Università degli studi di Padova
Facoltà di Ingegneria

Tesi di Laurea in

Ingegneria Informatica

Fusione dell'informazione di colore proveniente da diverse telecamere nel Free Viewpoint Video

Relatore
Prof. Pietro Zanuttigh

Candidato
Andrea Campanaro

Anno Accademico 2009/2010

A coloro che hanno condiviso anche brevi momenti della mia vita,
che mi hanno permesso di diventare quel che sono ora
e di arrivare fino a questo incredibile traguardo.

Sommario

Questa tesi riguarda come ottenere una rappresentazione in 3 dimensioni fedele al soggetto reale che si desidera modellizzare partendo da una *mesh* e dalle immagini relative al soggetto stesso acquisite in una stanza opportunamente allestita.
Vengono quindi brevemente presentati i software utilizzati per la creazione del modello 3D e per visualizzare lo stesso a video ed operare diverse trasformazioni. Verranno quindi spiegati gli inconvenienti incontrati nelle immagini di partenza dovuti a diversi tipi di distorsione ed i rimedi adottati per ovviare a tali problemi.
Successivamente verrà affrontato il tema della sovrapposizione delle immagini di uno stesso frame sul modello e verranno proposte diverse tipologie di soluzione al problema della scelta corretta delle immagini, in particolare nelle regioni sovrapposte di 2 fotogrammi provenienti da telecamere adiacenti. Infine verranno descritte le strutture dati utilizzate ed evidenziate alcune porzioni di codice C++ utili all'implementazione delle soluzioni adottate.

Indice

Capitolo 1

Introduzione

Lo scopo di questo lavoro è di trovare un buon metodo per fondere delle immagini di un modello in 3 dimensioni, acquisite da 4 telecamere fisse, in modo di assegnarle in maniera corretta ed evitando delle incongruenze nella sovrapposizione di immagini provenienti da telecamere vicine tra loro.

Gli strumenti di partenza sono un programma per ricostruire il modello in 3 dimensioni di un soggetto all'interno di una stanza, attrezzata con 4 telecamere fisse agli angoli che riprendevano i movimenti effettuati al suo interno, ed un programma per la visualizzazione del modello in 3 dimensioni con tecnologia *OpenGL*. Al momento di rappresentare il modello, in cui gli venivano sovrapposte le immagini corrispondenti, si notava però un vistoso disallineamento delle immagini rispetto alla struttura ed una mancata regolamentazione della sovrapposizione delle immagini, in cui era evidente il confine tra un'immagine ed un'altra appartenente alla telecamera adiacente, senza un particolare algoritmo per risolvere tale accavallamento anomalo.
Si è quindi cominciato ad analizzare il motivo di tale anomalia, prima nel codice sorgente del visualizzatore, poi nelle immagini stesse di partenza per capire dove venisse generato l'errore e trovarne una soluzione. Una volta appurato che le immagini di partenza erano affette da distorsione radiale e tangenziale, dovuta all'ottica delle telecamere utilizzate per l'acquisizione, si è proceduto a correggerle e ad applicarle nuovamente al modello. Il passo successivo è quindi stato quello di scrivere un programma, scritto in linguaggio C++, per regolamentare l'assegnazione del colore ad un determinato pixel, prima con una semplice formula di pesatura e successivamente introducendo ulteriori parametri per raffinare il risultato prendendo in considerazione anche il punto di vista dell'utente e la visibilità dei pixel in ombra, per poter così eliminare le immagini non correttamente visualizzate nel calcolo dei pesi finali, nel caso in cui non siano effettivamente visibili da una determinata telecamera. In questo modo si è arrivati ad ottenere una sovrapposizione

molto buona delle immagini sul modello, in cui le varie parti combaciano con il posto in cui sono posizionate nella realtà e le zone intermedie tra 2 immagini vengono smussate, correggendo le fastidiose imperfezioni dovute alle colorazioni dei punti non direttamente visibili dalle telecamere.

Per la risoluzione si è fatto affidamento al modello della telecamera prospettico *(pinhole)* per ciò che riguarda il modello geometrico della formazione delle immagini, si è utilizzato *GLview*, che si basa su *OpenGL*, come software per la visualizzazione del modello in 3 dimensioni. Il codice è stato sviluppato in C++, utilizzando una libreria *Open Source* per la gestione delle immagini Bitmap di nome *EasyBMP*.

Nel capitolo 2 verranno introdotti gli strumenti che stanno alla base dello sviluppo del progetto, in questo caso 2 software che si occupano uno della creazione del modello 3D e della costruzione della *mesh* ed un altro che ci permette di visualizzare tale modello a video ed effettuare una serie di operazioni di modifica e salvataggio sullo stesso. Nel capitolo 3 verrà presentata una soluzione all'inconveniente riscontrato in corso d'opera riguardo la distorsione radiale e tangenziale che affligge le immagini a nostra disposizione salvate dalle telecamere al momento dell'acquisizione all'interno della scena appositamente creata. All'interno del capitolo 4 verrà invece trattata la parte teorica riguardante i criteri di assegnazione del colore ad un determinato pixel che si trova sulla superficie del modello ricostruito, approfondendo i passi che hanno portato dal quesito principale ad un'affinazione della ricostruzione introducendo diversi parametri per migliorare il risultato finale. Infine il capitolo 5 contiene la descrizione dell'implementazione del concetto teorico discusso nel capitolo 4 e la spiegazione delle strutture dati utilizzate in **C++** ed approfondimenti di natura tecnica, oltre ad alcuni esempi del codice utilizzato.

Capitolo 2

Creazione e visualizzazione del modello 3D

2.1 Creazione del modello 3D

Per ottenere il modello tridimensionale su cui effettuare tutte le operazioni del lavoro bisogna utilizzare il programma *GetFrame*, creato da Ballan Luca come tesi di dottorato in Ingegneria dell'Informazione presso l'Università di Padova [1] [2], che permette di ricavare un modello 3D dalle 4 viste di un determinato frame scelto dall'utente nella linea di comando al momento del lancio del programma stesso. Tale modello altro non è che la ricostruzione della *mesh* dell'oggetto sul quale si vuole operare, nel nostro caso il fermo immagine dei movimenti di una persona. Una volta che il programma ha processato il frame di nostro interesse il programma restituisce in output diversi files, tra i quali un file WRL (.wrl è l'estensione di un file VRML). Un file con estensione .wrl, comunemente chiamato *world* (mondo), contiene al suo interno tutte le informazioni necessarie a descrivere una scena tridimensionale. Il file VRML contiene informazioni precise su come sono formati gli oggetti e sulla loro posizione nello spazio, in particolare gli aspetti geometrici dei vari elementi, composti da poligoni 3D definiti da vertici e spigoli, insieme alle loro proprietà (colore, *texture*, trasparenza o altro).

Lo scopo della costruzione di *GetFrame* è il recupero di una descrizione matematica tempo-variante di tutta la scena 3D utilizzando solo le informazioni estratte da sequenze video registrate da alcune telecamere disposte all'interno dell'ambiente. I video tridimensionali (3D-video) ed il video con punto di vista libero (FVV) sono i nuovi tipi di media che ampliano l'esperienza degli utenti al di la di quanto viene offerto dai media tradizionali. Il video 3D offre una sensazione di profondità tridimensionale della scena osser-

vata, mentre FVV consente una selezione interattiva del punto di vista e della direzione entro un certo intervallo di funzionamento. Il sistema FVV viene utilizzato per consentire una coinvolgente esperienza di un evento reale in cui ogni spettatore può cambiare dinamicamente il suo punto di vista durante la proiezione. Chiaramente, al fine di generare questo tipo di informazione, deve essere acquisito un modello parziale o completo dell'intero evento ed in questo caso le tecniche di ricostruzione 3D passive giocano un ruolo cruciale. Il sistema utilizzato nel software si occupa di catturare la forma, l'aspetto ed il movimento di persone ed oggetti che interagiscono tra loro, utilizzando solamente tecniche passive e non invasive. Dato uno scenario il sistema è in grado di fornire una descrizione tempo-variante dell'intera sequenza 3D considerando sia la sua geometria che il suo aspetto. Un utente è quindi in grado di navigare all'interno di questa rappresentazione e guardare l'azione da un qualsiasi punto di vista. In particolare la geometria della scena è modellata con *Mesh* tempo-consistenti ed alcune strutture scheletriche connesse ad entità articolate della scena. Il suo aspetto, invece, è modellato con semplici informazioni sul colore di tipo Lambertiano.

