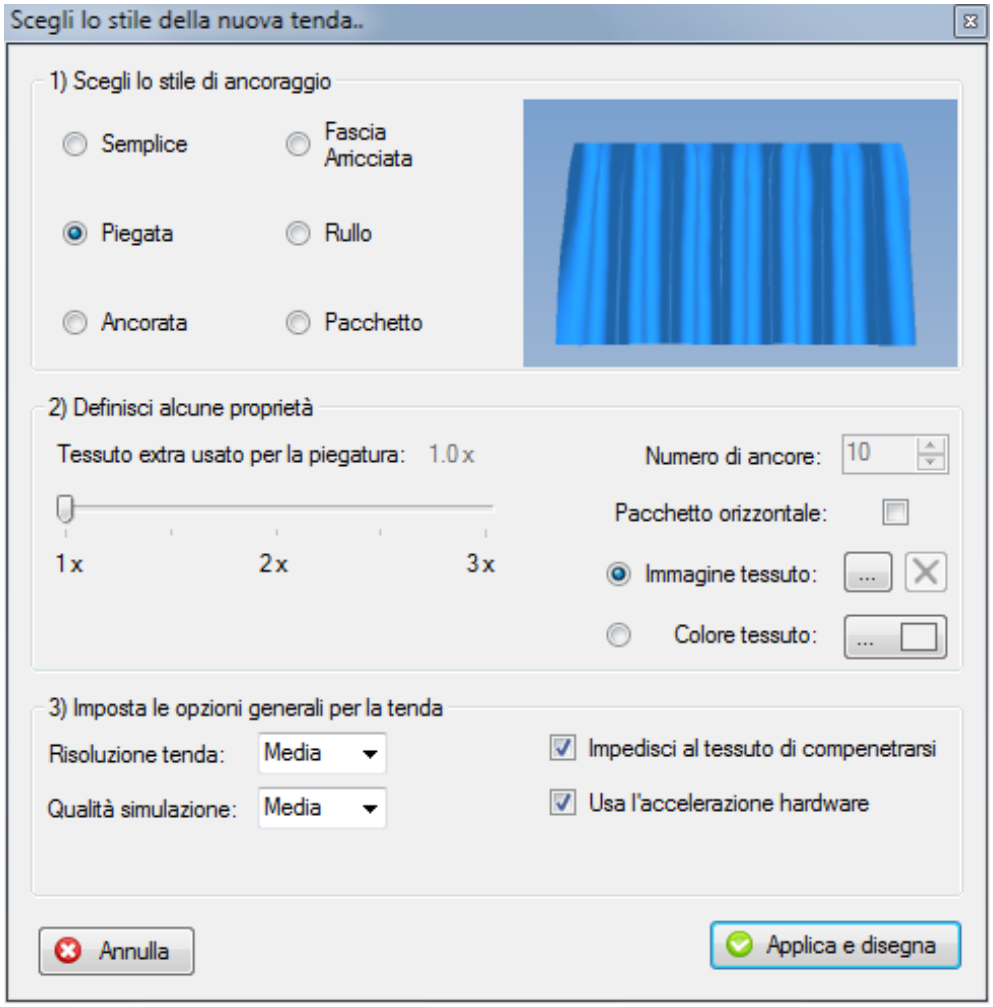
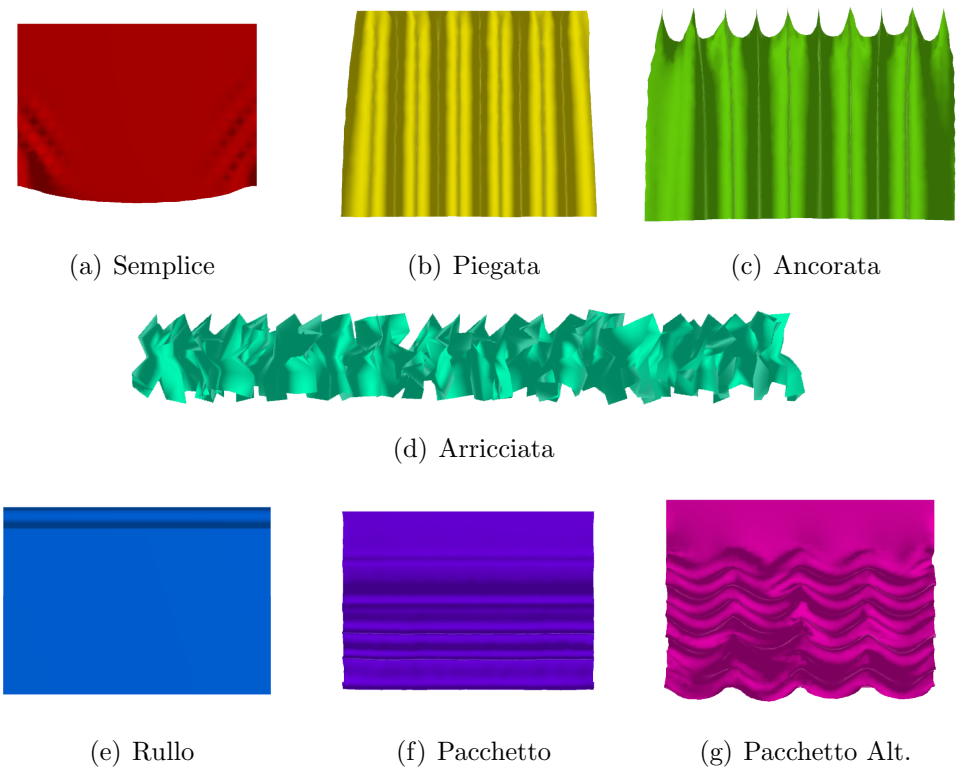


Figura 4.5: *La creazione di una tenda è affidata ad un'interfaccia simile ad un wizard: si sceglie dapprima il tipo di tenda, quindi si impostano i parametri che la caratterizzano ed infine si regolano alcune opzioni prestazionali per la simulazione del singolo tessuto.*

Qui è possibile definire l'apparenza della tenda già prima che venga disegnata. Nella prima sezione della schermata si definisce la tipologia base della tenda, come ad esempio ancoraggio o arrotolamento. Nella seconda fase invece si caratterizza maggiormente la tenda aggiungendole un effetto di piegatu-



(a) Semplice (b) Piegata (c) Ancorata

(d) Arricciata

(e) Rullo (f) Pacchetto (g) Pacchetto Alt.

Figura 4.6: *Le tende disegnabili in TSim-X*

ra, assegnandole un colore o una texture oppure definendo il tipo di chiusura che la tenda avrà. Nella terza ed ultima fase è possibile intervenire per regolare finemente alcuni parametri del motore di simulazione fisico, come ad esempio l'utilizzo della GPU per accelerare i calcoli o l'auto-compenetrazione del tessuto.

Le tende che è possibile creare mediante questa schermata sono raggruppate nella Fig.4.6.

Una volta confermate le opzioni nella schermata di creazione, sarà sufficiente tracciare un rettangolo sull'area di lavoro e la tenda sarà creata istantaneamente dal motore PhysX® .

La creazione di un accessorio invece si limita esclusivamente alla scelta, tramite una classica finestra di dialogo, della texture da associarvi. Lo stesso avviene per lo sfondo, senza però dover definire la dimensione dell'area, in quanto lo sfondo occupa l'intero schermo.

A qualsiasi oggetto, all'atto della sua creazione, viene assegnato un identificatore univoco, chiamato d'ora in avanti *ObjID*. Grazie a questo identificatore è possibile creare una corrispondenza fra gli oggetti gestiti a livello di interfaccia e quelli gestiti a livello applicativo dal TEngine. Ogni metodo che vorrà interagire con un oggetto infatti avrà come parametro un intero che accetterà tale identificatore.

Applicazione di una texture L'applicazione di una texture può avvenire in tre modalità:

- Durante la fase di creazione dell'oggetto stesso è possibile (necessario per gli accessori) definire come parametro la texture che sarà applicata.
- Selezionando l'oggetto e cambiando la texture cliccando sull'area apposita nella parte destra dello schermo.
- Utilizzando il tasto "Cambia texture a tutti" in modalità Modifica. In questo modo tutte le texture delle tende vengono sostituite contemporaneamente con quella scelta nella finestra di dialogo che appare.

Nel secondo caso, ovvero quando si modifica la texture del singolo oggetto dopo che è stato creato, è possibile accedere ad una schermata aggiuntiva (Fig.4.7) cliccando sull'icona relativa. Qui è possibile definire esattamente come verrà applicata la texture all'oggetto. Poter modificare la scala o il tipo di ripetizione di una texture direttamente dal programma consente, nella maggioranza dei casi, di avere una texture utilizzabile senza doverla necessariamente editare con programmi di fotoritocco.

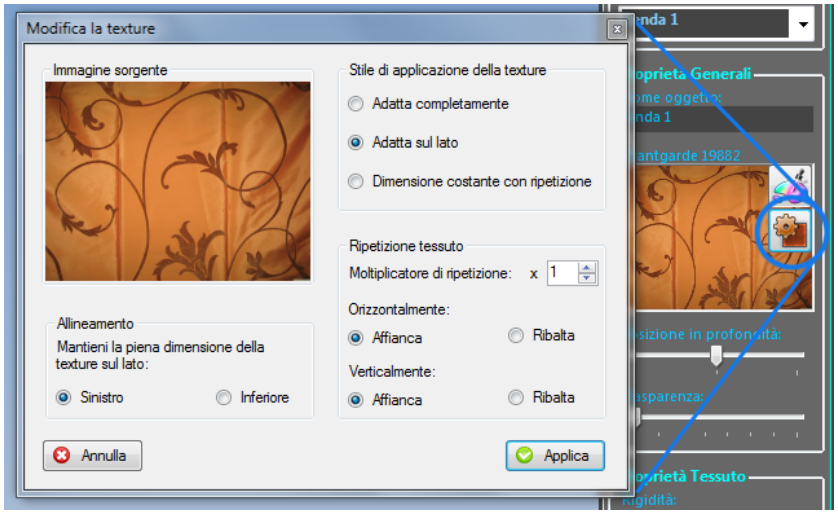


Figura 4.7: *Grazie alla schermata di modifica, è possibile definire dettagliatamente il metodo di applicazione delle texture.*

Interazione con la scena Nella modalità modifica, TSim-X mette a disposizione un insieme di strumenti per la manipolazione della scena creata. Alcuni comandi posso essere utilizzati su tutti gli oggetti, come ad esempio sposta o elimina; lo stesso vale per la funzione *duplica oggetto*, molto utile nel caso di tende fortemente personalizzate mediante drappeggio. Limitatamente agli accessori, è possibile utilizzare lo strumento ridimensiona: le tende infatti, per come sono gestite dalla libreria PhysX® , non possono essere ridimensionate in tempo reale dato che andrebbe a distruggere la simulazione del sistema di molle fra le particelle elementari del tessuto. Tuttavia, come spiegato in seguito, si è riusciti ad implementare qualcosa di molto simile nel caso del ridimensionamento multiplo in gruppo.

