Università degli Studi di Padova

Facoltà di Ingegneria

Ingegneria Informatica

Tesi di Laurea Magistrale

Realizzazione di un sistema di videoconferenza 3D real-time basato sul sensore MS Kinect

Relatore: Prof. Pietro Zanuttigh
Correlatore: Ing. Carlo Dal Mutto

Laureando: Mauro Donadeo

17 ottobre 2011

Questo documento è stato scritto in LaTeX su Debian GNU/Linux.

A mio nonno

"Volli, e volli sempre, e fortissimamente volli."
V. Alfieri

"Nuestra recompensa se encuentra en el esfuerzo y no en el resultado.
Un esfuerzo total es una victoria completa."
M.Gandhi

Indice

Elenco delle figure

Sommario

La computer vision è quell'area della ricerca che si occupa di elaborare immagini in estraendo ed elaborando da esse le informazioni sulla scena e oggetti che la rappresentano. In questi ultimi anni, inoltre, è aumentato in maniera esponenziale l'interesse per il 3D grazie all'uscita di all'arrivo sul mercato di dispositivi che permettono l'acquisizione e visualizzazione di dati 3D a costi decisamente abbordabili.

Questo lavoro si introduce all'interno della branca della computer graphic 3D*. Infatti è stato ideato un nuovo sistema di videoconferenza che acquisisce e visualizza i dati in 3D. Si tenterà di andare oltre alla solita visione dell'immagine piatta, anche se ad alta risoluzione, che una comune webcam può generare. Si mostrerà che oltre ai colori della scena è possibile stabilire la posizione di ogni punto all'interno di uno spazio 3D, ricostruendo in questo modo la posizione di ognuno nella scena. Inoltre l'immagine verrà riprodotta da uno schermo stereoscopico acuendo in questo modo la sensazione del 3D. Si è anche lasciata la libertà all'utente di muoversi all'interno della scena, permettendo proprio di poterci navigare, tramite zoom, movimenti e rotazioni della camera.*

Per ricreare la scena 3D ci si è avvalsi dell'aiuto del Microsoft Kinect, creato per la console di gioco Xbox, che grazie agli sforzi della comunità internet e ai driver rilasciati da Prime Sense è ora possibile utilizzarlo come una normalissima periferica del computer. Inoltre per aumentare la sensazione del 3D si è ricorsi al sistema di visione stereoscopica NVIDIA, che tramite un paio di occhiali sincronizzati con il monitor permette di avere l'illusione del 3D.

Inoltre in questo lavoro si mostrerà come ridurre la grandezza dei frame *utilizzando gli algoritmi di compressione* JPEG *e* JPEG2000 *che permettono di mantenere la comunicazione in real-time.*

Capitolo 1

Introduzione

La visualizzazione 3D negli ultimi anni ha fatto enormi passi avanti, basti pensare al gran numero di film prodotti in 3D. Naturalmente questo ha portato a produrre dispositivi che permettessero di poter ricreare le varie scene 3D anche a casa. Come ad esempio i primi monitor e telecamere stereoscopiche, che sono presenti all'interno della maggior parte dei rivenditori di informatica.

In questa tesi si è voluta sfruttare questa tecnologia a "basso costo": in particolare il *kinect* per l'acquisizione di dati 3D e il sistema di visione stereoscopico NVIDIA. Il kinect è dotato di due sensori che catturano la geometria della scena e uno che acquisisce i colori. Mentre il sistema di visualizzazione NVIDIA permette tramite un paio di occhiali sicronizzati con il monitor di creare l'illusione del 3D, aprendo e chiudendo le lenti.

L'idea si è voluta sviluppare in questa tesi è quella di progettare e implementare un nuovo sistema di *videochat* 3D. Ossia catturare l'immagine di profondità dal kinect associargli i colori e grazie al sistema di visione stereoscopico fornire un render della scena in 3D. Inoltre fornire all'utente un sistema di navigazione all'interno della scena che permetta all'utente di poter apprezzare la profondità dei singoli oggetti.

Ci sono vari aspetti che bisogna gestire in questo nuovo sistema di comunicazione, ad esempio la quantità di dati che il kinect genera che naturalmente senza previa elaborazione non possono essere spediti sulla rete. Va gestita anche la comunicazione tra due nodi di una rete che volessero incominciare a comunicare utilizzando il kinect. Bisogna anche preparare i dati da passare alla scheda video, cioè preparare la *mesh* di triangoli tenendo presente che punti contigui all'interno dell'immagine a colori potrebbero avere profondità differenti.

Il progetto è stato pensato in maniera modulare in modo da non limitare il riutilizzo del codice per eventuali sviluppi futuri. Infatti si è pensato di creare due moduli uno che è stato chiamato *server* e un altro *client*.

Il lato server di questa applicazione si occupa della gestione del kinect e di inviare i dati una volta effettuata la connessione con il client. Prima di inviare i dati questi vengono compressi utilizzando uno dei due algoritmi di compressione: *JPEG2000* o *JPEG*. Anche se non sembra la quantità di dati da gestire è tanta: il *kinect* è stato impostato per generare due immagini con risoluzione 640x480, con 2 byte per pixel per la *depth map* e 3 byte per pixel per quella a colori; generando 30 frame al secondo sono all'incirca 46MB al secondo di dati da gestire. Le librerie degli algoritmi di compressione sono state leggermente modificate affinché potessero grantire la scrittura dei dati compressi in memoria invece che su file; evitando di introdurre uno slowdown nell'applicazione che comporterebbe la lettura da essi.

Oltre ai driver di *Prime Sense* [5], per la gestione del kinect, è stato utilizzato il *framework* OpenNI [4]. Questo mette a disposizione un insieme di API che permettono di gestire i dati che il kinect fornisce. Ma inoltre permette di creare dei *middleware* che sono indipendenti dal tipo di hardware che si sta utilizzando.

Il *client* invece si occupa di tutto quello che riguarda la visualizzazione. Una volta acquisite le immagini le decomprime e su di esse crea una mesh di centinaia di migliaia di triangoli che permettono di discretizzare la scena e quindi poterla vedere anche in full screen. Il lato *client* si occupa anche di gestire i *driver NVIDIA* che permettono di visualizzare la scena in 3D gestendo parametri come la profondità della scena la separazione delle immagini nel render e la distanza dell'utente dal monitor. Inoltre il *client* si occupa della gestione della camera che l'utente può utilizzare effettuando movimenti, zoom e rotazioni di essa.

Una delle parti più delicate da gestire è stata quella della compressione dei dati, non tanto per le immagini a colori, ma quanto per le depth map. Questo perché esse contengono la geometria della scena, e come si vedrà all'interno di questo lavoro, alti *rate* di compressione fanno perdere gran parte delle informazioni e di conseguenza non permettono ricostruire perfettamente la scena. Quando si visualizza la scena con queste impostazioni la possibilità percepire il 3D è nettamente bassa, e poi venivano introdotti degli errori nella costruzione della *mesh* che si sono dovuti gestire.

Questo lavoro di tesi è strutturato nel seguente modo:

- **Capitolo 2**: verranno presentati gli strumenti utilizzati: panoramica dei sistemi di acquisizione delle informazioni 3D, alcuni dettagli sul kinect; verrà descritto come funziona il *framework OpenNI*; saranno accennate delle informazioni su cos'è un sistema di visione stereoscopico e come funziona quello NVIDIA.

- **Capitolo 3**: in questo capitolo verrà descritto il funzionamento degli algoritmi di compressione *JPEG2000* e *JPEG*; e come si è deciso di procedere per la compressione delle *depth map*.

- **Capitolo 4**: all'interno di questo capitolo verranno presentate e giustificate tutte le idee implementative, fino ad arrivare in dettaglio ai compiti svolti dal client e dal server.
- **Capitolo 5**: verranno riportati i risultati dei test percettivi fatti su un gruppo di persone; inoltre anche dei risultati fatti comparando due algoritmi di compressione, con qualche considerazione su come impostare i parametri degli algoritmi di compressione.

Capitolo 2

Acquisizione e visualizzazione 3D

In questo capitolo verranno presentati dei vari metodi per acquisire e visualizzare delle immagini stereoscopiche. Si scenderà più nello specifico sugli strumenti che sono stati utilizzati per lo sviluppo di questo lavoro di tesi.

2.1 Acquisizione depth map

Nella grafica 3D per *depth map* si intende un'immagine che contiene delle informazioni relative alla distanza delle superfici rispetto alla camera che sta inquadrando la scena. Esistono vari metodi per acquisire le *depth map*.

Uno strumento utilizzato per l'acquisizione è quello basato sul *tempo di volo* [1], [10], [2]. Queste tecnologie si basano su un principio di funzionamento abbastanza semplice e simile a quello dei RADAR: emettono un segnale infrarosso modulato sinusoidalmente e stimano la profondità dei dati misurando la differenza tra la trasmissione del segnale e la sua ricezione. I sistemi *time-of-flight (TOF)* quindi necessitano di grande precisione nelle misurazioni temporali, da cui dipende l'accuratezza della misura di profondità.

Un altro metodo per acquisire l'immagine di profondità è quello di adottare un sistema stereoscopico [9]. La visione stereoscopica permette di inferire la struttura tridimensionale di una scena osservata da due o più telecamere (nel caso di due telecamere si parla di visione binoculare). Il principio alla base della visione stereoscopica, consiste in una triangolazione mirata a mettere in relazione la proiezione di un punto della scena sui due piani immagine delle telecamere che compongono il sistema di visione stereoscopico. L'individuazione dei punti omologhi, problema noto in letteratura come *problema della corrispondenza*, consente di ottenere una grandezza denominata disparità (*disparity*) mediante

Figura 2.1: TOF - Mesa SwissRanger [7]

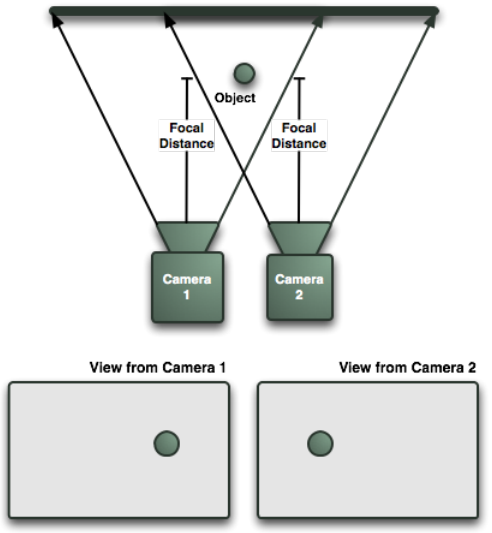

Figura 2.2: Schema acquisizione stereoscopica

la quale, conoscendo opportuni parametri del sistema stereoscopico, è possibile risalire alla posizione 3D del punto considerato.

2.1.1 Kinect

Il *Kinect* è prodotto dalla *Prime Sense* lasciato in licenza a Microsoft che lo ha lanciato sul mercato come accessorio per la sua console *Xbox 360*.

In figura 2.3 si può notare che il kinect: è formato da un sistema di acquisizione della depth map, da una telecamera a colori, da un array di microfoni e un motore che permette di poter muovere la barra in verticale. La telecamera RGB ha una risoluzione a 640x480 pixel. Il sensore infrarossi permette di acquisire *depth map* ad un alto frame rate e a costi relativamente ragionevoli.

Contrariamente da quanto si possa pensare il sistema di acquisizione della *depth map* del kinect non si basa su un sistema TOF. Ma è un sistema di luce strutturata: che proietta un pattern di pixel (griglia) a differente luminosità. Il modo con cui questi pattern si deformano quando impattano sulla superficie consente al kinect di calcolare le informazioni di profondità della scena.

Il grande problema del kinect è il rumore, e inoltre supporta solo basse risoluzioni. Il rumore è rappresentato da delle occlusioni, delle zone in cui il *kinect* non vede, che creano dei problemi in fase di ricostruzione 3D quanto si associa alla *depth map* i colori acquisiti dall'immagine VGA.

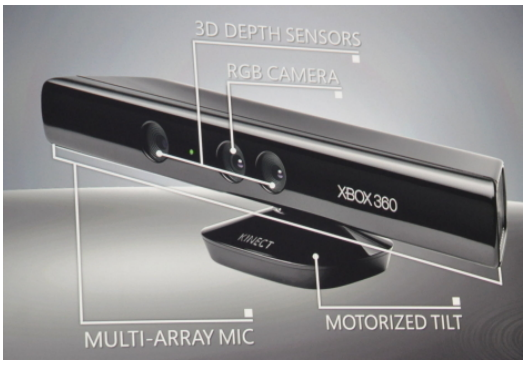

Figura 2.3: Kinect

Da quando è stato lanciato sul mercato la comunità si è subito messa al lavoro nel tentare di poter sfruttare il *kinect* come periferica per il PC. Il lavoro ha portato alla creazione di: *libfreenect* libreria che mette a disposizione delle classi che permettono di utilizzare il kinect. Il tutto si inserisce all'interno del progetto *OpenKinect*. Nel dicembre 2010, *Prime Sense*, ha rilasciato dei driver *open source* compatibili sia con *Windows* che con *Linux*. Questi driver consentono di poter avere il pieno controllo della periferica e sono basati su un API completa nota come OpenNI (*Open Natural Interaction*).

Per lo sviluppo di questa applicazione si è scelto il kinect, invece di un sistema combinato come sistemi stereoscopici o sistema TOF più camere, principalmente per una questione economica perché il costo tra questi due tipi di sensori sono nettamente differenti.

Purtroppo sul *kinect* non si possono riportare ulteriori informazioni in quanto tutto il progetto è protetto dal segreto dovuto ai vari brevetti.

2.1.2 Open Natural Interaction

Con il termine *Natural Interaction* si intende quella parte della scienza che si occupa della interazione uomo computer basata sui sensi dell'uomo ed in particolare si focalizza sull'ascolto e la visione. I dispositivi che permettono la *natural interaction* rendono obsoleti *device* come: tastiere, mouse, e controller remoti. Esempi di *natural interaction* li vediamo tutti i giorni come:

- riconoscimento di comandi vocali; dove il dispositivo agisce una volta riconosciuto il comando della nostra voce;
- riconoscimento di movimenti del corpo; utilizzato soprattutto per la realizzazione dei giochi.

