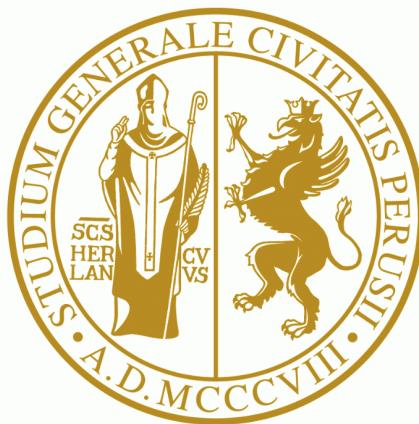


UNIVERSITÀ DEGLI STUDI DI PERUGIA  
Facoltà di Scienze Matematiche, Fisiche e Naturali

---

CORSO DI LAUREA IN INFORMATICA



Tesi di Laurea

**Realizzazione di un'interfaccia per l'interazione  
uomo-macchina basato su head-tracking**

Laureando:

*Michele Matriciani*

*Dott. Federico Frenguelli*

Relatori:

*Prof. Osvaldo Gervasi*

---

Anno Accademico 2012–2013

*Dedica...*

# Indice

<b>Introduzione</b>	<b>IV</b>
<b>1 Realtà virtuale</b>	<b>1</b>
1.1 Storia . . . . .	1
<b>2 Software utilizzati</b>	<b>4</b>
2.1 OpenCv . . . . .	4
2.1.1 Storia . . . . .	4
2.1.2 Computer vision . . . . .	5
2.1.3 Tracking . . . . .	6
2.1.4 Il training del cascade . . . . .	7
2.1.5 Face Recognition . . . . .	8
2.2 OpenGL . . . . .	12
2.2.1 Il funzionamento di OpenGL . . . . .	13
2.3 Ogre3D . . . . .	14
2.3.1 Storia . . . . .	14
2.3.2 Il funzionamento di Ogre3D . . . . .	15
<b>3 frustum</b>	<b>18</b>
3.1 Le trasformazioni . . . . .	19
3.1.1 ModelView Matrix . . . . .	19

3.1.2	Projection Matrix . . . . .	20
3.1.3	Viewport Matrix . . . . .	24
3.2	La teoria matematica delle trasformazioni . . . . .	25
3.2.1	Il view frustum . . . . .	25
3.2.2	Interpolazione . . . . .	27
3.2.3	Projection Matrix . . . . .	30
3.2.4	View Matrix . . . . .	33
<b>4</b>	<b>Il progetto</b>	<b>37</b>
4.1	Il funzionamento dell'applicazione . . . . .	38
4.1.1	Coordinate . . . . .	39
4.1.2	filtro . . . . .	41
4.1.3	3d . . . . .	44
4.1.4	perspective matrix . . . . .	45
4.1.5	view . . . . .	46
4.1.6	trasl . . . . .	48
4.1.7	fine . . . . .	49
4.2	Importazione scene in OpenGL . . . . .	50
4.3	Importazione in Ogre3D . . . . .	50
<b>5</b>	<b>Risultati e discussioni</b>	<b>53</b>
5.1	Difficoltà e problematiche riscontrate . . . . .	53
5.1.1	Lo studio delle trasformazioni . . . . .	53
5.1.2	I limiti dell'hardware . . . . .	54
5.1.3	I limiti di OpenCv . . . . .	55
5.1.4	Importazione modelli in OpenGL . . . . .	56
5.2	I requisiti dell'applicazione . . . . .	57
5.3	Risultati . . . . .	58

## **Introduzione**

---

<b>Conclusioni</b>	<b>58</b>
<b>Bibliografia</b>	<b>61</b>

# Introduzione

Da qualche anno a questa parte sono stati sviluppati e messi in commercio dispositivi per la realtà virtuale, i più famosi sono la Nintendo Wii, il Kinect della Microsoft o l’Oculus Rift sviluppato da Oculus VR. Periferiche come queste sono dette NUI, “natural user interface”[3]: essenzialmente sono interfacce per l’interazione con sistemi virtuali e hanno lo scopo di aumentare la sensazione della realtà durante l’utilizzo di applicazioni e simulazioni virtuali, permettendo all’utente di sfruttare più sensi.

La Wii permette, tramite il suo telecomando, di utilizzare indirettamente il corpo umano come dispositivo di input di comandi. Il Kinect utilizza dei sensori a infrarossi e una telecamera per rilevare il corpo umano e registrare direttamente i suoi movimenti, trasformandoli in comandi di input. L’Oculus Rift è un visore che presenta dei sensori che riconoscono l’orientamento della testa, i cui movimenti sono tradotti in rotazioni della telecamera virtuale nell’applicazione, rendendo immersiva la visione, anche grazie alla presenza di due schermi che producono un effetto tridimensionale.

Il progetto trattato in questa tesi mira ad emulare dispositivi di questo genere, utilizzando semplicemente un computer e una webcam. La posizione del volto, rilevata elaborando le immagini acquisite tramite una webcam, determina la prospettiva con cui viene visualizzata una scena tridimensionale, con lo scopo di collegare la telecamera virtuale agli occhi dell’utente.

L’obiettivo finale è quello di generare l’illusione della presenza di profondità all’interno dello schermo e di creare, in condizioni ottimali, un effetto tridimensionale senza l’utilizzo di occhiali appositi.

Dato l’utilizzo di dispositivi di qualità medio-bassa, l’effetto generato non è paragonabile a quello che può essere prodotto da sistemi più efficienti e dedicati allo scopo, tuttavia sono stati raggiunti comunque risultati notevoli.

Il progetto è stato sviluppato impiegando i seguenti software e librerie:

- OpenCV, per il rilevamento del volto.
- OpenGL, per lo studio di fattibilità e per lo sviluppo di un prototipo.
- Blender, per la creazione di scene 3D.
- Ogre3D, per un miglioramento dell’applicazione in termini di grafica ed efficienza.

Nel corso di questa tesi si farà un’introduzione sulla realtà virtuale, si parlerà delle tecnologie utilizzate, sarà spiegata la teoria matematica che sta dietro alle trasformazioni applicate, facendo un excursus sulle tecniche adottate dalla maggior parte delle applicazioni grafiche per renderizzare a schermo una scena tridimensionale, ed infine sarà trattato il progetto sviluppato, mostrando anche le problematiche riscontrate e le possibili soluzioni.

## Introduzione

---



# Capitolo 1

## Realtà virtuale

Al giorno d'oggi con realtà virtuale si intende quella disciplina che ha come obiettivo simulare la realtà, cioè permettere all'utente di interagire in modo realistico e naturale con un sistema puramente virtuale. A livello teorico, un sistema ideale dovrebbe permettere all'utente un'immersione completa, con la possibilità di utilizzare tutti i sensi. Le applicazioni che fanno parte di questa categoria sono ogni tipo di simulazione virtuale creata attraverso computer, schermi, dispositivi dotati di sensori, telecamera, etc. Tra queste i videogiochi stanno prendendo piede grazie all'introduzione del 3D e di periferiche di gioco sempre più immersive. Se, al giorno di oggi, con realtà virtuale si intende ogni forma di interazione con un sistema virtuale, nella storia essa ha avuto significati più ristretti.

### 1.1 Storia

Il termine realtà virtuale fu introdotto nel 1980 per indicare l'insieme dei fenomeni percettivi indotti da un'apparecchiatura cibernetica a più componenti che viene applicata a un soggetto umano. Ad introdurre il termine fu

un informatico statunitense che ha fondato la VPL Research, ricerca per i linguaggi di programmazione virtuale, ovvero Jaron Lanier.[6]

Tra i dispositivi creati in questo periodo possiamo citare il DataGlove, inventato da Thomas Zimmermann, un guantone-sensore collegabile ad un computer, oppure il display head-mounted (HMD), un visore comprendente degli schermi e dei sensori per l'orientamento, ideato da Scott Fisher, un ricercatore della Nasa.

Tuttavia, in passato, quando il concetto di realtà virtuale ancora non esisteva, già erano state costruite apparecchiature che possiamo definire adatte allo scopo.

Uno dei primi sistemi di realtà virtuale fu sviluppato nel 1962 da Morton Heilig, il quale costruì il prototipo di un dispositivo chiamato Sensorama, che proiettava cinque film e coinvolgeva sensi quali vista, olfatto, udito e tatto, creando, come la definiva lui, una sorta di cinema esperienza (Experience Theater).

Nel 1977 al MIT venne creato l'Aspen Movie Map, una simulazione in cui l'utente poteva camminare ed esplorare la cittadina di Aspen, in Colorado, tramite una serie di filmati che permettevano di visualizzare la maggior parte delle aree percorribili, nei limiti del possibile.

Al giorno d'oggi, uno dei dispositivi più efficienti per la realtà virtuale, tra quelli commercializzati, è l'Oculus Rift, che permette di vivere esperienze in prima persona grazie ad un visore da indossare, che possiede due schermi per avere una visione tridimensionale. I commenti di coloro che hanno provato questo dispositivo sono positivi, in quanto la qualità della grafica e del sensore di movimento rendono l'esperienza quasi realistica.

Per ora ci si concentra prevalentemente sullo sviluppo di applicazioni e dispositivi destinati ad utilizzare il senso della vista. Gli altri sensi ancora sono

agli esordi della realtà virtuale e stanno prendendo piede molto lentamente. Questo probabilmente perché la vista è il senso più sviluppato nell'uomo, e quindi il più adatto a percepire il realismo delle simulazioni virtuali.

# Capitolo 2

## Software utilizzati

### 2.1 OpenCv

OpenCV (Open Source Computer Vision Library) è una libreria open source che offre funzionalità per applicazioni di computer vision.

#### 2.1.1 Storia

Il progetto OpenCv è stato lanciato nel 1999 dalla Intel come una parte di varie iniziative volte a sviluppare applicazioni ad alta efficienza.[8]

La versione alpha è stata rilasciata nel 2000 mentre successivamente sono state rilasciate varie versioni beta; il rilascio della versione 1.0 è avvenuto nel 2006.

Dal 2008 il progetto è supportato dalla casa di produzione e ricerca Willow Garage, che si occupa di sviluppo di hardware e software open source per applicazioni robotiche. Nel 2009 è stata rilasciata una versione più efficiente, caratterizzata da migliori implementazioni e più funzionalità, delle quali una è la possibilità di parallelizzare il lavoro nei sistemi multi-core.

### **2.1.2 Computer vision**

La computer vision è una disciplina che ha come obiettivo principale ricreare la vista umana, utilizzando metodi per analizzare e processare immagini dal mondo reale al fine di estrarne determinate informazioni. Un sistema di computer vision opera in diverse fasi:

- acquisizione immagini tramite supporti ottici come telecamere.
- digitalizzazione dell'immagine.
- elaborazione dell'immagine da parte di software tramite algoritmi per rendere più marcate determinate caratteristiche, in modo da rendere più efficiente il riconoscimento (ad esempio aumentare il contrasto per rilevare meglio i contorni di oggetti).
- decisione da parte del sistema di scegliere o scartare il campione analizzato, in base allo scopo dell'applicazione, e raccolta di determinate informazioni derivate dall'analisi.

