SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTÀ DI INGEGNERIA DELL'INFORMAZIONE
TESI DI LAUREA MAGISTRALE IN INGEGNERIA DELLE COMUNICAZIONI

REAL-TIME DATA FUSION PER LA LOCALIZZAZIONE DI UN PARLATORE MEDIANTE TECNOLOGIA KINECT

Relatore
Prof. Massimo Panella

Candidato
Marco Barbato
Matricola: 802195

Anno Accademico 2011-2012

Ai miei genitori.

Indice

Elenco delle figure

Elenco delle tabelle

Listings

Acronimi

RGB	Red, Green and Blue
VGA	Video Graphics Array
IR	Infrared
QVGA	Quarter VGA
NUI	Natural User Interface
USB	Universal Serial Bus
PCM	Pulse Code Modulation
ADC	Analog-to-Digital Converter
WAVE	WAVEform Audio File Format
API	Application Programming Interface
MAEC	Multichannel Acoustic Echo Cancellation
APS	Active-Pixel Sensor
SDK	Software Development Toolkit
NIR	Near-Infrared
DMO	DirectX® Media Object
IDE	Integrated Development Environment
WASAPI	Windows Audio Session API
COM	Component Object Model
DSP	Digital Signal Processor
D2D	Direct2D
MAP	Microphone Array Processing
AEC	Acoustic Echo Cancellation

NS	Noise Suppression
AGC	Automatic Gain Control
MVDR	Minimum-Variance Distortionless Response
SNR	Signal-to-Noise Ratio
DOA	Direction of Arrival
MOCAP	MOtion CAPture
IFF	Interchange File Format
ISCTI	Istituto Superiore delle Comunicazioni e delle Tecnologie dell'Informazione
RMS	Root Mean Square
PESQ	Perceptual Evaluation of Speech Quality
SIR	Signal-to-Interference Ratio
DIET	Dipartimento di Ingegneria dell'Informazione, Elettronica e Telecomunicazioni
BSS	Blind Source Separation
ICA	Independent Component Analysis
DOA	Direction of Arrival

Introduzione

Un tema molto attuale negli ultimi anni è l'interazione tra uomo e macchina. Il progresso tecnologico mette a disposizione strumenti sempre più avanzati e potenti, contribuendo alla nascita di metodi innovativi nell'utilizzo degli stessi: uno degli aspetti più interessanti allora risiede nello sviluppo di nuove modalità di comunicazione tra le due entità. L'interfacciamento assume un ruolo cruciale in questo senso. Occorre tuttavia fare una riflessione: il miglioramento dei servizi spesso coincide con un aumento delle esigenze, così che vengano richiesti sempre più assiduamente prodotti all'avanguardia e al tempo stesso economici.

Quanto si sta per esporre ha l'obiettivo di creare una delle possibili applicazioni mediante l'utilizzo di Kinect, un dispositivo estremamente versatile messo in commercio da Microsoft come accessorio per videogiochi, ma grazie alle sue potenzialità trasformato in qualcosa di molto più esteso: una periferica collegabile a un qualsiasi personal computer, facilmente reperibile e con un costo molto accessibile.

È stato dunque necessario intraprendere un percorso che mettesse nella condizione di capire quali fossero le reali capacità offerte da Kinect nel suo complesso, e consentisse altresì di metterne in evidenza i possibili limiti. Si inserisce in questo senso il progetto che si sta per descrivere. Le intenzioni prestabilite erano quelle di realizzare un software che permettesse l'accesso a tutti i sensori, con i relativi dati, e che fosse in grado di sfruttare l'efficienza del dispositivo per operare in tempo reale, con l'obiettivo di localizzare e inseguire un parlatore all'interno di uno spazio chiuso limitato. In un situazione del genere i disturbi, oltre ai generici rumori d'ambiente, provengono anche dalle voci delle altre persone, andando a inserirsi nell'ambito del classico problema del *cocktail party*. L'aspetto innovativo del progetto consiste dunque nel cercare di offrire la più completa libertà di movimento all'interno di una

stanza, garantendo dei buoni risultati senza che le persone siano vincolate dall'indossare necessariamente un microfono. Il tutto inserito in questo nuovo contesto di interazione che è rappresentato dalle Natural User Interface.