OpenNI (Open Natural Interaction) è un *framework* multi-piattaforma che mette a disposizione delle *API* con cui effettuare il design di applicazioni che permettano l'interazione naturale con la macchina. Il principale obbiettivo di *OpenNI* è quello di fornire uno standard, attraverso le sue *API*, che permetta la comunicazione sia con dispositivi audio e video e anche attraverso del *middleware* percettivo (componenti software che analizzano l'audio e il video di una scena ed agiscano di conseguenza).

OpenNI fornisce un insieme di API che possono essere sfruttate dal dispositivo di acquisizione, e un altro insieme di API che può essere sfruttato delle componenti *middleware*. Si tenta in questo modo di spezzare la dipendenza tra sensore e *middleware*, infatti le API permettono di scrivere un'applicazione una sola volta e senza sforzi addizionali permette di operare con differenti moduli. Le API di questo *framework* consentono anche di lavorare direttamente con i dati raw, catturati dal sensore, tralasciando chi è il produttore del sensore; offrendo quindi la possibilità ai produttori di costruire dei sensori che siano *OpenNI* compilant.

write once, and deploy everywhere

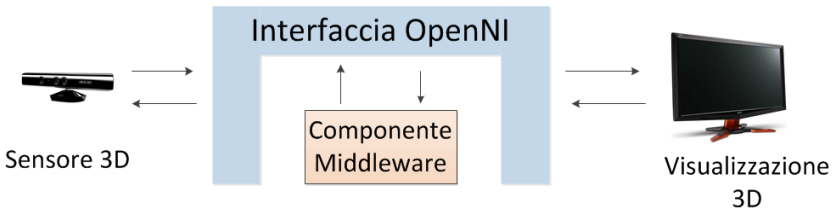

Figura 2.4: Principio di funzionamento applicazione con OpenNI

Il funzionamento di un'applicazione che sfrutta *OpenNI* è mostrato in figura 2.4 dove si può notare che ogni livello rappresenta un modulo indipendente:

- il livello più basso è rappresentato da un sensore di acquisizione 3D, in questo caso il *kinect*, che cattura gli elementi video e audio di una scena;
- il livello centrale è *OpenNI*, che si occupa della comunicazione tra il sensore e il *middleware*, che analizza i dati del sensore;
- il livello applicazione rappresenta il software che si occupa di raccogliere i dati di *OpenNI* e in questo caso li elabora e li visualizza su un monitor 3D.

Come si è potuto intuire *OpenNI* fornisce API sia per i sensori di acquisizione che per le componenti middleware. Le API consentono che più componenti siano registrate all'interno del framework. Per componente si intendono sia i sensori che i middleware che sono scritti. I componenti hardware che per ora supportati

sono: *sensori 3D (es.* kinect, *IR camera), webcam e dispositivi audio (microfoni o array di microfoni)*. Mentre le componenti middleware messe a disposizione da *OpenNI* sono:

- *full body analysis middleware:* un componente software che permette di processare i dati del sensore e generare le informazioni relative al corpo;
- *hand point analysis middleware:* componete software che processando i dati messi a disposizione dal sensore permette di generare la locazione dei punti della mano;
- *gesture detection middleware:* componente software che permette di identificare movimenti predefiniti e segnalarlo all'applicazione;
- *scene analyzer middleware:* componente software che analizza un'immagine della scena e nell'ordine produce le seguenti informazioni:
  - separazione del *background* dal resto della scena;
  - le coordinate del suolo;
  - l'individuazione di figure all'interno della scena.

2.1.3 Generazione e lettura dati

Generazione dati

I sensori possono essere intesi come dei veri e propri generatori di dati. Una volta che questi sono stati identificati e registrati all'interno del *framework*, non iniziano subito la generazione dei dati ma consentono di terminare le eventuali configurazioni. Questo garantisce che una volta aperto lo streaming dei dati questi siano conformi alle configurazioni richieste. I sensori non producono alcuna informazione fino a quando non viene creata la richiesta di inizio generazione dei dati (`StartGenerating()` se si ha un solo nodo che produce dati o `StartGeneratingAll()` se ci sono due o più nodi).

OpenNI inoltre mette a disposizione delle API che supportano la maggior parte delle *feature* dei sensori, chiamate *capabilities*. Naturalmente non tutte le *features* di tutte le case produttrici sono supportate, ma le più comuni tra di esse si. Per ora le più importanti *capatibilies* supportate sono:

- *punto di vista alternativo*: consente a qualsiasi sensore che genera mappe di profondità di trasformare i suoi dati come se il sensore fosse posizionato in un altro punto;

- *frame sync*: consente a due sensori di produrre dati sincronizzando i loro *frame rate* tale che i dati arrivino allo stesso tempo;
- *mirror*: consente di "specchiare" le informazioni i dati generati dal sensore;
- *User position*: consente a un generatore di immagini di profondità di ottimizzare l'output che è generato per specifiche aree della scena.

Lettura dei dati

I sensori generano in maniera continua nuovi dati, anche mentre l'applicazione sta ancora elaborando un vecchio *frame*; La conseguenza di questo è che: i sensori devono in qualche modo memorizzare internamente i dati, fino a quando non ricevono una richiesta esplicita di aggiornamento dei dati. Questo significa che i generatori di dati dovrebbero "nascondere" al loro interno i dati, finché l'applicazione che sta utilizzando il sensore non invoca la funzione `UpdateData`. *OpenNI* consente all'applicazione di mettere in attesa i nuovi dati disponibili fino a quando non viene invocata la funzione `xn::Generator::WaitAndUpdateData()`. Di questa funzione esistono diverse varianti a seconda di cosa si vuol fare: ad esempio consente di aggiornare i dati di uno solo dei sensori se si utilizzano vari oppure aggiornare i dati forniti da entrambi contemporaneamente. È consigliabile utilizzare la funzione di aggiornamento di tutti i sensori invece di aggiornarne solo uno questo perché: se ad esempio c'è un nodo che dipende da un altro, viene rispettata la gerarchia dei nodi aggiornando prima il nodo che non ha dipendenze e poi quello da cui dipende.

Un ulteriore tool di *debug* messo a disposizione da *OpenNI* è la registrazione dei dati. Infatti è possibile memorizzare su disco l'intera scena e successivamente riprodurre i dati. Questa funzione è risultata utilissima in fase di test, perché veniva riprodotto sempre lo stesso video con differenti tecniche di codifica.

2.2 Visione stereoscopica

Creare un'immagine 3D su di un monitor 2D è il sogno dei produttori da circa 160 anni, quando *sir Charles Wheastone* utilizzando coppie di disegni similari, e la nascente fotografia in seguito, tentò i suoi primi esperimenti di visone stereoscopica. La stereoscopia è la tecnica di realizzazione e visione di immagini finalizzata a trasmettere una illusione di tridimensionalità.

La stereoscopia lavora presentando a ciascun occhio due immagini sensibilmente differenti e utilizzando varie tecniche si cerca di trasmettere al cervello umano una sensazione di tridimensionalità.

Anaglifi è una delle prime tecniche create per la visione stereoscopica. È composto da due immagini stereoscopiche monocromatiche, ognuna con una dominante di colore differente, e sovrapposte sul medesimo supporto. Le immagini devono essere osservate con degli occhiali che hanno le lenti colorate con la stessa componente colore delle dominanti.

Sistemi di polarizzazione, utilizzato in molti cinema, sfruttano due proiettori ognuno dotato di una lente di polarizzazione che polarizza la luce in una direzione per l'immagine sinistra e in quella opposta per quella destra. Lo schermo deve mantenere la polarizzazione. Gli spettatori sono dotati di speciali occhiali con lenti polarizzate, in modo che all'occhio destro arrivi solo l'immagine destra (lo stesso per l'immagine sinistra).

Sistemi attivi, come gli occhiali venduti da *NVIDIA*, usano un insieme di lenti che in maniera alternata e sincronizzata con lo schermo oscurano una lente e poi l'altra, in modo che ogni frame venga visto solo da un occhio. Ogni occhio vede così delle immagini diverse creando in questo modo l'illusione 3D.

2.2.1 NVIDIA 3D Vision

Il sistema di visione stereoscopica NVIDIA, tenta di andare in contro allo sviluppatore di applicazioni stereoscopiche sollevandolo dal compito di fargli gestire il doppio render dell'immagine, e di come questi debbano interagire con la scheda video. Infatti lo sviluppatore creerà l'applicazione 3D come sempre e poi sarà il sistema NVIDIA con delle euristiche che permetterà la visualizzazione della scena.

Usando delle euristiche, i driver stereoscopici decidono quali oggetti dovrebbero essere disegnati per occhio e quali no, e costruiscono l'intera immagine per l'occhio sinistro e destro in maniera trasparente allo sviluppatore.

2.2.2 Visione stereo NVIDIA

Per capire come funziona la visione stereo NVIDIA, bisogna tener presente quale sia la pipeline utilizzata per visualizzare una qualsiasi scena. In figura 2.5 sulla sinistra è riportata una pipeline concettuale utilizzata dalle directX per la visualizzazione di una scena 3D. I passaggi sono quelli classici: nel *vertex shader* un elemento viene piazzato all'interno della scena 3D trasformando le coordinate del vertice dell'immagine secondo i vari step della visualizzazione. Subito dopo i vertici sono raggruppati in *index buffer* e vengono rasterizzati. Dopo aver preparato i frammenti da visualizzare, applicando la trasformata nel *viewport*, c'è la possibilità all'interno del *pixel shader* di modificare il colore di ogni singolo vertice.

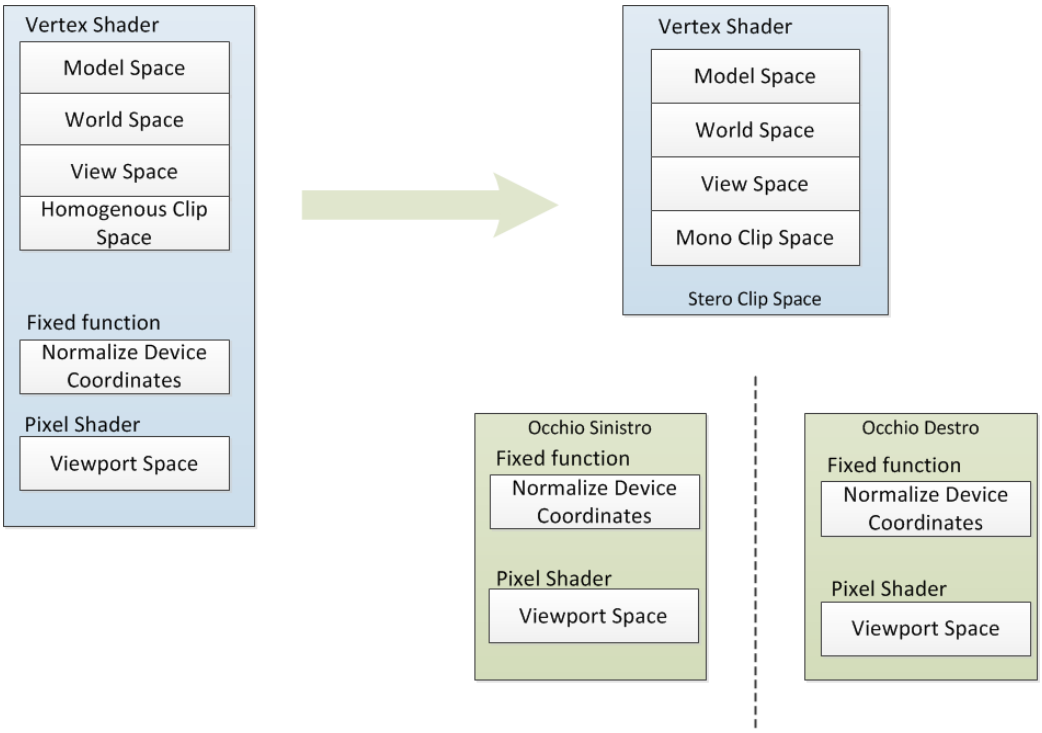

Figura 2.5: Modifica della pipeline concettuale della visualizzazione

La modifica della pipeline generale da parte di NVIDIA 3D Vision può essere vista come due passi fondamentali: duplicazione e modifica. Come si può vedere dalla figura il *viewport* è diviso in *viewport* occhio destro e *viewport* occhio sinistro. Per fare questo, il driver effettua le seguenti operazioni per conto dell'applicazione:

- Duplicare il render utilizzando le euristiche stereoscopiche
- Modificare il *vertex shader* affinché possa effettuare la trasformazione da spazio mono a spazio stereoscopico.
- Cambiare la chiamata per il render, con due chiamate per il render una per occhio.

Uno dei compiti più ovvi che il driver NVIDIA deve compiere al posto dell'applicazione è quello di duplicare le chiamate render dell'oggetto. Questo consente al driver di costruire il render dell'oggetto da presentare ad ogni occhio. In aggiunta l'oggetto da disegnare potrebbe essere duplicato basandosi sull'analisi euristica quando viene creato l'oggetto.

Duplicare il render dell'oggetto non è sufficiente. Infatti il driver modifica gli *shader* aggiungendo un pezzo di codice alla fine, questa aggiunta permette di poter gestire il *clip space* che ha una serie di proprietà speciali. Tra queste c'è quella che è orientato nello stesso modo dell'applicazione, indipendentemente da convenzioni locali. Ha inoltre una proprietà che permette di scalare le coordinate *x, y, z* e *w* di un vertice per accentuare l'illusione stereoscopica, senza dover intervenire modificando lo *z-buffer*, il processo di rasterizzazione e quindi i frammenti risultanti. Lo *shader* inserisce l'informazione in una variabile chiamata `PsInput` e questa viene modificata dai driver con la seguente equazione

$$PsInput.x = Separazione * (PsInput.w - Convergenza) \tag{2.1}$$

La convergenza è un valore relativamente piccolo, e può essere modificato dall'utente utilizzando il pannello di controllo NVIDIA. Valori piccoli della convergenza renderà negativo il mio valore di separazione creando in questo modo accentuato l'effetto *out of screen*.