I sistemi di computer vision sono sfruttati in moltissimi campi:

- controllo dei processi industriali: riconoscimento di prodotti, lettura di codici ed etichette, controllo dei nastri trasportatori e della merce, posizionamento ed orientamento di bracci meccanici.
- riconoscimento di eventi (esempio nella videosorveglianza).
- controllo di macchine autonome (robot, veicoli di vario genere ed, esteso al campo militare, droni, missili autoguidati etc).
- medicina: nelle analisi di immagini derivate da ecografie, radiografie e simili, per diagnosticare determinate malattie o rilevare la presenza di tumori.

- neurobiologia: lo studio della struttura del cervello con la computer vision permette di classificare determinati comportamenti o altre caratteristiche dell'uomo o degli animali. In particolare nell'ultimo secolo è stato condotto lo studio degli occhi, dei neuroni e della struttura del cervello per studiare i processi del sistema visivo.

### **2.1.3 Tracking**

Il video tracking consiste nell'individuare un oggetto e tracciare il suo movimento, dato un qualsiasi stream video, come la ripresa di una telecamera. I frame del video vengono processati in modo da rilevare nel tempo la posizione dell'oggetto, o degli oggetti in questione.

Il problema maggiore di questa applicazione si riscontra nel calcolare con una buona precisione la posizione dell'oggetto in ogni frame. Questo è dovuto a varie motivazioni:

- elevata velocità dell'oggetto rispetto al frame rate.
- qualità della telecamera.
- fattori ambientali quali luminosità, presenza di ostacoli o falsi positivi che possono interferire con il tracciamento.
- qualità del cascade (“classificatore”) utilizzato per riconoscere l'oggetto.

Tranne le prime due motivazioni, le altre dipendono dal cascade, un file XML che contiene varie informazioni per riconoscere determinati oggetti in un'immagine.

### **2.1.4 Il training del cascade**

In OpenCV esistono delle funzioni per addestrare classificatori, utilizzando vari algoritmi. Il training (“addestramento”) richiede una serie di immagini contenenti l’oggetto da rilevare, chiamate positivi, e una serie di immagini che non contengono l’oggetto in questione, chiamate negativi. In base all’algoritmo utilizzato vengono analizzati i positivi per ricavare le somiglianze, inoltre questi sono confrontati con i negativi per evidenziare le differenze.

Il processo di training richiede tempi lunghi, in quanto, per produrre buoni cascade, necessita di una grande quantità di immagini di buona qualità. I positivi dovrebbero raffigurare l’oggetto in ogni stato possibile, ovvero visto da ogni angolazione, in ogni sua variante, con presenza di una certa illuminazione, etc. Ad esempio se si vuole addestrare un classificatore per riconoscere volti umani, i positivi dovrebbero raffigurare volti umani di ogni età, sesso, razza, caratteristiche specifiche quali capelli, barba, occhi, etc, e dovrebbero essere inquadrati da ogni angolazione possibile.

Ovviamente più è complesso l’oggetto da riconoscere, e più lungo e dispendioso è il processo di addestramento. Questo processo non è banale, basti pensare che all’inizio del progetto si è cercato di creare un classificatore di mandarini (quindi un oggetto molto semplice) e, utilizzando circa 300 positivi e 2000 negativi, con un addestramento di una ventina di ore il risultato era di scarsa qualità.

OpenCv offre dei classificatori già addestrati, da utilizzare per le proprie applicazioni.

### 2.1.5 Face Recognition

I classificatori sono addestrati tramite diversi algoritmi; ognuno ha le sue caratteristiche e può produrre risultati diversi, dato uno stesso insieme di input. Per il volto i principali algoritmi sono:

- *PCA* (Principal Component Analysis): due immagini, supponiamo di risoluzione  $100 \times 100$ , formano uno spazio di 10.000 dimensioni, perciò il carico di lavoro per il confronto sarebbe insostenibile.

Questo metodo permette di ridurre le dimensioni dello spazio vettoriale prendendo in considerazione solamente le componenti principali, cioè quelle con la massima varianza. Queste componenti sono chiamate “eigenfaces” (da “eigenvalue” = “autovalore”) in quanto il metodo prevede il calcolo di autovettori ed autovalori.

Il difetto principale di questo metodo è che risente fortemente di rotazioni o traslazioni, ed è influenzato dall’illuminazione e dall’ambiente.

- *LDA* (Linear Discriminant Analysis): come nella PCA anche in questo metodo vengono diminuite le dimensioni del sottospazio vettoriale; il miglioramento rispetto al metodo precedente sta nel fatto che le componenti vengono suddivise in classi in cui la varianza è minima. In questo modo classi simili sono vicine tra loro nello spazio vettoriale, mentre classi diverse sono più lontane possibile le une dalle altre. Nella PCA può succedere che alcune componenti importanti vengano scartate in quanto alterate da fattori come la luce o una rotazione del volto. Nel LDA invece, essendo le componenti suddivise in classi, le componenti discriminanti vengono mantenute anche in presenza di piccole alterazioni, e questo permette un riconoscimento più efficiente, anche se il volto è ruotato o non è del tutto visibile alla telecamera.

- *Wavelet*: le wavelet (“piccole onde”) sono delle funzioni utilizzate specialmente in analisi armonica, che tra i vari scopi, permettono di creare dei filtri. Una famiglia di queste trasformate, chiamata Haar wavelet, dal suo creatore Alfred Haar (1909)[9], ha la caratteristica di avere un dominio compreso tra -1 e 1, con valori interi, ovvero gli unici valori di cui dispone sono 0, -1 e 1.

Paul Viola e Michael Jones [10] introdussero l’utilizzo di queste trasformate nel campo del riconoscimento facciale. Il loro sistema utilizza i cosiddetti Haar-like feature per riconoscere zone con determinate caratteristiche in un’immagine[11].

Un’ Haar-like feature è un rettangolo suddiviso in diverse aree, in modo da formare un pattern da riconoscere nell’immagine. Ogni feature rappresenta la differenza della somma dei pixel delle varie regioni nel rettangolo. Nel processo di addestramento si parte con delle feature di base, per riconoscere semplici spigoli o contorni. Nella figura 2.1 si possono osservare alcuni tipi di feature di base.

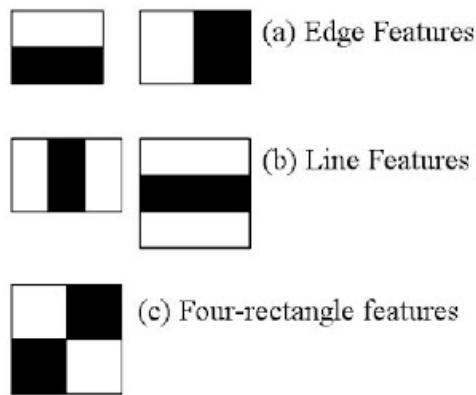


Figura 2.1: Alcune haar-features.

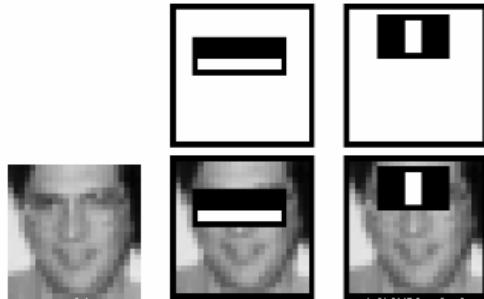


Figura 2.2: Particolari features riconosciute nel volto.

Le immagini vengono scansionate alla ricerca dei vari pattern di base. Ad ogni passaggio la ricerca viene ripetuta aumentando le dimensioni dei pattern. Il numero totale di feature calcolabili è molto grande, basti pensare che in un'immagine  $24 \times 24$  è possibile trovare più di 160.000 feature. Per questo si procede a prendere in considerazione solamente le feature più rilevanti. Per esempio nella figura 2.2 notiamo che nel volto sono stati riconosciute due particolari feature: una all'altezza degli occhi e una che ricalca la forma del naso.

Infatti la zona degli occhi solitamente è più scura rispetto alle altre parti del volto, ricalcando il primo pattern visibile nella figura 2.1, mentre il naso, posizionato tra le due zone scure degli occhi, presenta un pattern che ricalca la prima feature della seconda riga.

Dato un insieme di immagini positive, l'algoritmo procede a fare una classificazione per suddividere le feature simili in classi uguali, con una soglia prestabilita. Si selezionano solamente le feature con rapporto di errore minimo, in modo da suddividere quelle che possono ricondurre alle caratteristiche di un volto da quelle che non fanno parte di esso. Le immagini vengono scansionate fino a che il tasso di errore rimane sotto una certa soglia oppure in base al numero di stages di classifica-

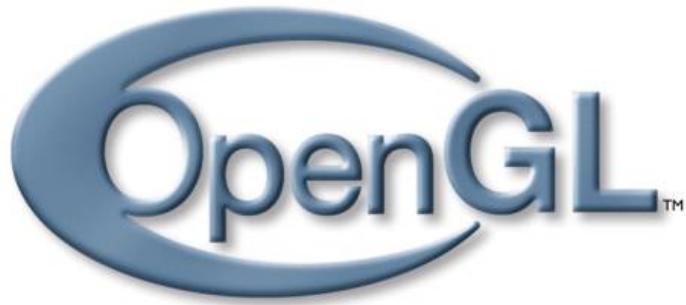
## **Software utilizzati**

---

zione deciso dall’utente. Alla fine del processo tutti i risultati, ovvero le features, vengono salvati all’interno di un file XML, che rappresenta il cascade desiderato.

OpenCv offre dei metodi, come `detect_multiscale()`, che permettono di riconoscere oggetti, scansionando un’immagine con lo scopo di cercare le features descritte nel cascade.

## 2.2 OpenGL



OpenGL (Open Graphics Library) è una specifica che definisce un'API (“Application Programming Interface”, “interfaccia di programmazione per applicazioni”) che fornisce metodi per sviluppare applicazioni di grafica 2D e 3D [13]. È stata introdotta nel 1992 dalla società californiana Silicon Graphics, è nata in ambiente Unix e, successivamente, è stata resa multi-piattaforma. La libreria OpenGL è usata per vari tipi di applicazioni, come animazioni, videogiochi, simulazioni 3D e di realtà virtuale.

Una specifica è un documento che descrive un insieme di funzioni ed il comportamento preciso che queste devono avere. I maggiori produttori hardware (Nvidia, AMD, Intel, etc) creano implementazioni di queste funzioni rispettando la specifica OpenGL, usando l'accelerazione hardware quando possibile. In questo modo il programmatore può usufruire di un'API unificata, utilizzando le funzioni senza preoccuparsi della loro implementazione.

Un'altra caratteristica fondamentale è che le funzioni di OpenGL devono essere sempre usufruibili, quindi le implementazioni richiedono un'emulazione software quando l'hardware non è in grado di fornire certe funzionalità.

### 2.2.1 Il funzionamento di OpenGL

OpenGL offre circa 250 chiamate di funzione per disegnare scene a due o tre dimensioni, a partire da delle primitive. Con primitive si intendono punti, linee e poligoni, i quali vengono convertiti in pixel tramite una serie di processi chiamata pipeline grafica. La pipeline rappresenta una catena di trasformazioni e operazioni, in cui ogni fase prende in input il risultato della fase precedente e produce l'output per la fase successiva.