L'acquisizione viene effettuata in due fasi distinte. Prima la forma e l'aspetto di ogni attore viene acquisita da uno scanner passivo del corpo. Successivamente gli attori sono invitati ad entrare in una seconda stanza in cui avrà luogo l'azione vera e propria. Un sistema di cattura del movimento di tipo *marker-less* viene utilizzato per catturare i loro movimenti ed il movimento di tutti gli oggetti con cui interagiscono. Più precisamente quest'ultimo sistema stima la posizione assunta da ciascun elemento del soggetto e di ogni oggetto in tutti i fotogrammi dell'azione registrata.

L'acquisizione dei video è stata effettuata in una stanza completamente rivestita di pannelli di colore blu elettrico al fine di aumentare il contrasto con il colore rosa della pelle di ciascun attore e le diverse telecamere, sincronizzate tra loro, sono state disposte secondo lo schema di figura 2.1.

Il passo successivo è stato quello di creare una struttura dello scheletro della persona al fine di rendere validi solamente determinati tipi di movimenti. La struttura dello scheletro è ad albero e prevede un numero ed un tipo di movimenti limitati alle reali movenze dell'essere umano e trascura determinati movimenti troppo specifici, quali le dita delle mani. Un esempio di scheletro ed una tabella di movimenti consentiti sono rappresentati in figura 2.2 ed in tabella 2.1.

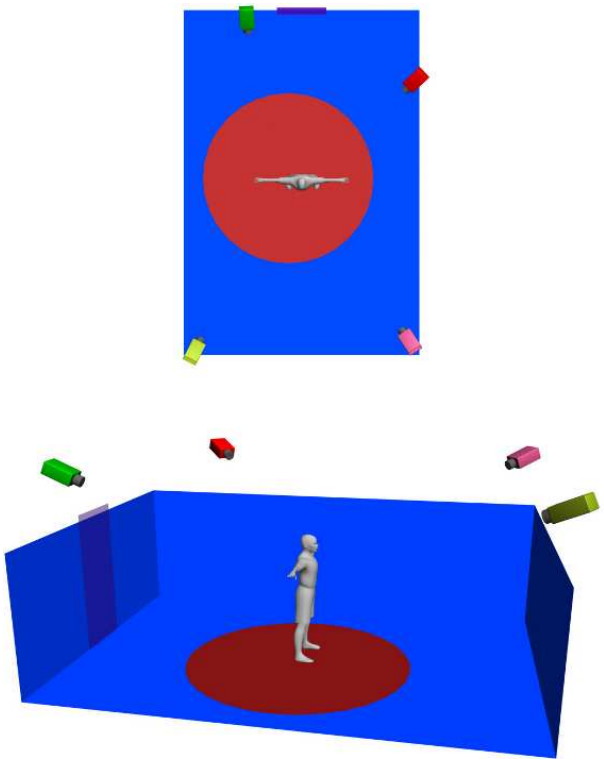

Fig. 2.1: Disposizione delle diverse telecamere all'interno della *Blue room*

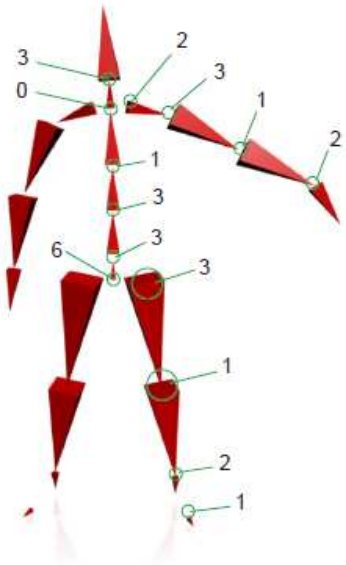

Fig. 2.2: Disposizione delle ossa nella struttura scheletrica utilizzata con i relativi gradi di libertà

	X-Axis		Y-Axis		Z-Axis	
	min	**max**	**min**	**max**	**min**	**max**
Pelvis	-180	180	-180	180	-180	180
Spine lv. 0	-15	15	-10	10	-10	60
Spine lv. 1	-20	20	-18	18	-10	60
Spine lv. 2	-180	180	-180	180	-180	180
Neck	-180	180	-180	180	-180	180
Head	-70	70	-50	50	-60	30
Clavicles	-30	30	-10	10	0	0
L Upper arm	-90	40	-40	85	-90	30
R Upper arm	-40	90	-85	40	-90	30
Forearms	0	0	0	0	-140	0
L Hand	-90	30	-80	80	0	0
R Hand	-30	90	-80	80	0	0
L Thigh	-90	90	-40	90	-160	90
R Thigh	-90	90	-90	40	-160	90
Lower legs	0	0	0	0	-140	0
Feet	-90	90	0	0	-90	50
Toes	0	0	0	0	-30	30

Tabella 2.1: Vincoli dello scheletro

Il concetto che sta alla base di tale programma è l'inseguimento del moto senza segnale *(marker-less motion tracking)* dell'intero corpo umano con un nuovo metodo che utilizza un modello di *skinned mesh* conosciuto a priori e tratta solamente spunti di moto in 2 dimensioni. La particolarità è l'utilizzo di una nuova funzione obiettivo che considera in modo univoco sia il flusso ottico che le informazioni della sagoma e stima le deformazioni non rigide della pelle del corpo, entro la struttura SSD *(Skeletal Subspace Deformation)*. Tale tecnica consente di tracciare movimenti umani complessi in maniera soddisfacente con un numero elevato di occlusioni in un sistema reale con 4 telecamere con modelli a 46 gradi di libertà. Per approfondimenti in merito vedere [2]
L'utilizzo di un modello basato su SSD nella minimizzazione permette di stimare correttamente anche piccole parti del corpo, quali clavicole ed ossa spinali, la cui deformazione è difficile da approssimare con sotto-elementi rigidi. Nel caso di movimenti umani veloci, che producono immagini con vistosi *motion blur*, è dimostrato che l'utilità del flusso ottico fornisce degli spunti per una corretta stima del movimento.

2.2 Visualizzazione del modello 3D

Il programma utilizzato per visualizzare il modello 3D e tutte le operazioni che verranno effettuate su di esso è **GLview**, sviluppato in **C++**, che non fa altro che caricare in **openGL** le *mesh* di *modeling* e mostrarne i risultati. All'interno del sorgente è possibile modificare diversi parametri per personalizzare la visualizzazione dell'output e definire la struttura del modello 3D da rappresentare.
Al momento del lancio del programma da linea di comando si possono scegliere diverse modalità di input a seconda dello scopo dell'utente, tra cui la possibilità di caricare o meno la *texture*, il modello e l'eventuale elenco dei files contenenti i parametri della telecamera e l'eventuale punto di vista scelto dall'utente all'interno di un file di testo. Una volta lanciato il programma con le impostazioni preferite dall'utente si apre la schermata vera e propria di visualizzazione del modello in cui è possibile effettuare diverse operazioni sulla vista a video. A seconda del tasto premuto sulla tastiera è possibile zoomare e ruotare l'immagine a video. L'elenco dei tasti e le azioni corrispondenti sono riportate nell'Appendice A.