L'effetto degli oggetti che vengono fuori dallo schermo è quello a cui sono interessanti tutti gli utenti, ma esagerare con questo effetto può risultare stancante per la vista dell'utente. Il cervello però fa fatica ad accettare che degli oggetti vengono fuori dallo schermo; per migliorare l'effetto sarebbe raccomandabile far muovere gli oggetti per tutta la profondità della scena e consentire all'utente di poter interagire con la scena muovendo la camera.

Capitolo 3

Compressione dei dati

Le immagini generate dal kinect *sono troppo grandi per poter essere spedite sulla rete; in questo capitolo verranno presentati gli standard di compressione* JPEG *e* JPEG2000 *. Inoltre verrà spiegato come si è pensato di comprimere le* depth map*.*

Il *kinect* genera due tipi di immagini, una di profondità, che definisce la geometria di quello che vede, e una a colori. La risoluzione delle immagini è di *640 x 480* con 16 bit per pixel per quella di profondità, e 24 bit (8 per colore) per pixel per l'immagine a colori. Considerando che il *kinect* genera circa trenta frame al secondo e che ogni frame pesa circa 1,5 MB sarebbe impensabile che una qualsiasi rete riesca ad gestire un traffico di 46MB al secondo. Da qui la necessità di comprimere le immagini prima di inviarle sulla rete. Per il momento si è pensato di utilizzare gli standard *JPEG* o *JPEG2000* per comprimere le singole immagini.

In questo capitolo verrà spiegato il funzionamento di questi protocolli di compressione e come si comprime le depth map in ognuno dei protocolli. Mentre la compressione delle immagini viene tralasciata in quanto si utilizzano i protocolli in maniera standard.

3.1 Compressione JPEG

Lo standard *JPEG* (*Joint Photographic Expert Group*) è uno dei più conosciuti e diffusi protocolli per la compressione delle immagini. L'approccio utilizzato da *JPEG* è basato su *discrete cosine transform (DCT)* ([6] pagg. 402-404). Un tipico uso del *JPEG* è la compressione *lossy* delle immagini, che in qualche modo introduce una minima perdita dei dati, ma che comunque garantisce una buona fedeltà di esse quando vengono decompresse. *JPEG* è considerato uno dei migliori standard per la compressione di fotografie.

Riportiamo in figura 3.1 uno schema a blocchi che illustra quali sono i passaggi messi a disposizione dallo standard per comprimere le immagini. L'algoritmo comprime in maniera indipendentemente blocchi 8x8, successivamente viene applicata la trasformata *DCT*; dopo questo ai coefficienti della trasformata viene applicato un processo di quantizzazione, ed infine i valori quantizzati vengono codificati ottenendo in questo modo l'immagine compressa. [11]

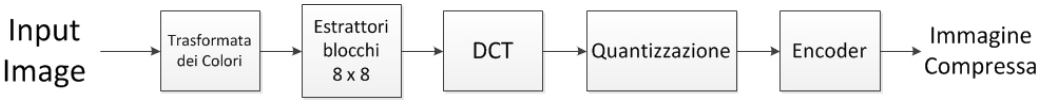

Figura 3.1: Schema a blocchi compressione *JPEG*

3.1.1 Come funziona

Il primo passo della compressione *JPEG* è è la suddivisione dell'immagine in input in blocchi 8 x 8 indipendenti; essi sono processati da sinistra a destra da sopra a sotto. Successivamente viene cambiato modello di colori: si passa da RGB a $Y^l C_B C_R$ (alcune volte questo passaggio viene saltato).Ad ognuno dei 64 pixel è effettuato un "*level shift*" sottraendo 2^{m-1} dove m è il numero di bit che sono utilizzati per descrivere ogni pixel. In questo modo se per esempio si ha una immagine in scala di grigi, ad 8-bit per pixel, i suoi valori saranno compresi tra 0 e 255; dopo questa operazione i valori saranno distribuiti intorno allo zero in un intervallo tra -127 a 128. Successivamente per ogni blocco viene calcolata la DCT. Dopo aver applicato la trasformata si può notare che in ogni blocco la componente a bassa frequenza (primo valore del blocco) hanno valori nettamente superiori alle componenti ad alta frequenza.

Il passo successivo è quantizzare i coefficienti ottenuti dopo la DCT. I passi di quantizzazione sono organizzati in una tabella, della quale si riporta un esempio nella tabella 3.1 (suggerito sempre da [11]). La quantizzazione avviene seguendo la seguente equazione:

$$I(u,v) = \left\lfloor \frac{T(u,v)}{Z(u,v)} + 0.5 \right\rfloor \tag{3.1}$$

dove $T(u,v)$ sono i coefficienti dopo aver calcolato la DCT, mentre $Z(u,v)$ sono i valori di quantizzazione. Come si può notare, dalla tabella di quantizzazione 3.1, i passi di quantizzazione crescono quando si passa dalle componenti a bassa frequenza a quelle ad alta frequenza. Ricordando che i valori dopo la trasformata *DCT* sono bassi per le alte frequenze e che gli step di quantizzazione sono alti per

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	19	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	21	120	101
72	92	95	98	112	100	103	99

Tabella 3.1: Tabella di quantizzazione

le stesse, dopo la fase di quantizzazione nel blocco dell'immagine si presentano lunghi *run* di zeri; utilissimi nell'ultima fase di *encoding*.

Nell'ultima fase vengono utilizzate due codifiche differenti per le componenti ad alta e bassa frequenza.

In particolare, le componenti ad alta frequenze sono codificate usando un codice a lunghezza variabile che definisce il valore del coefficiente e il numero di zeri che lo precedono. Per la componente a bassa frequenza, non si codifica il valore vero e proprio, ma si codifica la differenza relativa tra due blocchi consecutivi dell'immagine. Le tabelle per la codifica di *Huffman* delle componenti ad alta e bassa frequenza sono fornite dallo standard, ma viene anche lasciata libertà allo sviluppatore di costruire delle proprie tabelle; stesso discorso si può applicare alla tabella di quantizzazione, adattando le tabelle alle caratteristiche dell'immagine e alle esigenze dello sviluppatore.

3.1.2 Compressione depth map

Nello standard *JPEG* prima di comprimere le immagini bisogna specificare il tipo di immagine che si ha intenzione di comprimere; si deve indicare la grandezza dell'immagine, il numero di componenti colore ed anche il numero di bit con il quale è codificato ogni pixel. Per esempio se si volesse comprimere un'immagine a colori il numero di componenti sarà tre (se si usa come modello di colori RGB) e il numero di bit per componente sarà otto. Mentre se si comprime un'immagine in scala di grigi si specifica una componete soltanto per il colore e che il numero di per quella componente sono sempre otto.

Come detto in precedenza la *depth map* che il kinect fornisce ha 16 bit per pixel, quindi non è un'immagine che ricade in uno dei due esempi precedenti. Di seguito riportiamo i due approcci differenti che sono stati implementati per comprimere l'immagine di profondità.

Visto che ogni pixel della *depth map* è formato da 16 bit si è pensato di creare due immagini a partire dalla stessa. Per ogni pixel tramite una rotazione dei bit si creano due immagini: una considerando i bit meno significativi del pixel (`depthL`) e l'altra considerando i bit più significativi (`depthH`). In questo modo si possono comprimere in maniera indipendente le due immagini, e visto che ogni pixel ha una componente a otto bit, si può pensare di comprimerle come se fossero delle immagini in scala di grigi. `depthH` veniva compressa in maniera *lossless* mentre `depthL` in maniera *lossy*. Queste due immagini venivano successivamente spedite al *client* che si occupava di decomprimerle e di conseguenza ricostruire la *depth map* tramite la rotazione inversa dei bit.

La parte bassa della *depth map* si è deciso di comprimerla in maniera *lossy* perché degli otto bit di ogni pixel solo i primi 4 contengono informazioni, in quanto le informazioni che restituisce il kinect sulla *depth map* sono codificate nei 12 bit più significativi. Questo ha fatto nascere un'idea: pagare un po' di qualità della *depth map* comprimendo solo la parte alta ed inviarla al *client* evitando di inviare la parte di *depth map* che corrisponde alla parte bassa dei vari pixel. Va anche aggiunto che i 4 bit che vengono scartati dalla parte bassa non contengono informazione e quini a maggior ragione non vengono presi in considerazione

Figura 3.2: Depth non compressa

Figura 3.3: Depth map compressa

In figura 3.3 è riportata un'immagine compressa considerando la sola componente a bassa frequenza. Mentre in figura 3.2 è riportata la *depth map* che viene acquisita dal *kinect* e come si può notare le differenze sono minime e impercettibili ad occhio umano.

Per l'utilizzo dello standard *JPEG* si è utilizzata la libreria fornita da *Indipendent JPEG Group*.

3.2 Compressione JPEG2000

JPEG2000 è uno standard emergente per la compressione delle immagini. Le immagini diventano sempre più di alta qualità e quindi ci sono sempre più dati da manipolare. La compressione delle immagini non può, quindi, solo occuparsi di ridurre lo spazio che esse occupano, ma inoltre di estrarre informazioni che siano utili per un lavoro di *editing, processing* e *targeting* per particolari dispositivi e applicazioni. Il *JPEG2000* trae vantaggio dalle migliori performance in termini di *rate distorsion* rispetto alla compressione *JPEG* . Ma ancora di più *JPEG2000* permette di estrarre immagini a differente risoluzioni, regioni di interesse, componenti e molto di più: tutto dallo stesso *bitstream* compresso.

3.2.1 Come funziona

Forniremo una descrizione di *JPEG2000* . L'obbiettivo è quello di mettere a fuoco gli aspetti principali tentando di capire le scelte algoritmiche effettuate senza scendere troppo nei dettagli. La procedura di compressione si compone di vari step come mostrato in figura 3.4.

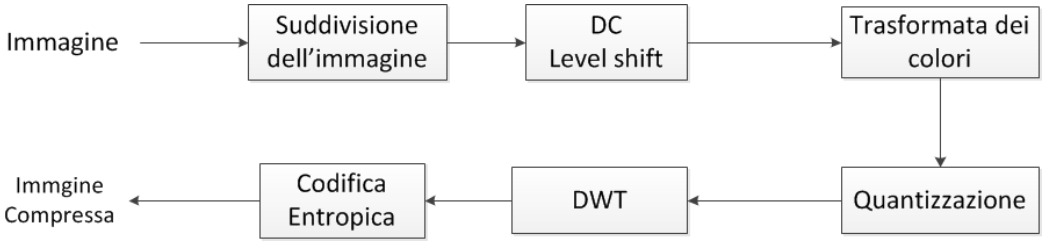

Figura 3.4: Schema a blocchi compressione *JPEG2000*

Concettualmente l'algoritmo divide l'immagine in blocchi. La grandezza di questi blocchi è arbitraria, si può anche considerare l'intera immagine. Ognuno di questi blocchi sarà compresso in maniera indipendente. Dopo aver effettuato la suddivisione in blocchi viene effettuato un *DC level shift* su ogni campione del blocco in questo modo ogni componente colore senza segno diventa un valore con segno centrato intorno allo zero (molto simile a quello che fa lo standard *JPEG*).

JPEG2000 può comprimere immagini con un numero arbitrario di componenti, nel caso di immagini standard RGB, può essere applicata una trasformazione del colore nello spazio dei colori YC_bC_R. Lo standard supporta due tipi di trasformazioni dello spazio dei colori: *reversible color transform (RCT)* che può essere usata sia per la compressione *lossy* e *lossless*; e *irreversible color transform(ITC)*

che può essere usata solo per la compressione *lossy*. Questa trasformazione ha fondamentalmente tre obbiettivi:

- rendere indipendenti le componenti colore in modo da ottenere una maggiore efficienza in compressione;
- permettere di enfatizzare la componente della luminanza (Y) nel passo di quantizzazione;
- nel caso RCT consente una perfetta ricostruzione dell'immagine nel caso si effettui una compressione *lossless*

La conversione irreversibile da RGB a YC_bC_R è data dalla seguente equazione:

$$\begin{pmatrix} Y \\ C_b \\ C_R \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.5 \\ 0.5 & -0.419 & -0.081 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{3.2}$$

Mentre la conversione reversibile è data dalla seguente equazione:

$$\begin{aligned} Y &= \left\lfloor \tfrac{R+2G+B}{4} \right\rfloor \quad C_b = B - G \quad C_R = R - G \\ G &= Y - \left\lfloor \tfrac{C_b - C_R}{4} \right\rfloor \quad R = C_R + G \quad B = C_b + G \end{aligned} \tag{3.3}$$

Questo passo non può essere applicato anche ai colori delle immagini in scala di grigi. Nella sezione successiva verrà spiegato come si è proceduto per comprimere le *depth map* usando *JPEG2000*

La successiva operazione è la *Discrete Wavelet Transform*. Lo standard prevede due differenti tipi di utilizzo della trasformata. Se si intende effettuare un tipo di trasformata *lossy* allora è il caso di utilizzare una trasformata (9,7) *floating point*, mentre se si intende effettuare una trasformata *lossless* si deve utilizzare la trasformata (5,3) intera. Ogni passo della trasformazione divide il blocco dell'immagine in quattro differenti *subband*, come mostrato in figura 3.5,la *subband* a bassa frequenza *(LL)*, corrisponde all'immagine a bassa risoluzione, mentre le tre *subband* ad alta frequenza rimanenti *(LH, HL, HH)* contengono i dettagli dell'immagine. La *LL subbands* è a sua volta decomposta. Il numero di livelli non è fisso, di solito ne bastano cinque. Dopo un livello L di decomposizione *wavelet* si ottiene la *subband* al più basso livello di risoluzione, che denoteremo LL_L, per ogni livello di risoluzione per $i = 1, \ldots, L-1$ si ha un insieme di *subband* ad alte frequenze LH_i, HL_i, HH_i. La decomposizione *wavelet* offre un modo semplice per ottenere vari livelli di qualità di immagini *JPEG2000* .