OpenGL opera a basso livello e richiede al programmatore di rispettare i passi precisi della pipeline che servono a renderizzare la scena. Tra i passi principali da seguire sono presenti :

- fornire le primitive che descrivano la scena.
- fornire le regole per generare una telecamera virtuale, che renderizzi solo determinate porzione della scena, con particolari modalità.
- fornire le regole per gestire texture, materiali, luci e ombre.

Queste regole vanno definite in particolari file chiamati shader, scritti in un linguaggio ad alto livello proprio di OpenGL, basato sul linguaggio C, chiamato GLSL (OpenGL Shading Language). Siccome la scena non può essere renderizzata senza aver fornito le regole basilari, è necessaria, da parte del programmatore, una buona conoscenza della suddetta pipeline grafica e del linguaggio GLSL.

Esistono anche framework che operano ad alto livello, i quali nascondono al programmatore le fasi più complesse del rendering, gestendole autonomamente, e richiedono solamente una descrizione generica della scena. Un'esempio è il framework Ogre3D, trattato in seguito.



## 2.3 Ogre3D

OGRE (Object-Oriented Graphics Rendering Engine) è un motore grafico open-source 3D che essenzialmente offre un'API che sfrutta l'accelerazione hardware per la creazione di applicazioni grafiche.

### 2.3.1 Storia

E' stato ideato nel 1999 da uno sviluppatore inglese di nome Steve Streeting, dopo che ebbe creato un progetto chiamato DIMClass, il quale aveva lo scopo di rendere più semplice l'utilizzo della libreria Direct3D. Si rese conto che il livello di astrazione raggiunto dalla sua libreria era così elevato che non aveva più bisogno di essere basata su Direct3D. Da qui nacque l'idea di creare una libreria che fosse indipendente da API [15].

Nel 2000 fu il progetto registrato su SourceForge<sup>1</sup> registrato il nuovo progetto dalla Sourceforge, con il nome di OGRE. Il 25 febbraio 2005 fu rilasciata la prima versione di Ogre, la 1.0.0, con il nome di Azathoth. La

---

<sup>1</sup>Piattaforma web per portare avanti progetti e software, facilitando la collaborazione tra sviluppatori

versione utilizzata in questo progetto è la 1.9.0, rilasciata il 24 novembre 2013, con il nome Ghadamon.

### **2.3.2 Il funzionamento di Ogre3D**

Questo framework è scritto in C++ e, a differenza di API quali OpenGL, permette di astrarre tutta la parte a basso livello che invece deve essere implementata negli altri casi.

Utilizza una struttura dati ad albero (scene graph), dove ogni nodo può avere più figli ma un solo padre. Partendo dalla radice, che permette di configurare il sistema, si possono poi creare dei figli per gestire tutta la scena. Il nodo principale è lo Scene Manager che, come dice il nome, gestisce tutto ciò che è presente nella scena, come telecamere, oggetti, luci, etc. Solitamente si utilizza un solo Scene Manager, ma è sempre possibile crearne di altri, magari in applicazioni più complesse, per gestire più scene. Questo può essere utilizzato ad esempio per suddividere la finestra in più parti, ognuna delle quali riprende una zona diversa della scena (il cosiddetto split screen).

Dallo Scene Manager è possibile appendere dei nodi chiamati Scene Node, che permettono di gestire più elementi della scena, come entità (i modelli), luci, telecamere, etc. In questo modo è possibile applicare trasformazioni ed effetti a gruppi di oggetti, invece che a ciascuno preso singolarmente.

Infine troviamo i singoli oggetti. Tra questi, come è stato già detto, vi sono i modelli, che vengono rappresentati da oggetti chiamati Entity, i quali devono essere imparentati ad almeno uno Scene Node, tramite il quale sono gestiti. In Ogre, a differenza di OpenGL che richiede una descrizione delle primitive, è possibile creare già dei modelli preimpostati, come ad

esempio cubi o piani, oppure possono essere importati dall'esterno. Inoltre Ogre permette di definire le luci e le telecamere come se fossero oggetti.

Da quello che abbiamo visto notiamo delle semplificazioni. Per esempio quando in OpenGL si importavano modelli esterni, questi dovevano essere prima processati, salvando nei buffer tutti i vertici e altre informazioni quali normali o coordinate uv per le texture, richiedendo diverse righe di codice. In Ogre invece sono presenti metodi già implementati che, grazie a pochissime righe di codice, permettono di importare modelli sotto forma di file con estensione .mesh.

Inoltre in OpenGL la gestione di texture, ombre e materiali va implementata a mano scrivendo gli algoritmi di shading, che contengono informazioni quali il colore o regole di comportamento in presenza di luce. Al contrario in Ogre la gestione di questi aspetti è quasi automatica in quanto di default vengono utilizzati shader già presenti nel framework.

Tuttavia, Ogre è anche versatile perchè lascia all'utente ampie libertà, permettendogli di personalizzare diversi aspetti. Ad esempio in Ogre la view matrix e la perspective matrix, che servono a definire la telecamera, sono generate in automatico, in modo trasparente al programmatore. Poichè la nostra trasformazione richiedeva che le due matrici fossero in funzione della posizione dell'utente, esse sono state calcolate in una funzioni scritte a mano, e poi passate come parametro a due metodi che impostano le trasformazioni a partire da matrici personalizzate.

Inoltre è sempre possibile utilizzare shader personalizzati, e definire i materiali degli oggetti in file con estensione .material, scritti con un linguaggio di facile comprensione, proprio di Ogre.

Per questo, una volta compresa la trasformazione che richiedeva il nostro progetto, è stata abbandonata l'idea di utilizzare OpenGL passando invece

## **Software utilizzati**

---

a Ogre3D, con lo scopo di concentrarci sulla qualità e la miglior resa della scena.

# Capitolo 3

## frustum

Come abbiamo già detto, le applicazioni grafiche eseguono una serie di operazioni e trasformazioni, chiamata pipeline grafica per renderizzare una scena prelevata da un mondo virtuale 3D così come noi la vediamo a schermo. Nella figura 3.1 sono presenti le principali fasi della pipeline.

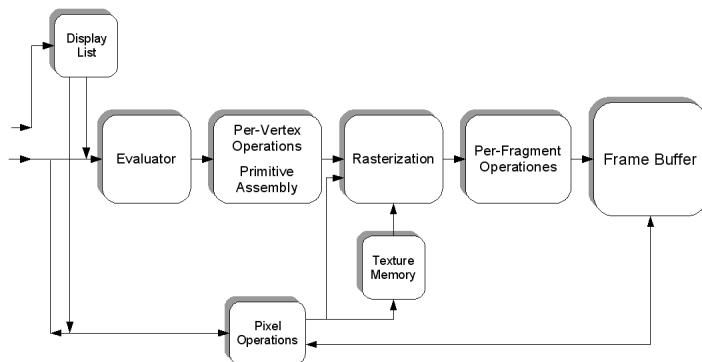


Figura 3.1: La pipeline grafica.

Ai fini di questo progetto, ci concentreremo sulle trasformazioni, più che su altre operazioni come la rasterizzazione (conversione da vettori a pixel, costruendo segmenti o porzioni di piano che riempiono lo spazio definito da due o più vertici) o l'applicazione di materiali e texture.

### 3.1 Le trasformazioni

Il diagramma in figura 3.2 mostra le principali trasformazioni eseguite nella pipeline.

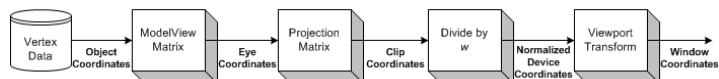


Figura 3.2: Le trasformazioni.

Per determinare la posizione di ciascun vertice all'interno del mondo virtuale, viene adottato un sistema di coordinate a 3 dimensioni (figura 3.3).

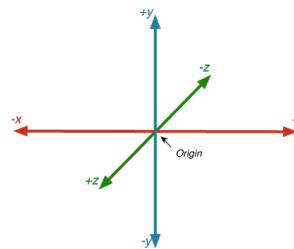


Figura 3.3: Il sistema cartesiano in OpenGL e Ogre3D.

Da notare l'asse Z, la cui direzione positiva è quella uscente, ovvero quella rivolta verso l'utente che sta di fronte allo schermo. Questa caratteristica è fondamentale perché determina il segno di alcuni elementi nel processo di trasformazione.

#### 3.1.1 ModelView Matrix

Nel prima fase si utilizza una matrice chiamata Modelview Matrix, che trasforma le coordinate locali degli oggetti presenti nella scena nel sistema globale di coordinate relativo al mondo virtuale. Di default, la telecamera

virtuale che inquadra la scena è posizionata nell'origine del sistema, ovvero nel punto  $(0, 0, 0)$ . La telecamera punta lo sguardo verso l'asse  $-Z$ , mentre gli assi  $X$  e  $Y$  sono paralleli al piano che rappresenta lo schermo (figura 3.4).

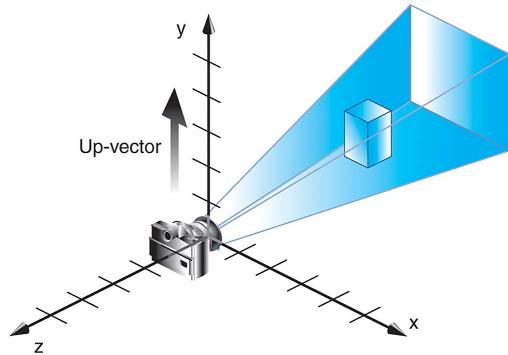


Figura 3.4: La posizione di default della telecamera virtuale.

### 3.1.2 Projection Matrix

Lo spazio che identifica la porzione del mondo che è possibile osservare, è chiamato view volume (figura 3.5); tutto ciò che si trova all'interno di esso viene renderizzato nella scena.

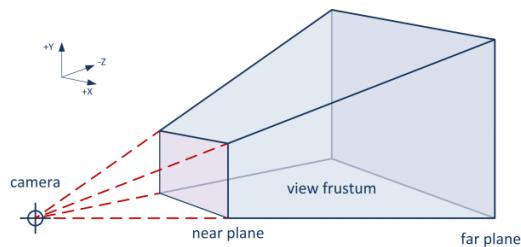


Figura 3.5: Il view volume nella proiezione prospettica.

Questo spazio è caratterizzato da un piano più vicino alla telecamera, il near plane, e uno più lontano, il far plane. Il near plane rappresenta la coordinata  $Z$  dalla quale sarà renderizzata la scena, ovvero tutto ciò che

sta dietro non sarà visibile. Il far plane rappresenta la coordinata Z di fine scena, perciò determina quanto in profondità essa è visibile.

La Projection Matrix ha il compito di trasformare il view volume in un cubo  $2 \times 2 \times 2$  centrato nell'origine, determinando anche la trasformazione di tutto ciò che è presente al suo interno. Ogni vertice all'interno di esso avrà coordinate  $-1 \leq x, y, z \leq 1$ .

Il modo in cui viene considerato il view volume, e quindi il modo in cui viene trasformato lo spazio, dipende dal tipo di proiezione. Essenzialmente esistono due tipi di proiezione, quella ortogonale e quella prospettica.

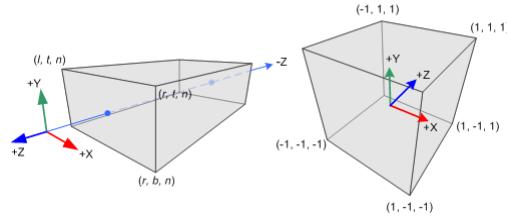


Figura 3.6: Trasformazione con proiezione ortogonale.