In realtà in fase iniziale si era partiti con l'idea di impiegare le caratteristiche dei microfoni e il procedimento del beamforming per implementare un algoritmo che provvedesse all'inseguimento di un soggetto parlante. In letteratura si trovano numerose ipotesi proposte da vari autori, ma il problema è ancora in fase di studio, dal momento che la soluzione di più facile impiego, allo stato attuale, resta sempre legata all'utilizzo di un microfono il più vicino possibile alla bocca della persona.

Tuttavia, per quanto il beamforming fosse molto efficace nella localizzazione di una sorgente, non veniva fornita nessuna forma di riconoscimento della voce, per cui la tecnica funzionava ma si focalizzava solo sul segnale che veniva percepito come predominante dal punto di vista energetico. Per questo, allora, si è pensato di riempire le suddette carenze beneficiando della funzionalità del body tracking, con il fine di realizzare un sistema di *gesture recognition*.

I contesti applicativi possono essere di diverso genere: oltre all'inserimento in sale conferenza, si potrebbe pensare ad un utilizzo informativo-culturale, potendo essere impiegato in uffici pubblici, musei o grandi eventi per esempio, oppure si potrebbe dare maggior rilievo alla non tangibilità dei comandi per muoversi tra le cartelle di un computer, adoperabile in ambito medico-chirurgico all'interno di sale operatorie e altri ambienti sterili; ancora, si potrebbero suggerire situazioni più creative, come l'interazione con installazioni e opere d'arte in generale, e tutti quei contesti sociali in cui si vuole offrire una discriminazione tra le persone presenti .

Nel corso della presente trattazione saranno presentati in maniera approfondita alcuni degli esempi che vengono forniti a seguito dell'installazione dei driver ufficiali Microsoft. La scelta di un modo di procedere così dettagliato potrebbe apparire a prima vista discutibile, data la quantità di argomenti da trattare, ma ha costituito una tappa fondamentale sia per raggiungere la dovuta padronanza di controllo dei vari flussi, sia per comprendere a fondo le modalità di realizzazione di un'interfaccia grafica, sia per veicolare le diverse informazioni all'interno di un unico algoritmo. La tesi si articola come segue.

Nel primo capitolo verrà fornita una descrizione delle caratteristiche tecniche delle varie componenti del dispositivo, con particolare riguardo alla do-

tazione hardware, per poi presentare l'architettura del Software Development Toolkit e introdurre alcuni concetti di base.

Nel secondo capitolo si porrà maggior rilievo all'array di microfoni, analizzando la creazione e l'accesso al relativo flusso di dati, per poi esaminare come utilizzarlo via software; sarà inoltra fatto qualche accenno alla teoria sottostante.

Nel terzo capitolo si proseguirà il discorso dedicandosi ai sensori di immagine, soffermandosi in particolare sulle informazioni di profondità e lo *skeletal tracking*.

Nel quarto capitolo verrà infine descritta nel dettaglio l'applicazione sviluppata, ne verrà spiegato il codice e infine saranno forniti i risultati di alcuni test.

Per concludere, si discuteranno alcuni dei possibili impieghi del software e si suggeriranno possibili estensioni future.

Capitolo 1

Componenti e principi di base

Presentato pubblicamente nel giugno 2010, Kinect™ di Microsoft® viene descritto come uno strumento per l'home entertainment, orientato a un utilizzo di tipo videoludico, essendo una componente aggiuntiva per la Xbox 360. Il suo nome deriva dalla fusione delle due parole *kinetic* e *connect*, e ne riassume in modo estremamente sintetico e immediato gli aspetti costitutivi: la cinetica, cioè il movimento, e la connessione, cioè l'interazione con la console; la periferica, infatti, è un dispositivo di input sensibile al movimento, superando il concetto stesso di controller, essa consente di interagire con la *console* di gioco attraverso i movimenti del corpo e alcuni comandi vocali.