Dopo la trasformata tutti i coefficienti sono soggetti a quantizzazione uniforme utilizzando una *dead-zone* fissata intorno all'origine. Questo è ottenuto

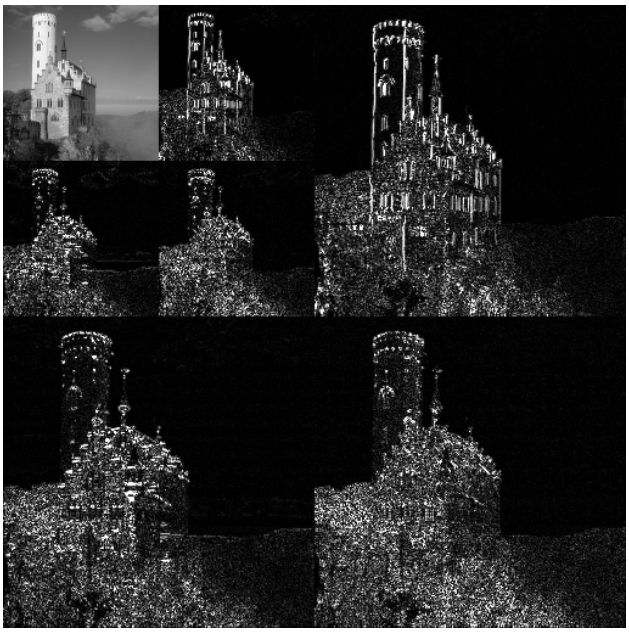

Figura 3.5: Wavelet 2-level decomposition

dividendo il valore di ogni coefficiente per uno step di quantizzazione e prendere la parte bassa di questa quantità. Un differente *step* di quantizzazione è utilizzato per ogni *subbands*. Lo step di quantizzazione può essere scelto in base alla qualità dell'immagine che si vuole ottenere.

Dopo la quantizzazione ogni *subbands* è soggetto ad una *packet partition*, che divide divide ogni *subbands* in rettangoli regolari senza sovrapposizione. Tre rettangoli spazialmente cooerenti (uno per ogni *subband* allo stesso livello di risoluzione) formano una *pocket partition*. Questa *pocket partition* formano i *precinct*. La suddivisione in *precincs* consente un efficiente accesso random ai dati compressi. Infine i *precinct* sono divisi a loro volta in rettangoli chiamati *codeblock* che sono elementi fondamentali per l'ultima fase di *encoding*.

Ricapitolando: ogni immagine è divisa in blocchi (*tile*) e ognuno di questi viene trasformato. Le *subbands* (di un *tile*) sono divisi *packet partition*. Infine ognuno di questi blocchi viene diviso in *codeblock*. Questa situazione è illustrata dalla figura 3.6

L'ultimo passo è la codifica entropica dei *codeblock*. Ogni *codeblock* è codificato in maniera indipendente usando tecniche di codifica aritmetica (*MQ coder*) e il paradigma *EBCOT* (*Embedded Block Coding with Optimal Truncation*). Questo efficiente sistema di codifica produce un fine *stream* di bit che può essere troncato in qualsiasi punto, consentendo un buon trade-off tra qualità

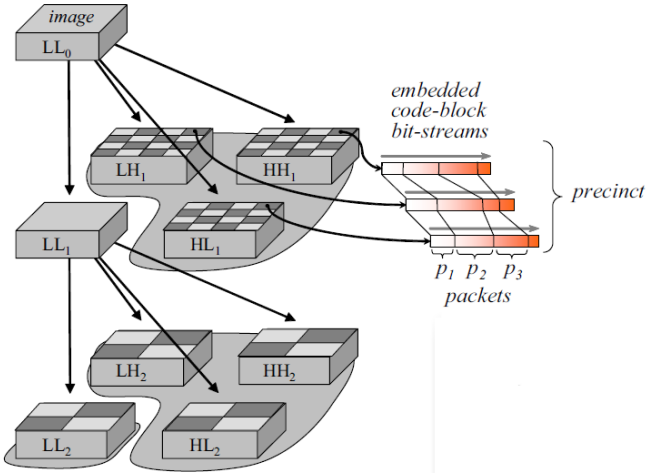

Figura 3.6: Elementi della compressione *JPEG2000*

dell'immagine e lunghezza del codice.

Il vantaggio principale dei file compressi con *JPEG2000* consiste nel fatto che racchiudono al loro interno più risoluzioni differenti della stessa immagine. Una simile caratteristica è importantissima per i futuri usi di questo formato su Internet. I file *JP2*, che sarebbe l'estensione proposta per identificare lo standard *JPEG2000* , consentiranno insomma di racchiudere in un unico documento l'anteprima, la bassa la media e l'alta risoluzione della stessa immagine, senza però moltiplicare proporzionalmente il peso del file.

Dettagli sullo standard *JPEG2000* possono essere trovati in [3] per uno studio più approfondito, mentre per una lettura veloce di questo standard [8] [12].

3.2.2 Compressione depth map

Per comprimere la *depth map* acquisita dal *kinect* utilizzando lo standard *JPEG2000* si è deciso di utilizzare il *Kakadu software*. *Kakadu* è una completa implementazione in *C++* dello standard *JPEG2000* , Parte 1, come ISO/IEC 15444-1, più le principali componenti della Parte 2 e Parte 3.

Lo standard *JPEG2000* non impone vincoli sul numero di bit che si devono utilizzare per componente; quindi il problema che si è presentato nello standard *JPEG* di dividere l'immagine in parte alta e parte bassa con questo standard non sussiste.

Nella compressione della *depth map* non si sono utilizzate tutte le specifiche che *Kakadu* mette a disposizione. Anzi si è prediletto un approccio semplice

principalmente per capire come utilizzare questo software. Sono state utilizzate le principali caratteristiche, tralasciando le impostazioni più complesse, che per il momento non interessano, ma questo ha comunque portato a dei risultati interessanti che illustreremo nel Capitolo 5.

Riportiamo uno pseudocodice delle istruzioni principali che servono per comprimere la *depth map*

Algoritmo 1 Compressione Depth map

Creazione dei parametri per il codestream

$Scomponents \leftarrow 1$ *Numero di componenti dell'immagine*
$Sdims[0] \leftarrow 640$ *Larghezza dell'immagine*
$Sdims[1] \leftarrow 480$ *Altezza dell'immagine*
$Sprecision \leftarrow 16$ *Numero di bit per componente dell'immagine*
$Ssigned \leftarrow false$ *Indica se i bit sono con segno o meno*

Crea il target di output
Crea il codestream con le impostazioni precedenti e il target di output
Stabilisci il numero di livelli dell'immagine, e indica
se la compressione è lossy o lossless

Creazione dell'oggetto di compressione specificando
i paramatri delle componenti colore e il codestream

$comrpessor.pushstripe(buffer, 640);$ *effettiva compressione dell'immagine*

Ripulisci memoria

Come si può notare la compressione della *depth map* sembra abbastanza facile e intuitiva.

Le difficoltà riscontrate utilizzando sia *JPEG* che *JPEG2000* sono state nel reindirizzare l'output; cioè modificare lo standard che in automatico salva l'immagine compressa su file. Si è dovuto quindi intervenire nella libreria modificando le istruzioni che scrivono l'output su file ad aree di memoria molto più consono per questo tipo di applicazione.

In appendice sono riportate le note tecniche per ottenere il risultato della compressione in memoria anziché su file.

Capitolo 4

Architettura di sistema

In questo capitolo verrà approfondita l'idea di videochat esponendo e giustificando le scelte implementative. Fino ad arrivare nel dettaglio dei compiti svolti dal lato client e dal lato server.

4.1 Idea generale

Come si può intuire dai capitoli precedenti, prima di passare all'implementazione di un nuovo sistema di videochat, bisogna pensare bene alle caratteristiche che l'implementazione deve offrire.

Va inoltre precisato che si è optato, dove è stato possibile, ad effettuare delle scelte che non fossero troppo limitative per il riutilizzo del codice. Infatti si è pensato ad una struttura dell'applicazione modulare che non chiudesse nessuna possibilità di modifica dell'ossatura dell'applicazione.

Di seguito si spiegheranno alcune scelte effettuate fino a scendere nel dettaglio dell'implementazione.

4.1.1 Comunicazione client-server

Onde evitare di appesantire eccessivamente l'applicazione di moduli si è pensato di dividerla in due parti. La prima parte, che per comodità chiameremo *server*, si occuperà della gestione del *kinect*, che dopo un processing dei dati acquisiti, li invia tramite un *socket* sulla rete. La seconda parte che chiameremo *client* si occuperà di connettersi ad un server e dopo un lavoro di *processing* mostrerà a video la scena tridimensionale acquisita dal server.

Con questa divisione dei compiti è possibile che più applicativi *client* possano connettersi allo stesso *server*, prevedendo in questo modo uno sviluppo futuro dell'applicazione come ad esempio una multi videochat.

Si riporta uno schema generale del funzionamento dell'applicativo tra una coppia di computer.

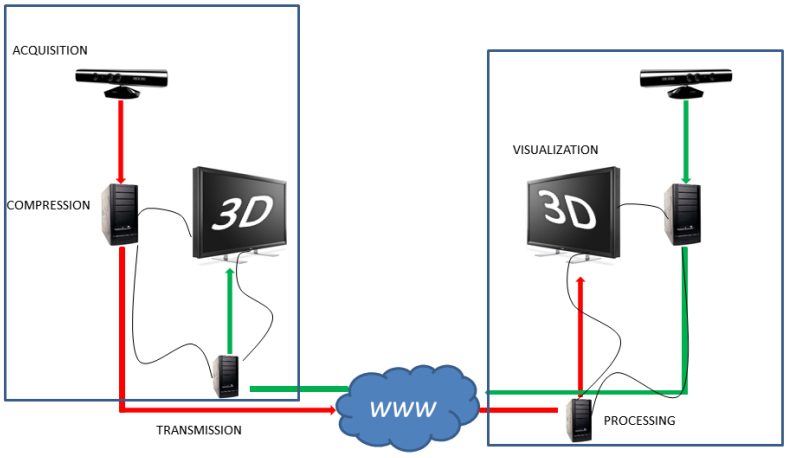

Figura 4.1: Schema principale di comunicazione

Come si può notare nella figura non si è fatta distinzione tra lato *client* e *server* perché si ipotizza che risiedano entrambi sulla stessa macchina. Evitando di gestire una comunicazione non affidabile, che potrebbe risultare più veloce, si è pensato nell'implementazione di appoggiarsi al protocollo *TCP/IP*.

Dopo verrà spiegato in maniera più approfondita i compiti che vengono svolti sia dal server che dal client.

4.2 Design lato server

Quello che è stato definito come *server* svolge principalmente tre compiti: accettare le richieste di connessioni, acquisire e comprimere i dati dal *kinect* e infine li invia al *client*. Della comunicazione se ne occupa un *socket* si è scelto di appoggiarsi alla classe *winsock* fornita dalle librerie di *Windows*.

4.2.1 Socket handler

Windows socket (winsock) è una libreria che consente agli sviluppatori di creare applicazioni per reti intranet, internet indipendentemente dal protocollo di co-

municazione utilizzato. La programmazione dei *winsock* si basa principalmente sul protrocollo *TCP/IP*.

Il server, quando viene avviato, la prima operazione che effettua è quella di creare un socket e di di porsi in ascolto di eventuali connessioni. Quando il client si connette al *socket* in ascolto il server riceve una notifica da parte di *winsock*; accettando la connessione, il server, lancia un *thread* ed inizia ad acquisire i dati dal *kinect*. Come è stato detto in precedenza il *server* è in grado di gestire più connesioni da parte di più client; il modo con cui lo fa è quello di lanciare un *thread* per ogni connessione *client*. Questo consentirà a più *client* di visualizzare la stessa scena acquisita dal *server*.

```
int send(SOCKET s,
const char *buf, int
len, int flags);
```

Per trasferire i dati tra *client* e *server* si usano due primitive `send` e `recv`. La primitiva `send` invia i dati nel buffer del socket e ritorna il numero di *byte* messi in coda. La funzione `recv`, legge i dati che sono a disposizione nel socket e li memorizza all'interno di un buffer.

```
int recv(SOCKET s,
char *buf, int len,
int flags);
```

Queste funzioni sono di base non bloccanti, cioè non si mettono in attesa fino al completamento dell'invio/ricezione dei dati; in teoria questo comportamento dovrebbe velocizzare l'applicazione. Visto che vengono generati e inviati circa trenta *frame* al secondo, più le rispettive dimensioni di ogni frame; in fase di avvio della comunicazione si verifica che vengono letti nei buffer dei valori non coerenti. In fase di decompressione questo è causa di errori da parte degli algoritmi di decompressione, che leggono dei valori non conformi. Pertanto si è optato di utilizzare delle `recv` bloccanti, modificando il campo `flags` della primitiva, in questo modo il *client o server* si pone in attesa fino a quando non riceve tutti i dati.

4.2.2 Protocollo di comunicazione

A questo punto sembra doveroso presentare il protocollo di comunicazione che si è voluto creare, affinché la comunicazione tra *client* e *server* proceda senza intoppi.

Dopo tutte le inizializzazioni effettuate dal *server*, dopo che il server ha ricevuto una richiesta di connessione da parte del *client* ed aver lanciato il *thread* che servirà quella connessione: il *server* è pronto a procedere con l'invio dei dati al *client*

Subito dopo che la connessione è stata instaurata il *server* invia al *client* un valore che indica il tipo di compressione che si è deciso di utilizzare. Dopodiché il server incomincia ad acquisire e inviare i dati del *kinect* al *client*. Ogni compressione produce buffer di grandezza differente, per ogni frame si è reso necessario dover inviare la grandezza dei buffer della compressione e poi i buffer contenenti le immagini compresse. Quindi viene inviata per prima la grandezza della *depth map* compressa, poi il buffer che la contiene; analogo discorso avviene per l'im-

magine. Ogni volta che il *server* effettua un invio si mette in attesa di un *ack* da parte del *client*; questa attesa fino di una conferma da parte del *client* può essere vista come: una conferma di buona ricezione di quello che è stato inviato e l'autorizzazione a procedere con l'invio del frame successivo. Infine il *server* si metterà in attesa di un ultimo *ack* da parte del *client* che verrà inviato quando questi avrà finito di preparare il frame, che dovrà essere visualizzato. In figura 4.2 è riportato un esempio del protocollo di comunicazione sopra descritto.