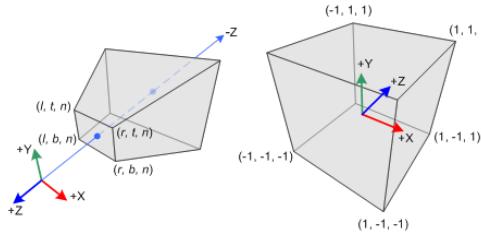


Figura 3.7: Trasformazione con proiezione prospettica.

In quella ortogonale (figura 3.6) il view volume è rappresentato da un parallelepipedo; in questo caso tutti i vertici vengono proiettati sul near plane, parallelamente alla direzione della telecamera. Il risultato sarà quello di avere una visione non prospettica della scena.

La proiezione prospettica (che vedremo in dettaglio in seguito, in quanto il suo uso è fondamentale in questo progetto) presenta un view volume che

ha la forma di un tronco di piramide, in inglese frustum. Il vertice della piramide è dato dalla posizione della telecamera, mentre il near plane e il far plane sono rispettivamente la base minore e maggiore del frustum. In questo caso i vertici vengono proiettati sul near plane seguendo la direzione che li congiunge alla telecamera. Dopo aver applicato questa trasformazione, il frustum diventerà un cubo, e tutti gli oggetti al suo interno saranno deformati per come sono visti a schermo.

Nelle figure 3.8 e 3.9 è possibile vedere un esempio di questa trasformazione.

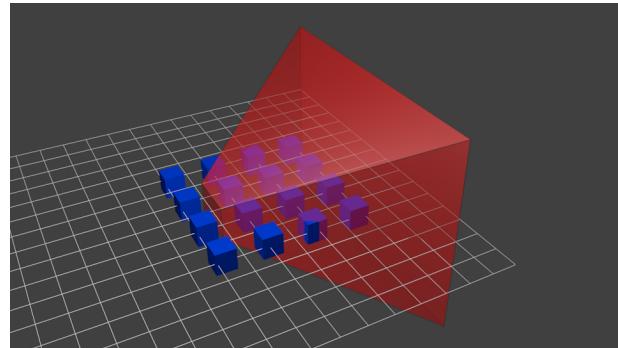


Figura 3.8: Il frustum prima della trasformazione.

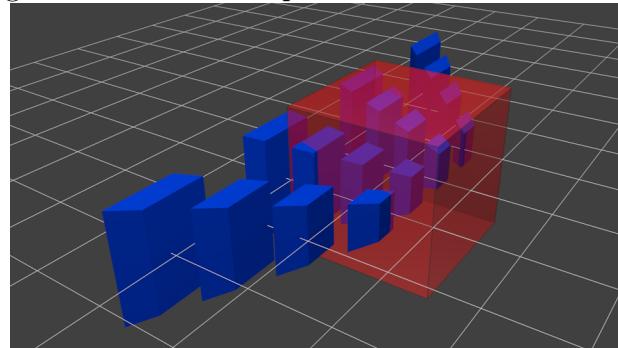


Figura 3.9: Il frustum trasformato nel sistema di coordinate omogeneo.

Nella figura 3.10 il cubo è visto dal suo piano frontale.

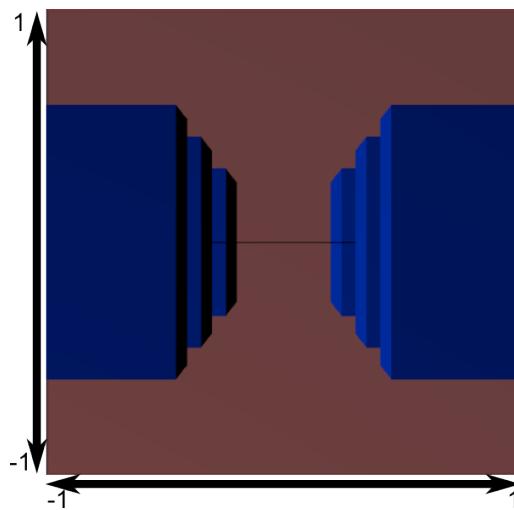


Figura 3.10: Il nuovo frustum visto in prospettiva.

Nella projection matrix è compresa l'operazione che normalizza i vertici, mappando le coordinate in modo che siano comprese tra  $-1$  e  $1$ . Questa normalizzazione è data da una divisione delle componenti di ogni vertice per la quarta componente, cioè la coordinata  $W$ .

L'utilizzo della quarta coordinata per definire i vertici dà origine al sistema di coordinate omogenee. Questo sistema, introdotto da August Ferdinand Möbius intorno al 1837, prevede la rappresentazione di  $N$  coordinate con vettori di  $N + 1$  dimensioni, ed è usato per descrivere i punti nella geometria proiettiva. Inoltre la quarta coordinata permette di rappresentare tutte le trasformazioni affini (trasformazioni lineari più una traslazione) tramite le matrici.

Considerando che la traslazione di un vettore a due dimensioni  $v = (x, y)$  per un vettore  $v_0 = (a, b)$  è data dalla loro somma, è possibile rappresentare

questa trasformazione anche tramite una matrice  $3 \times 3$ :

$$T_{v_0} \times v = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + a \\ y + b \\ 1 \end{pmatrix}$$

E' possibile estendere questo concetto ad ogni tipo di trasformazione, permettendo di raggruppare più trasformazioni in una singola matrice uguale al prodotto delle matrici che le rappresentano.

### 3.1.3 Viewport Matrix

L'ultimo passaggio è dato dalla Viewport Matrix, che trasforma i vertici in coordinate relative allo schermo. Dopo questa fase, il nuovo sistema di riferimento è relativo all'angolo in basso a sinistra della finestra aperta dopo aver lanciato l'applicazione, e le coordinate sono date dai pixel dello schermo.

Dati i valori  $w$  (width = larghezza) e  $h$  (height = altezza), in pixel, il canonical view volume viene ridotto, scalandolo di  $w/2$  nella direzione orizzontale e di  $h/2$  nella direzione verticale. Dopo di ciò il tutto viene traslato di un vettore  $(-w/2, -h/2, -1)$  per allineare l'angolo in basso a sinistra di fronte del canonical view volume (cioè il punto  $(-1, -1, -1)$ ) con l'angolo in basso a sinistra della finestra visualizzata a schermo.

Tutte le operazioni che si svolgono dopo, come la rasterizzazione e lo shading, vengono fatte usando questo sistema di riferimento.

## 3.2 La teoria matematica delle trasformazioni

### 3.2.1 Il view frustum

Con view frustum indichiamo il view volume utilizzato nella proiezione prospettica, in quanto il frustum rappresenta il tronco di piramide che racchiude lo spazio visibile del mondo 3D; in genere viene chiamato allo stesso modo anche il view volume considerato nella proiezione ortogonale, ma in questo caso occorre notare che si tratta di un parallelepipedo e non un tronco di piramide.

Il view frustum è posizionato in modo che il vertice della piramide, rappresentante l'osservatore o la telecamera, giaccia sull'origine. Esso è composto da sei piani, due di questi li abbiamo già visti, e sono il near e il far plane. Gli altri quattro rappresentano le coordinate che definiscono i bordi del near plane, e sono chiamati left, right, bottom e top planes.

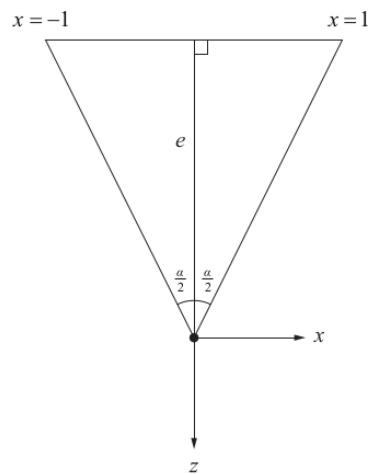


Figura 3.11: La distanza focale  $e$  dipende dall'angolo dell'apertura  $\alpha$  della telecamera.

Si considera, in generale, che il piano di proiezione sia situato ad una

distanza  $e$  dall'osservatore, in modo che le coordinate del left e right plane siano rispettivamente  $x = -1$  e  $x = +1$  (figura 3.11).

La distanza  $e$  è la distanza focale, che è determinata dall'angolo di apertura  $\alpha$  della telecamera: valori più alti di  $\alpha$  determinano valori di  $e$  più piccoli, e ciò si traduce in un accorciamento della distanza e allargamento della visuale, creando un effetto grandangolare, in quanto il piano di proiezione è più vicino all'osservatore; viceversa valori alti di  $e$  determinano un allontanamento del piano di proiezione, con effetto di allungamento della distanza e restringimento della visuale.

Grazie alla trigonometria possiamo calcolare la distanza focale a partire dall'angolo di apertura, come

$$e = \frac{1}{\tan(\alpha/2)}$$

Un altro valore da tenere in considerazione è l'aspect ratio , cioè il rapporto tra la lunghezza e l'altezza del piano. Per esempio in uno schermo  $1920 \times 1080$  l'aspect ratio è uguale  $\frac{1080}{1920} = \frac{16}{9}$ , oppure in uno schermo  $640 \times 480$  è  $\frac{640}{480} = \frac{4}{3}$ . Grazie a questo possiamo determinare il formato della finestra che vogliamo visualizzare a schermo.

Considerando che il left e right plane hanno coordinate  $x = -1$  e  $x = +1$ , il bottom e top plane vengono posizionati alle coordinate  $y = -a$  e  $y = +a$ , dove  $a$  è il reciproco dell'aspect ratio; nel caso  $\frac{16}{9} \rightarrow a = \frac{9}{16} = 0.5625$  (figura 3.12).

Possiamo calcolare l'angolo di apertura verticale  $\beta$  come:

$$\beta = 2 \tan^{-1}(a/e)$$

Possiamo assegnare arbitrariamente un valore  $n$  come coordinata per il near plane, in modo da determinare la distanza da cui inizia il frustum. Le

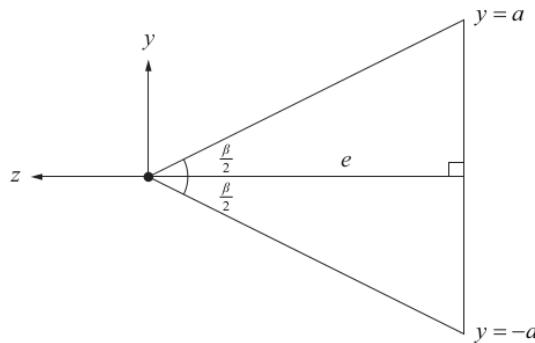


Figura 3.12: L'angolo di apertura verticale  $\beta$  dipende dall'aspect ratio.

coordinate dei quattro piani ora devono essere ricalcolate come

$$x = \pm \frac{n}{e}$$

e

$$y = \pm \frac{an}{e}$$

### 3.2.2 Interpolazione

Consideriamo il piano  $xz$  e il segmento formato dai punti  $(x_1, z_1)$  e  $(x_2, z_2)$  (vedi figura 3.13).

Definiamo l'equazione della retta sulla quale giace il segmento come

$$ax + bz = c$$

supponendo che non passi per l'origine, cioè con  $c \neq 0$ . Tracciamo le rette (cioè le proiezioni) che congiungono l'origine (dove sta la telecamera) con i due punti, come in figura.