1.1 Caratteristiche generali

Kinect è un dispositivo molto avanzato e versatile. Esso si compone di una telecamera RGB con risoluzione 640×480 pixel (VGA) e di due sensori di profondità a raggi infrarossi IR – un proiettore e un ricevitore – con risoluzione 320×240 pixel (QVGA), che consentono di operare mediamente intorno ai 30 fps[1]. Per quanto riguarda la parte audio, dispone di un array di quattro microfoni a supercardioide disposti in linea. Per concludere la dotazione hardware, all'interno della base di appoggio della struttura è presente un motorino elettrico che ne consente il tilt fino a $\pm 27°$ di angolazione, in modo da permettere eventuali aggiustamenti per ottimizzare il processo di acquisizione

[1] In realtà, se si diminuisce la velocità di elaborazione dei frame, è possibile raddoppiare la risoluzione per entrambi i flussi, portandole rispettivamente a 1280×960 e 640×480.

dei dati; è presente infine un led che fornisce un feedback visivo sul corretto funzionamento del dispositivo. Tutte le caratteristiche sono riassunte nella figura 1.1.

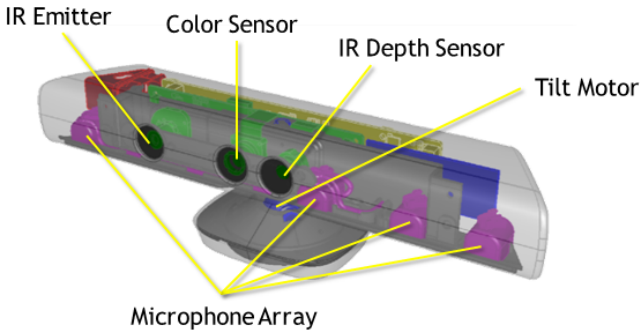

Figura 1.1: Kinect: scomposizione interna.

Tra le specifiche tecniche, va evidenziata anche la presenza di un accelerometro con una sensibilità 2g/4g/6g (dove con g si intende la ben nota accelerazione gravitazionale), configurato in modalità 2g e con un limite inferiore di accuratezza di 1°: secondo quanto dichiarato da fonti ufficiali, esso serve principalmente a determinare di volta in volta sia l'orientamento che l'inclinazione del sensore, per prevenirne posizionamenti instabili o inusuali e garantire ricostruzioni virtuali il più accurate possibile.

Ciò che rende estremamente interessante lo studio della costituzione e dei possibili impieghi di tale periferica è la filosofia con cui è stata concepita: superando il concetto stesso di controller, essa consente di interagire con la *console* di gioco attraverso i movimenti del corpo e alcuni comandi vocali. A partire da queste basi, allora, si è sviluppato il concetto di Natural User Interface (NUI), ovvero la possibilità di creare via software degli algoritmi che sfruttino il riconoscimento dei gesti, della voce e delle espressioni del volto, al fine di associargli delle specifiche istruzioni da eseguire. Tale concetto, intrinsecamente semplice e al tempo stesso versatile, ha rappresentato un importante stimolo per diverse centinaia di migliaia di individui nel mondo della ricerca, trascendendo così gli scopi videoludici originariamente stabiliti in fase di progettazione: un'idea così semplice da permettere la sperimentazione nei contesti più disparati, consentendo applicazioni praticamente di qualsiasi tipo, dalle più immediate alle più raffinate. In aggiunta, il connettore Universal

Serial Bus (USB) permette il collegamento con qualsiasi personal computer[2]. È grazie a tutti questi fattori che Kinect rappresenta l'apparato elettronico commerciale più venduto di sempre.

1.2 Specifiche Hardware Audio

Come anticipato in precedenza, la parte destinata all'acquisizione di onde sonore è delegata a un array di quattro microfoni, tutti rivolti verso il basso e con una disposizione geometrica di tipo lineare; essi sono collegati alla scheda madre attraverso un connettore a cavo unico. La disposizione prevede tre microfoni inseriti sul lato sinistro, mentre il quarto è situato sul lato destro (per una illustrazione dettagliata, si veda la figura 1.2).