A seconda del tasto premuto sulla tastiera è possibile zoomare e ruotare l'immagine a video, oltre ad eseguire delle traslazioni della stessa. Oltre a comandi da tastiera è possibile ruotare il modello anche attraverso l'utilizzo del

mouse, tenendo premuto il tasto sinistro e trascinando secondo la direzione di movimento desiderata.

Premendo invece il tasto destro del mouse si aprirà un menu a tendina in cui sono elencati una serie di comandi di notevole utilità da utilizzare sulla visualizzazione del modello, quali la possibilità di visualizzare la struttura della *mesh* evidenziata con triangoli di colore diverso, salvare l'immagine a video, salvare i valori della vista in cui ci si trova, visualizzare il modello con i soli bordi dei triangoli della *mesh* colorati coerentemente con l'immagine da input e con l'interno trasparente, oppure riportare il modello alla posizione iniziale.

Lo studio di tale programma e del sorgente in linguaggio C++ è di notevole importanza per capire il funzionamento del visualizzatore ed aiuta il programmatore a capire le operazioni che esso svolge e l'ordine in cui esse vengono eseguite permettendo di capire dove intervenire per migliorare la qualità del risultato.

Capitolo 3

Compensazione della distorsione delle lenti

A questo punto, dopo aver introdotto i due programmi che stanno alla base della modellizzazione 3D ed il visualizzatore del modello stesso, si passa alla soluzione pratica del problema di far aderire nel miglior modo possibile le immagini delle 4 telecamere per ciascun frame al modello ricostruito. Al momento di cominciare il lavoro si nota fin da subito che visualizzando le immagini salvate al momento dell'acquisizione nella camera adibita allo scopo non combaciano, come si penserebbe, al modello ricostruito e devono quindi essere adottati dei provvedimenti affinché le immagini diventino il più aderenti possibili allo stesso. Il primo intoppo che si è incontrato consiste nel trovare dove sia la sorgente di tale disambiguità ed analizzando tutti i dati in possesso si è cercato di capire se il problema fosse nel codice sorgente di GLview oppure nelle immagini salvate. Come primo passo si è quindi pensato di confrontare le immagini salvate da ciascuna telecamera con le sagome ottenute segmentando i dati delle camere, dette di tipo Render, e successivamente con le immagini ottenute rimuovendo la distorsione dalle immagini delle camere, salvate con il nome di Undistorted. Si è quindi notato che il problema non risiedeva nelle immagini ottenute trasformando le immagini originali secondo quanto detto in precedenza, ma proprio in un difetto relativo alle immagini originali acquisite con le telecamere all'interno della Blue room detta distorsione focale. Una volta appurato che il problema consiste nel fatto che le immagini sono affette da distorsione focale bisogna applicare una serie di funzioni che permettono di trasformare ciascuna immagine per compensare la distorsione radiale e focale dovuta alla lente della telecamera. All'interno di glview viene utilizzato openCV (OPEN Source Computer Vision), una libreria che fornisce funzioni di programmazione per la visione computerizzata in real time [4]. Tale libreria si basa sul modello di camera

detto "*pinhole*" in cui la vista di una scena è formata proiettando punti 3D nel piano immagine usando una trasformazione prospettica del tipo evidenziato in (3.1)

$$
\begin{aligned}
s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} &= \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} . \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \\
s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} &= A[R|t] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}
\end{aligned}
\tag{3.1}
$$

Dove (X, Y, Z) sono le coordinate di un punto 3D nello spazio di coordinate reali, (u, v) sono le coordinate del punto proiettato in pixel, A è la matrice dei parametri intrinseci, (c_x, c_y) è il punto principale (solitamente al centro dell'immagine) e (f_x, f_y) sono le lunghezze focali espresse in pixel. Quindi, se un'immagine dalla telecamera è scalata di un qualche fattore, tutti questi parametri devono essere scalati (moltiplicati/divisi, rispettivamente) dello stesso fattore. La matrice dei parametri intrinseci non dipende dalla scena visualizzata e, una volta calcolata, può essere riusata (finché la focale della lente rimane fissata). La matrice di roto-traslazione $[R|t]$ è chiamata matrice dei parametri estrinseci ed è utilizzata per descrivere il movimento della telecamera attorno ad una scena statica, o viceversa, il movimento rigido di un oggetto di fronte alla telecamera. Perciò $[R|t]$ traduce le coordinate di un punto (X, Y, Z) in un qualche sistema di coordinate, fisso rispetto alla telecamera [3] [5].

La trasformazione (3.1) è equivalente a (3.2) (quando $z \neq 0$)

$$
\begin{aligned}
&\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \\
&x' = x/z \\
&y' = y/z \\
&u = f_x * x' + c_x \\
&v = f_y * y' + c_y
\end{aligned}
\tag{3.2}
$$

in cui x' è lŠinverso della dimensione efficace del pixel lungo la direzione u e y' è lŠinverso della dimensione efficace del pixel lungo la direzione v. Le lenti reali solitamente hanno qualche distorsione, soprattutto distorsione

radiale e in misura minore distorsione tangenziale. Quindi il modello è esteso come in (3.3)

$$\begin{aligned}
&\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \\
&x' = x/z \\
&y' = y/z \\
&x'' = x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\
&y'' = y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y' \\
&\text{dove} \quad r^2 = x'^2 + y'^2 \\
&u = f_x * x'' + c_x \\
&v = f_y * y'' + c_y
\end{aligned} \tag{3.3}$$

(k_1, k_2, k_3) sono i coefficienti di distorsione radiale, (p_1, p_2) sono i coefficienti di distorsione tangenziale, ed i valori di x'' ed y'' si ottengono moltiplicando i vari fattori per i coefficienti di distorsione radiale prima e tangenziale poi.
Per eliminare la distorsione focale dalle immagini che ne sono affette bisogna quindi calcolare tali coefficienti e mandarli in pasto ad una funzione creata ad hoc all'interno di **OpenCV**, detta **Undistort2** che prende in input proprio tali parametri nella forma

$$(k_1, k_2, p_1, p_2, [k_3])$$

in cui, se il vettore contiene 4 elementi, significa che $k_3 = 0$.
Tale funzione prende come parametri di ingresso l'immagine di origine (distorta), l'immagine di destinazione (della stessa dimensione dell'immagine di origine), la matrice di input della telecamera del tipo (3.4) ed, appunto, il vettore $(k_1, k_2, p_1, p_2, [k_3])$.

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

Una volta calcolato il vettore dei coefficienti di distorsione si passa al processo vero e proprio delle varie immagini, in cui vengono caricate da input le diverse immagini affette da distorsione, vengono riprocessate ed infine vengono salvate.

Capitolo 4

Modifica dei criteri di assegnazione del colore di un pixel

Dopo aver spiegato il meccanismo per risolvere i problemi di distorsione che affliggono le immagini registrate dalle diverse telecamere affronteremo in questa parte il problema vero e proprio dell'assegnazione corretta di un determinato colore ad un pixel al fine di ottenere una visualizzazione corretta delle immagini applicate al modello.

Il metodo utilizzato finora dal visualizzatore nella ricostruzione della scena reale prevede che le immagini vengano sovrapposte al modello con il solo vincolo di proiettare l'immagine di una telecamera concordemente con l'angolazione della telecamera stessa rispetto al modello, senza curare i confini in cui due immagini adiacenti vengono in contatto e lasciando quindi delle vistose incongruenze, o addirittura dei buchi, sulla superficie del modello, come mostrato in 6.5.