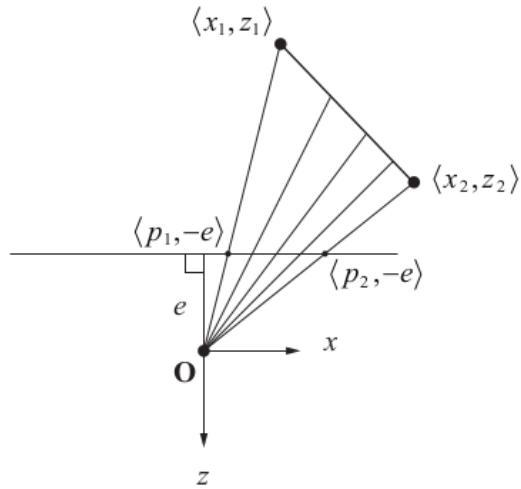


Figura 3.13: Un segmento nel frustum proiettato sul near plane.

Le due rette intersecano il piano di proiezione nei punti  $(p_1, -e)$  e  $(p_2, -e)$  (si usa il segno meno in quanto  $e$  è la distanza tra l'origine e il piano di proiezione, situato lungo l'asse negativo delle  $z$ ).

Per provare che la proiezione è valida, dimostriamo che tutti i punti compresi nel segmento iniziale siano interpolati correttamente nel segmento proiettato.

Il triangolo rettangolo formato dai punti  $\{O, (0, -e), (p_1, -e)\}$  è simile a quello formato dai punti  $\{O, (0, z_1), (x_1, z_1)\}$ , in quanto hanno tre angoli congruenti. Lo stesso vale per gli altri due triangoli, comprendenti i punti  $(x_2, z_2)$  e  $(p_2, -e)$ .

Possiamo definire la relazione derivata dalla proprietà dei triangoli simili

$$\frac{p}{x} = \frac{-e}{z}$$

Mettiamo in evidenza  $x$

$$x = \frac{pz}{-e}$$

e sostituiamo la  $x$  nell'equazione precedente della retta

$$\left( -\frac{ap}{e} + b \right) z = c$$

Mettiamo in evidenza la quantità  $1/z$

$$\frac{1}{z} = -\frac{ap}{ce} + \frac{b}{c}$$

Consideriamo il punto generico  $(p_3, -e)$  compreso tra  $(p_1, -e)$  e  $(p_2, -e)$ ; possiamo esprimere  $p_3 = (1-t)p_1 + tp_2$  con  $0 \leq t \leq 1$ . Riprendendendo la relazione precedente scriviamo

$$\frac{1}{z_3} = -\frac{ap_3}{ce} + \frac{b}{c}$$

Sostituendo  $p_3$  con la precedente uguaglianza e facendo qualche passaggio otteniamo:

$$\frac{1}{z_3} = -\frac{ap_1}{ce}(1-t) - \frac{ap_2}{ce}t + \frac{b}{c}$$

Poichè  $(1-t)+t = 1$ , possiamo sostituire  $\frac{b}{c}(1-t+t) = \frac{b}{c}(1-t) + \frac{b}{c}(t)$ .

Raccogliendo i termini simili scriviamo

$$\left( -\frac{ap_1}{ce} + \frac{b}{c} \right) (1-t) + \left( -\frac{ap_2}{ce} + \frac{b}{c} \right) t$$

Perciò

$$\frac{1}{z} = \frac{1}{z_1}(1-t) + \frac{1}{z_2}t$$

Con ciò abbiamo dimostrato che anche il reciproco di  $z$  è interpolato correttamente.

### 3.2.3 Projection Matrix

Come abbiamo detto all'inizio del capitolo, per renderizzare a schermo un punto tridimensionale, esso deve essere trasformato in un punto a due dimensioni. Il sistema di coordinate omogenee, che si utilizza per identificare i punti nello spazio, possiede quattro dimensioni. La projection matrix lo trasforma in un sistema a tre dimensioni, proiettando lo spazio omogeneo in uno spazio 3D chiamato canonical view volume (un cubo).

Dopo aver proiettato i punti omogenei sul near plane, le nuove coordinate vengono normalizzate, cioè divise per la quarta coordinata  $w$ ; questo permette di mappare le nuove coordinate in uno spazio  $[-1, 1]$ .

Consideriamo un punto omogeneo  $P = \langle P_x, P_y, P_z, 1 \rangle$  situato nel view frustum. Riprendendo il discorso del paragrafo precedente riguardante i triangoli simili, le coordinate  $x$  e  $y$  del punto proiettato sul near plane sono

$$x = -\frac{n}{P_z} P_x \quad \text{e} \quad y = -\frac{n}{P_z} P_y$$

$$x = -\frac{n}{P_z} P_x \quad y = -\frac{n}{P_z} P_y$$

Allora  $l \leq x \leq r$  e  $b \leq y \leq t$ , dove  $l, r, b, t$  sono le coordinate dei quattro piani del frustum, descritti in precedenza. Per mappare le nuove coordinate  $x$  e  $y$  si usano le funzioni lineari:

$$x' = (x - l) \frac{2}{r - l} - 1$$

e

$$y' = (y - b) \frac{2}{t - b} - 1$$

Sostituendo alla  $x$  e alla  $y$  le espressioni scritte in precedenza e semplificando otteniamo:

$$x' = \frac{2n}{r-l} \left( -\frac{P_x}{P_z} \right) - \frac{r+l}{r-l}$$

e

$$y' = \frac{2n}{t-b} \left( -\frac{P_y}{P_z} \right) - \frac{t+b}{t-b}$$

Per quanto riguarda la mappatura della coordinata  $z$ , si deve trovare una funzione che porti  $-n \rightarrow -1$  e  $-f \rightarrow 1$ . Considerando che le coordinate  $z$  hanno i reciproci interpolati, possiamo scrivere una funzione di  $1/z$ :

$$z' = \frac{A}{z} + B$$

Abbiamo detto che  $-n \rightarrow -1$  e  $-f \rightarrow 1$ , perciò possiamo ottenere il sistema:

$$-1 = \frac{A}{-n} + B$$

e

$$1 = \frac{A}{-f} + B$$

Risolvendo il sistema otteniamo:

$$A = \frac{2nf}{f-n}$$

e

$$B = \frac{f+n}{f-n}$$

Sostituendo  $A$  e  $B$ :

$$z' = -\frac{2nf}{f-n} \left( -\frac{1}{P_z} \right) + \frac{f+n}{f-n}$$

Si può notare che nelle equazioni di  $x'$ ,  $y'$  e  $z'$  è presente una divisione per  $-P_z$ : il punto tridimensionale  $P' = \langle x', y', z' \rangle$  è equivalente al punto omogeneo 4D  $P = \langle -x'P_z, -y'P_z, -z'P_z, -P_z \rangle$  dopo che esso viene diviso per

la sua coordinata w, cioè per  $-P_z$ . Perciò possiamo esprimere le precedenti equazioni mettendo in evidenza  $-x'P_z$ ,  $-y'P_z$ , e  $-z'P_z$ :

$$-x'P_z = \frac{2n}{r-l}P_x + \frac{r+l}{r-l}P_z$$

,

$$-y'P_z = \frac{2n}{t-b}P_y + \frac{t+b}{t-b}P_z$$

e

$$-z'P_z = -\frac{f+n}{f-n}P_z - \frac{2nf}{f-n}$$

Essendo funzioni lineari delle coordinate del punto  $P$ , possiamo utilizzare una matrice  $4 \times 4$  per rappresentare la trasformazione prospettiva; moltiplicando la matrice per il punto  $P$ , otteniamo la sua proiezione  $P'$  sul near plane:

$$P' = M * P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

In OpenGL esistono essenzialmente due funzioni per generare una matrice di proiezione prospettica: la libreria GLM (OpenGL Mathematics), usata in questo progetto (nella versione sviluppata in OpenGL), offre i metodi `perspective(float fov, float aspect, float n, float f)` e `glFrustum(float l, float r, float t, float b, float n, float f)`.

La prima funzione restituisce una matrice di proiezione con frustum simmetrico, ovvero il near plane assume la figura di un rettangolo; questo è il tipo di frustum maggiormente utilizzato; come visto all'inizio del paragrafo relativo al view frustum, tramite l'angolo di apertura di campo `fov`, e l'`aspect ratio`, è possibile calcolare le coordinate dei quattro piani del frustum, oltre

al near e il far. Dopo aver calcolato le coordinate, la funzione genera la matrice desiderata.

La seconda funzione invece prende in input le coordinate dei sei piani del frustum e calcola direttamente la matrice; in questo modo è anche possibile creare un frustum asimmetrico, generando una prospettiva distorta (il near plane può assumere la forma di un quadrilatero qualsiasi).

Quest'ultima funzione è stata utilizzata in questo progetto, in modo da avere la possibilità di distorcere la prospettiva della scena in base alla posizione dell'osservatore.

### 3.2.4 View Matrix

La Projection Matrix trasforma lo spazio considerando che la telecamera sia situata nell'origine e che la direzione dello sguardo sia rivolto lungo l'asse negativo delle Z. Per questo, se la telecamera è posizionata in punti diversi, si deve applicare un'altra trasformazione, che porti la telecamera nella sua posizione di default. Questa trasformazione è data dalla View Matrix, che ha il compito di trasformare lo spazio di coordinate del mondo 3D nello spazio di coordinate relativo alla telecamera. In pratica la telecamera viene ruotata e traslata per essere posizionata nell'origine, con il corretto orientamento, e nello stesso momento tutto il mondo 3D subisce la stessa trasformazione (figure 3.14 e 3.15).

Dopo aver applicato questa trasformazione è possibile applicare anche la proiezione.

Il calcolo della view matrix necessita di tre vettori:

- **eye**: identifica la posizione della telecamera.
- **target**: identifica il punto verso cui è rivolto lo sguardo.

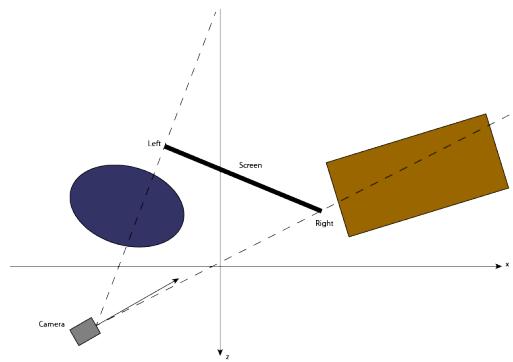


Figura 3.14: Una scena con la telecamera posta in un punto qualsiasi.

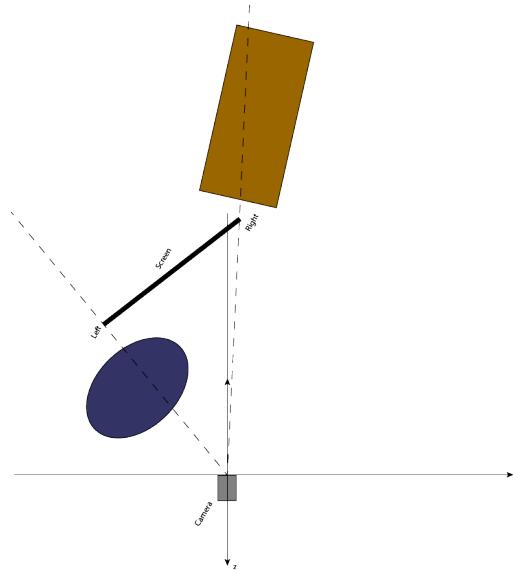


Figura 3.15: La scena trasformata con la view matrix.