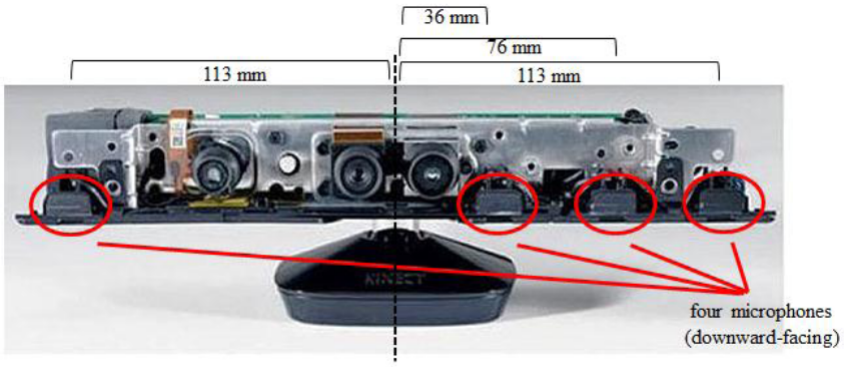

Figura 1.2: Distanze tra i quattro microfoni.

Anche il design di Kinect è stato concepito tenendo ben presenti le prestazioni di insieme. Innanzitutto, la scocca esterna è stata disegnata in modo da risultare acusticamente trasparente, a seguito di un'attenta analisi della propagazione delle forme d'onda intorno al dispositivo. Discorso analogo è stato fatto per garantire una corretta ventilazione all'interno della periferica, in modo da limitare il deterioramento dovuto a polvere e umidità e al tempo stesso mantenere invariate le proprietà desiderate. La struttura stessa appare visibilmente allungata: ciò ha consentito l'alloggiamento dei vari microfoni in modo che la distanza tra gli estremi fosse di circa 226

[2] A voler essere precisi, Kinect non può essere considerata una periferica USB a tutti gli effetti: pur adottando un connettore di tipo USB, utilizza un differente firmware (non noto), essendo stata concepita e ottimizzata per lavorare con la *console* Xbox 360®.

mm. Ciascun trasduttore è poi avvolto in una capsula di gomma, al fine di limitare eventuali interferenze dovute a rumori meccanici e vibrazioni. Anche il posizionamento dei singoli sensori non è casuale ed è il frutto di studi di ottimizzazione, incentrati sull'analisi dei microfoni a due a due, così da ottenere la più efficiente copertura in frequenza che agevolasse il lavoro svolto lato software.

Per la conversione della forma d'onda nel dominio numerico, si utilizza la modulazione di tipo mono Pulse Code Modulation (PCM), in cui il segnale viene campionato a 16 KHz e quantizzato con 24 bit per campione; tali operazioni sono effettuate da un unico Analog-to-Digital Converter (ADC) a 24 bit. Il formato di codifica è il `WAVE_FORMAT_PCM`, un tipo di WAVEform Audio File Format (WAVE) di proprietà della Microsoft, i cui valori di riferimento saranno considerati esplicitamente durante l'analisi del procedimento di acquisizione; dal momento che le informazioni provengono contemporaneamente da quattro canali, per la formazione dei vari pacchetti si effettua un *interleaving* tra i campioni relativi ai singoli canali.

Sebbene i dati vengano riconosciuti correttamente dal sistema operativo – per conferma basta vedere la sezione Audio del Pannello di Controllo, selezionare Kinect come dispositivo per la registrazione e controllarne le proprietà – vengono subito in evidenza i limiti dovuti all'utilizzo della scheda audio in dotazione alla macchina utilizzata per la sperimentazione, in quanto viene reso disponibile solo un ADC a due canali. Nel caso di impiego di Kinect come strumento puro di acquisizione, ciò si traduce nella possibilità di utilizzo dei microfoni solo per registrazioni in modalità mono o stereo. Anche se non è chiaro capire quali dei quattro canali entrino a far parte della registrazione stereo, non essendo mai stato reso noto il meccanismo di selezione, si può verosimilmente assumere che i microfoni attivi siano quelli alle estremità del dispositivo [1]. Ad ogni modo, il problema può essere relativamente arginato lato codice sorgente, in quanto l'accesso ai singoli canali è garantito attraverso la funzionalità di gestione dei flussi audio offerta dall'Application Programming Interface (API).