4.1 Pesatura dei contributi di una telecamera

L'idea che sta alla base del progetto in analisi è quella di prendere per ciascun pixel che si trova sulla *mesh* l'immagine, o la combinazione tra le quattro immagini, più "diritta" in modo da tenere solamente quelle rilevanti per ciascun pixel e tralasciare i contributi delle telecamere che non sono ben posizionate e che possono quindi introdurre errori al momento della sovrapposizione sul modello. Per rilevare quali siano le telecamere più rilevanti per ciascun pixel basterà mettere in relazione la normale di ciascun pixel sulla *mesh* con le normali delle 4 telecamere, opportunamente pesate, e scegliere

quindi il contributo di ciascuna immagine per poterlo poi assegnare al pixel corrispondente. Ovviamente viene privilegiata l'immagine che si trova ad avere la normale parallela alla normale del pixel, quella che meglio si sovrappone al modello in quel determinato punto. A tal fine bisogna calcolare le normali di ciascuna telecamera, che rimarranno poi immutate, e le normali dei diversi pixel sulla *mesh*. Per ciascun pixel bisogna effettuare un confronto con ciascuna delle 4 immagini del frame corrispondente e calcolare il *coseno* dell'angolo che si forma tra le normali, prese a coppie pixel-telecamera. Nell'immagine 4.1 viene mostrata la relazione tra la normale di un punto sulla superficie e le normali delle telecamere ai quattro angoli della stanza.

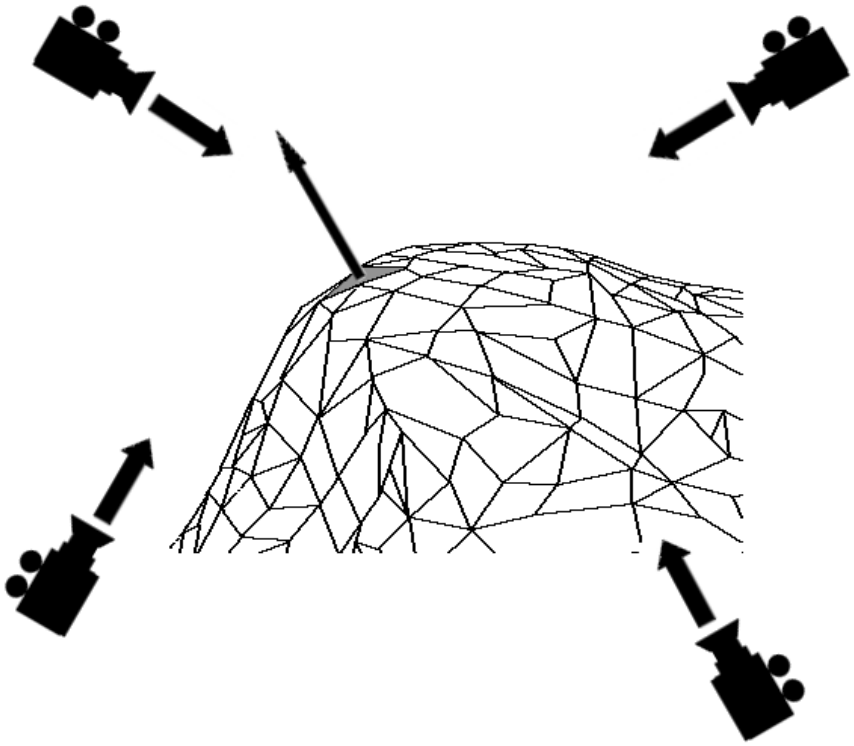

Fig. 4.1: Relazione tra la normale di un pixel sulla superficie e le normali delle 4 telecamere

Finché il pixel si trova perfettamente allineato con una determinata telecamera l'idea più logica consiste nel selezionare l'immagine relativa da sovrapporre al punto stesso, ma come si può effettuare una buona scelta per un punto che, ad esempio, si trova sulla superficie a metà strada tra due diversi punti di vista? Per ogni punto occorre prendere l'immagine che presenta la vista "migliore", che è quindi più in linea con il punto tra due o più punti di vista. A tal proposito, calcolando il coseno dell'angolo tra la normale del

punto sulla *mesh* e le normali delle viste, si riesce ad avere un valore che esprime il grado di bontà di ciascuna vista e, partendo da tali valori, è possibile applicare diversi algoritmi per la scelta del colore da assegnare al punto sulla superficie.
Una volta intuito il concetto che sta alla base della scelta corretta bisogna però fare attenzione ad alcuni piccoli particolari che possono essere fonte di errore nei calcoli e vanificare l'intera procedura, nello specifico parliamo delle dimensioni dei vettori che vengono messi in relazione e della rilevanza di un punto di vista nel calcolo dei coseni. Per quel che riguarda le dimensioni dei vettori occorre fare attenzione al fatto che siano normalizzati, che abbiano cioè norma pari ad 1, al fine di non introdurre, e poi propagare, errori nei calcoli e nella successiva scelta delle immagini. A questo proposito bisogna porre l'attenzione sulle normali delle viste e sulle norme dei pixel, controllare ed eventualmente correggere le norme degli stessi.
La rilevanza di una telecamera al calcolo dei pesi è altresì importante in quanto, essendo le telecamere disposte ai quattro angoli della stanza, non tutte contribuiscono a portare informazioni su ciascun pixel, anzi si rischia di introdurre nel calcolo immagini sbagliate che possono però poi contribuire alla formazione del colore del pixel finale. A tal proposito occorre prestare la massima attenzione ai criteri con cui si mettono in relazione le normali, stando attenti a scartare i contributi delle viste che si trovano alle "spalle" del punto sulla superficie e che quindi non porteranno certamente informazioni corrette. Nel calcolo del coseno dell'angolo tra i vettori normalizzati verrà quindi preferito il punto di vista che presenta il valore più vicino ad 1, che è il massimo che può assumere la funzione.
Nella scelta del colore che verrà assegnato al pixel non bisogna però scegliere la sola vista che presenta il coseno maggiore, altrimenti si rischia di creare un vistoso stacco nei punti che si trovano a metà tra due telecamere adiacenti. L'assegnazione del colore al pixel sulla mesh dovrà quindi essere pesata in base al contributo di ciascun coseno calcolato in precedenza. La funzione utilizzata per risolvere il problema dei pesi e delle assegnazioni è riportata in (4.1)

$$p = \frac{\cos\alpha_1 C_1 + \cos\alpha_2 C_2 + \cos\alpha_3 C_3 + \cos\alpha_4 C_4}{\cos\alpha_1 + \cos\alpha_2 + \cos\alpha_3 + \cos\alpha_4} \tag{4.1}$$

In cui α_1 è l'angolo tra la normale del pixel e la normale della telecamera i, mentre C_i è il colore dell'immagine relativa alla telecamera i di quel determinato frame. Nella formula precedente i coseni vanno calcolati facendo attenzione che le normali delle diverse telecamere vengano girati, ossia prendendo i vettori opposti, affinché si possano avere dati corretti e nel caso in cui l'angolo tra una coppia di normali pixel-telecamera superi i 90° è bene

toglierlo dal calcolo affinché non introduca valori negativi nella scelta del colore del pixel; sarà quindi opportuno porre a zero il valore di un determinato α_i quando questo diventa negativo, in modo da non influenzare il calcolo sia al numeratore che al denominatore.

4.2 Pesatura coerente con il punto di vista dell'utente

Dopo aver calcolato i pesi seguendo la modalità descritta in precedenza si è passati ad apportare alcune migliorie all'assegnazione delle immagini, governate principalmente dalla formula dei pesi enunciata in (4.1). Per prima cosa si è pensato di aggiungere un elemento che permette di analizzare il punto di vista scelto dall'utente, che non coincide quasi mai con le coordinate delle diverse telecamere, e da questo effettuare una serie di scelte per garantire un miglior risultato visivo per ciascun punto di vista. È infatti preferibile che il modello presenti una ricostruzione il più perfetta possibile proprio dal punto di vista scelto dall'utente ed è così stato necessario introdurre ulteriori fattori per aumentare il rendimento della ricostruzione.
Si è quindi pensato di far intervenire nel calcolo dei pesi anche un altro fattore che prenda in considerazione il punto di vista scelto dall'utente e ne calcoli l'incidenza nella scelta delle immagini. Un buon parametro per giudicare la bontà di una vista consiste nel calcolo del coseno dell'angolo che vi è tra il vettore normalizzato del punto di vista ed i 4 vettori che rappresentano le direzioni delle telecamere. Tale valore ci permetterà di privilegiare nella scelta delle immagini quelle che si riferiscono alla telecamere più "vicina" al punto di vista, in modo da escludere dal calcolo le immagini che si trovano "dietro" al punto di vista e che non apportano informazioni corrette alla ricostruzione.