- **up:** identifica la direzione che punta verso l'alto, di solito coincide con l'asse y, quindi è  $(0, 1, 0)$ .

La matrice consiste in due trasformazioni: una rotazione, per allineare l'orientamento della telecamera con l'asse negativo delle Z, e una traslazione, per spostare la telecamera nell'origine. Per la rotazione si calcola una base ortonormale che rappresenta il sistema di coordinate relativo alla telecamera, tramite i tre vettori, ponendo

$$n = \text{normalize}(eye - target)$$

$$u = \text{normalize}(\text{cross}(up, n))$$

$$v = \text{cross}(n, u)$$

dove  $\text{cross}(a, b)$  rappresenta il prodotto vettoriale tra  $a$  e  $b$  e  $\text{normalize}(a)$  la normalizzazione di  $a$ . I vettori  $u, v, n$  formano la base ortonormale desiderata.

La matrice che orienta la scena nel modo corretto è:

$$R = \begin{bmatrix} u.x & u.y & u.z & 0 \\ v.x & v.y & v.z & 0 \\ n.x & n.y & n.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Per quanto riguarda la traslazione basta utilizzare la matrice

$$T = \begin{bmatrix} 0 & 0 & 0 & -eye.x \\ 0 & 0 & 0 & -eye.y \\ 0 & 0 & 0 & -eye.z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

dove il meno permette di traslare la scena in modo che la telecamera finisca nell'origine.

---

La View Matrix è data dal prodotto della matrice dell'orientamento per quella di traslazione. Possiamo perciò scriverla in modo compatto come:

$$V = R \times T = \begin{bmatrix} u.x & u.y & u.z & -dot(u, eye) \\ v.x & v.y & v.z & -dot(v, eye) \\ n.x & n.y & n.z & -dot(n, eye) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

dove  $dot(a, b)$  rappresenta il prodotto scalare tra  $a$  e  $b$ .

# Capitolo 4

## Il progetto

Dopo aver fatto un'introduzione sulla teoria matematica e sui programmi utilizzati, possiamo ora esaminare il progetto trattato in questa tesi.

L'obiettivo del progetto è quello di offrire un'interfaccia uomo-macchina che permetta all'utente di interagire con la telecamera virtuale presente nella scena utilizzando il proprio volto come controller. Utilizzando la tecnica dell'head-tracking viene rilevato il movimento del volto, che causa la trasformazione della prospettiva con cui è vista la scena, generando l'illusione della presenza di profondità al di là dello schermo. Con un rilevamento efficiente del volto, e con una scena ben fatta, è addirittura possibile creare una visione tridimensionale senza l'uso di occhiali 3D.

Per aumentare l'effetto desiderato, la scena creata inizialmente è l'interno di una scatola, con la parte frontale aperta posizionata sul near plane, in modo da dare l'illusione che lo stesso schermo del pc sia l'apertura della stessa. All'interno si aggiungono degli oggetti per apprezzare meglio i cambiamenti della prospettiva. Ovviamente si possono utilizzare scene di qualsiasi genere ma, per ottenere un effetto maggiore, conviene che i bordi frontali della scena siano fissati ai bordi dello schermo.

L'applicazione è stata sviluppata in due versioni, utilizzando due API differenti: OpenGL e Ogre3D. Poichè OpenGL opera più a basso livello, le trasformazioni devono essere esplicitate calcolando le matrici e moltiplicandole nel modo opportuno; questo può aiutare nella comprensione delle trasformazioni utilizzate. Tuttavia, dati i miglioramenti che sono stati riportati nella seconda versione, la prenderemo in esame ed analizzeremo il suo codice, trattando le varie fasi in dettaglio. In seguito sarà dedicata una sezione all'importazione delle scene in Ogre.

L'applicazione è stata scritta in C++ ed è divisa in due parti principali:

- Un modulo OpenCv separato da tutto il resto, che offre un metodo che fornisce le coordinate del volto rilevate nel momento in cui viene richiamato.
- La parte grafica sviluppata usando Ogre3D.

### **4.1 Il funzionamento dell'applicazione**

Il cuore dell'applicazione risiede nelle trasformazioni da applicare alla scena. In ogni frame vengono eseguite diverse fasi, a partire dall'ottenimento delle coordinate del volto, arrivando a trasformare la telecamera nel modo voluto.

In seguito vediamo in breve le fasi:

1. Ottenimento coordinate del volto in pixel (relative alla webcam).
2. Applicazione di un filtro per eliminare il rumore dato dall'imprecisione del rilevamento.
3. Trasformazione delle coordinate in punti dello spazio 3D.

4. Calcolo dei sei piani del frustum in base alle coordinate dell'osservatore, e creazione della projection matrix.
5. Calcolo dei vettori necessari per la view matrix, e creazione della stessa.
6. Applicazione di una traslazione per aggiustare lo spostamento della telecamera, in modo che la scena sia sempre fissa.
7. La matrice di trasformazione finale relativa alla telecamera è data, in ordine, dal prodotto della matrice di traslazione per la view matrix per la projection matrix.

Analizzando in dettaglio le varie fasi:

### 4.1.1 Coordinate

Grazie al modulo OpenCv, richiamando una semplice funzione possiamo ottenere le coordinate del volto in un dato frame.

Di seguito un estratto di codice della funzione nel modulo OpenCv

```
bool getFaceCoord(int* x, int* y, int* z) {  
    ...  
    if ( detectAndDisplay( pt, z ) ) {  
        *x = pt.x;  
        *y = pt.y;  
        *z = z;  
        return true;  
    }  
    return false;  
}
```

```
}

...

bool detectAndDisplay( cv::Point& center, int& z ) {
    ...

face_cascade->detectMultiScale( frame_gray, faces, 1.1, 3,
                                0|cv::CASCADE_SCALE_IMAGE,
                                cv::Size(90,110) );

    if (faces.size() == 1) {
        center.x = faces[0].x + faces[0].width*0.5;
        center.y = faces[0].y + faces[0].height*0.1;
        z = faces[0].width;
    }
    return true;
}
```

La funzione detectMultiscale rileva i volti e inserisce i risultati in un vettore di matrici, faces. Si controlla che sia rilevato un solo volto, altrimenti si potrebbe generare confusione tra le coordinate.

In seguito sono salvate le tre coordinate:

- Per la x si considera la coordinata x del centro del volto rilevato.
- Per la y si considera la coordinata y a 1/10 dell'altezza del volto, partendo dall'alto. Con varie prove si è notato che gli occhi sono rilevate più o meno a questa altezza, perciò si è adottata questa misura per collegare nel miglior modo la telecamera agli occhi dell'utente.
- Per la z si considera la larghezza del volto in pixel. Purtroppo non è stato trovato un modo migliore per stimare la distanza dallo schermo, perciò ci si è accontentati di questa misura abbastanza grossolana.

In seguito il codice presente nel parte in Ogre3D, in cui viene richiamata la funzione descritta in precedenza:

```
Ogre::Real x,y,z;  
int coordX = 0 ,coordY = 0,coordZ = 0;  
if (!getFaceCoord( &coordX , &coordY , &coordZ ) ) {  
    x = f1X->interpolate(prevX);  
    y = f1Y->interpolate(prevY);  
    z = f1Z->interpolate(prevZ);  
}
```

Il metodo `getFaceCoord()` viene richiamato passando come parametri tre interi. Se la funzione ritorna true si continua con le fasi successive, altrimenti c'è stato un errore nel frame corrente e vengono prese in considerazione le coordinate del frame precedente, per evitare crash nell'applicazione. `x`, `y` e `z` sono di tipo `Real`, un tipo proprio di Ogre, equivalente al `double`. Le tre funzioni all'interno del blocco `if` saranno trattati nella fase successiva.

### 4.1.2 filtro

Come abbiamo detto nel paragrafo relativo ad OpenCv, vari fattori determinano un riconoscimento non sempre perfetto; ad esempio, rimanendo fermi, le coordinate rilevate oscillano sempre di minimo 2 o 3 pixel, e questo determina una fastidioso tremolio nella scena.

Per ovviare (almeno in parte) a questi problemi, è creato un piccolo algoritmo che filtra il rumore generato dal rilevamento, cercando di preservare la fluidità nei movimenti. In seguito la funzione che elimina il rumore di fondo.

```

void Filter::filter(Ogre::Vector2 &cVec, int erMin,
                     int erMax) {
    int coord = cVec.x; // coordinata attuale
    int p      = cVec.y; // coordinata precedente

    // se coord = 0 il rilevamento non e' avvenuto
    // perciò si considera la coordinata precedente
    if (coord == 0) {
        cVec.x = p;
        return;
    }

    int dif = abs( coord - p );

    // se e' sotto la soglia minima salva
    // la coordinata precedente
    if ( dif <= erMin ) {
        cVec.x = p;
    }
    // altrimenti pone la precedente uguale all'attuale
    else {
        cVec.y = coord;
    }
}

```

La funzione prende in input un vettore con la coordinata attuale e precedente, e una soglia di errore. Se la differenza in valore assoluto tra le due coordinate è minore della soglia, allora si scarta la nuova coordinata e si utilizza la precedente, non generando movimento nella scena. Questo elimina il rumore, ma genera un altro problema: quando l'utente si sposta normalmente la scena si trasforma con movimenti a scatti, dovuti proprio

al filtro che elimina comunque dati utili.

Per risolvere questo secondo problema si è creato un secondo algoritmo che interpola i movimenti, rendendoli più fluidi generando coordinate intermedie tra l'attuale e la precedente. Analizziamo il codice.

```
Ogre::Real FluidFilter::interpolate(Ogre::Real c) {
    // se la coordinata rilevata e' la stessa
    // si avvicina ad essa
    if ( lastCoord == c ) {

        if ( count == 1) {
            count--;
            last = c;
        }
        else if ( count > 0) {
            count--;
            last += gap;
        }
    }

    // altrimenti ricalcola il gap
    else {
        lastCoord = c;
        count = n;
        gap    = ( c - last ) / n ;
        last += gap;
    }

    return last;
}
```

La classe FluidFilter contiene, fra gli attributi, un contatore e la coordi-

nata interpolata nel frame precedente. Il contatore diminuisce quando le coordinate non cambiano, in modo da rendere l'interpolazione finita. Se le coordinate cambiano invece si calcola un gap dato dalla differenza tra quella attuale e quella interpolata precedentemente, diviso per n; n influisce sulla qualità dell'interpolazione: all'aumento di n i movimenti sono più fluidi, ma, come effetto contrario, sono più rallentati. Con un valore di n pari a 3 o 4 l'effetto finale è accettabile, rendendo il delay quasi impercettibile e la fluidità efficiente.

### 4.1.3 3d

Le coordinate fornite da OpenCv sono interi che rappresentano il centro del volto rilevato. Per esempio in una webcam con risoluzione  $640 \times 480$  la x oscilla tra 0 e 640 e la y tra 0 e 480; inoltre l'origine si trova nell'angolo in alto a sinistra.