Drappeggio e ribalta sono invece due comandi esclusivi nell'editing di tende. Se da un lato è facile capire perché un oggetto bidimensionale non possa essere drappeggiato in TSim-X, le motivazioni alla base della impossibilità di ribaltare un accessorio sono meno immediate. Un accessorio è infatti costituito da una texture applicata su di una geometria rettangolare:

ribaltarlo vorrebbe dire solamente invertire la texture lungo l'asse verticale. Nell'attuale versione del programma non è stata data la possibilità di farlo direttamente dall'interfaccia dato che è sufficiente modificare l'immagine con un qualsiasi programma di fotoritocco. Non si esclude, tuttavia, la possibilità per le future versioni di TSim-X di abilitare il ribaltamento anche per gli accessori.

Salvataggio e ricaricamento di una scena Una delle caratteristiche maggiormente richieste dopo il rilascio della versione 1.0 del programma fu la possibilità di salvare e successivamente ricaricare la scena composta su file. In tal modo un venditore di tende può prepararsi le scene *tipiche* nel suo ufficio per poi averle immediatamente a disposizione sul suo portatile quando, chiamato da un cliente, gli viene chiesto di mostrare loro un'anteprima.

Per consentire la massima interoperabilità possibile con eventuali evoluzioni future dell'architettura del programma, TSim-X esporta le scene sottoforma di file di testo formattato in linguaggio XML. La struttura del file può essere schematizzata come rappresentato in Fig.4.8.

In seguito si analizzeranno le strutture degli elementi *RClothObj* e *RGenericObj*.

Raggruppamento di oggetti La funzione di raggruppamento degli oggetti permette di vincolare nello spostamento più oggetti. Non è necessario che gli oggetti siano tutti dello stesso tipo ed anzi è possibile unire più gruppi fra loro. Attualmente non sono supportate gerarchie di gruppi: ciò vuol dire che un gruppo, se unito ad un altro, perde la sua struttura originaria unendo i propri elementi con quelli dell'altro gruppo, formandone uno completamente nuovo.

La creazione di un gruppo avviene selezionando più oggetti contemporaneamente, premendo il tasto CTRL mentre si evidenziano gli oggetti che

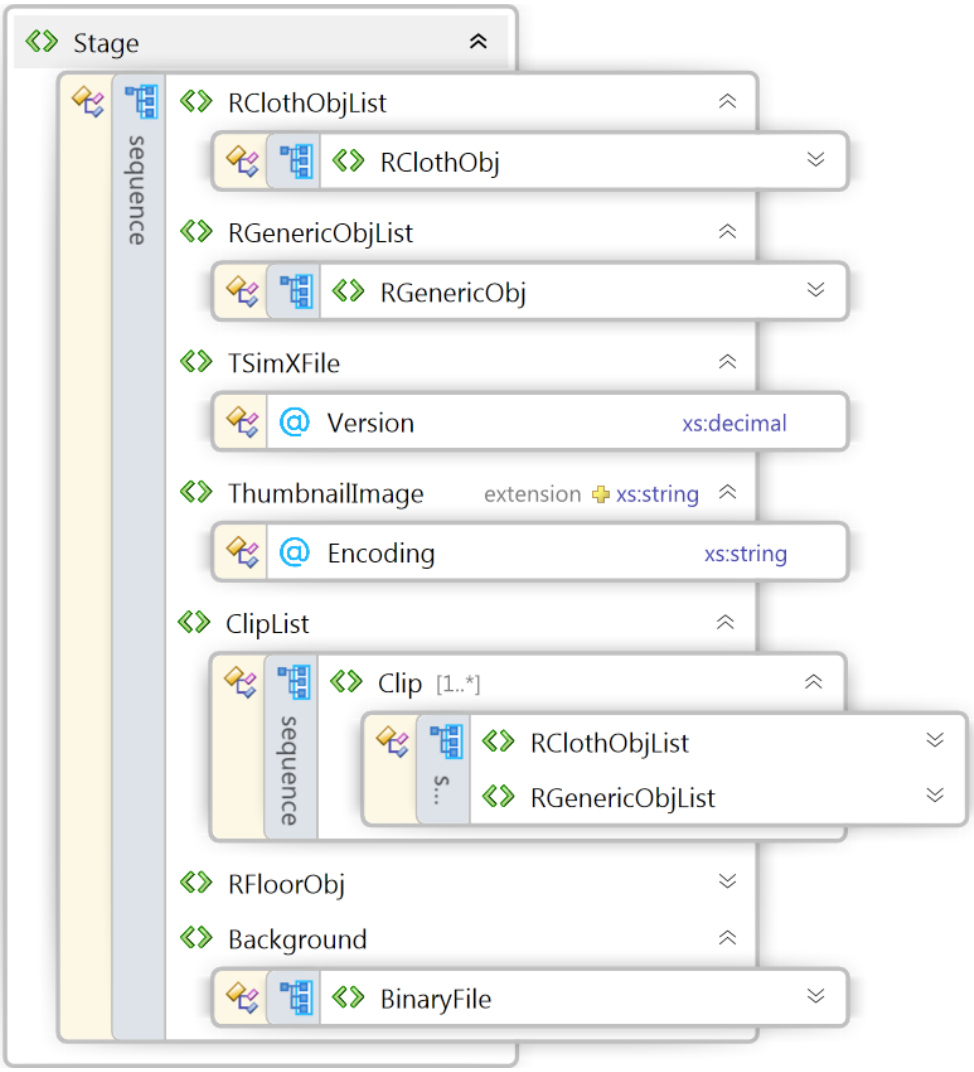


Figura 4.8: *Uno schema della struttura di un file XML prodotto da TSim-X 2.0.*

faranno parte di esso. Durante tale operazione, TSim-X popola una struttura monodimensionale (un ArrayList) con gli *ObjID* degli oggetti selezionati. Spostando il gruppo, TSim-X itera su tutti gli oggetti presenti nella lista e per ognuno di essi invoca il metodo per lo spostamento con il medesimo offset: tuttavia alcuni problemi sono sorti nell'implementazione di questa funzionalità. Il problema, che verrà analizzato e risolto nella sezione riguardante il

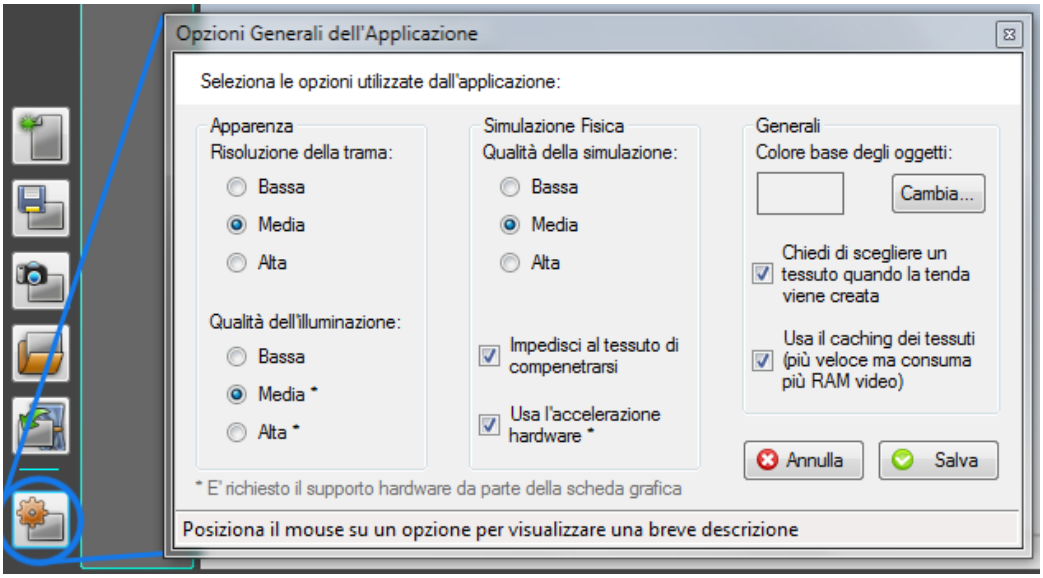


Figura 4.9: *La schermata di opzioni di TSim-X, utile per ottimizzare le prestazioni e la qualità della simulazione.*

controller, riguarda la gestione degli eventi da parte del motore fisico; parrebbe infatti che PhysX® non riesca a gestire più eventi contemporaneamente sullo stesso oggetto fra un passo di simulazione ed il successivo.

Modifica delle opzioni del programma Attraverso l'apposito pulsante è possibile accedere alla schermata di opzioni di TSim-X, riportata in Fig.4.9. Nella parte sinistra della schermata trovano spazio una coppia di opzioni per la regolazione della qualità visuale della simulazione. È possibile regolare la densità della trama utilizzata nella rappresentazione di tendaggi, modificando quindi il dettaglio poligonale della mesh tridimensionale. Aumentando il dettaglio della trama, ogni tenda sarà costituita da più vertici anche se, parallelamente, il motore fisico dovrà iterare su più particelle nella fase di simulazione: da un lato si migliora la qualità visiva quindi, mentre dall'altro si peggiorano le performance. Diminuendo il dettaglio poligonale, viceversa,

ci saranno meno particelle da dover simulare, anche se sarà possibile notare la geometria del tessuto abbastanza facilmente.

Modificando invece l'opzione relativa alla qualità di illuminazione, TSim-X abiliterà o disabiliterà gli effetti discussi nel capitolo 3.1.6 nel seguente ordine:

- Bassa: non viene utilizzato alcun effetto di quelli discussi. TSim-X in questa modalità diventa compatibile con OpenGL 1.4 in modo da garantire la massima compatibilità possibile. A titolo di curiosità, in azienda è stato possibile avviare il programma in questa modalità su di un vecchio portatile dei primi anni 2000 con grafica integrata Intel. Nonostante le pesanti limitazioni grafiche, in questa modalità TSim-X permette di utilizzare tutte le funzionalità implementate.

- Media: viene abilitato il rendering dei tessuti con il cloth shader, e dei Framebuffer Objects per ottimizzare le performance dell'applicazione. I requisiti dell'applicazione salgono, imponendo all'utilizzatore di avere una scheda grafica e dei driver video compatibili con le specifiche OpenGL 2.0.

- Alta: viene infine abilitato il rendering con light blooming, parallelamente al cloth shader. In futuro altri shader potrebbero venire sviluppati ed utilizzati all'abilitazione della modalità di rendering ad alta qualità.