La formula utilizzata è simile alla (4.1) ed è riportata in (4.2)

$$s = \frac{\cos\beta_1 C_1 + \cos\beta_2 C_2 + \cos\beta_3 C_3 + \cos\beta_4 C_4}{\cos\beta_1 + \cos\beta_2 + \cos\beta_3 + \cos\beta_4} \tag{4.2}$$

In cui β_i è l'angolo tra il punto di vista e la telecamera i, mentre C_i è sempre l'immagine relativa alla telecamera i di quel determinato frame. Anche in questo caso bisogna prestare attenzione al fatto che tutti i vettori siano orientati nella giusta maniera e ricordarsi di scartare dal calcolo i coseni minori di 0, in quanto non apportano informazioni ma errori, sia al numeratore che al denominatore. Le due formule dei pesi (4.1) e (4.2) vengono poi fuse assieme e regolate da due variabili come mostrato in (4.3)

$$w = K_1 p + K_2 s \tag{4.3}$$

In cui K_1 e K_2 sono le variabili di controllo, che presentano valori all'interno del *range* $[0, 1]$ ed in cui vale la relazione (4.4).

$$K_1 = 1 - K_2 \tag{4.4}$$

4.3 Pesatura e visibilità del pixel

Dopo aver perfezionato la ricostruzione, anche aiutati dalla scelta da parte dell'utente del punto di vista preferito, siamo andati a studiare gli *output* della ricostruzione ed è evidente che in certe immagini vi è una specie di strisciata dell'immagine di una telecamera sul modello dovuta al metodo di ricostruzione ed analisi delle immagini colori-triangoli (immagini in cui sono evidenziati i triangoli della *mesh* ed in cui ogni triangolo è colorato con un colore univoco che lo identifica), che sono quelle che contengono le informazioni sul rapporto tra pixel e triangoli della *mesh*. In determinati punti, che si trovano in zone d'ombra, viene infatti propagata l'immagine che si trova "sopra" nel modello, cioè ad esempio nella zona del torace sotto al braccio viene propagata l'immagine del braccio stesso in quanto è quello che viene visto dalla telecamera.

Per ovviare a questo inconveniente si è pensato di acquisire in input anche le 4 immagini colori-triangoli di ciascuna telecamera ed analizzarle al fine di determinare se un determinato pixel è visibile da una o più telecamere. Si sono quindi introdotte delle variabili booleane di tipo $True = 1$ e $False = 0$ in cui se un determinato triangolo della *mesh* è visibile da una telecamera allora il controllo viene messo a $True$, altrimenti resta a $False$, come di default. In questa maniera è possibile per ciascun pixel calcolare il coseno tra la sua normale e la normale di una telecamera solo se risulta essere visibile, altrimenti viene messo a 0 e non parteciperà alla pesatura all'interno di (4.3).

Capitolo 5

Implementazione del modello

In questo capitolo viene spiegata nel dettaglio l'implementazione della teoria sulla corretta assegnazione dei colori ai vari pixel presenti sulla *mesh* del modello che verrà creato al momento della scelta del *frame* di interesse da parte dell'utente.

Al momento di iniziare la risoluzione di tale problema abbiamo a disposizione i seguenti elementi:

- Un programma per il calcolo di un modello 3D a partire da determinati elementi di input;
- Un programma per la visualizzazione di oggetti in 3D e ricco di funzioni per personalizzare la visualizzazione e per effettuare il salvataggio di importanti parametri;
- 2000 immagini a colori in formato BMP che rappresentano i *frame* di un video di una persona che effettua determinati movimenti all'interno di una stanza appositamente preparata (500 immagini per ognuna delle 4 telecamere posizionate ai quattro angoli dei tale stanza);
- 2000 immagini in bianco e nero ottenute *renderizzando* le immagini originali;
- 2000 immagini in bianco e nero ottenute "un-distorcendo" le immagini *renderizzate*;
- 4 files TXT che contengono le *texture* delle diverse telecamere;
- Un file TXT che contiene le matrici delle 4 telecamere ed un altro che contiene i coefficienti di distorsione relativi, definiti per 3 diverse impostazioni testate all'interno della stanza di acquisizione;

- Diverse librerie di appoggio per visualizzare le immagini in 3D ed effettuare diverse operazioni sui dati.

I passi da seguire per poter risolvere tale problema sono nell'ordine:

- Analizzare il programma **GetFrame** che crea la *mesh* ed il modello per ciascun *frame*;
- Studiare il programma **GLview** per la visualizzazione del modello 3D e capirne le funzionalità ed i comandi di base;
- Controllare se e come le diverse immagini sono applicate sulla *mesh*;
- Correggere eventuali errori sulle immagini stesse o sul modello;
- Creare un'applicazione che assegni a ciascun pixel sulla *mesh* l'immagine della camera più corretta a seconda del punto sulla *mesh* stessa.

Verrà ora approfondito l'ultimo punto relativo all'applicazione creata partendo dai dati in nostro possesso e dalle intuizioni ottenute nei capitoli precedenti. Per prima cosa è necessario definire delle costanti che compariranno spesso all'interno del codice che sono le dimensioni, in pixel, delle immagini che verranno processate, imposte per congruenza con le immagini di partenza a 1032 pixel di larghezza e 778 pixel di altezza.

5.1 Calcolo della normale di un pixel sulla *mesh*

Uno degli elementi fondamentali per tutti i calcoli e le assegnazioni dei colori ai diversi pixel è la normale di ciascun pixel presente sulla superficie del modello; per calcolare tale valore non vi è a disposizione una funzione già pronta, ma bisogna attuare una serie di passaggi che ci permetta di arrivare al risultato voluto utilizzando le funzioni e le librerie già presenti all'interno di **GLview** ed, eventualmente, costruirne altre *ex novo* per raggiungere il nostro scopo. Per prima cosa occorre definire una struttura adatta a contenere i valori relativi alle normali di ciascun pixel e la prima cosa che istintivamente viene da creare è una matrice di interi di dimensione $1032 * 778$ che verrà utilizzata ripetutamente nel codice; si è però notato che utilizzando questa struttura dati il compilatore, al momento di eseguire il codice, e quindi dopo che il programma è stato compilato correttamente, si bloccava dando come errore un *memory overflow* a causa della larga quantità di spazio allocato.

Per risolvere tale inconveniente si è quindi deciso di utilizzare come struttura dati un *array bidimensionale* allocato dinamicamente, che consente di immagazzinare la stessa quantità di dati e permette di accedere agli stessi utilizzando i medesimi comandi di una matrice, ma che permette di allocare lo spazio in memoria in maniera dinamica, aggirando quindi il problema del *memory overflow*. La definizione del codice è riportata in 5.1

```
1 point3D** pixel_normale;
2 pixel_normale = new point3D*[LARGHEZZA];
3 for (int r=0; r< LARGHEZZA; r++){
4     pixel_normale[r] = new point3D[ALTEZZA];
5 }
```

Listing 5.1: Definizione di un array bidimensionale allocato dinamicamente

in cui si nota che l'*array bidimensionale* è definito come un *array di array* (righe 1 e 2), di elementi *point3D*, una struttura dati utilizzata all'interno di **GLview** che rappresenta un punto nello spazio 3D ed è costituito di 3 elementi, uno per ciascuna dimensione, di tipo *float*.