Il sistema di coordinate del mondo 3D è differente ed ha origine nel punto  $(0, 0, 0)$ . Perciò le coordinate devono essere trasformate per essere utilizzate nel nuovo sistema di coordinate

Per una maggiore efficienza, le coordinate dovrebbero essere calcolate anche in funzione della distanza dalla telecamera; inoltre dovrebbero variare anche in base alle dimensioni della scena importata. Per ora questo studio non è stato effettuato perciò, utilizzando una scena di default, sono stati adottati i seguenti calcoli:

```
z = z / 150.0f;
x = -( ( ( x - xCam/2 ) / 15.0f ) / (xr*ar) );
y = -( ( ( y - yCam/2 ) / 15.0f ) / (yr*ar) );
```

$xCam$  e  $yCam$  rappresentano le dimensioni della webcam in pixel. La prima sottrazione nella x e nella y spostano l'origine di OpenCv nell'origine del mondo virtuale.

$ar$  è il reciproco dell'aspect ratio della finestra aperta.

$xr$  e  $yr$  sono rispettivamente il rapporto delle dimensioni della finestra per quelle della webcam, sempre in pixel. Questa divisione finale dovrebbe permettere di mantenere una proporzionalità dei movimenti utilizzando risoluzioni diverse.

### 4.1.4 perspective matrix

Si calcolano i 6 piani del frustum in base alle coordinate rilevate, in modo da modificare la prospettiva in ogni frame.

```
n = -face.z + nP;      //near plane
f = n + sceneLength;   //far plane
r = width - face.x;   //right plane
l = r - 2*width;       //left plane
t = height - face.y;  //top plane
b = t - 2*height;      //bottom plane
```

Il vettore *face* contiene le coordinate calcolate.

Il near plane è calcolato in base alla distanza (approssimata) dell'osservatore, in modo che, quando si avvicina allo schermo, si produce un effetto di allungamento della scena . Per fare un esempio immaginiamo di vedere una finestra da lontano; più ci avviciniamo alla finestra e più ampia diventa la vista di ciò che c'è al di fuori. Inoltre esso varia in base alla costante *nP* definita dall'utente, che permette di modificare la distanza focale di default a piacimento.

Il far plane dipende dal near plane e dalla lunghezza della scena.

I successivi quattro piani sono calcolati in modo che formino sempre un rettangolo di dimensioni variabili, con lo scopo di ottenere un frustum asimmetrico che distorce la prospettiva della visuale. *width* e *height* coincidono rispettivamente con la metà della larghezza e dell'altezza della scena. Come vedremo in seguito, queste dimensioni dovrebbero rispettare l'aspect ratio della finestra aperta.

In seguito viene richiamata una funzione che prende in input le sei coordinate e, nello stesso modo che abbiamo visto nella teoria, calcola la matrice di proiezione, restituendola:

```
Ogre::Matrix4 frustum = Ogre::Matrix4(
    2*n/(r-l), 0, (l+r)/(r-l), 0,
    0, 2*n/(t-b), (t+b)/(t-b), 0,
    0, 0, f/(n-f), (n*f)/(n-f),
    0, 0, -1, 0
);
```

### 4.1.5 view

Come abbiamo visto nel paragrafo relativo alla view matrix, occorrono tre vettori per costruire quest'ultima. In seguito il codice relativo al calcolo dei primi due vettori:

```
eye     = Ogre::Vector3(-face.x, -face.y, -face.z);
```

```
target = Ogre::Vector3(r - width, t - height, -n);
```

*eye* utilizza le coordinate del volto, con il segno opposto. Quando il frustum viene spostato lungo una direzione, l'effetto prodotto è uno spostamento della scena nel verso opposto. Questo perchè il frustum rappresenta la porzione di spazio da visualizzare, non la scena. Ad esempio se noi stiamo guardando un oggetto nel frattempo che ci stiamo muovendo verso destra, dal nostro punto di vista l'oggetto si starà spostando verso sinistra. Perciò, per mantenere allineati i movimenti, il vettore *eye* deve essere l'inverso di *face*. Senza questa accortezza si crea un effetto di rotazione, come è possibile vedere nella figura 4.1.



Figura 4.1: Trasformazione ottenuta calcolando in modo errato il vettore *eye*.

*target* è il centro del rettangolo che rappresenta il near plane, in quanto il punto osservato deve essere sempre il centro della scena.

In seguito viene richiamato un metodo che calcola la view matrix:

```
Ogre::Vector3 up = Ogre::Vector3(0, 1, 0);
Ogre::Vector3 z = (eye - target);
```

```

z.normalise();      // forward vector.
Ogre::Vector3 x = (up.crossProduct(z));
x.normalise();    // right vector.
Ogre::Vector3 y = z.crossProduct(x); // up vector.

Ogre::Matrix4 viewMatrix = Ogre::Matrix4(
    x.x, x.y, x.z, -(x.dotProduct(eye)),
    y.x, y.y, y.z, -(y.dotProduct(eye)),
    z.x, z.y, z.z, -(z.dotProduct(eye)),
    0,     0,     0,           1
);

```

Il vettore *up* equivale all’asse Y; la base ortonormale la matrice sono calcolati come è stato visto nella parte teorica.

#### 4.1.6 trasl

Con le trasformazioni calcolate in precedenza otteniamo colleghiamo la prospettiva della visuale con la posizione dell’utente, però vengono generati anche degli spostamenti indesiderati della scena che la fanno scomparire oltre lo schermo, in quanto la telecamera virtuale in realtà sta fluttuando in base ai nostri movimenti.

Vediamo nella figura 4.2 l’effetto indesiderato.

La vera forza della trasformazione sta nel fatto che la scena deve invece rimanere ferma, fissata ai bordi dello schermo, e per questo occorre una traslazione che annulli gli spostamenti precedenti.

```

Ogre::Vector3 translVec = Ogre::Vector3(
    -face.x + r - width,
    -face.y + t - height,

```



Figura 4.2: Scena fluttuante, dovuta all'assenza di traslazione.

```

    0
);

Ogre::Matrix4 translMat = Ogre::Matrix4::IDENTITY;
translMat.makeTrans(translVec);

```

La matrice di traslazione è calcolata con la funzione *makeTrans*, che prende come parametro il vettore che annulla lo spostamento della telecamera.

#### 4.1.7 fine

In OpenGL la trasformazione della telecamera è rappresentata da una matrice, che è il prodotto tra la traslazione, la matrice di proiezione e la view matrix:

```

camera = translMat * frustum * viewMatrix;

```

In Ogre questa operazione è eseguita automaticamente dal framework, quindi basta solamente impostare le due matrici, utilizzando come view matrix il prodotto tra la matrice di traslazione per la view matrix calcolata a mano.

```
mCamera->setCustomViewMatrix(true, translMat*view);  
mCamera->setCustomProjectionMatrix(true, frustum);
```

## 4.2 Importazione scene in OpenGL

Un’ulteriore feature che si voleva aggiungere all’applicazione era quella di poter caricare una qualsiasi scena, modellata con programmi come Blender, e poterla visualizzare in modo più interattivo, creando l’illusione 3D.

In OpenGL è utilizzato un parser di file con estensione .obj, in modo da caricarli nel buffer e renderizzarli a schermo. All’avvio del programma viene esaminata una directory apposita e, per ogni file .obj trovato, ne viene fatto il parsing e vengono inserite caratteristiche quali coordinate dei vertici, coordinate uv per il mapping delle texture, etc in diversi buffer.

Purtroppo questo processo si è rivelato oneroso in OpenGl e ha portato anche ad alcune problematiche che hanno determinato la scelta di compiere questo ulteriore passo utilizzando il framework Ogre3D.

## 4.3 Importazione in Ogre3D

Esistono diversi exporter per software di modellazione quali Blender, Maya, 3D Studio Max, etc, per esportare scene e convertirle nel formato riconosciuto da Ogre. I modelli hanno estensione .mesh, mentre le informazioni relative ai materiali sono raccolte in file con estensione .material, tramite un linguaggio proprio di Ogre, facilmente comprensibile.

Per ottenere effetti più apprezzabili nelle trasformazioni, conviene che la scena rappresentata sia racchiusa da un box, con l’apertura frontale. Come

esempio è stata preparata una scena in Blender, raffigurante un salotto, come si può vedere nella figura 4.3.



Figura 4.3: Scena modellata in Blender.

Da notare il fatto che le misure del box che racchiude la scena, in questo caso, ha un aspect ratio di 16/9, infatti la lunghezza è pari a 16 unità mentre l'altezza è di 9 unità. Questo perché lo schermo utilizzato ha lo stesso aspect ratio. In Ogre è possibile scegliere la risoluzione della finestra all'avvio dell'applicazione, perciò si deve fare attenzione a rispettare il formato, in quanto rapporti diversi possono creare effetti di stretching o di rimpicciolimento lungo una o entrambe le dimensioni.

Per ottenere una trasformazione corretta, il nostro frustum dovrebbe avere le stesse dimensioni della scena importata, ed inoltre il near plane va posizionato ad una distanza pari alla distanza da dove comincia la scena, in modo che il near plane coincida con l'apertura del box.

Un'ulteriore operazione da fare, all'avvio dell'applicazione, è quella di traslare la scena in modo che sia centrata nell'origine, per rientrare nella porzione di spazio considerato nel frustum. Dopo la traslazione la scena

sarà posta come è visibile nella figura 4.4. L'origine è rappresentato dai tre assi cartesiani colorati.



Figura 4.4: La scena dopo la traslazione.

Per quanto riguarda la Z, la nostra scena inizia alla coordinata  $y = 2$  ( la coordinata Y di Blender corrisponde alla Z in Ogre3D), perciò il near plane sarà posto ad una distanza pari a 2 lungo l'asse z, rispetto all'origine. Ovviamente il near plane è variabile, per determinare gli effetti di allontanamento o avvicinamento, ma comunque lo spostamento è corretto dalla view matrix, così che il near plane è sempre coincidente con l'apertura del box.

Tutte queste condizioni sono necessarie affinchè i bordi della scena siano sempre fissati ai bordi dello schermo del pc, in modo da illudere maggiormente l'occhio. Ovviamente senza questi accorgimenti la trasformazione continuerebbe comunque a funzionare in modo corretto, ma gli effetti ottenuti non garantirebbero l'illusione che l'applicazione ha lo scopo di creare.

# Capitolo 5

## Risultati e discussioni

### 5.1 Difficoltà e problematiche riscontrate

Nello sviluppo del progetto sono state riscontrate varie problematiche, dovute anche ai limiti del software e dell'hardware utilizzato.

#### 5.1.1 Lo studio delle trasformazioni

Lo studio per formulare il metodo usato per creare le giuste trasformazioni, ha occupato diverse giornate di lavoro, all'incirca una trentina. Questo soprattutto perchè la computer grafica, quando si tratta di teoria matematica, non è sempre facilmente comprensibile.

Infatti, dopo aver studiato le basi delle trasformazioni, trattate anche in questa tesi, esse sono state sviluppate procedendo sia per ragionamenti dovuti a calcoli preliminari, sia per tentativi puramente casuali. Fare prove inserendo valori casuali nelle varie operazioni è stato d'aiuto, in effetti, per comprendere meglio le variazioni del comportamento del sistema. Una volta trovata la trasformazione corretta, tuttavia, il procedimento ancora non era ben chiaro, perciò si è continuato a fare ulteriori prove. Tutto ciò è stato utile

a comprendere quando la trasformazione funzionasse, o meglio, a stabilire le condizioni necessarie per produrre l'effetto desiderato e quindi rendere la trasformazione efficace.