Centralmente nella schermata di opzioni è possibile trovare le impostazioni che influenzano il motore fisico: alterando il parametro di qualità della simulazione, verranno rispettivamente eseguiti più o meno step di rilassamento, come spiegato al capitolo 3.2.1. Praticamente, un basso numero di iterazioni comporterà nelle tende una risposta al movimento molto elastica, caratteristica assolutamente non presente in realtà. D'altro canto su macchi-

ne poco potenti senza possibilità di utilizzare l'accelerazione hardware fornita dalle PhysX® , potrebbe essere un buon metodo per guadagnare in fluidità nella fase di editing. Viceversa, impostando il parametro su *Alta*, farà in modo che il comportamento della tenda sia quanto più realistico possibile da parte della libreria PhysX® .

Già viste in fase di creazione di una tenda, anche nella schermata di opzioni generali è possibile abilitare l'accelerazione hardware (se supportata dalla macchina, altrimenti risulterà inibita) e l'algoritmo che impedisce al tessuto di auto-compenetrarsi in movimento o drappeggio.

Infine, nella parte destra della finestra, è possibile definire un colore default che sarà applicato a tutti gli oggetti privi di texture; è possibile inoltre far in modo che TSim-X utilizzi più memoria video per tenere memorizzate, fino alla chiusura del programma, le texture di tende e accessori. Grazie a questa opzione TSim-X non dovrà ricaricare una seconda volta in memoria una texture precedentemente usata, velocizzando quindi le operazioni di editing.

4.2.2 TSim-X Touch

Diversamente da quanto visto con l'Editor, l'interfaccia grafica di TSim-X *Touch* o semplicemente *XT*, risulta estremamente minimalista. L'applicazione, come mostrato in Fig.4.10, è costituita da una o più schermate di rendering prive di qualsiasi controllo e da una piccola finestra fluttuante composta da appena quattro pulsanti in tutto.

Il minimalismo estremo, al contrario di quanto visto con l'Editor, si è reso necessario in base allo scenario d'utilizzo dell'applicazione. L'idea, suggeritaci da un cliente, era quella di mettere a disposizione di un utente in visita ad un negozio, una postazione con monitor touchscreen in cui avesse potuto cambiare il tessuto della tenda mostrata ed eventualmente anche l'ambien-

Figura 4.10: *Una videata dell'interfaccia minimalista usata in TSim-X Touch.*

te di cui la scena si componeva. Il monitor touchscreen avrebbe permesso all'utente di interagire con la tenda in maniera maggiormente personale e naturale, oltre che divertente; contemporaneamente, un secondo monitor di grandi dimensioni, avrebbe mostrato agli altri clienti del negozio ciò che in tempo reale l'altra persona stava facendo. In questo modo si punta a coinvolgere ed invogliare maggiormente gli altri clienti a provare la postazione, fidelizzandolo maggiormente al negozio. Il programma sarebbe diventato quindi uno strumento di *marketing*.

Attualmente l'interfaccia Touchscreen è sotto pesante lavoro di reingegnerizzazione per via di nuove prospettive di utilizzo ancora in fase di definizione.

4.3 Supervisione e gestione degli eventi - Controller

All'inizio pensato come un sottile layer che supervisionasse l'esecuzione del programma, man mano cresciuto e diventato la parte più importante di TSim. Oltre 2500 righe di codice C++ che gestiscono eventi, coordinano le operazioni di rendering e simulazione fisica. In Fig.4.11 è stato riportato uno schema UML della classe e delle strutture dati da essa definite; non sono stati riportati tutti i metodi di cui si compone la classe per ragioni di semplicità: sono stati esclusi i metodi *overload*, metodi di utility e *stub*.

All'avvio di TSim, l'interfaccia grafica istanzia l'oggetto Controller specificando nei parametri del costruttore, attraverso gli *iFlag*, quali caratteristiche abilitare nel programma. Volendo è possibile creare un istanza del controller privo di simulazione fisica o privo del motore grafico: naturalmente, i metodi che per essere eseguiti necessitino di una o dell'altra caratteristica non verranno richiamati dal Controller, impedendo quindi che il programma vada in crash. Questo metodo parametrizzato di creazione del Controller permette, ad esempio, di creare una seconda istanza del motore di rendering che visualizzi soltanto delle texture, senza dover occupare ulteriori risorse di memoria con la libreria PhysX® che comunque non verrebbe utilizzata.

Successivamente, l'interfaccia specifica l'area su cui il rendering della scena dovrà avvenire, insieme alle sue dimensioni in pixel, attraverso il metodo `initALL()`. L'area di rendering viene identificata mediante un *Window Handler*, spesso abbreviato dalla sigla *HWND*. Grazie ad esso OpenGL è in grado di creare un contesto di rendering dove avverranno tutte le operazioni di visualizzazione. È tuttavia possibile definire solamente le dimensioni dell'area, senza specificare la destinazione dell'output video: in questo caso TEngine provvederà a creare una finestra in maniera completamente autonoma, passando all'OpenGL lo HWND di quest'ultima; non sarà tuttavia

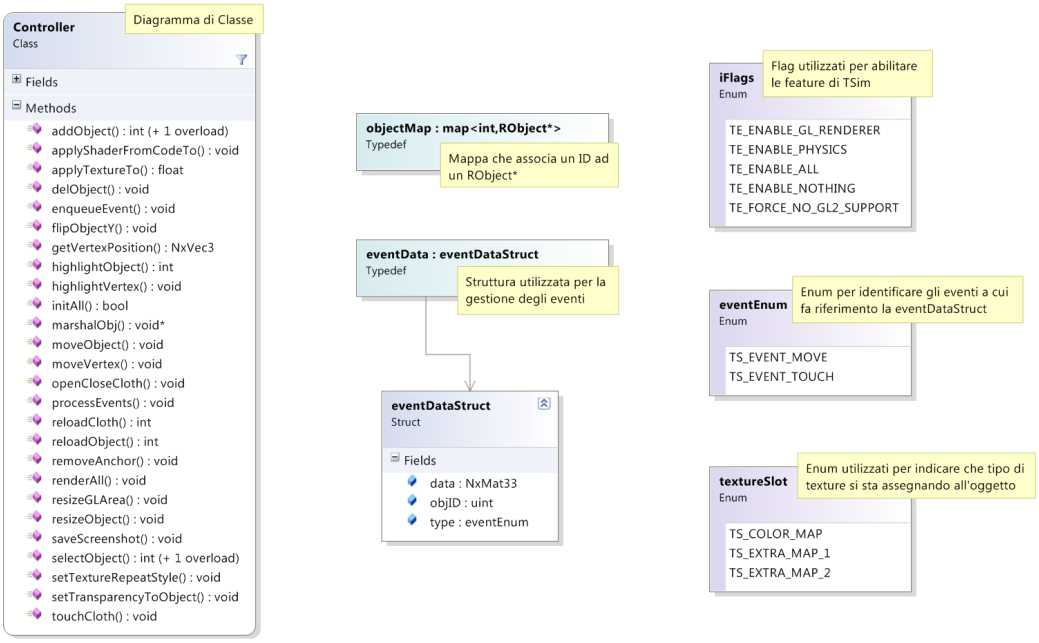


Figura 4.11: *Lo schema UML della classe Controller e delle strutture ad esso associate.*

possibile interagire con la scena, ma solo visualizzarla. L'idea in fase di progettazione di questa feature era di poter permettere un avvio parametrizzato da riga di comando di TSim-X per visualizzare una scena precedentemente salvata su di un client remoto. In questa fase vengono inizializzati, se supportati dall'hardware, anche Shader e Framebuffer Object, utilizzati durante il rendering a media ed alta qualità di illuminazione; viene infine istanziato il motore PhysX.

4.3.1 L'interfaccia verso il mondo esterno: ControllerCaller

Fino a questo punto, si è sempre lasciato intendere che le classi Controller e CtrlWrapper fossero in diretta comunicazione tra loro. Al fine di rendere il

codice più leggibile e semplice è stata introdotta una classe intermedia che avesse come unico compito quello di esporre i metodi pubblici del Controller verso l'esterno. Il controller, lo ricordiamo, è compilato come libreria a collegamento dinamico (*dll*): una libreria dinamica, per poter esportare un metodo, deve esplicitarlo mediante identificatori anteposti alla dichiarazione del metodo. Onde evitare di creare troppa confusione nel già affollato header della classe Controller, oltre che fornire una classe che si preoccupasse di convertire i tipi di dato da e verso il CtrlWrapper, è stata introdotta la classe ControllerCaller.

4.3.2 Comunicazione tra GUI e applicativo

Per meglio spiegare il collegamento esistente fra le tre classi appena citate, si analizzerà a titolo di esempio il metodo `getVertexPos()` che accetta come parametro un intero e restituisce le coordinate spaziali di un vertice.

L'interfaccia, interrogando la PhysX, genera un indice relativo al vertice di cui vuole conoscere le sue coordinate. Quindi richiama il metodo:

```
coords vCoords = mainGLArea.GetVertexPosition(
                        selectedObjectID,
                        vertexID );
```

In questo esempio, `mainGLArea` è l'oggetto istanziato della classe CtrlWrapper all'interno della classe `MainApp` discussa all'inizio di questo capitolo. I parametri forniti al metodo permettono di identificare con precisione l'oggetto a cui appartiene il vertice di indice `vertexID`. Il wrapper a questo punto richiamerà la *controllercaller.dll* tramite il metodo:

```
[DllImport("ControllerLayer.dll")]
static private extern void CallGetVertexPos(
```

```
            IntPtr pControllerObject,
            int objectID,
            int vertexID,
            float[] vertPos );

   public coords GetVertexPosition(
            int objectID,
            int vertexID)
   {
      coords vertPosMan;
      float[] vertPosUnman = new float[3];
      CallGetVertexPos(
            this.m_pNativeObject,
            objectID,
            vertexID,
            vertPosUnman );

      vertPosMan.X = vertPosUnman[0];
      vertPosMan.Y = vertPosUnman[1];
      vertPosMan.Z = vertPosUnman[2];

      return vertPosMan;
   }
```

Dall'altro lato, il ControllerCaller fornisce la seguente definizione del metodo `CallGetVertexPos()`:

```
   extern "C" __declspec(dllexport) void CallGetVertexPos (
            Controller *pObject,
            int objectID,
```

```
            int vertexID,
            float *vertexPos )
{
   if (pObject != NULL)
   {
      NxVec3 pos = pObject->getVertexPosition(
         objectID,
         vertexID );

      vertexPos[0] = pos.x;
      vertexPos[1] = pos.y;
      vertexPos[2] = pos.z;
   }
}
```

L'oggetto puntato da `pObject` è l'istanza della classe Controller. Pertanto, il metodo `CallGetVertexPos()` riportato sopra, richiama a sua volta il metodo `GetVertexPos()` del controller:

```
NxVec3 Controller::getVertexPosition( int objID, int vertID )
{
   objectMap::iterator objIT = myObjectMap.find(objID);
   if (objIT != myObjectMap.end())
      return objIT->second->getVertexPosition(vertID);

   return NxVec3(0.0f);
}
```

Il Controller tiene traccia di tutti gli oggetti creati grazie ad una mappa, definita come visto già in Fig.4.11. Quindi per poter richiamare l'oggetto corrispondente all'identificatore `objID`, è necessario dapprima effettuare una

ricerca sulla mappa e solo successivamente è possibile accedere all'oggetto richiesto. Il metodo `getVertexPosition()` richiamato sull'oggetto RObject fornirà come output un `NxVec3` contenente le coordinate del vertice `vertID`-iesimo.