Un input può essere riconosciuto come tale da Kinect solo se l'angolazione di provenienza, sul piano orizzontale, è di $\pm 50°$ rispetto all'asse perpendicolare al sensore (il range complessivo è suddiviso in 11 beam prefissati ottenuti per incrementi di 10°). È prevista una procedura di cancellazione (monofonica)

del rumore di ambiente fino a 20 dB[3]. Se il suono proviene da dietro il sensore, esso subisce un'attenuazione di ulteriori 6 dB dovuta al posizionamento frontale dei microfoni [2].

Grazie all'elevato numero di bit utilizzati dall'ADC, il dispositivo consente una distinzione piuttosto accurata della dinamica della voce umana, garantendo un buon livello di differenziazione tra il parlato e l'urlato. Bisogna tener presente, però, che in presenza di più voci Kinect riconosce e traccia il suono più alto. A questo proposito, occorre sempre considerare il ruolo svolto dall'ambiente circostante: in presenza di lievi rumori provenienti dall'esterno e scarsi suoni emessi da un PC (sotto i 50 dB), la massima distanza di utilizzo garantita è di circa 3 m. Nel caso di ambienti rumorosi, come ad esempio una conversazione di sottofondo nella stanza (circa $60 - 65$ dB) l'interazione tra l'utente e il dispositivo dovrà avvenire a distanza ridotta.

1.2.1 Array di microfoni: funzionalità e vantaggi

Un singolo microfono, per sua natura, capta tutto ciò che ne mette in moto la membrana: oltre al segnale di interesse, entrano a far parte della registrazione anche il rumore dell'ambiente circostante, il riverbero causato dalla stanza più eventuali rumori dovuti alle componenti elettroniche nelle vicinanze. Nonostante le numerose tecniche di Signal Processing avanzato, per poter avere risultati soddisfacenti l'utente sarebbe comunque costretto a generare suoni a una distanza estremamente ravvicinata rispetto al microfono, dell'ordine di pochi centimetri, imponendo vincoli anche sull'auspicabile libertà di movimento del parlatore. La soluzione più semplice a tali questioni è rappresentata proprio dall'utilizzo contemporaneo di più microfoni, giacché l'array-processing è un'operazione lineare e dunque non introduce distorsioni sul segnale: se posizionati a distanze ravvicinate tra loro, infatti, i singoli sensori sono raggiunti da un'onda acustica incidente con tempi leggermente diversi. Grazie alla combinazione dei diversi segnali, è possibile, mediante la tecnica del *beamforming*[4], gestire elettronicamente l'array come se fosse

[3] Tale valore corrisponde all'incirca al livello di pressione sonora di un bisbiglio.

[4] Il beamforming è una tecnica di Signal Processing utilizzata per la trasmissione/ricezione direzionale di un segnale. Avvalendosi della presenza contemporanea di più sensori, essa consente di ricavare una mappa acustica dell'area in esame e quindi "vedere" le direzioni di provenienza di un suono, sfruttando le differenti fasi con cui l'onda arriva a ogni microfono. Gli elementi dell'array sono allora combinati in modo da ottenere interferenze

un unico microfono ad elevata direzionalità: in questo modo, una volta individuata la posizione della sorgente, si può puntare direttamente il beam di acquisizione in quella direzione, escludendo tutto ciò che proviene dal resto dell'ambiente. In aggiunta, se la sorgente si muove, è possibile inseguirla, riposizionando il beam di volta in volta. Anche la cancellazione del rumore stazionario risulta agevolata: avendo a disposizione un segnale più pulito, è possibile operare più selettivamente rispetto al caso di microfono singolo.