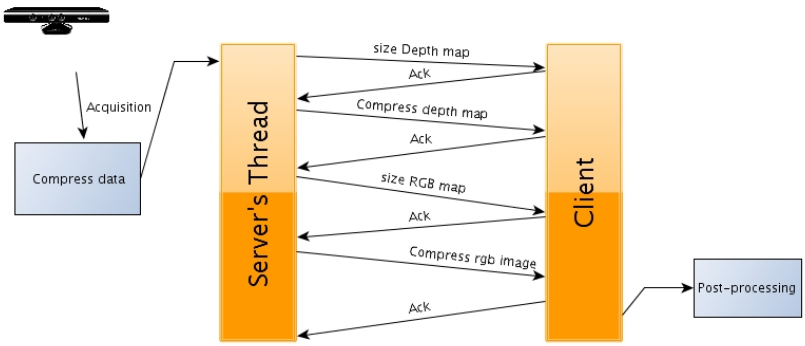

Figura 4.2: Esempio di scambio dei dati client server

4.2.3 Gestione del kinect

Il *server* oltre all'aspetto della comunicazione si occupa anche della gestione del kinect. Infatti, dopo aver creato il *socket* per la comunicazione e prima di mettersi in ascolto di eventuali richieste di connessione da parte dei client, una delle operazioni che server effettua è quella di inizializzare il *kinect*. Come spiegato in precedenza OpenNI (si veda 2.1.2) mette a disposizione delle funzioni per poter inizializzare e acquisire i dati dal *kinect* tramite degli script xml. Questo ha consentito di scrivere un semplice file indicando i nodi del kinect che si intendono utilizzare: specificando le risoluzioni con il quale operano i sensori, il numero di frame al secondo che generano ed altre specifiche che possono essere utili per l'inizializzazione del *kinect*.

Nell'inizializzazione del kinect si è posta, inoltre, l'attenzione su due piccoli particolari: sincronizzare per quanto possibile il *frame rate* delle due telecamere; per non ottenere due immagini totalmente distinte in fase di acquisizione: si è tentato di limitare il ritardo di una delle due immagini rispetto all'altra. In questo modo si è attenuato molto il fenomeno di vedere muovere, ad esempio, i colori della mano e successivamente vedersi muovere la geometria della mano.

Il secondo particolare al quale si è prestato attenzione è stato quello di fare "vedere" alle due telecamere la stessa cosa, cioè tentare di fare avere lo stesso *view point* ad entrambe le camere. Questo ha permesso, in fase di ricostruzione della scena 3D, che i colori aderissero in maniera, quasi, perfetta sulla *depth map* che il *kinect* genera. Dopo l'inizializzazione del *kinect* e dopo che il server ha accettato una connessione da parte di un *client*, il server incomincia ad acquisire i dati dal kinect e a spedirli sulla rete. Le immagini che genera il *kinect* hanno una grandezza pari a *614 kB* per la *depth map* e *921 kB* per l'immagine a colori. Quindi è necessario comprimere ogni singolo frame di poterlo spedire sulla rete.

Ogni frame che viene generato viene compresso utilizzando uno dei standard di compressione presentati nel capitolo precedente. Anche qui si sono adottate delle piccole accortezze. Una su tutte comprimere le immagini in memoria anziché su file. Questo perché un'applicazione che deve funzionare in *real-time* non può permettersi di perder tempo nell'accesso in lettura da file, che come è noto introduce un enorme *slowdown* all'interno di qualsiasi applicazione.

In appendice sono riportate le modifiche che sono state introdotte nel protocollo di compressione reindirizzando l'output da file in area di memoria. In figura 4.3 è riportato uno schema generale dell'applicazione lato server.

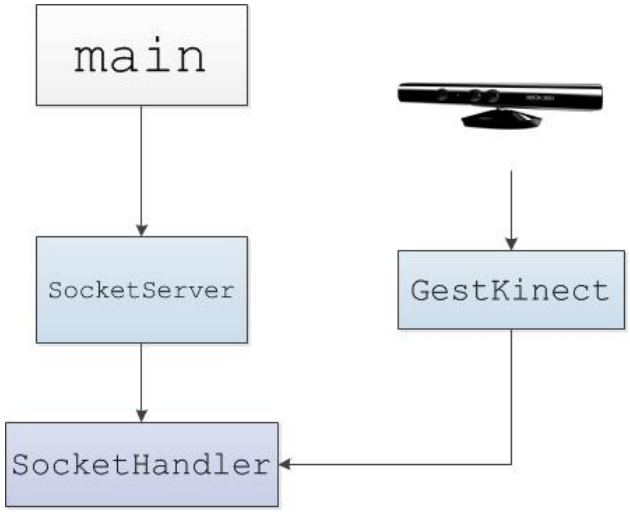

Figura 4.3: Lato server dell'applicazione

4.3 Design lato client

Come si può intuire sarà il lato *client* ad occuparsi della parte che riguarda la visualizzazione dei dati ricevuti. In questa sezione presenteremo in maniera generale

i moduli che compongono il *client* motivando anche alcune scelte implementative, e si scenderà nello specifico nella parte che si occuperà della creazione dell'immagine da visualizzare.

Esistono differenti librerie grafiche che permettono di creare delle applicazioni nell'ambito di *computer graphics*. Negli ultimi anni sono andate affermandosi per la maggiore *OpenGL* (multi-piattaforma e *free*) e le *DirectX (Direct3D)* (proprietarie *Microsoft*).

Per lo sviluppo dell'applicativo *client* la scelta è ricaduta sulle *DirectX* in quanto il driver *Nvidia* gestisce il 3D pilotando l'hardware GPU attraverso i file *HLSL* che sono utilizzati per la creazione di *shader*.

4.3.1 Framework client

Prima di partire con la spiegazione delle classi fondamentali del client, presenteremo un semplice *framework* che è alla base di questa applicazione. L'obbiettivo di questo *framework* è quello di fornire un sistema di base per la gestione delle finestre e fornire un modo semplice di espandere il codice in maniera modulare.

Il *framework* principale è composto da quattro moduli come si può vedere nella figura 4.4: il `main` che incapsula tutto il codice, la classe `system` che si occupa di gestire i due tronconi principali dell'applicazione; un modulo che si occupa di controllare gli `input` dell'utente, ed infine il modulo `graphic` che conterrà tutte le funzionalità grafiche che verranno sviluppate.

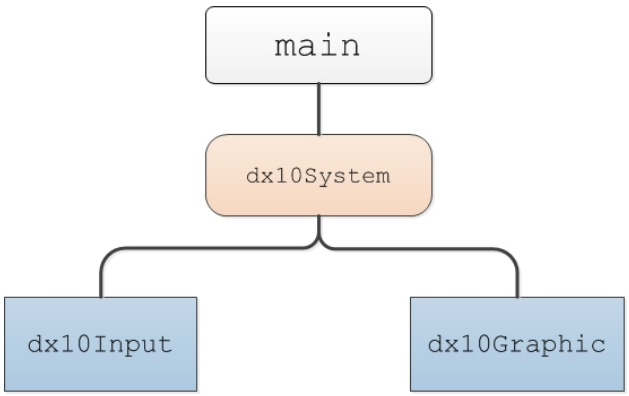

Figura 4.4: Framework principale applicazione

Come si può notare dalla figura 4.4 tutti i nomi delle classi hanno un suffisso `dx10` che indica la versione di *directX* utilizzata (*directX10*); questo perché la scheda grafica Nvidia ed i driver utilizzati permettono di utilizzare questa versione o versioni precedenti.

Il modulo `dx10Input` si occupa di aiutare la gestione dell'interazione dell'utente con l'applicazione. Per ora l'utente può navigare all'interno della scena muovendo il punto di vista, ricevendo una sensazione di muoversi all'interno della scena.

Il modulo `dx10Graphic` si occupa di tutto quello che viene visualizzato a video. Infatti in figura 4.5 sono mostrati i moduli che sono necessari per poter visualizzare a schermo i vari *frame* della videochat.

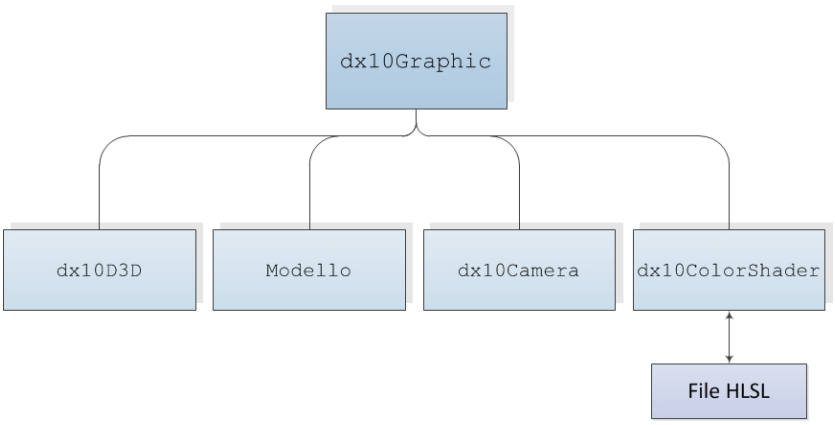

Figura 4.5: Moduli utilizzati per la visualizzazione

Molto brevemente verrà descritto di cosa si occupa ogni singolo modulo.

`dx10D3D`

Questo modulo è molto importante; in quanto all'interno è stata creata la finestra che ospiterà i frame da visualizzare. Vengono stabilite le grandezze della finestra (si ricorda che per poter visualizzare l'applicazione in 3D essa deve funzionare in *fullscreen*), il numero di frame che devono essere visualizzati al secondo (cioè la frequenza con il quale verrà effettuato il *refresh* della scheda video), anche il tipo di telecamera che verrà utilizzata per visualizzare la scena e la grandezza del *frustum*. In pratica viene creato il dispositivo 3D che ospiterà i vari *frame* della scena.

`dx10Camera`

In questo modulo viene creato il punto di vista dell'osservatore. Cioè viene posizionata una telecamera all'interno della scena. Sono stati implementati anche dei metodi che aiutano a muoversi all'interno della scena. Viene memorizzata

la posizione e la rotazione della telecamera, all'interno di una matrice che verrà utilizzata in seguito.

Modello

Questa classe è utilizzata per creare la geometria del modello 3D. Nella prossimo paragrafo spiegheremo nel dettaglio i passi che porteranno alla creazione del modello 3D.

`dx10ColorShader`

Ci sono dei concetti fondamentali che bisogna introdurre per poter capire come funziona il render 3D, utilizzando le *directX* e per poter intuire i compiti svolti da questo *frame*.

Ogni frame che viene disegnato è modellato da centinaia di migliaia triangoli. Ogni triangolo è composto da tre punti, chiamati *vertici*. Per effettuare il render del nostro modello questi vertici devono essere inseriti all'interno di *array* speciale che si chiama *vertex buffer*. Altro concetto fondamentale è: *index buffer*. Lo scopo di questo buffer è quello di memorizzare l'ordine con cui devono essere interpretati i vertici. La GPU utilizza *index buffer* per accedere velocemente ad uno specifico vertice del vertex buffer. In sostanza *index buffer* ha la stessa funzione di un indice del libro. *Vertex shader* sono dei programmi che trasformano il *vertex buffer* in modello 3D. Mentre *pixel shader* sono delle procedure utilizzate per colorare il modello creato dai *vertex shader*.

Gli *HLSL shader* sono dei programmi (*C-like*) utilizzati in *DirectX 10* per codificare i programmi *vertex e pixel shader*.

I moduli prima descritti creano delle matrici utilizzate dai file *HLSL* che servono per posizionare nello spazio 3D i vari vertici.

Quindi la classe `dx10ColorShader` si occupa di interagire con la *GPU* attraverso i file *HLSL* passando tutti i dati relativi alla scena da visualizzare

4.3.2 Creazione del modello 3D

Il client acquisisce dati del *kinect* dal server. Nella fattispecie riceve dal server le grandezze delle due immagini (*depth* e a colori), insieme ai buffer contenenti le due immagini compresse.

La prima operazione che viene effettuata una volta acquisiti i buffer, è naturalmente quella di decomprimere le immagini cioè passare dagli standard *jpeg o jpeg2000* a i dati *raw*. Questo perché sulle immagini compresse non possono essere fatte delle elaborazioni come spiegheremo in seguito.

Prima di costruire il *vertex buffer*, utile per creare la scena 3D, bisogna prestare attenzione ad una piccola operazione che si deve effettuare. Bisogna convertire le coordinate dal sistema di riferimento del kinect al sistema di riferimento dell'applicazione. Questo perché esistono due sistemi di riferimento uno del *kinect* e uno dell'applicazione; quindi serve una mappa che permetta di cambiare riferimento. Per fare questo, inizialmente, si sfruttava una funzione messa a disposizione dalle API *OpenNI*, in seguito è stata trovata una funzione che permette questa trasformazione in maniera *offline*; cioè senza l'utilizzo del kinect. Si è optato per questa scelta per due motivi: il primo è che il kinect non può essere utilizzato da due applicazioni contemporaneamente (*client e server*); la seconda è dare l'opportunità all'utente di poter utilizzare il lato *client* anche senza possedere un *kinect* (l'utilizzo di questo tipo di funzioni messe a disposizione da *OpenNI* prevedono che il *kinect* sia collegato fisicamente al computer e che si sia effettuata tutta la procedura di inizializzazione).

Fatta questa conversione delle coordinate, si è pronti alla creazione del *vertex buffer*: si è scelto di creare una struttura che contenesse sia la posizione che il colore del vertice che andrà a comporre l'immagine. Dopo aver creato questo array contente le posizioni dei vertici e i colori; teoricamente si sarebbe è pronti alla visualizzazione della scena. Però se si visualizzasse questa scena, ipotizzando di aver creato un *index buffer* che semplicemente numera i vertici, sullo schermo si vedrebbe una scena piena di puntini, che rende l'idea di quello che si sta visualizzando, ma ricordando la differenza di risoluzione tra le immagini acquisite dal kinect e quella del monitor l'effetto che si otterrebbe non permetterebbe di poter apprezzare tutti i particolari della scena.