Seguendo a ritroso il percorso appena tracciato si noterà come la struttura NxVec3 venga scomposta in un semplice array di float all'interno del ControllerCaller in modo da poter serializzare le informazioni tra l'applicazione e l'interfaccia. Viceversa accade se vi è la necessità di convertire dati *grezzi* ricevuti dall'interfaccia, in strutture utilizzate all'interno del Controller.

4.3.3 La gestione degli eventi

In occasione dell'analisi dell'interfaccia di TSim-X Editor era stato fatto presente come la libreria PhysX® presentasse delle limitazioni per quanto riguarda la gestione degli eventi. Il motore fisico infatti viene fatto avanzare di uno step di simulazione la volta in corrispondenza di ogni passata di rendering, in modo da avere a disposizione i dati calcolati dalla simulazione appena prima che gli oggetti vengano disegnati su schermo.

Questo approccio deve scontrarsi con un problema di fondo: il motore grafico aggiorna la scena *appena può*; in pratica, non appena la scheda grafica avrà finito di visualizzare un frame della scena, inizierà immediatamente a processare il frame successivo. Gli eventi generati dal movimento del mouse o della pressione di un tasto dell'interfaccia, tuttavia, non sono sincronizzati con il rendering. Sarebbe come chiedere all'utente di sincronizzarsi con la frequenza di aggiornamento della scena (*Devo inserire il dito nella porta USB per la sincronizzazione..?*).

Se a questo si somma l'incertezza relativa a quando gli eventi generati dall'interfaccia saranno effettivamente processati dal sistema operativo, è facile capire come mai gli eventi avvengano in maniera completamente scorrelata

dal ciclo di rendering dell'applicazione. Inoltre può capitare che, durante il movimento del mouse in corrispondenza dell'operazione di trascinamento di una tenda, vengano lanciati più eventi `OnMouseMove` in rapida successione.

Un mouse, collegato tramite porta USB al computer, è in grado di fornire dati ogni 8ms (125Hz di *polling rate*). Ci sono alcuni mouse per *Gamers* che sono addirittura in grado di comunicare un movimento ogni 1ms (1000Hz di polling rate). Fortunatamente questi sono i casi *peggiori* o migliori, dipende dal punto di vista; in ogni caso, il sistema operativo potrebbe ritrovarsi a gestire un evento `OnMouseMove` ogni 8ms. Come descritto nel primo capitolo, un programma di grafica real-time come TSim-X produce un fotogramma ogni 30ms, circa. Quindi fra un ciclo di rendering ed il successivo potrebbero arrivare al Controller, e quindi alla PhysX® , fino a tre eventi in sequenza.

Le analisi teoriche di questo comportamento sono state successivamente verificate attraverso output di debug su file, anche se in genere gli eventi contemporanei erano al massimo due ($\simeq 30\%$), con picchi di tre estremamente rari ($< 5\%$).

Quando la libreria PhysX® riceveva due o tre eventi contemporaneamente, tuttavia, processava solamente l'ultimo, segno di come la libreria non gestisca una coda di eventi. Il motivo è facilmente intuibile: se si chiede alla libreria di simulare uno spostamento, questa semplicemente lo simula. Tutto ciò che riguarda gli eventi deve essere gestito all'esterno dato che non è un problema che riguarda direttamente la fisica, quanto piuttosto l'interfaccia. Inoltre, essendo la libreria multipiattaforma, il problema menzionato potrebbe non manifestarsi su tutte le piattaforme supportate.

Per risolvere l'inconveniente è stato necessario definire una lista di eventi da processare prima di lanciare la simulazione fisica. Gli eventi che popoleranno la lista pertanto non dovranno più richiamare direttamente l'oggetto interessato dall'evento ma dovranno utilizzare il metodo `enqueueEvent()`:

```
void Controller::enqueueEvent(
          eventEnum type,
          int objID,
          NxMat33 data)
{
   eventData ed;
   ed.type = type;
   ed.objID = objID;
   ed.data = data;

   eventQueue.push_back(ed);
}
```

Per maggiori informazioni sulla struttura della coda `eventQueue` si rimanda al diagramma di Fig.4.11. La coda di eventi così popolata verrà processata prima dell'inizio della simulazione: tutti gli eventi con medesimo `objID` e `type` verranno compattati sommando fra loro i campi `data`. Così facendo, ad esempio, invece di spostare un oggetto di `vec2(0.23, -0.45)` con il primo evento e di `vec2(0.10, 0.15)` con il secondo, si avrà un unico spostamento di `vec2(0.33, -0.30)`, risolvendo di fatto il problema della gestione degli eventi.

4.3.4 Il ciclo di rendering

Il Controller, come detto nelle precedenti sezioni, coordina il rendering degli oggetti, definendo l'ordine con cui questi vengono calcolati e abilitando l'utilizzo di shaders e framebuffer object in base alla qualità dell'illuminazione scelta.

Le operazioni di rendering avvengono quindi secondo un ordine ben preciso, in cui si sceglie un *rendering path* piuttosto che un altro in base alle

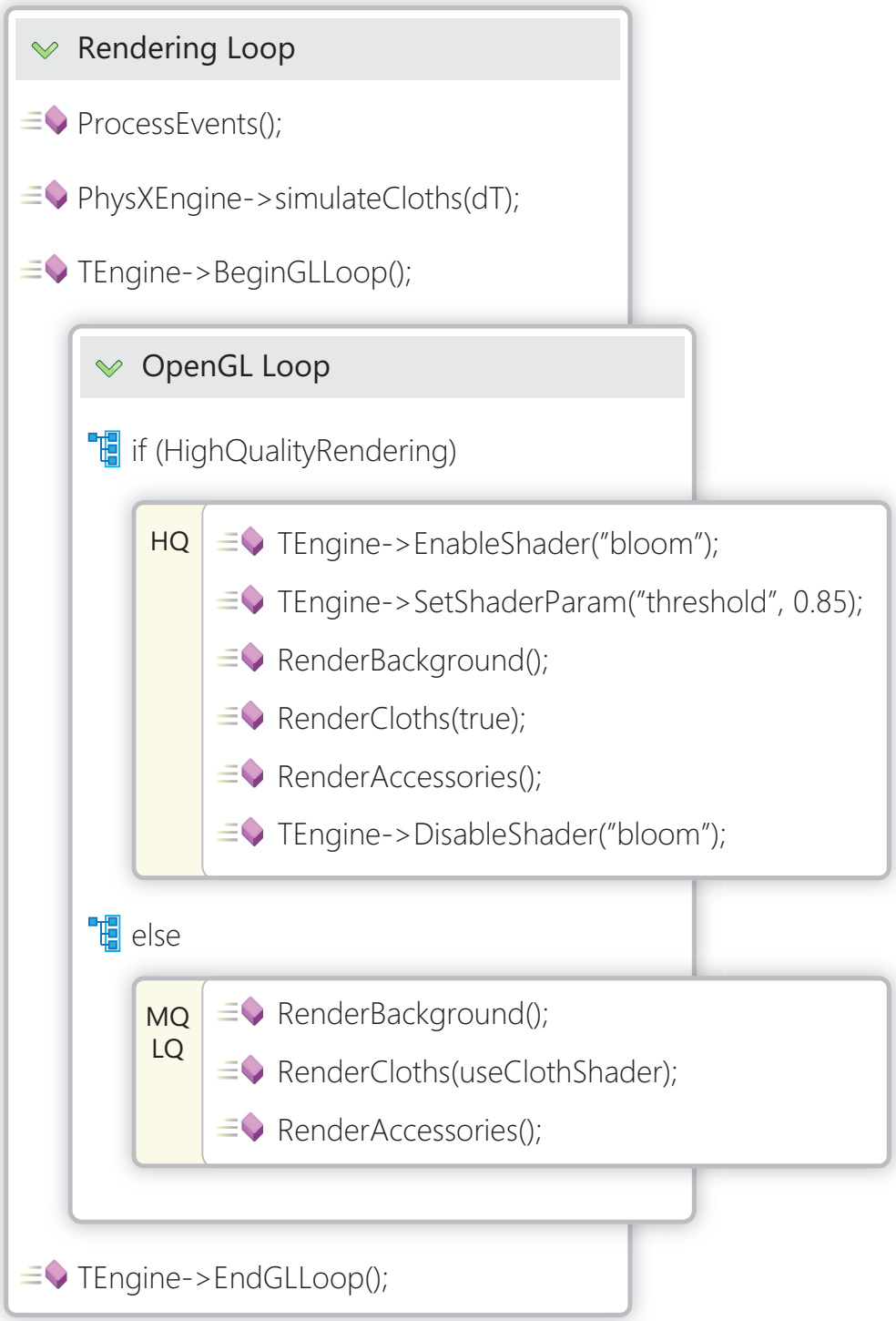


Figura 4.12: *Uno schema a grandi linee che mostra l'ordine con cui vengono eseguite le operazioni di rendering nel metodo renderAll().*

opzioni selezionate dall'utente e in base alle capacità della macchina. In questo caso i rendering path sono soltanto tre, ma in grossi progetti real-time, come è il caso dei videogames, possono arrivare anche a 10 o più.

Si descrivono più in dettaglio le operazioni di rendering sintetizzate in Fig.4.12:

1. come spiegato nel capitolo 4.3.3 la prima operazione da effettuare nel rendering loop è l'elaborazione degli eventi;

2. il motore fisico calcolerà gli step di rilassamento per un tempo massimo specificato da `dT`. Nel caso di TSim-X questo valore è pari a 16ms permettendo un movimento dei tessuti molto fluido (fino a 60 step al secondo);

3. si chiede al TEngine di inizializzare la fase di rendering: viene pulito il framebuffer, vengono preparati gli eventuali framebuffer objects, si impostano le luci e la posizione della camera;

4. se è stata selezionata la modalità di rendering ad alta qualità, si abilita il bloom shader impostando il *threshold* di livello al 85% ;

5. si renderizzano in ordine: sfondo, tende e infine gli accessori. In particolare, l'ultimo accessorio ad essere disegnato è la linea del pavimento;

6. infine si disabilita il bloom shader per non interferire con altre operazioni di rendering (se ad esempio si deve salvare la scena come bitmap);

7. se invece non è stata impostata la modalità di rendering ad alta qualità si renderizzeranno semplicemente gli oggetti nello stesso ordine precedente. Nel caso delle tende, si specifica se utilizzare o meno il cloth shader attraverso un booleano.

8. nell'ultima fase di rendering si pulisce lo stato della macchina OpenGL per poter essere riutilizzata correttamente nel successivo frame.