Calcolare la normale di un pixel non è così immediato come sembra, in quanto non è prevista una funzione diretta che partendo dal modello restituisca l'insieme delle normali di ciascun pixel. Con gli elementi a nostra disposizione è però possibile costruire tale collezione di dati partendo da un determinato punto di vista scelto dall'utente. Per fare ciò bisogna creare un modello con **GetFrame**, scegliendo il *frame* di interesse all'interno dell'intervallo di immagini a nostra disposizione e, una volta ottenuto il file prova.wrl, utilizzare **GLview** per poterlo visualizzare; una volta che l'immagine in 3 dimensioni appare a video è sufficiente posizionarsi nel punto di vista desiderato, come spiegato nel relativo capitolo, e dopo aver premuto il tasto destro del mouse scegliere dal menu a tendina l'opzione "Colori-triangoli", che visualizzerà la struttura della *mesh* e colorerà ciascun triangolo con un colore univoco che rappresenta l'identificativo di ciascun triangolo. Una volta certi del corretto posizionamento del modello è sufficiente premere nuovamente il tasto destro del mouse e scegliere dal menu l'opzione "salva z-buffer" che ci permetterà di salvare l'immagine che è presente a video ed i parametri relativi al punto di vista scelto.

A questo punto si può chiudere la finestra di **GLview** e copiare i valori della matrice di proiezione completa che compaiono nel *prompt dei comandi* in un file testo al fine di salvare i valori del punto di vista scelto. Tra i vari files che vengono salvati con il comando "salva zbuffer" vi è *gl_out.bmp*, che non è altro che l'immagine con colori e triangoli che compariva a video in **GLview**. Utilizzando un programma di manipolazione di immagini basterà quindi aprire tale file Bitmap, riflettere verticalmente l'immagine in questione

(poiché viene salvata capovolta) e salvarla con un nome che sia esplicativo di ciò che rappresenta, come ad esempio "colori-triangoli.bmp", poiché ogni qualvolta verrà effettuato un nuovo salvataggio dello z-buffer l'immagine con il nome *gl_out.bmp* verrà sovrascritta. L'immagine che si ha ora rappresenta il modello dal punto di vista scelto in cui sono evidenziati i triangoli della *mesh* ed i colori che identificano i triangoli in maniera univoca. Poiché all'interno di **GLview** vi è una funzione che restituisce la normale di un determinato triangolo passato come input dovremo ora cercare di assegnare correttamente un determinato pixel al triangolo in cui esso è contenuto nella *mesh*. Per poter calcolare la normale di ciascun pixel dovremo quindi prima correlare tutti i pixel con i triangoli che li contengono, calcolare le normali di tali triangoli ed infine assegnare a ciascun pixel la normale relativa. Come è stato evidenziato in precedenza ad ogni triangolo sulla mesh è assegnato un colore univoco che lo identifica; leggendo quindi il colore di ciascun pixel dell'immagine "colori-triangoli" appena salvata possiamo risalire al triangolo a cui esso appartiene.

Per analizzare nel dettaglio l'immagine salvata si è scelto di utilizzare **EasyBMP**[1], una semplice libreria, multi-piattaforma, *open source*, che permette di leggere, scrivere e modificare facilmente files di immagini di Windows in formato BMP [6]. Utilizzando due cicli *for* innestati è quindi possibile effettuare la scansione completa dell'immagine desiderata pixel per pixel e salvare le informazioni relative in una nuova struttura dati da definire. Vengono quindi creati altri due *array bidimensionali* allocati dinamicamente di dimensione $1032 * 778$ denominati "pixel_triangolo" e "triangolo_normale" che verranno poi fusi al fine di creare l'*array bidimensionale* "pixel_normale". Per prima cosa viene assegnato il percorso ed il file che si desidera scansionare, nel nostro caso l'immagine *colori-triangoli.bmp*, ad un file *const char* che verrà utilizzato per creare un nuovo oggetto BMP utilizzando la libreria **EasyBMP**. In 5.2 viene mostrata la parte del codice relativa al caricamento e all'analisi dell'immagine precedentemente salvata.

[1] http://easybmp.sourceforge.net/

```
1  const char * path_img = "triangoli.bmp";
2  BMP immagine;
3  immagine.ReadFromFile(path_img);
4
5  for( int i=0 ; i < immagine.TellWidth() ; i++ ){
6      for( int j=0 ; j < immagine.TellHeight() ; j++ ){
7          RGBApixel Temp = immagine.GetPixel(i,j);
8          indice_triangolo = (Temp.Red+((Temp.Green) << 8)+((Temp.
               Blue) << 16))/3;
9          if (indice_triangolo >0)pixel_triangolo [i][j] =
               indice_triangolo;
10         else pixel_triangolo [i][j] = -100;
11     }
12 }
```

Listing 5.2: Caricamento ed analisi di una immagine BMP con EasyBMP

Si nota che l'immagine viene caricata in un file BMP "immagine" (righe 1-3) che viene poi scansionato con due cicli *for* prima in larghezza e poi in altezza a causa delle proprietà del formato Bitmap. Ad ogni passo viene letto il valore del pixel relativo e assegnato ad un oggetto di tipo **RGBApixel** di nome *Temp* (riga 7); tale struttura è definita all'interno di **EasyBMP** ed è costituita di 4 elementi di tipo "unsigned char ebmpBYTE" che contengono i valori cromatici *Red*, *Green*, *Blue* **(RGB)** e la trasparenza *Alpha*.
L'elemento indice_triangolo (aggiornato nella riga 8) è un intero temporaneo che riceve il valore della funzione (5.1) che effettua la conversione dal colore del pixel all'indice del triangolo corrispondente sulla *mesh*.

$$indicetriangolo = \frac{(R + (G << 8) + (B << 16))}{3} \tag{5.1}$$

Su tale valore appena calcolato viene poi effettuato un controllo per verificare che il triangolo ottenuto dalla funzione sia realmente esistente, sia cioè presente nell'intervallo $[0, totale - triangoli]$; siccome non si sa a priori quale sia il valore massimo dei triangoli su una *mesh* è importante verificare che il risultato ottenuto sia almeno positivo, in quanto un valore negativo o uguale a zero rappresenta un punto all'esterno della *mesh*. Viene quindi effettuato un controllo di tipo *if* e, se viene superato, si assegna il valore appena calcolato all'ingresso corrispondente al pixel processato nell'*array bidimensionale pixel_triangolo* (come mostrato nella riga 9), altrimenti si assegna a tale posizione un valore di controllo pari a -100 (riga 10) che servirà più avanti per identificare il triangolo non trovato e permetterà di risparmiare diverse operazioni di calcolo.

Il passo successivo consiste nel caricare la *mesh* del modello, nel nostro caso il file *prova.wrl* che viene salvato da **GetFrame**, ed estrarre il

numero di triangoli che compongono la *mesh* salvando in un intero il valore ritornato dalla funzione *mesh.GetNumberOfFaces()*. Bisogna poi costruire un vettore di point3D grande quanto il numero dei triangoli della *mesh* in cui verranno salvate le normali di ciascun triangolo usando la funzione *triangolo.normale()*;

Infine bisogna creare un nuovo *array bidimensionale* allocato dinamicamente, "pixel_normale", in cui viene salvata la fusione tra l'*array bidimensionale* "pixel_triangolo" ed il vettore "triangolo-normale" stando attenti al fatto che se il valore di un elemento di "pixel_triangolo" è pari a -100 il valore assegnato alla normale di quel determinato pixel sarà $(0, 0, 0)$.