Discretizzazione della superficie con una mesh di triangoli

Per descrivere in maniera digitale il volume di spazio occupato da un oggetto tridimensionale, è quello di rappresentare tale volume, per mezzo di un'approssimazione discreta, definita da un insieme di facce poligonali. Per descrivere tale approssimazione si ha bisogno di codificare l'insieme dei vertici su cui le facce poligonali risiedono e quindi l'insieme delle facce. L'insieme dei vertici calcolato in precedenza fornisce la così detta *geometria* dell'immagine, mentre il modo con cui le facce sono costruite a partire dai vertici fornisce l'informazione *topologica*, questa è la così detta *mesh*. Le ragioni che ci hanno portano a scegliere le *mesh* con dei poligoni *triangoli* sono varie: i triangoli permettono di rappresentare qualsiasi modello descrivibile per mezzo di poligoni (ogni poligono può essere infatti decomposto in triangoli); inoltre le primitive triangolari rendono più semplice e robusta la progettazione di algoritmi, di modellazione e *rendering*.

A partire dalla geometria creata, si deve quindi procedere con la costruzione della *mesh* di triangoli. Nella creazione della *mesh* si è tenuto conto anche della posizione nello spazio di ogni vertice. Questo perché quando il *kinect* restituisce una *depth map* inserisce degli artefatti. Dove non "vede" il kinect, il valore di profondità che calcola è nullo. Quindi se inquadra un oggetto la profondità dietro

è posta a zero. La conseguenza di creare un triangolo prendendo dei vertici con profondità nettamente differenti è un errore perché la scena visualizzata presenterà delle aree in cui la *mesh* non è ben definita. Si è aggiunto un ulteriore controllo nella creazione dei triangoli, se la profondità tra due vertici è superiore oltre una certa soglia questi non vengono presi in considerazione nella creazione dell'*index buffer*. In figura 4.6 si può notare che ci sono delle parti completamente in nero; quelli sono gli artefatti di cui si parlava prima. Sono i punti in cui il *kinect* "non riesce a vedere". Mentre in figura 4.8 si può vedere l'immagine che si ottiene associando ad ogni pixel dell'immagine a colori, un valore di profondità dopo che è stata creata la *mesh* di triangoli.

Figura 4.6: Depth map

Figura 4.7: Immagine a colori

Figura 4.8: Fusione delle immagine di profondità e immagine a colori

Ci sono altri problemi che si sono riscontrati nella creazione della scena da visualizzare. Uno è puramente tecnico e riguarda l'aggiornamento del *vertex buffer* e di *index buffer*. Visto che vengono generati trenta *frame* al secondo le configurazioni standard delle directX per questi due buffer non vanno bene in quanto perché la memoria. E allora si sono dovute cambiare le configurazioni base dei buffer descrivendo e il tipo di utilizzo che la CPU farà di questi buffer. L'altra problematica che si è dovuta affrontare riguarda la *depth map* dopo averla decompressa. Quando si comprime la *depth map* in maniera *lossy* inevitabilmente vengono introdotti degli errori; i punti dove il *kinect* "non vede" vengono posti a zero, ma quando vengono decompressi non tutti questi valori sono a zero. Questo genera dei problemi in fase di creazione della *mesh*, i triangoli confluiscono tutti in un punto indefinito della scena, e soprattutto nella visualizzazione *fullscreen* (quindi 3D) questo problema viene maggiormente accentuato. È stato imposto un controllo per ripristinare gli artefatti del *kinect*. Siccome il *kinect* *"non vede"* se gli oggetti sono troppo vicini alla camera allora si è pensato di fare un controllo della depth map decompressa imponendo che i valori sotto una certa soglia siano zero. Questo non ha risolto completamente il problema, ma lo ha ridotto di molto

Capitolo 5

Test e risultati

In questa ultima parte faremo un'analisi delle prestazioni degli algoritmi di compressione JPEG *e* JPEG2000*, confrontando diversi tipi di configurazione di qualità delle immagini. Nell'ultima parte invece varrà presentata un'analisi percettiva su un campione di persone a cui è stato presentato un video con diverse configurazioni*

5.1 Analisi delle prestazioni

In questa parte verrà effettuata un'analisi delle prestazioni di *JPEG2000* e *JPEG*.

Come è stato spiegato in precedenza per l'algoritmo di compressione *JPEG2000* permette di stabilire la grandezza dei frame che si ha intenzione di ottenere dopo la fase di compressione. Mentre l'algoritmo *JPEG* fa variare la grandezza dei frame in base alla qualità di compressione che si imposta. Quindi per tentare di comparare questi due algoritmi di compressione si è deciso di scegliere delle qualità di compressione per il *JPEG* e dopo aver fatto una media della grandezza dei frame per le varie configurazioni si sono scelte di conseguenza le configurazioni per *JPEG2000*.

Per confrontare le gli standard di compressione si è scelta come metrica il *peak signal-to-noise ratio (PSNR)*. Il *PSNR* misura il rapporto tra il valore massimo della potenza del segnale e la potenza del rumore che può corrompere il segnale. Il *PSNR* utilizzato per comparare le qualità di ricostruzione di diversi algoritmi di compressione. Il segnale, in questo caso, è quindi rappresentato dai dati originali, e il rumore è rappresentato dall'errore introdotto nella compressione.

Prima di definire come si calcola il *PSNR*, bisogna introdurre il *mean square error (MSE)*, che è una misura statistica, utile per calcolare il rumore che è introdotto dall'algoritmo di compressione. *MSE* può essere definito nella seguente

maniera:

$$MSE = \frac{1}{XRES * YRES} \sum_{i=1}^{XRES} \sum_{j=1}^{YRES} [I(i,j) - D(i,j)]^2 \qquad (5.1)$$

Con $XRES$ e $YRES$ che indicano la risoluzione dell'immagine, e I indica l'immagine originale mentre D indica l'immagine decompressa. Quindi il calcolo del *PSNR* può essere effettuato seguendo la seguente formula:

$$PSNR = 10 \log \left(\frac{MAX_I^2}{MSE} \right) \qquad (5.2)$$

dove MAX_I^2 è il valore massimo che un pixel può avere. In generale se un'immagine per ogni pixel è rappresentata da B bit allora $MAX_I = 2^B - 1$. Per le immagini a colori si hanno 8 *bit per pixel (bpp)* (il risultato del PSNR per le immagini a colori è la media di 3 PSNR calcolati su ogni componente), mentre per le *depth map* si hanno 16 bpp.

5.1.1 Configurazioni scelte

In tablea 5.1 sono riportate le configurazioni che sono state scelte per il *JPEG*. I valori che sono riportati sulla taglia dei frame per qualità dell'immagine è un valore medio su un video che è stato registrato previamente e utilizzato per fare i test.

TYPE IMAGE	QUALITY COMPRESSION	SIZE COMPRESSED DATA
RGB	15	*16KB*
	25	*21KB*
	35	*25.6KB*
	75	*47.7KB*
Depth	10	*4.3KB*
	50	*5.3KB*
	75	*7.4KB*
	90	*10KB*
	100	*25KB*

Tabella 5.1: *JPEG* grandezza dei frame di compressione

Una volta ricavate le grandezze dei frame è possibile anche procedere con la compressione *JPEG2000*

Per confrontare gli algoritmi di compressione sono stati salvati 200 frame, 100 *depth map* e 100 immagini a colori. Quindi un totale di 1800: 100 per ogni configurazione di *JPEG2000* e *JPEG*, e per ognuno di questi è stato calcolato il valore di *MSE* e *PSNR*. I risultati ottenuti per ogni tipo di configurazione sono riportati in tabella 5.2 (chiaramente sia i valori di *MSE* e *PSNR* sono dei valori medi).

Type Image	Size Compr	Type Compr	MSE	PSNR
RGB (24 bpp)	*16kB*	*JPEG*	177.0	25.9dB
		JPEG2000	106,9	27,9 dB
	21kB	*JPEG*	148.0	26.4dB
		JPEG2000	84.6	28,9 dB
	26kB	*JPEG*	134.4	26.9dB
		JPEG2000	66.2	29.3 dB
	48kB	*JPEG*	100.2	28.1dB
		JPEG2000	34.9	32.7 dB
Depth Map (16 bpp)	*4.3kB*	*JPEG*	415785.2	40.15dB
		JPEG2000	326743.4	51.3 dB
	5.8kB	*JPEG*	100597.9	46.3dB
		JPEG2000	18880.8	53.7 dB
	7.4kB	*JPEG*	55086.7	48.9dB
		JPEG2000	7932.38	57.51 dB
	10.3kB	*JPEG*	33249.34	51.12dB
		JPEG2000	3020.6	61.7 dB
	25.3kB	*JPEG*	10691.6	56.1dB
		JPEG2000	130.8	75.3 dB

Tabella 5.2: Tabella dei test effettuati

Ma ancora meglio si può notare l'andamento delle due compressioni nei grafici in figura 5.1 e 5.2

Come si può notare la compressione delle immagini a colori, a parità di grandezza del frame, con *JPEG2000* hanno un PSNR nettamente superiore rispetto alle immagini compresse con il *JPEG* . Questo significa che quando il lato client ricostruisce le immagini inviate dal server queste saranno maggiormente fedeli rispetto all'originale se ricostruite con *JPEG2000* anziché con *JPEG*.

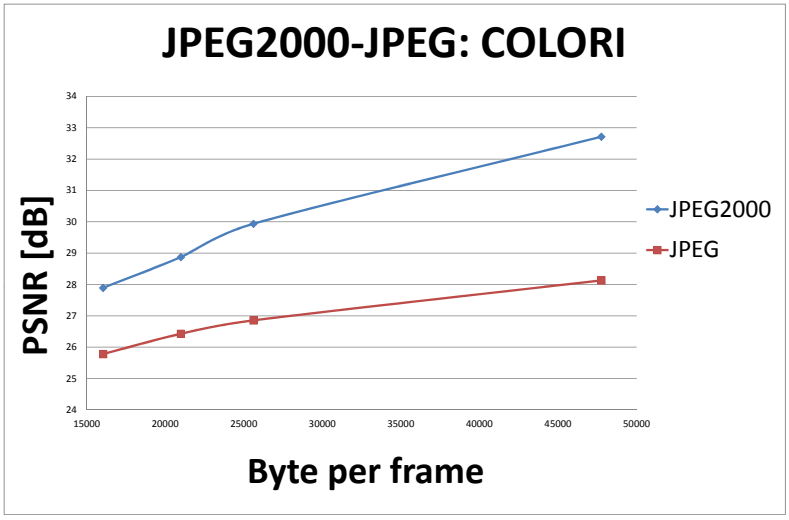

Figura 5.1: Confronto tra *JPEG2000* e *JPEG* per i colori

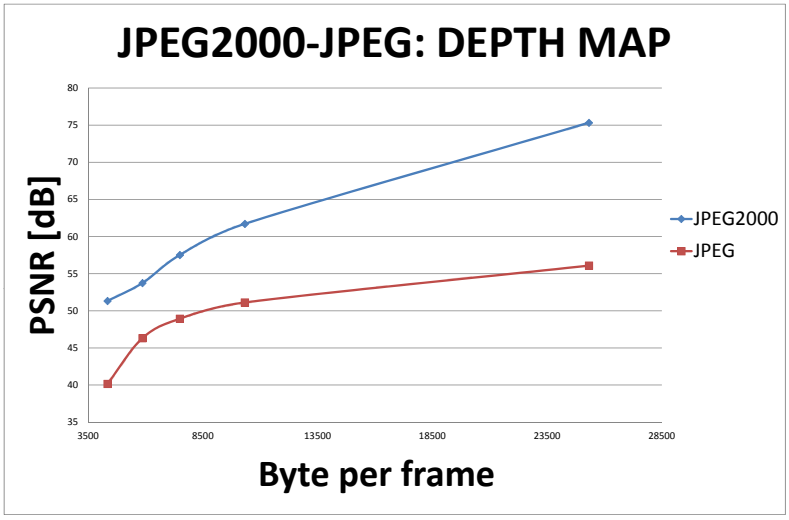

Figura 5.2: Confronto tra *JPEG2000* e *JPEG* per le depth map

Per le *depth map* il divario è maggiormente accentuato, infatti la compressione *JPEG2000* ha maggior rendimento ad alti rate di compressione (pochi byte

per frame).

5.2 Test percettivi

In questa fase si è chiesto ad un gruppo di persone di visionare un video in diverse configurazioni di compressione pescate da quelle presentate in tabella 5.2. Per evitare di appesantire l'occhio delle persone che hanno fatto i test si sono scelte 4 tipi di configurazione per algoritmo di compressione.

Prima di far visionare il video nelle varie configurazioni di compressione gli intervistati hanno visionato lo stesso nella versione originale. In questo modo gli utenti hanno avuto un metro di paragone con cui confrontare i vari sistemi di compressione. Di seguito sono riportati dei frame, nelle varie configurazioni di compressione, che hanno visionato i vari utenti.