4.4 Il motore di rendering - TEngine

Fin'ora si è parlato diverse volte del motore di rendering TEngine. Come già anticipato nei precedenti capitoli, il motore è piuttosto semplice, supporta shader multipasso e l'utilizzo dei framebuffer objects per velocizzare alcune operazioni di rendering. Il sistema di illuminazione di default è quello dell'OpenGL, lasciando che sia ciascun RObject a dichiarare ed implementare eventuali altri metodi di illuminazione. Infine, il TEngine permette di effettuare il rendering della scena su più display contemporaneamente, con scarso impatto sulle performance generali dell'applicazione. Questo è reso possibile renderizzando la scena, alla massima risoluzione fra quella dei monitor collegati al computer, su di una texture: in questo modo, per ciascun display collegato, si renderizzerà un semplice rettangolo che copra per intero l'area di rendering con associata la texture contenente la scena precedentemente calcolata. In questo modo l'*overhead* è minimo.

Il TEngine prende il nome dall'omonima classe che ne implementa le funzioni di base. Uno schema della classe è riportato in Fig.4.14.

La classe definisce una struttura chiamata `Globals` che memorizza le imformazioni associate a ciascun display su cui il rendering avrà luogo. Per il caching delle textures discusso nel capitolo 4.2.1 quando si è analizzata la schermata di opzioni del programma, la classe TEngine definisce una mappa di associazione. In essa si correlano l'hash calcolato sul percorso del file ed un intero, generato dall'OpenGL quando si carica in memoria la texture per la prima volta. In questo modo, caricando per la seconda volta la stessa texture, si avrà un hash generato identico ad uno già presente nella mappa: TEngine non caricherà nuovamente il file ma semplicemente fornirà come valore di

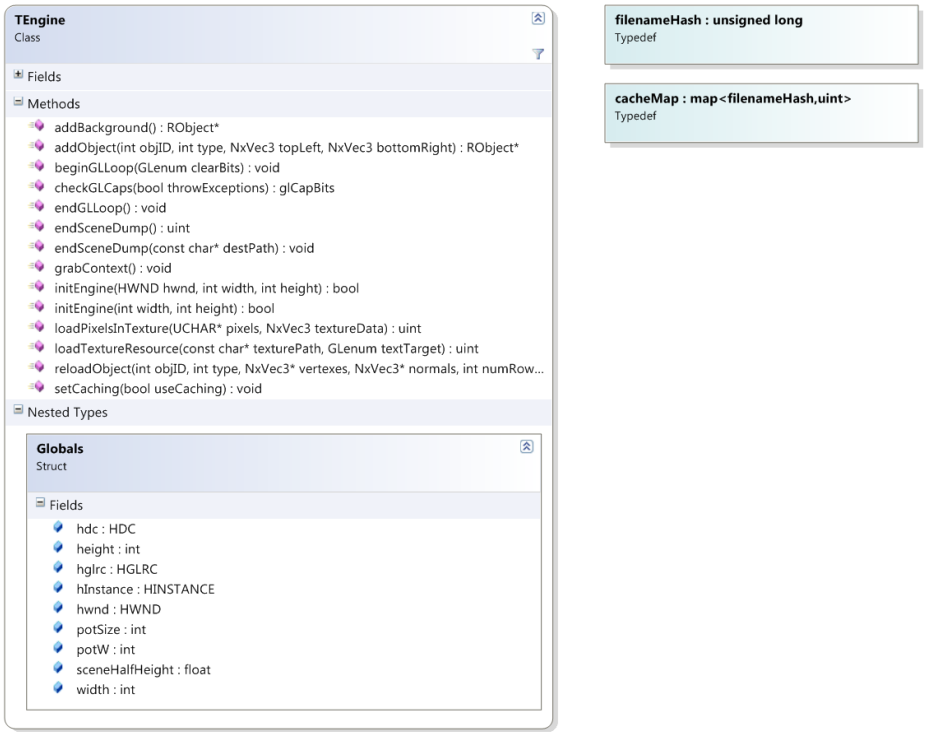


Figura 4.13: *Lo schema UML della classe TEngine e delle strutture ad esso associate.*

ritorno lo stesso ID già generato in passato, rendendo il meccanismo del caching totalmente trasparente al resto del programma.

Come già anticipato nella precedente sezione, sono definiti anche i metodi `beginGLLoop()` e `endGLLoop()` che hanno il compito di inizializzare e finalizzare le fasi di rendering. In particolare, `beginGLLoop()` pulisce i buffer dell'OpenGL (*framebuffer* e *z-buffer*), imposta l'illuminazione e definisce la posizione della camera; `endGLLoop()` invece si preoccupa esclusivamente di finalizzare il rendering inviando il comando `wglSwapBuffer()` all'OpenGL.

La classe TEngine implementa infine il metodo `addObject` che permette di creare un RObject, specificando come parametri del metodo le dimensioni volute. L'RObject creato può essere attualmente o un RGenericObject o un RFloorObject. Per maggiori chiarimenti su questi oggetti si rimanda alla

prossima sezione.

4.5 Gli oggetti renderizzabili - RObject

In TSim, la classe `RObject` definisce sostanzialmente qualsiasi tipo di oggetto renderizzabile. Lo sfondo, il pavimento, gli accessori e le tende sono tutte istanze della classe astratta RObject. Essa implementa l'interfaccia `ISavable`, stando ad indicare che qualunque oggetto derivato da RObject sarà salvabile su file: l'interfaccia richiede che la classe derivata implementi un metodo chiamato `marshalObj()` che esporti le informazioni necessarie a TSim per ricreare l'oggetto in un secondo momento.

RObject dichiara una serie di metodi: per alcuni di essi è presente una definizione nel file *.cpp* associato alla classe, mentre altri sono lasciati astratti in modo da indicare al programmatore quali metodi possono avere una propria implementazione specifica solo nella classe derivata. Si pensi a titolo di esempio al metodo `render()`. La classe astratta RObject non può in alcun modo sapere come implementarlo se non si è prima precisato che tipo di oggetto sarà *realmente*.

Tutti i metodi *setter* dichiarati in RObject sono stati già definiti, in quanto non fanno altro che impostare il valore di un campo ad un certo valore. Lo stesso si può dire anche dei metodi *getter*, tuttavia con un eccezione: il metodo `getVertexPosition()` deve essere necessariamente implementato dalla classe derivata, in quanto la struttura dati che contiene i vertici differisce da quella utilizzata nelle altre classi.

Vengono infine forniti tre metodi per la conversione a run-time di un generico `RObject*` nel proprio tipo derivato. Questi metodi sono `isCloth()`, `isFloor()` e `isGeneric()` e ritornano tutti come valore un puntatore a RObject.

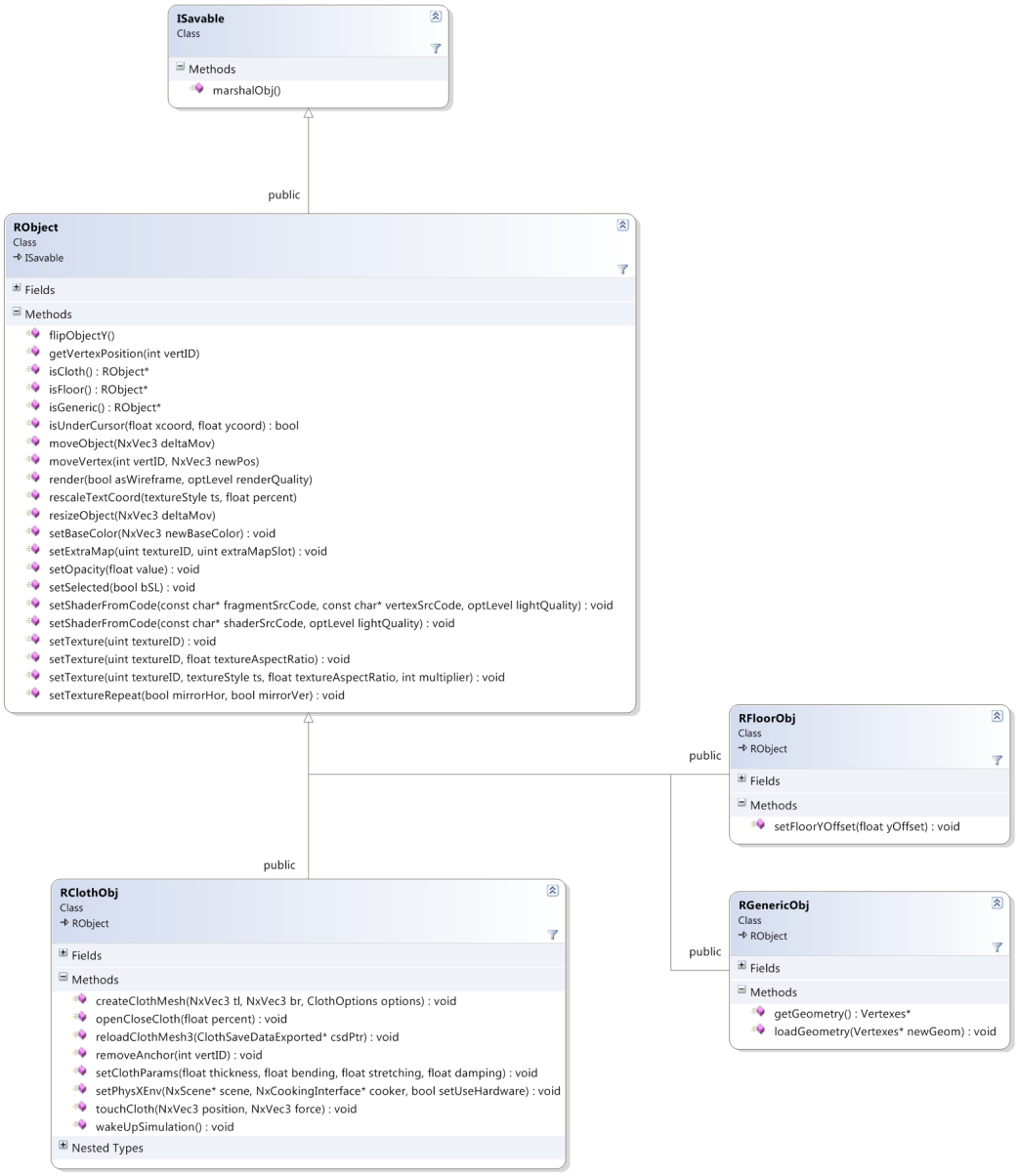


Figura 4.14: *Lo schema UML della classe astratta RObject, dell'interfaccia ISavable e delle classi derivate RClothObj, RGenericObj e RFloorObj.*

Poniamo ad esempio il seguente caso:

```
RClothObj* myClothObj = new RClothObj();
RObject* tempObject = myClothObj->isCloth();
```

Il puntatore a `RObject` identificato da `tempObject` avrà come valore la locazione di memoria in cui l'oggetto `myClothObj` è stato allocato in quanto il suo tipo derivato è proprio RClothObj. Se invece poniamo quest'altro caso:

```
RClothObj* myClothObj = new RClothObj();
RObject* tempObject = myClothObj->isGeneric();
```

Si sta chiedendo all'oggetto `myClothObj` se è di tipo `RGenericObj`: non essendo questo il suo vero tipo, la funzione ritorna un valore `NULL`. Grazie a questo meccanismo è possibile verificare a run-time il tipo dell'oggetto ed effettuare quindi un *Cast* in maniera totalmente sicura. Il cast va effettuato utilizzando l'operatore `dynamic_cast<type>`:

```
RClothObj* myClothObj = new RClothObj();
RObject* tempObject = myClothObj->isCloth();


if (tempObject != NULL)
   RClothObj* newCloth = dynamic_cast<RClothObj*>(tempObject);
```

In questo modo `tempObject` viene convertito con successo da `RObject*` a `RClothObj*`.