5.2 Estrazione dei valori delle telecamere

A questo punto abbiamo le normali di ciascun punto sul modello e dobbiamo procurarci i valori dei punti di vista delle diverse telecamere. Per ciascuna delle 5 visuali, 4 telecamere ed il punto di vista scelto dall'utente, dobbiamo perciò caricare i file TXT corrispondenti, estrarne la direzione e normalizzare tale valore. La parte di codice relativa a tale operazione è mostrata in 5.3.

```
1 projectionpinhole p;
2 float fx,fy,cx,cy;
3 p = projectionpinhole("matrice-telecamera-xxxx.txt");
4 p.getCameraParameters(fx,fy,cx,cy);
5 point3D direzione_camera_xxxx = p.Direction(cx,cy);
6 direzione_camera_xxxx = normalize(direzione_camera_xxxx);
7 direzione_camera_xxxx = reverseDirection(direzione_camera_xxxx);
```

Listing 5.3: Normalizzazione direzione telecamera

Dal codice importato si vede che innanzitutto occorre definire una variabile di tipo *pinhole* (riga 1) e 4 valori di tipo *float* (riga 2) che rappresentano i valori del centro di osservazione e la direzione di osservazione come definito in (3.2) ed in (3.3). Bisogna quindi estrarre i parametri dal file caricato ed assegnarli ad un oggetto *point3D* usando la funzione di **GLview** *Direction(cx,cy)* in cui viene calcolata la direzione a partire dal centro dell'immagine passato come parametro (riga 5). Il passo successivo consiste nel normalizzare tale valore (riga 6) e crearne l'opposto al fine di renderlo confrontabile con la normale di un pixel sulla *mesh* (riga 7); le ultime due funzioni sono state create appositamente per il calcolo di normali ed opposti di elementi *point3D*.

5.3 Calcolo del coseno tra 2 vettori

In questa sezione viene mostrato come si è implementato il calcolo dei coseni tra i diversi vettori delle telecamere, del punto di vista e delle normali dei pixel sul modello. La formula che è stata utilizzata è la (5.2) in cui si riesce ad estrarre il valore del coseno a partire dai valori dei vettori di direzione presi a coppie.

$$\begin{aligned} \overrightarrow{a} \cdot \overrightarrow{b} &= \|a\| \, \|b\| \cos\theta \\ \cos\theta &= \frac{\|a\|\|b\|}{\overrightarrow{a} \cdot \overrightarrow{b}} = \frac{a_x b_x + a_y b_y + a_z b_z}{\sqrt{a_x^2 + a_y^2 + a_z^2}\sqrt{b_x^2 + b_y^2 + b_z^2}} \end{aligned} \tag{5.2}$$

5.3.1 Angolo tra punto di vista e direzione telecamera

Il calcolo del coseno dell'angolo tra la direzione di una telecamera ed il punto di vista scelto dall'utente è rappresentata dalla semplice applicazione della formula (5.2) facendo attenzione ad applicarla per ciascun valore delle 3 dimensioni; il valore ottenuto come risultato viene quindi salvato con il nome *betaX* in cui X rappresenta il numero della telecamera confrontata con il vettore del punto di vista e che andrà a far parte della funzione (4.2) nel posto opportuno. Alla fine di ogni calcolo del coseno è stato introdotto il controllo di positività del coseno, in cui se il valore è negativo, e quindi l'angolo non è nel range $[-\pi/2, \pi/2]$, lo stesso valore *betaX* viene forzato a 0 in modo da evitare che l'immagine proveniente dalla telecamera X contribuisca all'assegnazione del colore finale del pixel.

5.3.2 Angolo tra direzione telecamera e normale del pixel

Anche in questo caso è opportuno applicare la formula (5.2) con l'unica differenza che, se per il punto di vista essa è calcolata solamente una volta per ciascuna telecamera, in questo caso dovremo applicare la formula per ciascun pixel presente sull'immagine ed è quindi preferibile accorpare in gruppi di 4 tali calcoli in modo che per ciascun pixel sia possibile estrarre in minor tempo i valori dei coseni per ciascuna telecamera. Come per il punto precedente, anche in questo caso bisogna fare attenzione al fatto che il coseno sia positivo al fine di non introdurre errori al momento di assegnare il colore finale al pixel. Il valore finale in questo caso è salvato con il nome di *alphaX*, in cui X è sempre il numero della telecamera con cui viene effettuato il confronto, e concorre a formare il risultato dell'equazione (4.1). Nel calcolo di

tale coseno occorre però fare attenzione che tra i diversi valori di *alpha* per ciascuna delle 4 telecamere ve ne sia almeno uno diverso da zero, altrimenti il valore del pixel in questo caso verrà settato come colore nero, creando vistose macchie nere nei punti in cui non vi è nessuna telecamera che riprende un determinato pixel, con una angolazione al di fuori di $[-\pi/2, \pi/2]$. Per risolvere tale inconveniente viene effettuato un controllo prima di assegnare un colore ad un pixel in cui i valori dei diversi coseni, se negativi, vengono salvati in una variabile temporanea e, nel caso in cui tutti e 4 tali valori siano negativi, piuttosto che assegnare un colore nero si assegna al pixel corrispondente l'immagine della telecamera "meno peggio", cioè quella che presenta il valore di coseno più grande tra i 4 valori. In questo caso si è notato che l'approssimazione funziona molto bene e che i valori assegnati a tali zone d'ombra provengono dalle stesse telecamere che contribuiscono a formare le immagini nell'intorno di tale buco, evitando di inserire immagini errate e creare quindi disambiguità nella figura.

5.4 Visibilità di un pixel

Come spiegato in 4.3 la visibilità di un pixel dipende dal punto scelto sulla *mesh* e sul punto di vista che viene scelto, per cui lo stesso punto non visibile per 3 telecamere può essere visibile per un'altra. Nel nostro caso è necessario acquisire le 4 immagini del tipo "colori-triangoli" per ciascuna delle diverse visuali delle telecamere ed analizzarle nel dettaglio al fine di trovare se un pixel è visibile oppure no. Vengono quindi creati 4 vettori di booleani di dimensione pari al numero di triangoli presenti sulla *mesh* ed inizializzati tutti a *False*. A questo punto vengono analizzati tutti i pixel delle 4 immagini "colori-triangoli" ricavando per ciascun pixel il triangolo a cui esso appartiene, ne viene analizzato il colore ed estratto il numero associato e, nel caso in cui sia positivo, allora nella posizione relativa all'interno del vettore di booleani verrà cambiato il valore da *False* a *True*. Una volta terminata l'analisi delle 4 immagini avremo così 4 vettori che indicano, per ciascuna telecamera, quali siano i punti visibili e quali no. Ora possiamo utilizzare tali informazioni per perfezionare il calcolo dei colori da assegnare ai pixel, in quanto basterà moltiplicare il valore relativo del vettore di visibilità (che vale 1 nel caso sia *True* e vale 0 nel caso sia *False*) al numeratore nell'equazione degli *alpha* . In questo modo entreranno nel calcolo dei vari pesi solamente i contributi dei pixel effettivamente visibili da una telecamera, ottenendo un risultato più pulito visivamente e più corretto formalmente.

Capitolo 6

Risultati ottenuti

In questa sezione vengono presentati i risultati ottenuti dall'applicazione scritta come progetto di tesi e verranno visualizzati di seguito alcuni esempi di immagini ottenute e ne verranno descritte le caratteristiche.

Le immagini 6.1, 6.2, 6.3 sono 3 delle 4 immagini di un frame salvate da 3 telecamere diverse di uno stesso soggetto, preso ovviamente da angolature diverse, con uno sfondo creato *ad hoc* per risaltare il contrasto con il rosa della pelle del viso; tali immagini rappresentano i dati di partenza sui quali si è applicato il programma scritto per la corretta sovrapposizione.

L'immagine 6.5 rappresenta il risultato della sovrapposizione delle immagini sul modello effettuate dalla semplice esecuzione di *GLview*, in cui si nota che vi sono delle vistose disambiguità sulle zone di intersezione di due immagini vicine, soprattutto sul volto e sulla parte superiore del busto, ed alcune zone in cui vi è una mancanza di informazione tipo sopra la spalla sinistra e lungo il braccio sinistro del modello.