Un altro aspetto su cui è necessario porre l'attenzione riguarda le caratteristiche spettrali dei segnali da trattare: il parlato ha un'occupazione in banda che va all'incirca dai 200 Hz ai 7 000 Hz, il che si traduce in una estrema variabilità in termini di lunghezze d'onda corrispondenti, arrivando fino a un fattore 35 se si considerano i due estremi. Ciò comporta notevoli difficoltà nel mantenere l'apertura del beam costante per tutte le frequenze. Anche in questo caso, avere a disposizione più sensori consente di mantenere quasi costanti sia il beam che il guadagno almeno nella porzione più importante della banda (tra i 300 Hz e i 5 000 Hz) [3].

I vantaggi dal punto di vista economico non sono da meno: si pensi che l'indice di direzionalità complessiva ottenuta con il beamforming risulta maggiore persino rispetto a un microfono a ipercardioide di alta qualità, posizionandosi nell'ordine dei 10.1 dB[5] [4]. Discorso analogo si può fare riguardo al caso di inseguimento del parlatore: per ottenere un equivalente "meccanico", sarebbero necessari due microfoni altamente direzionali – uno che analizza costantemente l'ambiente di lavoro misurandone i livelli di pressione e l'altro che acquisisce il segnale nella direzione in cui il livello risulta più alto. Nella figura 1.3 si mostra visivamente l'andamento dell'indice di direttività in funzione del numero di microfoni.

Infine, per quanto riguarda la cancellazione dell'eco acustica, l'attenzione è stata posta principalmente sulla rimozione di tutti quei suoni noti – come ad esempio quelli emessi da un altoparlante – che possono interferire qualora captati dai microfoni. Il problema è che un cancellatore standard è in grado di operare sul singolo canale, risultando più adatto solo per conversazioni

costruttive per i segnali provenienti da certe direzioni e distruttive per altre, risultando in un orientamento virtuale della struttura.

[5] L'Indice di Direttività caratterizza la bontà dell'array nel ricavare la direzione di provenienza di un segnale contemporaneamente alla soppressione di altri suoni (come il rumore di sottofondo o il riverbero). Tale indice si misura in decibel, con 0 dB che indica la mancanza assoluta di direzionalità. Un microfono a cardioide ideale dovrebbe aggirarsi intorno ai 4.8 dB.

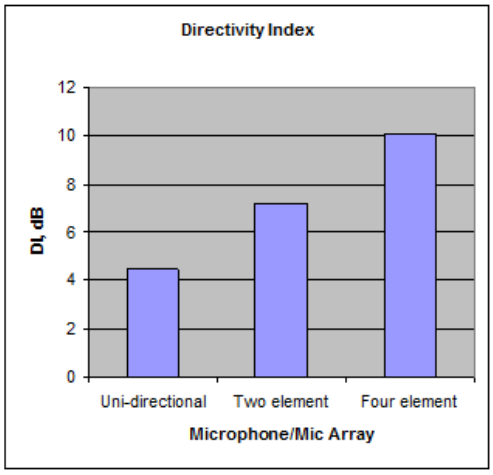

Figura 1.3: Valori della direttività in funzione del numero di microfoni.

di tipo telefonico, mentre mostra tutti i suoi limiti già in presenza di suoni stereofonici. La soluzione è stata raggiunta mediante il Multichannel Acoustic Echo Cancellation (MAEC): per sfruttare nella maniera più efficiente possibile la particolare architettura della pipeline audio, viene avviato un processo di calibrazione, che, attraverso l'emissione di impulsi specifici, stima la funzione di trasferimento tra microfoni e altoparlanti per ogni frequenza e per ogni posizione di interesse [5]. Tali funzioni vengono poi memorizzate e utilizzate come riferimento per operare un filtraggio adattativo che va a compensare sia rumori di sottofondo (prodotti da persone in movimento, porte, strada, etc.) che eventuali cambi di temperatura. In Kinect, questa architettura è implementata in modo da supportare un sistema di riproduzione di tipo surround a cinque canali e utilizzare tutti e quattro i microfoni.