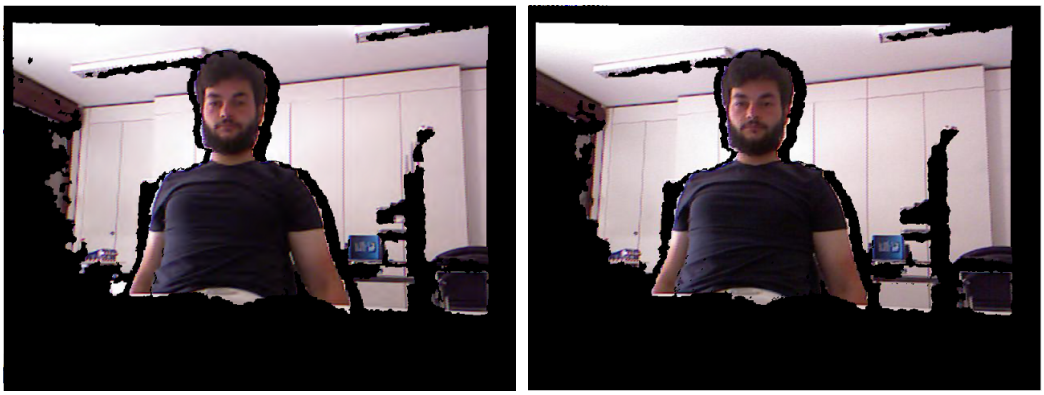

Figura 5.3: *JPEG2000* e *JPEG* alta qualità

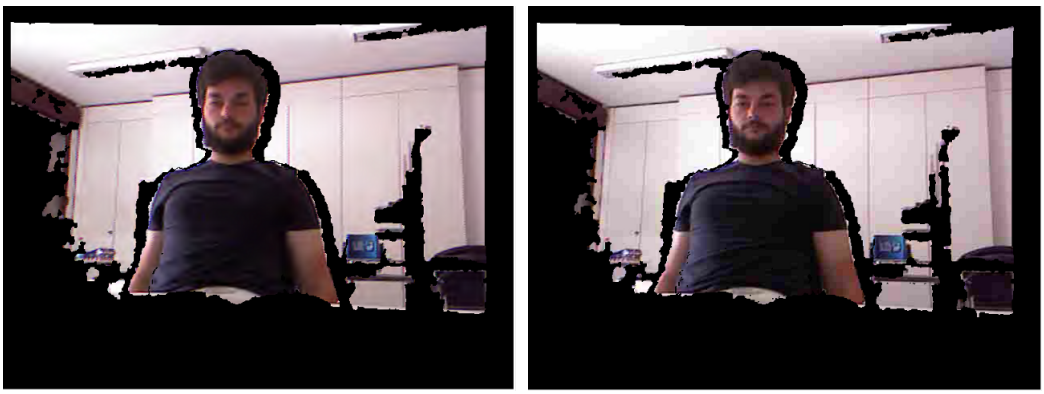

Figura 5.4: *JPEG2000* e *JPEG* colori bassa qualità, depth alta qualità

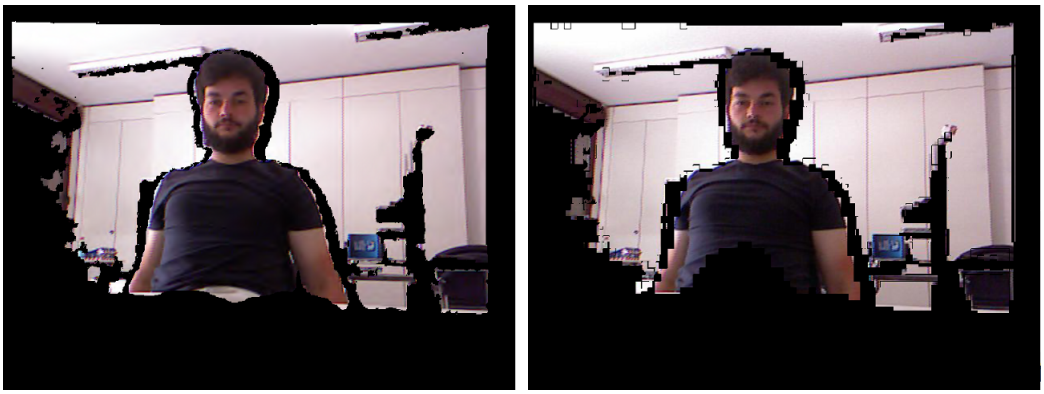

Figura 5.5: *JPEG2000* e *JPEG* colori alta qualità, depth bassa qualità

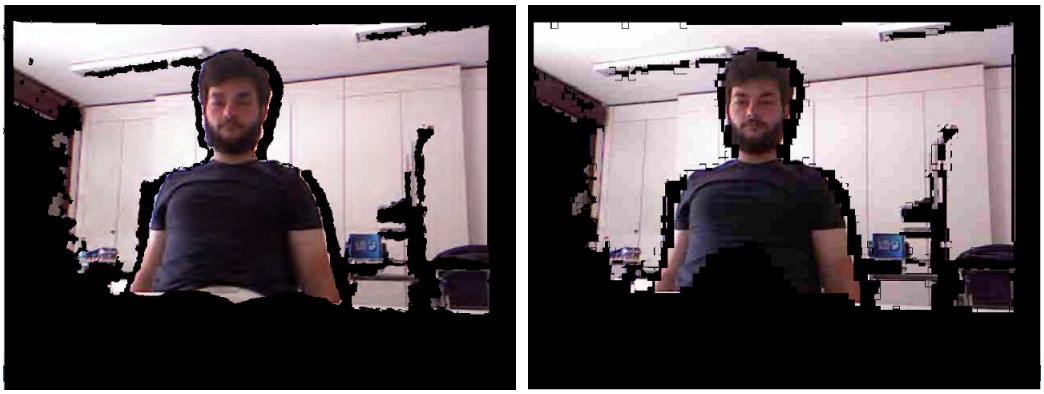

Figura 5.6: *JPEG2000* e *JPEG* colori bassa qualità, depth bassa qualità

5.3 Risultati raccolti

Dall'indagine svolta la prima cosa che si notata è quanto riportato nei diagrammi in figura 5.1 e 5.2 oltre ad avere un riscontro nei numeri ha un riscontro anche nella percezione dell'occhio umano: il *JPEG2000* risulta essere un algoritmo di compressione migliore rispetto al *JPEG*. Anche se la compressione *JPEG2000* introduceva un minimo rallentamento nel movimento delle immagini, dovuto a parametri di compressione che possono essere migliorati, queste erano maggiormente gradite quando compresse.

Quando veniva mostrato il video con *depth map* a bassa qualità quello che gli utenti notavano, pur cambiando la qualità di compressione dei colori, è che si stesse guardando una immagine piatta. La sensazione di tridimensionalità era quasi, o completamente, assente. Eccessivo rumore sui contorni delle immagini. Questo è dovuto anche al fenomeno di che si è descritto alla fine del capitolo precedente. In queste situazioni a contribuire ad una maggiore perdita delle

informazioni è il lavoro di *cropping* fatto per eliminare i pixel incoerenti. Inoltre con alti rate di compressione, percependo poco o nulla della profondità, avevano quasi una sensazione di fastidio nelle zone di confine dove il *kinect* non vede e il resto della scena. Il fenomeno era meno accentuato nel *JPEG2000* rispetto al *JPEG* che presentava la geometria dell'immagine come un vero e proprio blocco di rettangoli uno accanto all'altro. Ad ogni partecipante a questa intervista è stato chiesto di dare inoltre un giudizio da 1 a 5. Per quanto riguarda la *depth map* la media dei voti è stata 1.75 se compressa con *JPEG2000* , mentre con *JPEG* il voto è stato di: 1.5.

Diversamente quando l'intervistato osservava il video compresso con una immagine di profondità ad alta qualità la sensazione di tridimensionalità era ben percepita. L'utente riusciva a distinguere vari livelli di profondità, cioè tra il personaggio in primo piano e l'armadio sullo sfondo riuscivano la differenza di profondità tra le sedie e il carrello, oppure che le parti laterali dell'armadio fossero su due profondità differenti. In questo caso anche i colori hanno portato ulteriori informazioni. Infatti quando si mostravano le immagini con basso *rate* di compressione la sensazione è stata sempre a vantaggio del *JPEG2000* in quanto: nonostante la poca nitidezza delle immagini ma comunque continue erano preferite a le immagini compresse con *JPEG* che permettono quasi di contare i quadrati dei pixel presenti. Il lavoro di *cropping* su queste immagini è stato pressoché inesistente in quanto comunque garantivano una buona ricostruzione delle geometrie con entrambi gli algoritmi di compressione. Quando venivano mostrate anche i colori ad alta qualità un quarto degli intervistati non faceva distinzione tra *JPEG2000* e *JPEG* , mentre la parte restante si è accorta anche in questo caso che il *JPEG2000* era più vicino al video originale mostrato in principio.

Da questa intervista si può dedurre, in conclusione, che un basso rate di compressione per le depth map e la qualità dei colori relativamente bassa sarebbe un giusto compromesso per questa applicazione. Questo rende più realistica la scena che si sta visualizzano. È meglio spendere un po' di banda in più per la *depth map* altrimenti il monitor stereoscopico fa fatica a presentare una buona scena 3D, in quanto le informazioni riguardanti la geometria sono ricavate da questa.

In tabella 5.3 a supporto di quanto riportato in precedenza le medie dei voti raccolti durante le interviste.

Come si può notare dai dati riportati dalle medie: il *JPEG2000* risulta migliore di *JPEG* anche a basse qualità di compressione; sottolineare che una *depth map* di buona qualità con dei colori a bassa qualità può migliorare l'effetto visivo dell'immagine si può riscontrare anche nella valutazione dei colori.

Qualità immagine	Tipo di compressione	Media depth	Media colori
Depth alta qualità, colori alta qualità	*JPEG2000*	4.25	3.75
	JPEG	3.75	3.5
Depth alta qualità, colori bassa qualità	*JPEG2000*	3.25	2.25
	JPEG	3	2.0
Depth bassa qualità, colori bassa qualità	*JPEG2000*	1.75	1.5
	JPEG	1.5	1.5
Depth bassa qualità, colori alta qualià	*JPEG2000*	1.75	2.75
	JPEG	1.5	2.5

Tabella 5.3: Media dei voti su un campione di 20 persone

Conclusioni

Il lavoro fatto fino ad adesso ha portato allo sviluppo di entrambi gli applicativi lato client e lato server. Le modifiche apportate alle librerie di compressione hanno reso possibile il funzionamento in real-time dell'applicazione, senza scartare alcun frame. L'applicazione garantisce un buon funzionamento in 3D con il monitor stereoscopico con immagini ad alta qualità, che permettono di distinguere nitidamente vari livelli di profondità dell'immagine.

Una demo di questa applicazione è stata anche presentata anche alla *Notte Europea della ricerca* che si è tenuta il 23 settembre a Padova. La gente è stata attirata da questa demo e ha anche accolto con sorpresa un'applicazione del genere. Naturalmente è stata un'occasione per catturare nuove idee magari anche per sviluppi futuri, e testare se un prodotto del genere può interessare al mercato. Ed effettivamente sono stati molti pareri positivi e anche delle critiche che hanno portato a pensare a dei possibili sviluppi futuri. Gli spunti sono stati vari, e hanno fatto intendere che la strada tracciata sembra quella giusta. Per molte persone le zone d'occlusione del *kinect* sono un fastidio, rovinano la scena, questo era dovuto principalmente al fatto che le immagini erano soggette ad alto rate di compressione. Questo ha spinto a pensare anche che forse con un algoritmo di interpolazione potrebbe essere utile per ricostruire buona parte dei punti che il *kinect* "non vede". Un'altra cosa che stata notata è bisogna trovare il modo per far aumentare l'effetto *out of screen*. Una buona idea potrebbe essere quella di tentare di aumentare la profondità della scena. Nel complesso, l'idea che il 3D potrebbe aiutare a rendere più realistica la videochat sembra essere una buona strada da percorrere. Inoltre la possibilità di poter muovere la camera della scena ha reso l'applicazione più accattivante, in quanto permette di apprezzare maggiormente la tridimensionalità della scena fornita dal *kinect*.

Un'altra strada che si potrebbe percorre per migliorare la qualità delle immagini è quella di provare ad utilizzare un algoritmo di compressione *video*, come ad esempio MPEG-4 e H.264: che utilizzano compressione temporale o inter-frame. Questo garantirebbe maggiore qualità e quantità di informazioni.

Una ulteriore miglioria da apportare è aggiungere il segnale voce a questa applicazione, il *kinect* ha un array di microfoni che permettono di registrare

l'audio, ed *OpenNI* permette di gestire l'audio, mancano ancora però i driver da parte di *Prime Sense* che permettano di poterli catturare. Si attende, infatti, a breve un aggiornamento dei driver che consenta di muoversi in questa direzione. Da qualche mese Microsoft ha rilasciato la *SDK* per sviluppare applicazioni che sfruttino il kinect; ma essa non garantisce nessuna compatibilità con i driver *Prime Sense* e tanto meno con OpenNI. Ancora non esistono molti tutorial sul quale lavorare e prendere spunto; e la documentazione fornita da Microsoft sulla SDK lascia molto a desiderare. Per ora è fortemente sconsigliato migrare verso la SDK in quanto si chiuderebbe la possibilità di poter portare questa applicazione su sistemi operativi come *OSX* e *Linux*.

Appendice A

Scrittura in memoria JPEG e JPEG2000

In questa appendice verranno descritti di dettagli tecnici che si sono messi a punto per reindirizzare l'output degli standard di compressione JPEG e JPEG2000

Gli standard di compressione *JPEG2000* e *JPEG* prevedono che il loro output venga salvato su file. Mentre per l'applicazione, visto che deve funzionare in *real-time*, sarebbe più congeniale che il processo di compressione e decompressione avessero come output e input, rispettivamente, delle aree di memoria. Questo perché la lettura e scrittura di un file è notevolmente molto più lenta di un accesso ad una locazione di memoria RAM.

Le difficoltà principali riscontrate nel caso di *JPEG* è stato quello di riscrivere delle funzioni che agissero sulla memoria invece che sul file e reindirizzare le chiamate dello standard a queste; mentre nel caso di *JPEG2000* bisogna creare delle classi che permettano oltre a fornire l'indirizzo di memoria del buffer di disporre di metodi che possano modificare quest'area.

A.1 JPEG: redirect output

A.1.1 Compressione

Per comprimere in memoria utilizzando lo standard *JPEG* , seguendo le direttive dello standard, si è creata una struttura di memoria:

```
1 typedef struct{
2   struct jpeg_destination_mgr pub;
3   //buffer dell'immagine da comprimere
4   JOCTET* buffer;
```

```
5     //grandezza dell'immagine
6     int buffSize;
7     size_t dataSize;
8     //grandezza dell'imgine in uscita
9     int* outSize;
10    int errorCount;
11  } memory_destination_mgr;
12  typedef memory_destination_mgr* memory_dest_ptr;
```

dove buffSize indica la grandezza dei dati *raw*, mentre outSize indica la grandezza del buffer risultante. Le direttive dello standard prevedono inoltre l'implementazione dei seguenti tre metodi:

- `init_destination(j_compress_ptr cinfo)`: utilizzato per inizializzare la destinazione. Questo metodo viene invocato da `jpeg_start_compress()`, prima che qualsiasi dato venga scritto, e deve inizializzare i campi della struttura di memoria pub;
- `empty_output_buffer(j_compress_ptr cinfo)`: chiamato ogni volta che il buffer è pieno cioè lo spazio allocato per il buffer non è sufficiente. Un'applicazione dovrebbe gestire anche questa evenienza salvare il contesto, re-allocare lo spazio e ripristinare il buffer e continare a riempire la memoria con l'immagine compressa.
- `term_destination(j_compress_ptr cinfo)`: invocato dalla funzione `j_finish_compress()` per inserire nel buffer gli eventuali dati residui e liberare la memoria.