L'immagine 6.6 è il risultato dell'applicazione della formula dei pesi (4.1) ed è evidente che alcune imperfezioni dell'immagine precedente sono state corrette, soprattutto nella zona delle intersezioni delle immagini adiacenti sul volto, sul busto e sul braccio sinistro del modello e le mancate assegnazioni di colore sulla spalla sono state sfumate e si è riusciti a creare una continuità tra le immagini anteriori e posteriori al modello.

L'immagine 6.7 rappresenta invece il risultato della correzione dovuta all'introduzione del parametro "punto di vista", oltre alla formula di pesatura, in cui si è scelto di assegnare un uguale contributo tra l'informazione apportata dalle normali sulla *mesh* e l'informazione derivante dall'introduzione del punto di vista. Tale scelta deriva dal fatto che in questa maniera non vi è un fattore dominante e partecipano entrambi in egual misura alla formazione del colore del pixel; lo spostamento di tale indice a favore delle sole normali dei pixel riporterebbe il modello all'immagine precedente, mentre uno sposta-

mento a favore del solo punto di vista creerebbe un'immagine in cui i colori dei pixel vengono ottenuti con una media tra i colori delle due telecamere che si trovano rispettivamente a destra e a sinistra del punto di vista, appiattendo l'informazione dell'immagine. Si nota in questo caso che l'immagine risulta sempre ben definita rispetto al risultato di GLview e migliora la sfumatura sul braccio sinistro del modello, eliminando qualche residuo di colore blu, dovuto allo sfondo della stanza, ancora presente nel caso precedente.

Infine nell'immagine 6.8 viene rappresentato il risultato dell'introduzione della visibilità dei pixel nascosti, oltre alle altre appena applicate, in cui si nota che i contorni dell'immagine sono più definiti, specialmente la distanza che separa il braccio destro del modello dal busto, anche se vengono introdotte delle macchie di pixel di colore nero nelle zone d'ombra non viste dalle telecamere, specialmente nella zona interna del braccio sinistro.

Fig. 6.1: Immagine salvata dalla telecamera 2

Fig. 6.2: Immagine salvata dalla telecamera 3

Fig. 6.3: Immagine salvata dalla telecamera 4

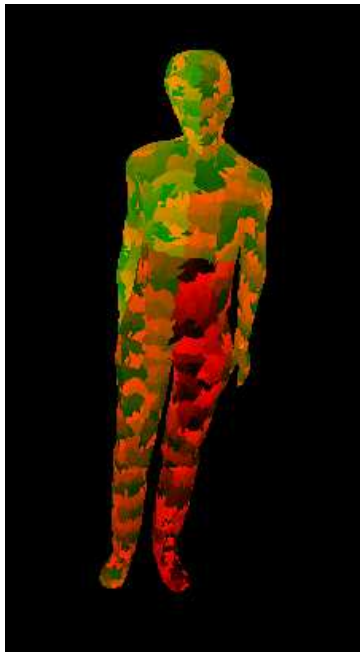

Fig. 6.4: Immagine colori-triangoli

Fig. 6.5: Output di GLview

Fig. 6.6: Immagine pesata

Fig. 6.7: Immagine pesata con punto di vista

Fig. 6.8: Immagine pesata con punto di vista e visibilità

Capitolo 7

Conclusioni

In questa tesi si è cercato di trovare un metodo, o una serie di metodi, per visualizzare correttamente 4 immagini in 2 dimensioni sulla ricostruzione in 3D del modello originale, facendo in modo di ottenere una sovrapposizione il più precisa possibile, evitando buchi sul modello ed errate sovrapposizioni.

Con la sola pesatura dei contributi di ciascuna immagine su ogni pixel si è creato un buon punto di partenza, in quanto sono state eliminate le imperfezioni più evidenti di *GLview* e da queste nuove immagini si è cominciato a cercare di perfezionare il risultato finale.
Introducendo poi il concetto di punto di vista dell'utente si è focalizzata l'attenzione sulla porzione di immagini più interessanti per l'utente finale, cercando di bilanciare i contributi di pesatura e punto di vista in modo da arricchire la qualità dell'immagine ottenuta, stando attenti a non privilegiare troppo l'una o l'altra per non ottenere invece l'effetto contrario.
Aggiungendo infine il concetto di visibilità sono state corrette alcune imperfezioni presenti nelle immagini ottenute, ottenendo l'eliminazione degli errori introdotti da immagini provenienti da telecamere "sbagliate" per determinati pixel, introducendo però delle discontinuità che in altri punti, sulla superficie del modello, risultano più marcate.

Una cosa fondamentale da evidenziare riguarda però le immagini di partenza sulle quali sono state applicate tutte le teorie descritte in precedenza, in quanto erano affette da diverse tipologie di distorsione dovute alle lenti delle telecamere che le hanno acquisite; si è cercato di modificarle, correggendone le imperfezioni, e si è raggiunto un buon risultato nel confronto con le immagini obiettivo, anche se non ottimale e proprio questo fattore ha contribuito a creare alcune discontinuità specialmente nelle zone di confine di ciascuna immagine.
L'aspetto interessante sarebbe applicare il programma creato per questo lavoro a delle nuove immagini ottenute con una perfetta calibrazione ed analizzar-

ne i risultati, ma purtroppo la stanza che era stata allestita per le acquisizioni è stata smantellata e si attende a breve il suo ripristino.

L'intero lavoro finora presentato non deve però essere considerato un punto di arrivo per la ricostruzione di modelli in 3D, ma un buon punto di partenza, ricco di variabili che possono essere combinate tra loro in moltissimi modi per ottenere il risultato più adatto allo scopo che si vuole raggiungere; il codice scritto per questa applicazione è anch'esso migliorabile, soprattutto per ciò che riguarda l'efficienza in termini di memoria utilizzata e velocità di esecuzione.

A questo proposito applicando un programma come quello presentato, opportunamente modificato, su di un calcolatore dedicato è possibile pensare di svolgere le applicazioni descritte in questa tesi in real time, sempre però dopo aver acquisito la struttura desiderata con uno scanner passivo, ed applicare tali soluzioni in diversi campi, che vanno dall'industria dei videogames alla sicurezza.

Bibliografia

[1] Luca Ballan. Acquiring shape and motion of interacting people from-videos. Master's thesis, Università degli studi di Padova, Gennaio 2009.

[2] Luca Ballan and Guido Maria Cortelazzo. Marker-less motion capture of skinned models in a four camera set-up using optical flow and silhouettes. In *3DPVT*, Atlanta, GA, USA, June 2008.

[3] Andrea Fusiello. *Visione Computazionale*. 1st edition, 2008.

[4] OpenCV User Group. Opencv wiki. http://opencv.willowgarage.com/wiki/, 2010.

[5] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[6] Paul Macklin. The easybmp project. http://easybmp.sourceforge.net/, 2006.

Appendice A

Elenco tasti funzione di GLview

Vengono di seguito elencati l'elenco dei tasti e le azioni corrispondenti in **GLview**come introdotto nel capitolo 2.2.

- tasto 'q' ruota l'immagine verso sinistra sull'asse z;
- tasto 'e' ruota l'immagine verso destra sull'asse z;
- tasto 'w' ruota l'immagine verso sinistra sull'asse y;
- tasto 's' ruota l'immagine verso destra sull'asse y;
- tasto 'a' ruota l'immagine verso destra sull'asse x;
- tasto 'd' ruota l'immagine verso sinistra sull'asse x;
- tasto 'z' zooma in avanti;
- tasto 'c' zooma indietro;
- tasto '2' trasla l'immagine di (0,0,1);
- tasto '8' trasla l'immagine di (0,0,-1);
- tasto '3' trasla l'immagine di (1,0,0);
- tasto '9' trasla l'immagine di (-1,0,0);
- tasto '4' trasla l'immagine di (0,1,0);
- tasto '6' trasla l'immagine di (0,-1,0).