1.3 Specifiche Hardware Video

Relativamente a ciò che Kinect è in grado di "vedere", l'immagine viene pre-elaborata in modo che le informazioni raccolte comprendano due categorie di dati: colore e profondità. Analizzando le caratteristiche dei sensori di profondità, lo strumento si basa sulle ricerche operate in campo *Range Imaging*

dalla società israeliana PrimeSense™ (da cui Microsoft ha acquistato la relativa licenza): prendendo spunto dal meccanismo con cui noi osserviamo il mondo circostante, lo scopo era quello di creare delle immagini bidimensionali in cui il valore di ogni pixel rappresentasse una distanza nello spazio reale. In questo modo viene offerta la possibilità di classificare la scena in base agli oggetti inquadrati: pavimenti, arredamento e persone, gettando le basi per la rilevazione e il tracciamento di movimenti e gesti[6]. Come mostrato nella figura 1.4, il sensore di profondità si compone di un proiettore IR OG12 / 0956 / D306 / JG05A e di un sensore CMOS monocromatico[7] Microsoft / X853750001 / VCA379C7130, il quale si occupa di leggere i dati relativi all'ambiente circostante, convertirli in un segnale elettrico ed effettuarne una ricostruzione sotto qualsiasi condizione di illuminazione.

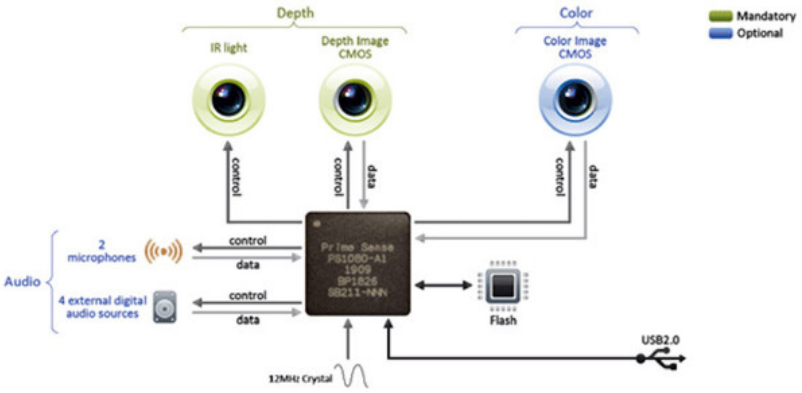

Figura 1.4: Sensori video (e audio): schema generale di riferimento (secondo PrimeSense).

A completare la dotazione video, è inserito un sensore CMOS a colori VNA38209015, simile a una webcam, con una lente piuttosto larga e dotato di autofocus.

[6] Tale tecnologia è stata poi denominata dall'azienda stessa Light Coding™.

[7] Il sensore CMOS è un particolare tipo di Active-Pixel Sensor (APS), ossia un circuito integrato costituito da un array di pixel-sensors, ciascuno contenente un fotorivelatore e un amplificatore.

Figura 1.5: I tre sensori video: proiettore, ricevitore e telecamera.

1.3.1 Ricostruzione 3D: generazione delle mappe di disparità

Attualmente, una delle tecniche più usate per la ricostruzione di ambienti 3D è il *Time of Flight*. Esso racchiude genericamente un insieme di metodi atti a misurare il tempo che una particella impiega per coprire una certa distanza attraversando un certo mezzo. Data la sua trasversalità, uno degli impieghi tipici si ha ad esempio in ambito grafico: viene utilizzato per stimare la distanza degli oggetti o della scena inquadrati, misurando il tempo che occorre a un impulso luminoso per percorrere il tragitto telecamera-oggetto-telecamera; la scena è dunque acquisita in maniera completa come avviene per una foto, ma la distanza è ricavata indipendentemente per ciascun pixel, consentendo così la ricostruzione 3D dell'oggetto o dell'intera scena scena.