4.5.1 RClothObject

La più utilizzata delle classi derivate da RObject è sicuramente la RClothObj. Rispetto alle altre classi derivate è anche la più complessa in quanto deve supportare una forma di interazione che va oltre lo spostamento o il ribaltamento: a differenza degli altri oggetti renderizzabili, un RClothObj supporta

il *drappeggio* con cui si possono spostare ed ancorare i singoli vertici che compongono la mesh tridimensionale. Versione alternativa del drappeggio è il *tocco*, introdotta a seguito dell'implementazione della seconda interfaccia grafica, consente di applicare una forza direzionale ad un intorno del punto toccato, simulato tutto in tempo reale dalla fisica: in questo modo si rende possibile un interazione diversa e più realistica con le tende, anche grazie alle tecnologie touchscreen discusse alla fine di questo capitolo. Altra possibilità di interazione con le tende è data dal metodo `openCloseCloth()` che permette di impostare il livello di apertura per le tende di tipo *Rullo* e *Pacchetto* mostrate in Fig.4.6.

Una volta istanziato l'oggetto, RClothObj richiama il metodo `createClothMesh()` che, con una serie di chiamate alla libreria PhysX® , crea ed inizializza la geometria della mesh tridimensionale, definisce le coordinate texture ed imposta i parametri di simulazione di default. Successivamente l'utente, mediante lo slider *Grammatura* posto sull'interfaccia, potrà modificare i parametri della tenda in modo da renderla più o meno pesante, in base al tipo di simulazione che si vuole raggiungere.

4.5.2 RGenericObject

La seconda classe derivata più utilizzata subito dopo la RClothObj. In questo caso le operazioni possibi con cui interagire con l'RGenericObj calano, limitandosi esclusivamente allo spostamento ed al ribaltamento lungo l'asse verticale.

La geometria della mesh associata a ciascun RGenericObj viene definita mediante un array di vertici generati manualmente nel codice o avvalendosi della classe d'utility `GeometryGen` che può generare attualmente solo rettangoli bidimensionali. Per ciascun vertice sono definite le informazioni sulla

posizione spaziale, sulle normali e sulle coordinate texture, oltre che alcuni flag booleani di controllo.

4.5.3 RFloorObject

L'ultima delle tre classi derivate da RObject è la RFloorObj. La classe dichiara un solo metodo oltre quelli ereditati ed è rappresentata graficamente da una semplice linea nera con bordo bianco.

Il pavimento dal punto di vista fisico è un piano perpendicolare all'asse verticale che taglia quindi la scena in orizzontale. La fisica simulerà le interazioni tra il piano e le tende presenti fornendo quindi uno strumento per impedire che le tende scendano al di sotto di una soglia. Gli sfondi utilizzati in TSim non presentano uno standard per quanto riguarda l'altezza del pavimento rispetto alla scena, si rende pertanto necessario questo meccanismo per rendere immediata la regolazione del piano orizzontale. L'unico metodo che dichiara RFloorObj, oltre quelli ereditati, è `setFloorYOffset()` che permette quindi di regolare in altezza il piano.

4.6 Periferiche di input alternativo

Come anticipato all'inizio del capitolo, durante la fase di definizione dei requisiti per l'interfaccia Touchscreen, è stata richiesta la possibilità di interagire con il programma mediante tag RFID e monitor touchscreen. In quest'ultima sezione del capitolo verranno analizzate queste due tecnologie di imput alternativo

4.6.1 RFID per la selezione interattiva di tessuti

L'acronimo RFID sta per *Radio Frequency IDentification*: è una tecnologia per l'identificazione automatica di oggetti, animali o persone basata sulla

capacità di memorizzare e accedere a distanza a tali dati usando dispositivi elettronici, chiamati *transponder*, che sono in grado di rispondere comunicando le informazioni in essi contenute quando interrogati. In questo modo è possibile effettuare una lettura remota senza fili.

Applicando un piccolo talloncino, chiamato TAG, ad un oggetto è possibile creare una corrispondenza fra l'oggetto reale ed un codice digitale.

Nel caso particolare di TSim, i TAG RFID sono stati applicati ai tessuti di un campionario ed è quindi stata creata un associazione fra TAG e texture corrispondente al tessuto reale: avvicinando il tessuto con il TAG al lettore, è possibile cambiare istantaneamente la texture di tutte le tende presenti nella scena.

Pur essendo una feature perfettamente funzionante in TSim-XT, l'implementazione della tecnologia RFID è tuttavia ancora in fase sperimentale. Al fine di proporre un programma pilota al cliente, si è scelto di implementare un lettore economico che fornisse esclusivamente questo tipo di funzionalità, con delle API piuttosto scarne di documentazione e nessun supporto post-vendita.

A titolo di completezza, tuttavia, si descrive la modalità con cui il lettore RFID interagisce con il software. Analogamente con quanto fatto con il ControllerLayer, l'interfaccia grafica per poter comunicare con la *.dll* del lettore RFID necessita di un wrapper che importi le chiamate della libreria e le renda accessibili all'interno del codice. In questo caso le chiamate importate sono soltanto tre:

```
DllImport("MF_API.dll", CharSet = CharSet.Unicode)]
static private extern int MF_InitComm(
          byte[] portname,
          long baud );
```

```
[DllImport("MF_API.dll", CharSet = CharSet.Unicode)]
static private extern int MF_Request(
          int DeviceAddr,
          char mode,
          byte[] alarm );

[DllImport("MF_API.dll", CharSet = CharSet.Unicode)]
static private extern int MF_Anticoll(
          int DeviceAddr,
          byte[] snr );
```

Il metodo `MF_InitComm()` viene utilizzato per instaurare una connessione con il lettore collegato tramite porta USB. Se il valore di ritorno della funzione è pari a zero, la connessione è stata stabilita con successo. A questo punto è sufficiente effettuare una chiamata a `MF_Request()` che verificherà l'eventuale presenza di un TAG nelle vicinanze del lettore RFID: analogamente al primo metodo, se il valore di ritorno della funzione è pari a zero, il lettore sta comunicando al programma che un TAG è in attesa di essere letto. Chiamando infine `MF_Anticoll()` si potrà leggere l'identificatore univoco del TAG, memorizzato nel parametro `byte[] snr` del metodo.

Sulla base degli ID associati a ciascun TAG è stato possibile definire una mappa di corrispondenza: da un lato è presente l'identificatore del TAG; dall'altro il percorso su disco della texture associata al tessuto reale.

Di conseguenza, avvicinando un tessuto con TAG al lettore, si genererà una corrispondenza tra TAG e tessuto, sulla base della quale TSim-XT provvederà ad aggiornare la texture di ciascuna tenda presente in scena.

4.6.2 Touchscreen

Il mercato dei monitor cosiddetti *Touchscreen* sta vivendo attualmente un periodo fiorente: le proposte di computer notebook e desktop previsti di monitor touchscreen stanno aumentando e, dall'altra parte, il mercato, sebbene con qualche perplessità, sta accettando sempre più calorosamente questa novità.

Un touchscreen, o schermo tattile, è un particolare dispositivo, frutto dell'unione di uno schermo ed un digitalizzatore, che permette all'utente di interagire con una interfaccia grafica mediante le dita o particolari oggetti.

Dal punto di vista tecnologico non esiste un solo metodo per realizzare il digitalizzatore, ma una moltitudine. I primi schermi tattili usavano raggi di luce infrarossa proiettati secondo una disposizione a griglia immediatamente sopra la superficie dello schermo; appoggiando il dito allo schermo l'utente interrompe alcuni fasci orizzontali e alcuni fasci verticali, consentendo così l'identificazione delle coordinate a cui è avvenuto il *contatto*.

Il digitalizzatore di tipo **resistivo**, presente nella maggior parte dei dispositivi moderni, è composto invece da due strati di materiale conduttivo che, nel momento in cui un oggetto viene premuto sullo schermo, entrano in contatto permettendo al dispositivo di determinare la posizione dell'oggetto.

Il digitalizzatore **capacitivo**, presente ad esempio su molti smartphone, sfrutta la variazione di capacità dielettrica tipica dei condensatori sul vetro del telefono stesso, che viene ricoperto da un sottile strato di ossido metallico sulla parte esterna. Ai quattro angoli del pannello viene applicata una tensione che si propaga uniforme su tutta la superficie dello schermo per via dell'ossido di metallo; quando il dito o un materiale conduttore di elettricità tocca lo schermo avviene una variazione di capacità superficiale, che viene letta da una matrice di condensatori a film posizionati su un pannello posto sotto la superficie del vetro.

Ogni tecnologia presenta dei vantaggi e degli svantaggi che ne decidono il miglior campo d'utilizzo. In genere, un utente inesperto, quando messo a diretto confronto con uno schermo touchscreen, si aspetta una risposta dallo schermo non appena questo viene sfiorato. Se invece il display richiede una pressione per funzionare, l'utente potrebbe percepire il tutto come troppo macchinoso e lento da utilizzare. In virtù di quanto appena detto, la tecnologia preferita per l'utilizzo con TSim-XT è sicuramente quella capacitiva, che permette inoltre di avere uno strato di vetro temprato a diretto contatto con l'esterno, garantendo quindi anche una certa resistenza all'usura.

Uno schermo touchscreen, da parte dei software, agisce esattamente come un mouse: toccando un punto dello schermo equivale a spostare il mouse in quello stesso punto e fare click con il tasto sinistro. Le operazioni di trascinamento diventano quindi estremamente comode ed intuitive, perdendo tuttavia gran parte della precisione possibile attraverso il mouse. Per sfruttare i vantaggi di usabilità forniti dal touchscreen e limitare nel contempo gli svantaggi di precisione, è stato necessario rivedere le modalità con cui interagire con i tessuti.