Bisogna inoltre implementare un metodo `jpeg_memory_dest (j_compress_ptr, JOCTET*, int, int*)`; che andrà a sostituire la funzione `jpeg_stdio_dest(& cinfo, outfile)` utilizzata per salvare l'output su file. In questa funzione bisognerà indicare che i metodi da utilizzare per il controllo della memoria sono quelli indicati precedentemente.

```
1   GLOBAL(void) kinectController::
2       jpeg_memory_dest(j_compress_ptr cinfo, JOCTET* buffer,
3        int buffSize, int* outSize){
4     memory_dest_ptr dest;
5
6     if(cinfo->dest == 0){
7       cinfo->dest =  (struct jpeg_destination_mgr *)
8         (*cinfo->mem->alloc_small)
9         ((j_common_ptr)cinfo, JPOOL_PERMANENT,
10        sizeof(memory_destination_mgr));
11    }
12    dest = (memory_dest_ptr) cinfo->dest;
```

```
13    // Qui specifico i metodi che JPEG deve utilizzare
14    //sono stati sovrascritti.
15    dest->pub.init_destination = init_destination;
16    dest->pub.empty_output_buffer = empty_output_buffer;
17    dest->pub.term_destination = term_destination;
18
19    dest->buffSize = buffSize;
20    dest->buffer = buffer;
21    dest->outSize = outSize;
22
23  }
```

A.1.2 Decompressione

Anche per la decompressione bisogna implementare dei metodi per poter reindirizzare l'output. Come fatto per la compressione anche in questa fase è stata creata una struttura dati di appoggio che ci aiuterà nella gestione della memoria:

```
1  typedef struct{
2    struct jpeg_source_mgr pub;
3    JOCTET* buffer;
4    int buffSize;
5    int lastUsed;
6  } memory_source_mgr;
```

I metodi da implementare sono i seguenti:

- `init_source (j_decompress_ptr dinfo)`: ha lo stesso scopo del metodo descritto in precedenza per la compressione;
- `fill_input_buffer(j_decompress_ptr dinfo)`: chiamata se `pub->byte_in_buffer = 0` mentre nel buffer sono ancora presenti dei dati.
- `skip_input_data(j_decompress_ptr dinfo, long num_bytes)`: salta un `num_bytes` di dati. Utilizzato per saltare dei dati che l'algoritmo non ritiene interessanti.
- `term_source (j_decompress_ptr dinfo)`: ha lo stesso scopo del metodo descritto in precedenza per la compressione;

Anche in questo caso bisogna sostituire la funzione `jpeg_stdio_src()` che legge l'input da decomprimere su file con:

```
1  GLOBAL(void) clientKinect::jpeg_memory_src(
2    j_decompress_ptr dinfo, unsigned char* buffer, size_t size){
3
```

```
4   memory_source_mgr* src;
5   if(dinfo->src == 0){
6     dinfo->src = (struct jpeg_source_mgr*)
7     (*dinfo->mem->alloc_small)
8     ((j_common_ptr)dinfo,JPOOL_PERMANENT,
9     sizeof(memory_source_mgr));
10  }
11  src = (memory_source_mgr *)dinfo->src;
12  src->pub.init_source = init_source;
13  src->pub.fill_input_buffer = fill_input_buffer;
14  src->pub.skip_input_data = skip_input_data;
15  /* ci sarebbe anche resync_to_start da implementare
16  * ma si e' deciso di utilizzare quello di default
17  * che lo standard mette a disposizione */
18  src->pub.resync_to_restart = jpeg_resync_to_restart;
19  src->pub.term_source = term_source;
20  src->pub.next_input_byte = src->buffer = buffer;
21  src->pub.bytes_in_buffer = src->buffSize = size;
22  src->lastUsed=0;
23 }
```

A.2 JPEG2000: redirect output

A.2.1 Compressione

Per creare la compressione in memoria anziché su file in *JPEG2000* si è creata la classe `kdu_simple_buff_target` derivata da `kdu_compressed_target`. Una volta creata questa classe si può procedere con la creazione del `codestream` e proseguire con la compressione normalmente come spiegato nello pseudocodice del capitolo 3.

I metodi accessori che aiutano la compressione nella gestione del buffer di memoria sono:

- `open(int num_bytes)`: viene creato un buffer di memoria di taglia `num_bytes`; si ricorda che nel protocollo di compressione *JPEG2000* è possibile specificare la grandezza del buffer che si vuole in output.

- `write(const kdu_byte *buf, int num_bytes)`: copia da `buf` un numero di `num_bytes` all'interno dell'area di memoria. Quindi vengono aggiornata la grandezza dell'area di memoria e anche la posizione del puntatore.

- `start_rewrite(kdu_long backtrack)`: permette di poter tornare indietro nel puntatore di memoria di `backtrack` byte. Questo è utile per

completare la scrittura de `header` del buffer. Naturalmente a questa funzione è accompagnata un'altra che permette di ristabilire il puntatore della memoria all'area destinata.

- `close()`: libera la memoria;
- `sizeMemory()`: restituisce la grandezza del buffer compresso;
- `getBuffer()`: restituisce il buffer di memoria compresso.
- `exitst() e operator!()`: utilizzati per verificare se il buffer di memoria esiste ed è stato istanziato correttamente.

Di seguito vengono riportate le istruzioni più importanti per comprimere la *depth map* in memoria utilizzando la classe `kdu_simple_buff_target`:

```
1  kdu_simple_buff_target* bufferDepth
2    = new kdu_simple_buff_target();
3  /* istanzio il buffer di memoria che conterra'
4   * la depth compressa*/
5  bufferDepth->open(XN_VGA_X_RES*XN_VGA_Y_RES*2);
6  /*creazione codestream indicando il buffer
7  di memoria invece che file su disco*/
8  codestreamDepth.create(&sizDepth,bufferDepth);
9  codestreamDepth.access_siz()->parse_string("Clayers=12");
10 codestreamDepth.access_siz()->parse_string("Creversible=no");
11 /*...*/
12 /* acquisizione depth map*/
13 outDepth = (kdu_int16*)pDepthMap;
14 /*inizio compressione*/
15 compressorDepth.start(codestreamDepth,1,size,NULL);
16 compressorDepth.push_stripe(outDepth,stripe_heights,NULL,
17   NULL,NULL,p1,p2,NULL);
18 compressorDepth.finish(1, layer_sizes_out, layer_slopes_out);
19 codestreamDepth.destroy();
```

A.2.2 Decompressione

Anche per la decompressione si è dovuta creare una classe che gestisse la memoria, e nello specifico si è creata la classe che si chiama `kdu_simple_buff_source` che deriva la classe `kdu_compressed_source`. Anche in questo caso una volta creata la classe che aiuta alla gestione della memoria, si può procedere con la decompressione creando il `codestream` specificando l'indirizzo del buffer di memoria che si deve decomprimere.

I metodi che sono stati introdotti nella gestione della memoria per la fase di decompressione sono i seguenti:

- `open(kdu_byte* ptrMemory, int size)`: viene istanziato un puntatore al buffer di memoria che contiene l'immagine compressa, e `size` indica la grandezza dell'area di memoria.
- `read(kdu_byte* buff, int num_byte)`: copia dal buffer di memoria contenente un'immagine all'interno di `buff` un numero di byte pari al secondo parametro.
- `seek()`;
- `get_pos()`: restituisce la posizione del puntatore in memoria;
- `close()`;

Per completezza riportiamo le istruzioni più importanti per la decompressione della *depth map*:

```
1  kdu_simple_buff_source* sourceDepth =
2    new kdu_simple_buff_source();
3  /*depth e' la depth map ricevuta dal server
4  e size e' la relativa taglia*/
5  sourceDepth->open(depth,size);
6  codestream.create(sourceDepth);
7  /*...*/
8  decompr.start(codestream);
9  decompr.pull_stripe(buffer,stripe_heights,NULL,
10   NULL,NULL,p1,p2);
11 decompr.finish();
12 codestream.destroy();
13 sourceDepth->close();
```

I parametri che sono stati utilizzati in *JPEG2000* per la compressione sono i seguenti:

- stripe_ heights: indica quante righe ci sono da comprimere;
- p1 indica che si tratta di numeri a 16 bit;
- p2 booleano che indica che si tratta di numeri senza segno.

Bibliografia

[1] A. Kolb, E. Barth, R. Koch, R. Larsen. Time-of-Flight Sensors in Computer Graphics. *Eurographics*, 2009.

[2] B. Buttgen, P. Seitz. Robust Optical Time-of-Flight Range Imaging Based on Smart Pixel Structure. *IEEE*, 2008.

[3] D.S. Taubman, M.W. Marcellin. *JPEG2000 Image compression fundmanetal, standards and practice*. Cambrige, United Kingdom: Kluwer Academic Publisher,2002, 2002.

[4] http://www.openni.org/. Open Natural Interaction. 2011.

[5] http://www.primesense.com/. Primse Sense Natural Interaction. 2011.

[6] Khalid Sayood. *Introduction to data compression*. Morgan Kuffman Publishers, 2006.

[7] MESA. *SR4000 Data Sheet*. Mesa Imaging, 2011.

[8] Michael W. Marcellin, Michael J. Gormish, Ali Bilgin, Martin P. Boliek. An overview of jpeg-2000. *Proc. of IEEE Data Compression Conference, pp. 523-541, 2000.*, 2001.

[9] Stefano Matoccia. *Introduzione alla Visione Stereo*. Università degli Studi di Bologna, 2008.

[10] T. Moller, H. Kraft, J. Frey, M. Albrecht, R. Lange. Robust 3D Measurement with PMD Sensor. 2005.

[11] W.B. Pennebaker and J.L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, 1993.

[12] Pietro Zanuttigh. A rate-distorsion framework for trasmission and remote visualization of 3d model. *Phd Thesis, pp. 27-29.*, 2007.

Ringraziamenti

È giunto il momento che io mi prenda un paio di pagine solo per me, lontano da tutti quei tecnicismi che hanno caratterizzato queste pagine. È un po' di tempo che penso a cosa scrivere in queste pagine e più o meno un'idea me l'ero fatta, ma ora che sono qua a scrivere ho dimenticato tutto quindi improvviserò spero di non dimenticare nessuno.

In primo luogo vorrei ringraziare la mia famiglia: i miei genitori che sono state le colonne portanti per questi anni che mi hanno fornito un appoggio incondizionato in tutti questi sette anni, mio fratello Luca che, anche se in disparte, c'è sempre stato, mio primo sostenitore al quale forse non gliel'ho mai detto ma: grazie per tutto; e grazie a mia sorella Silvia splendida e solare compagna di viaggio. Ringrazio anche il resto della mia famiglia: zii nonni e cugini che oggi avrebbero voluto essere qua, ma capisco anche le loro esigenze.

Ci tengo a ringraziare il prof. Cortellazzo, Pietro e Carlo che mi hanno accolto in laboratorio a scatola chiusa, dimostrandosi pazienti nei miei confronti e per avermi aiutato a proseguire a lavorare in una delle poche branche dell'informatica che mi piacciono.

Un ringraziamento va a coloro che ogni giorno hanno lottato con me per arrivare alla fine di questi lunghi sette anni di lacrime e sudore, il gruppo di studio: Totò per gli n fine settimana passati a studiare a discutere e a lamentarci: grazie amico mio; Mauro per essersi prestato come modello la mia tesi e le sue perle di "saggezza" che spezzavano l'aria pesante che si veniva a creare in laboratorio; Riccardo per aver condiviso con me quelle pesanti ore di GSO e Mattia per avermi spianato la strada verso la Spagna. Grazie a coloro che si sono liberati del DEI e hanno deciso, giustamente, di stare alla larga da quel posto ma che hanno condiviso con me ore di studio cicche e chiaccherate: Brus, Enrico, e Samuele.

Devo ringraziare anche le tre fanciulle che hanno reso meno traumatico il mio rientro dall'Erasmus: Marta, Elisa e Delia, non credevo che avrei trovato in voi delle persone così speciali: grazie per gli interminabili discorsi e per quelli oltre ogni limite di decenza, e soprattutto per gli aperitivi da una mezz'ora che finivano puntualmente dopo 3 ore. Grazie per avermi sopportato nei miei sfoghi su ogni minima cazzata, grazie per i mille consigli che reciprocamente ci siamo

dati e che nessuno ha mai seguito.

Ya tambìen tengo que decir gracias a quien ha hecho especial mi Erasmus. Gracias a Trevi he descubierto una nueva persona que me ha aiudado a cambiare y pensar positivo. Gracias a el he llegato a Barcelona, gracias a el he llegato a Vallcarca y gracias a el ahora he conocido personas de todo el mundo. Un gracias a mi hermanita Kuni que cada dia me alegraba con su sonrìsa, peli y ademas pasta; mis hermanos Angel, Damien y Alfonso me han dado tanto y espero que he hecho lo mismo para ellos; gracias a Audrey y Kris para la enorme cantidad de chocolate que hemos comido todos juntos y buenas compañera de viaje: Andalusia sin vosotras no era la misma. Y tambien gracias a Filo, Fra, Giovanni, Giacomo, Nicola e Antonio.

Ringrazio anche i ragazzi e ragazze della federazione poker ex Corso Milano insieme a quelli del sabato sera: grazie per le serate e risate passate assieme.

Ed infine ma non ultima grazie a Laura che superando i 1000 km che ci separano non mi ha fatto mancare il suo affetto nonostante il mio essere cazzone. Grazie anche a: Antonio, Andrea e Mirko.

Un ringraziamento, va a tutti coloro che non ci credevano che in questi anni, a partire dalla terza media, mi hanno dato per spacciato; grazie mi avete dato ancora più forza mi avete dato gli stimoli per dimostrare che sono molto di più di quello che voi pensavate.

Perdonatemi se ho dimenticato qualcuno ma sono anche le tre di notte e sono veramente cotto.

Grazie veramente di cuore a tutti.