### 5.1.2 I limiti dell'hardware

L'obiettivo del progetto era quello di creare un'applicazione usufruibile da ogni utente, disponendo solamente di un pc e una webcam. Tuttavia si è notato che la qualità dell'hardware incide nella resa dell'applicazione.

Le prime prove erano fatte utilizzando le coordinate del mouse, al posto delle coordinate del volto rilevato. Si è notato che, importando il modulo OpenCv, la velocità di esecuzione veniva fortemente rallentata, perché il solo rilevamento è un processo molto dispendioso. Aggiungendo il carico di lavoro dovuto alla parte grafica, il risultato era inaccettabile, renderizzando circa 15 frames al secondo (utilizzando un processore Intel I5 e una scheda grafica ATI Radeon HD 5650, quindi hardware di qualità medio-alta).

Fortunatamente esisteva un parametro, nella funzione *detect\_multi\_scale()* situato nel modulo OpenCv, che ha permesso di migliorare la velocità senza diminuire la qualità. Questo parametro era relativo alle dimensioni dell'area minima da analizzare per il rilevamento; più è piccola l'area e più calcoli deve eseguire l'algoritmo per riconoscere i volti. All'inizio era impostato a (30, 30), in questo modo i volti venivano ricercati in aree di minimo 30 pixel per 30. Aumentando questo valore a circa (100, 100), la velocità di esecuzione è stata portata a 30 frames al secondo, utilizzando anche scene complesse, in cui erano presenti alcune centinaia di migliaia di vertici. Questi valori permettono, in una webcam di risoluzione  $640 \times 480$ , di rilevare volti fino ad una distanza di circa un metro e mezzo dallo schermo, perciò il compromesso è accettabile.

La qualità della webcam può incidere nel miglioramento della qualità, potendo diminuire il rumore generato dall'imprecisione del rilevamento. Tuttavia, aumentare la risoluzione della webcam potrebbe incidere nella velocità, per il motivo precedente, ovvero perché i frames che OpenCv deve analizzare, hanno dimensioni più elevate, perciò deve effettuare più calcoli.

Allo stesso modo può influire l'angolo di apertura della webcam: più è elevato e meglio si può apprezzare l'illusione, in quanto l'utente ha più spazio per potersi muovere senza essere perso dall'obiettivo della webcam, generando interruzioni nella trasformazione.

### 5.1.3 I limiti di OpenCv

Come è stato già accennato, OpenCv fornisce dei metodi che hanno delle limitazioni: il rilevamento non è preciso, generando sempre una sorta di oscillamento nelle coordinate rilevate durante il tracking, e non è in grado di stimare la distanza dallo schermo.

Per quanto riguarda il primo limite si può dire che, grazie all'aggiunta dell'algoritmo che funge da filtro, il problema è stato risolto senza incidere troppo sulla qualità finale.

Per quanto riguarda il secondo limite invece non è stato raggiunto il risultato sperato. Per un effetto più realistico infatti, la trasformazione dovrebbe tenere conto in modo preciso della distanza dell'utente, perché le coordinate x e y andrebbero calcolate in funzione anche di essa.

Questo perchè anche la webcam possiede un proprio frustum, e quindi spostamenti lungo l'asse x o y non rimangono proporzionati al variare della distanza, ovvero la coordinata z. Infatti, all'aumentare della distanza dallo schermo, gli spostamenti rilevati lungo gli assi x e y variano più lentamente, di conseguenza la prospettiva sarà trasformata più lentamente.

Il calcolo della distanza non è banale con una semplice webcam. Bisognerebbe effettuare una calibrazione ad ogni esecuzione per misurare la grandezza della testa dell’utente in modo da trovare un coefficiente per ric算olare la x e la y. Poichè anche il rilevamento non è preciso e varia in base a molti fattori, questa caratteristica richiede uno studio molto approfondito, e un gran numero di prove.

Il problema potrebbe essere risolto utilizzando un sensore apposito, come il Kinect della Microsoft, che stima in modo abbastanza preciso le distanze grazie a raggi infrarossi, ma questo andrebbe contro la logica dell’applicazione, che mira ad offrire questa forma di interazione utilizzando semplicemente una webcam.

### 5.1.4 Importazione modelli in OpenGL

In OpenGL è stato riscontrato un problema nell’importazione dei modelli 3D con estensione .obj. Infatti quando erano renderizzati a schermo, risultava che la trasformazione che essi subivano fosse ritardata di uno o due frames rispetto agli altri oggetti disegnati direttamente in OpenGL. Questo generava un’irregolarità nel movimento apparente degli oggetti, producendo un’effetto simile ad una sorta di tremolio.

Il difetto probabilmente era legato ai driver della scheda grafica, tuttavia non sono state fatte prove su altre macchine quindi il problema è rimasto irrisolto.

Successivamente lo sviluppo è passato ad Ogre3D, che permette di gestire più efficientemente scene importate dall’esterno, perciò la parte OpenGL è stata messa da parte. Con Ogre3D il problema non è ricomparso.

## 5.2 I requisiti dell'applicazione

In questo paragrafo riassumiamo le condizioni che devono essere rispettate per la miglior resa dell'applicazione:

- *Ambiente*: L'ambiente in cui si usa l'applicazione deve essere ben illuminato, per riconoscere meglio i volti. La scarsa illuminazione rende anche più faticoso il rilevamento, causando una caduta di prestazioni in termini di velocità di esecuzione.

Inoltre lo spazio intorno all'utente dovrebbe essere libero da cose che possono creare interferenze o falsi positivi (altre persone, magliette, disegni o poster raffiguranti volti, etc).

- *Webcam*: La webcam dovrebbe essere di buona qualità, ma con una risoluzione non troppo alta, per evitare che le dimensioni dei frame rendano il rilevamento troppo complesso (questo se si utilizzano un processore e una scheda grafica di media potenza).

Inoltre la webcam deve essere posizionata sopra lo schermo, al centro, cioè nella posizione di default dei computer portatili. Se viene posizionata in altri punti, il volto rilevato renderebbe la prospettiva dell'utente non più allineata con quella della telecamera virtuale.

- *Scena 3D*: Per rendere l'illusione più efficiente (come già detto la trasformazione è corretta anche se la scena non rispetta queste condizioni), la scena 3D importata dovrebbe essere racchiusa in un box, le cui dimensioni dovrebbero essere uguali alle dimensioni del frustum, con l'apertura giacente sul near plane. Inoltre l'aspect ratio del box dovrebbe rispettare quella dello schermo, per evitare stiramenti o rimpicciolimenti della scena.

Nel progetto sviluppato in Ogre3D è stato aggiunto un file di configurazione, dove l'utente può specificare le coordinate dei punti che identificano il suo box. Tramite queste coordinate il programma imposta le dimensioni del frustum e trasla la scena centrandola nell'origine, allineandola così con il frustum.

Grazie a queste accortezze l'apertura del box rimane coincidente con lo schermo (se si usa lo schermo intero), e, durante la trasformazione di prospettiva, i bordi rimangono allineati, creando l'illusione che la scatola abbia profondità.

### **5.3 Risultati**

Nel corso dello sviluppo del progetto sono stati apportati notevoli miglioramenti nel funzionamento dell'applicazione. Essa è stata provata da alcuni utenti (familiarì e conoscenti), e tutti sono rimasti affascinati dall'effetto prodotto. Arrivati al punto attuale, perciò, si può dire che i risultati sono accettabili, in quanto l'illusione che si vuole creare è più che apprezzabile.

# Conclusioni

In questa tesi è stata trattata un'applicazione che presenta un'interfaccia uomo-macchina per l'interazione con un sistema virtuale. In particolare essa trasforma lo stesso utente nel controller del sistema visivo per una scena virtuale. La prospettiva con cui è vista la scena è trasformata in funzione delle coordinate del volto rilevato. L'obiettivo è creare l'illusione che ci sia profondità all'interno dello schermo, generando un effetto 3D.

L'applicazione è stata sviluppata utilizzando software completamente open-source. Sono stati usati: OpenCv per il rilevamento del volto, OpenGL per lo studio della trasformazione adottata, Ogre3D per il miglioramento della resa della scena. Inoltre per la modellazione 3D si è usato Blender.

L'applicazione offre un'ulteriore feature, ovvero la possibilità di importare modelli dall'esterno. Esistono exporter che permettono di esportare scene modellate con diversi software, nel formato riconosciuto da Ogre3D. Tuttavia per ora si è preso in considerazione solamente l'exporter di Blender, perciò il funzionamento è garantito solo per scene esportate con questo software.

Richiedendo solamente un pc e una webcam, il programma è accessibile da tutti, ma presenta dei limiti. Il limite che influenza maggiormente è l'assenza del calcolo preciso della distanza dell'utente dallo schermo. Questo problema rende non uniformi gli spostamenti dell'utente, se si trova a

distanze differenti.

Al di là dei limiti, l'applicazione produce un effetto positivo, soprattutto nell'utente medio, magari non abituato ad illusioni di questo genere, perciò il risultato si può considerare più che accettabile.

L'applicazione presenta grandi potenzialità, potendo estendere il suo utilizzo per svariati scopi, a partire da semplice forma di intrattenimento, fino ad essere impiegato in simulazioni virtuali più complesse, come ad esempio i videogiochi. Il fatto che l'applicazione sia stata sviluppata con software open-source la rende completamente aperta a miglioramenti e sviluppi futuri.

# Bibliografia

- [1] Vella, F., Cefalà, R., Costantini, A., Gervasi, O., Tanci, C.: Gpu computing in egi environment using a cloud approach. In: 2011 International Conference on Computational Science and Its Applications. pp. 150–155. IEEE (2011)
- [2] intro
- [3] NUI [http://it.wikipedia.org/wiki/Interfaccia\\_utente\\_naturale](http://it.wikipedia.org/wiki/Interfaccia_utente_naturale)
- [4] realta
- [5] [http://it.wikipedia.org/wiki/Realtà\\_virtuale](http://it.wikipedia.org/wiki/Realtà_virtuale)
- [6] [http://www.treccani.it/enciclopedia/realta-virtuale\\_\(Encyclopædia\\_Italiana\)/](http://www.treccani.it/enciclopedia/realta-virtuale_(Encyclopædia_Italiana)/)
- [7] opencv
- [8] <http://en.wikipedia.org/wiki/OpenCV>
- [9] [http://it.wikipedia.org/wiki/Wavelet\\_Haar](http://it.wikipedia.org/wiki/Wavelet_Haar)
- [10] [http://en.wikipedia.org/wiki/Viola-Jones\\_object\\_detection\\_framework](http://en.wikipedia.org/wiki/Viola-Jones_object_detection_framework)

## **Conclusioni**

---

[11] [http://en.wikipedia.org/wiki/Haar-like\\_features](http://en.wikipedia.org/wiki/Haar-like_features)

[12] opengl

[13] <http://it.wikipedia.org/wiki/OpenGL>

[14] ogre

[15] <http://en.wikipedia.org/wiki/OGRE>

[16] frustum

[17] progetto

[18] risultati

[19] conclusioni