Come anticipato in precedenza, è possibile interagire con una tenda in vario modo, fra cui il *tocco*. Per TSim-XT *toccare* un punto dello schermo equivale a generare due vettori tridimensionali: nel primo sarà presente la posizione attuale del dito, mentre nel secondo il vettore secondo cui la *forza* agisce. Quindi, ogni volta che sullo schermo verrà spostato il dito, TSim-XT provvederà ad aggiornare questi valori ed inviarli al Controller: quest'ultimo richiamerà il metodo `touchCloth()` della tenda toccata, la quale a sua volta richiamerà la libreria PhysX® :

```
cloth->addDirectedForceAtPos(
          position,
          force * touchMagnitude,
```

```
touchWidth );
```

Come evidenziato, posizione e forza non sono gli unici due parametri necessari a definire uno spostamento. Le variabili `touchMagnitude` e `touchWidth` permettono rispettivamente di potenziare o ridurre la forza con cui interagire con la tenda ed il raggio d'azione di questa forza, in modo da poter coinvolgere più vertici in un intorno del punto toccato, simulando di fatti la larghezza del dito.

In questo caso, tuttavia, l'operazione di *tocco* sostituisce la classica operazione di drappeggio. Non sarà possibile infatti definire nuove ancore con cui modellare la tenda: l'idea del tocco è quella di rendere la scena più interattiva e divertente. Con l'Editor sarà possibile creare scene e definire tutto nei minimi particolari, mentre con la versione Touch si darà modo all'utente di poter cambiare immediatamente il tessuto attraverso l'uso di TAG RFID e di interagire liberamente mediante monitor touchscreen, senza dover perdere tempo nella creazione dell'ambiente.

Capitolo 5

Analisi di usabilità

In quest'ultimo capitolo si analizzeranno i comportamenti di un utente inesperto durante l'utilizzo del software realizzato. La motivazione principale che ha spinto la stesura di questo report è stata la necessità di valutare quanto ed in che modo gli accorgimenti di usabilità studiati ed implementati nell'interfaccia grafica hanno reso l'esperienza utente positiva. In particolare verranno analizzati tre casi d'uso:

- reazione dell'utente al primo avvio dell'applicazione;
- tempo di completamento e analisi dei comportamenti del soggetto a fronte di una specifica richiesta
- tempo di completamento di specifici task e analisi dei comportamenti del soggetto dopo un breve periodo di apprendimento.

Il primo punto mira ad analizzare le reazioni emotive dell'utente inesperto quando messo a confronto l'applicazione: TSim-X viene eseguito in modo tale da caricare una scena predefinita in cui sono presenti tende e accessori; viene abilitato lo strumento di spostamento e si chiede all'utente di interagire con la scena. Non sono state date restrizioni sugli strumenti utilizzabili. In questo

modo si può analizzare per quanto tempo e in che modo l'utente è stato attratto dalla scena visualizzata prima di passare ad utilizzare gli strumenti messi a disposizione. La durata della prova sarà di 5 minuti.

Nel secondo punto invece sarà chiesto al soggetto di portare a compimento una serie di task nel minor tempo possibile. Saranno analizzati i tempi di completamento delle operazioni, le reazioni emotive suscitate durante il compimento di specifiche richieste e saranno riportate le opinioni del soggetto al termine della prova.

Infine, il terzo punto prevede che l'utente venga *istruito* sull'utilizzo dell'applicazione per poi ri-eseguire il test del punto precedente. L'apprendimento dell'utente avviene in due fasi: nella prima, sarà fornito il manuale dell'applicazione per creare nel soggetto una consapevolezza degli strumenti presenti nell'applicazione; la seconda fase invece prevede un tutorial dal vivo in cui un utente esperto spiega e fornisce consigli sull'utilizzo ottimale del software. Si ripete quindi il test effettuato nel secondo punto in modo da valutare se l'utente preferisce ripercorrere la strada fatta in precedenza o ottimizza le operazioni sfruttando l'esperienza maturata nell'apprendimento. Confrontando i risultati della seconda e terza fase è possibile delineare quanto l'interfaccia risulta intuitiva e permetta ad un utente inesperto di essere efficiente nell'uso del software già nelle prime fasi di utilizzo.

5.1 Primo impatto con l'applicazione

La scena caricata prevede due tende piegate e drappeggiate verso l'esterno in modo da lasciar vedere la finestra posta sullo sfondo; nella parte superiore è posta una terza tenda piegata, molto larga e corta, chiamata in gergo *calata*. Due oggetti bidimensionali con la texture di un fiocco sono stati posti sulla scena, ciascuno su una delle due tende drappeggiate. Una timeline riassuntiva del risultato del test è mostrata in Fig.5.1.

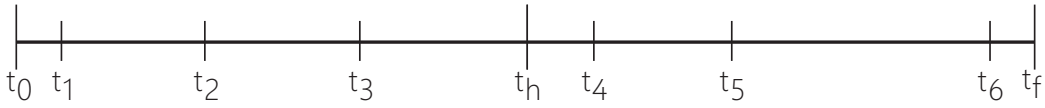


Figura 5.1: *La timeline relativa al primo test di usabilità. In corrispondenza di ciascun t_x l'utente inizia una nuova attività.*

Si riassume di seguito il risultato del test:

0:00 - 0:09 (t_1) L'utente prende confidenza con l'interfaccia, guardandola solamente. Prende consapevolezza di come gli strumenti sono stati disposti e di come la scena è visualizzata.

0:09 - 1:02 (t_2) Interessante è stato notare come la prima operazione dell'utilizzatore sia stata quella di creare una nuova scena, cliccando sul pulsante posto in basso a sinistra. Quindi inizia la fase di creazione degli oggetti: naturalmente è la prima operazione che il soggetto esegue in quanto il programma, una volta lanciato, è in modalità creazione. Il primo oggetto creato è un accessorio con la texture di un nastrino. L'operazione di selezione della texture è immediata, grazie alla schermata di sistema per la selezione dell'immagine che l'utente dichiara di conoscere già. La prima creazione dell'accessorio non va come l'utente desiderava: una volta scelta la texture, l'utente è stato per qualche istante fermo, forse in attesa che il programma disegnasse l'oggetto per lui. L'utente tuttavia, non appena ha notato che la forma del puntatore del mouse è cambiata in una croce ha intuito che avrebbe dovuto tracciare un'area o un segmento. L'operazione avviene senza un preciso movimento, risultando in un'area molto grande. Dopo questo primo errore l'utente dichiara di aver capito come tracciare correttamente gli oggetti e dimostra la volontà di cancellare quanto appena disegnato: dopo un rapido sguardo alla parte sinistra dello schermo, ha spostato con decisione il puntatore sulla modalità Modifica, cliccato e quindi eli-

minato l'oggetto senza difficoltà. Con sicurezza ha quindi ridisegnato l'accessorio con dimensioni ben precise.

1.02 - 1:51 (t_3) Il secondo oggetto disegnato è stata una tenda, in cui l'utente non si è soffermato molto ad osservare la finestra di opzioni ma ha dato subito conferma per poter passare alla fase di disegno. Ha quindi interagito con essa mediante l'operazione di spostamento.

1:51 - 2:47 (t_4) Tenendo selezionata la tenda, il soggetto ha iniziato l'interazione con la parte destra del programma, quella relativa alle opzioni dell'oggetto selezionato. Ha dapprima cliccato sull'area celeste vuota, avendo capito che tramite quell'area era possibile cambiare tessuto. L'operazione è stata eseguita due volte di seguito. In ordine, dall'alto verso il basso, l'utilizzatore ha iniziato ad interagire con ciascuno dei controlli presenti in modo da rendersi conto direttamente delle possibilità di intervento sulla tenda.

2:47 - 3:25 (t_5) Una volta terminata la fase di modifica delle opzioni della tenda, l'utente ha provveduto a cambiare sfondo alla scena, scorrendone in totale 6 diversi e rendendosi conto di come le dimensioni dell'area di lavoro vengono modificate in base allo sfondo selezionato.

3:25 - 4:42 (t_6) Successivamente l'utilizzatore ha ricominciato con la fase di creazione tende, soffermandosi per qualche istante sulla finestra di opzioni: è stata cambiata la tipologia della tenda in *Pacchetto* ed è stata assegnata una texture direttamente dalla schermata di opzioni. In questo modo, disegnata la tenda, l'utente ha notato come questa fosse già dotata di texture proprio in virtù della sua scelta nella schermata di creazione. In questa fase l'utente alterna interazione con gli oggetti mediante spostamento e modifica dei parametri, in particolare quello di *Apertura–Chiusura* per la tenda a Pacchetto.

4:42 - 5:00 (t_f) Negli ultimi secondi di test l'utilizzatore ha iniziato a prendere familiarità con i comandi per il raggruppamento di oggetti, leggendo i suggerimenti che appaiono dopo aver lasciato il cursore del mouse per qualche secondo sul controllo.

Terminato il test, è stato chiesto all'utente come mai, nei primi istanti di utilizzo del programma, abbia voluto cancellare la scena caricata per avere a disposizione una nuova area di lavoro. La risposta è stata tutto sommato intuibile: l'utilizzatore aveva necessità di prendere confidenza con l'interfaccia e con gli strumenti a disposizione, pertanto la prima cosa che ha voluto fare è stata quella di pulire la scena per cominciare da zero.

Analizzando questo comportamento si evince come l'interfaccia abbia trasmesso subito sicurezza e ordine, anche ad un utente inesperto. Normalmente, quando messo di fronte ad un software complesso e ricco di opzioni, l'utente evita di doversi scontrare con tali opzioni ed è pertanto invogliato maggiormente ad interagire con l'area di lavoro. Nel caso studiato, invece, è avvenuto l'esatto contrario.

Chiedendo direttamente all'utilizzatore un parere sulla fruibilità dell'interfaccia grafica, questi ha portato l'attenzione di come abbia trovato di grande utilità i suggerimenti a comparsa sui controlli dell'interfaccia; ha tuttavia lamentato di come all'inizio abbia avuto qualche difficoltà nel capire come avviene la creazione di oggetti. Ha tuttavia ammesso che dopo il primo tentativo mal riuscito, abbia subito capito come disegnare gli oggetti tramite trascinamento dell'area.

5.2 Comportamento dell'utente inesperto di fronte ad una richiesta esplicita

Il secondo test invece chiedeva all'utente di realizzare una scena quanto più simile possibile a quella mostratagli all'inizio, durante il primo test. Per semplicità, gli è stata fornita una stampa del risultato finale che avrebbe dovuto ottenere. Inoltre, avrebbe dovuto produrre un totale di 3 files:

- Nel primo si sarebbe dovuta salvare la tenda ed il fiocco posti sul lato sinistro della finestra
- Nel secondo la tenda ed il fiocco sul lato destro
- Nel terzo invece la scena completa

Il test viene svolto 15 minuti dopo il primo ed avviene come segue:

0:00 - 0:15 Scelta dello sfondo.

0:15 - 0:43 Disegno della tenda superiore: il tessuto è stato scelto durante la fase di creazione.

0:43 - 2:10 Disegno della tenda sinistra, drappeggio ed applicazione del tessuto in fase di creazione.

2:10 - 2:25 Duplicazione e ribaltamento della tenda

2:25 - 2:40 Creazione del fiocco come oggetto bidimensionale e duplicazione in modo da completare la scena.

2:40 - 2:44 Salvataggio della scena completa (**obiettivo 3 completato**).

2:44 - 3:18 Creazione di un nuovo documento; successivamente è stato ricaricato il file salvato al passo precedente; sono state quindi eliminate le tende superiore e destra, unitamente al fiocco destro.

3:18 - 3:22 Salvataggio della scena (**obiettivo 1 completato**).

3:22 - 3:49 Creazione di un nuovo documento; è stato caricato nuovamente il documento contenente la scena completa e sono stati eliminati gli elementi non richiesti nel punto 2.

3:49 - 3:53 Salvataggio della scena (**obiettivo 2 completato**).

Il soggetto si è dimostrato sicuro nelle operazioni durante il loro compimento. Secondo l'utente, il primo test è stato sufficiente ad apprendere le meccaniche base dell'applicazione. Il task più impegnativo è stato tuttavia l'operazione di drappeggio: se da un lato è stato immediato per l'utilizzatore capire dove lo strumento fosse stato posizionato nell'interfaccia, dall'altro è stato un po' macchinoso riuscire ad ottenere una tenda drappeggiata in maniera similare a quanto visto nella scena d'esempio. In ogni caso, l'utente dichiara di non aver riscontrato particolari problemi nel portare a compimento le operazioni, anche se ammette che il dover caricare la scena completa per poi eliminare gli oggetti superflui gli ha causato rallentamenti.

5.3 Comportamento dell'utente dopo un periodo di apprendimento

Al termine del precedente test, viene consegnato al soggetto il manuale del programma e 10 minuti di pausa nella quale avrebbe avuto la possibilità di studiarlo. Nei restanti 5 minuti di intervallo fra un test ed il successivo, l'autore di questo documento di tesi provvede ad istruire l'utente sulle caratteristiche del software che ancora non ha avuto modo di utilizzare.

Viene dunque avviato il terzo ed ultimo test di usabilità:

0:00 - 0:09 Scelta dello sfondo.

0:09 - 1:24 Disegno della tenda sinistra, drappeggio ed applicazione del tessuto in fase di creazione.

1:24 - 1:38 Creazione del fiocco sinistro come oggetto bidimensionale.

1:38 - 1:42 Salvataggio della scena (**obiettivo 1 completato**).

1:42 - 2:01 Ribaltamento della tenda e riposizionamento degli elementi.

2:01 - 2:05 Salvataggio della scena (**obiettivo 2 completato**).

2:05 - 2:27 Importazione nel documento corrente del file prodotto per il primo obiettivo.

2:27 - 2:56 Creazione della tenda superiore con applicazione del tessuto in fase di creazione.

2:56 - 3:00 Salvataggio della scena (**obiettivo 3 completato**).

Come evidenziato, l'intera operazione ha richiesto circa un minuto in meno per essere portata a compimento. L'utente in particolare ha dimostrato particolare apprezzamento per la funzionalità di importazione della scena, appresa durante la lettura del manuale e successivamente illustrata nel tutorial. L'operazione di drappeggio è stata percepita, inoltre, come meno difficoltosa dopo che nel tutorial sono stati presentati una serie di accorgimenti per far si che il tessuto si deformi più realisticamente.

Prima di dichiarare conclusa la fase di testing, l'utente ha voluto esprimere il suo apprezzamento nei confronti del software realizzato, avendolo trovato semplice ma potente allo stesso tempo. Ha voluto far notare come sarebbe il caso di spiegare maggiormente alcune meccaniche del software proprio durante la fase di utilizzo. Se da un lato crediamo che quanto proposto possa sicuramente incontrare l'apprezzamento degli utilizzatori inesperti, dall'altro

reputiamo maggiormente opportuno lasciare che tali problematiche vengano affrontate nella guida del programma, per evitare che utilizzatori esperti debbano ritrovarsi i suggerimenti sull'uso dell'applicazione costantemente visualizzati.

Capitolo 6

Conclusioni ed evoluzioni future

Giunti alla conclusione del documento di tesi, emergono alcune considerazioni circa l'applicativo realizzato. Tali considerazioni sono legate a doppio filo con quelle che saranno le evoluzioni future del programma.

Una delle possibili evoluzioni infatti riguarderà la possibilità di creare oggetti tridimensionali con i quali far interagire i tessuti. Questo, tuttavia, potrebbe scontrarsi con la struttura analizzata nei capitoli precedenti. Spesso, per motivi di espandibilità e compatibilità, si realizzano gli oggetti tridimensionali come un insieme di mesh renderizzabili. Attualmente, invece, TSim prevede una semplificazione di questo modello, in cui ciascun oggetto nella scena è composto da un unico elemento visualizzabile, sè stesso. Per garantire spazio all'evoluzione del software, sarà necessario rivedere pesantemente la classe RObject, in modo da poter assegnargli un insieme di mesh generiche che saranno renderizzate contemporaneamente alla chiamata del metodo `RObject::render()`. Inoltre dovrà essere possibile accedere alle singole mesh, o fornire un meccanismo per farlo attraverso l'RObject, in modo da poterne modificare lo stato durante l'esecuzione del programma.

Un'altra considerazione che è stata maturata solo in tempi recenti riguarda la gestione degli shaders. Attualmente infatti ciascun RObject prevede che ogni shader ad esso associato duri soltanto una passata di rendering. Come analizzato in occasione del Bloom Shader, ci sono alcuni effetti che richiedono più passate di rendering per essere calcolati. Il motore di rendering deve sapere tuttavia quali sono gli effetti che devono essere calcolati e deve fare in modo da sincronizzarli in modo che il risultato finale dei singoli effetti venga generato correttamente. Pertanto, ci dovrà essere un gestore che conosce lo stato della scena e riordinerà il rendering degli elementi in base all'effetto desiderato.

Concludendo, vorrei far notare come l'applicativo fin qui discusso sia in sviluppo continuo da due anni e che conta oramai oltre *ventimila* righe di codice. Nel corso dei mesi molte cose nuove sono state apprese ed implementate per ottimizzare il funzionamento del programma. Attualmente il programma può considerarsi stabile e maturo, anche se ci sono ancora diversi spazi di miglioramento. Le sfide per il futuro del software realizzato si sposteranno sempre più sul lato dei servizi offerti nel pacchetto quanto piuttosto che verso nuove features. Il veicolo di questa evoluzione sarà il web, con il quale fornire aggiornamenti al software ed alla libreria di tessuti e accessori a cui il programma ha accesso.

Elenco delle figure

Bibliografia

[1] Bjarne Stroustrup. *C++ - Linguaggio, libreria standard, principi di programmazione.* Addison-Wesley Longman Italia Editoriale s.r.l., Milano, Italia, third edition, 2000.

[2] The C++ Resources Network. cplusplus.com - The C++ Resources Network.
`http://www.cplusplus.com/`.

[3] OpenGL Shading Language. OpenGL.org.
`http://www.opengl.org/documentation/glsl/`.

[4] General Purpose GPU. GPGPU.org.
`http://www.gpgpu.org/`.

[5] Edward Angel. *Interactive Computer Graphics, A Top-Down Approach Using OpenGL.* Pearson, Addison-Wesley, Boston, Massachusetts, third edition, 2003.

[6] OpenGL. OpenGL.org.
`http://www.opengl.org/`.

[7] RenderMonkey. AMD in collab. con 3Dlabs.
`http://ati.amd.com/developer/rendermonkey`.

[8] W3 Consortium. XML.
`http://www.w3.org/XML/`.

[9] Andrew S. Glassner. *An Introduction to Ray tracing (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, San Fransisco, USA, first edition, 1989.

[10] Claude Puech François Sillion. *Radiosity and Global Illumination (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, San Fransisco, USA, first edition, 1994.

[11] K. E. Torrance et al. C. Goral. Modeling the interaction of light between diffuse surfaces. In *In Computer Graphics' Proceedings of ACM SIGGRAPH 1984*, volume 18, pages 213–222, 1984.

[12] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley, first edition, 2004.

[13] Susan F. Marseken Lambert M. Surhone, Miriam T. Timpledon. *Stencil Buffer: Stencil, Computer Graphics, Hardware, Data Buffer, Z-buffering, Integer*. Betascript Publishing, first edition, 2010.

[14] Samuel R. Buss. *3D Computer Graphics: A Mathematical Introduction with OpenGL*. Cambridge University Press, first edition, 2003.

[15] James T. et al. Kajiya. The rendering equation. In *In Computer Graphics' Proceedings of ACM SIGGRAPH 1986*, volume 20(4), pages 143–150, 1986.

[16] Katja Daubert, Hendrik P. A. Lensch, Wolfgang Heidrich, and Hans-Peter Seidel. Efficient cloth modeling and rendering. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 63–70, London, UK, 2001. Springer-Verlag.

[17] Yanyun Chen, Stephen Lin, Hua Zhong, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Realistic rendering and animation of knitwear. *IEEE Transactions on Visualization and Computer Graphics*, 9:43–55, 2003.

[18] Stephen H. Westin, James R. Arvo, and Kenneth E. Torrance. Predicting reflectance functions from complex surfaces. *SIGGRAPH Comput. Graph.*, 26(2):255–264, 1992.

[19] Takami Yasuda, Shigeki Yokoi, and Jun ichiro Toriwaki. A shading model for cloth objects. *IEEE Computer Graphics and Applications*, 12:15–24, 1992.

[20] Michael Ashikmin, Simon Premože, and Peter Shirley. A microfacet-based brdf generator. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 65–74, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[21] Wolfgang Heidrich, Katja Daubert, Jan Kautz, and Hans-Peter Seidel. Illuminating micro geometry based on precomputed visibility. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 455–464, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[22] Peter-Pike Sloan Michael and Michael F. Cohen. Interactive horizon mapping. In *In Rendering Techniques '00 (Proc. Eurographics Workshop on Rendering)*, pages 281–286. Springer, 2000.

[23] Murat Kurt, László Szirmay-Kalos, and Jaroslav Křivánek. An anisotropic brdf model for fitting and monte carlo rendering. *SIGGRAPH Comput. Graph.*, 44(1):1–15, 2010.

[24] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, 1982.

[25] Christophe Schlick. An inexpensive brdf model for physically-based rendering. In *In Computer Graphics Forum's Proceedings of Eurographics 1994*, volume 13(3), pages 149–162, Oslo, Norway, 194.

[26] Brand M. Matusik W., Pfister H. and McMillian L. A data driven reflectance model. *ACM Trans. Graph.*, 1(1):759–769, 2003.

[27] Dr. Charles Kreitzberg and Ambrose Little. Usability in practice: Strategies for designing application navigation. *Msdn magazine : the Microsoft journal for developers.*