La versione più semplice di un dispositivo che implementa tale metodo è ottenuta mediante impulsi luminosi: viene emesso un fascio di luce per un periodo di tempo molto breve, così da illuminare la scena e in particolare gli oggetti all'interno di essa. La lente della telecamera raccoglie la luce riflessa e riproduce il tutto sul piano del sensore: a seconda della distanza, l'impulso luminoso presenterà un ritardo sulla cui base ricavare le informazioni desiderate.

Per quanto riguarda Kinect, la situazione cambia: essa ricava informazioni sulla scena mediante la creazione della cosiddetta *Mappa di Disparità*.

Quest'ultima si ottiene in genere tramite una tecnica di ripresa di tipo stereoscopico (analogamente a quanto avviene nel sistema visivo umano), ed è rappresentata da una matrice che contiene le disparità dei singoli punti nel piano immagine relativo alla telecamera sinistra rispetto al piano destro (o viceversa).

Nel caso in esame, però, il procedimento è del tutto differente: l'emettitore IR proietta nell'ambiente un insieme di punti luminosi – per poterli percepire, basta spegnere le luci, puntare Kinect verso una superficie piana e inquadrare il tutto con una fotocamera digitale – la cui disposizione segue un ben preciso pattern di riferimento. All'interno del dispositivo è memorizzato come il pattern dovrebbe tornare al sensore se fosse proiettato su una superficie perfettamente parallela al piano della depth camera e a una distanza prefissata. Di conseguenza, per ciascun punto si può calcolare la relativa lontananza dal sensore confrontando la sua posizione effettiva (in pixel) con quella che avrebbe assunto nel pattern di riferimento (la disparità, appunto).

Per estendere il procedimento a tutta la scena e al tempo stesso garantire una scansione univoca di tutti i punti, l'analisi procede per blocchi di 64 punti contigui, individuati mediante una funzione di correlazione.

Come precedentemente esposto, prima di arrivare al prodotto commercializzabile Microsoft si è avvalsa degli studi effettuati in ambito ricostruzione 3D da PrimeSense. In realtà, un altro contributo in questo senso si era avuto, nel 2009, con l'acquisizione della società 3DV Systems Ltd., anch'essa israeliana, la quale aveva brevettato qualche anno prima la tecnologia *ZCam*, che si basava proprio sul tempo di volo per l'acquisizione in tempo reale delle informazioni di profondità[8], avvalendosi di impulsi luminosi di tipo Near-Infrared (NIR).

Ciò premesso, occorre ribadire che, essendo tutti i progetti proprietari, non sono mai stati ufficialmente rivelati i meccanismi interni che consentono la mappatura e la definizione delle distanze. Dopo un'attenta documentazione [6][7][8], tuttavia, si può ragionevolmente supporre che l'idea principale alla base dell'intero sistema consiste nell'utilizzo di un proiettore IR che irradia la superficie da ricreare con un pattern ben definito e noto alla macchina (si pensi alla proiezione di una griglia di un numero molto elevato di punti su

[8] Trattandosi di un'ibridazione rispetto a una classica telecamera RGB, la 3DV si riferiva al prodotto chiamandolo RGBD, dove la *D* indicava la presenza appunto di un canale dedicato alla distanza.

tutta la scena) la cui risposta (cioè il raggio riflesso dagli oggetti presenti) viene captata dalla telecamera nella stessa banda. Così facendo, si ottiene quello che in fisica si chiama *speckle* (letteralmente macchiolina, puntino), cioè un'immagine che rappresenta i salti di fase (casuali) che compie un'onda coerente nell'attraversare un mezzo disordinato; le onde generate per effetto della diffrazione, sovrapponendosi tra loro, creano quindi dei puntini luminosi o scuri a seconda che l'interferenza sia di tipo costruttivo o distruttivo; il pattern ha inoltre la proprietà di essere costante lungo l'asse Z. La ricostruzione tridimensionale di un oggetto avviene dunque confrontando pattern inviato e pattern ricevuto, attraverso algoritmi di computazione paralleli. A tal proposito, l'approccio può essere di differenti tipologie: si può procedere secondo una ricerca esaustiva reticolare, oppure ci si può servire delle più sofisticate metodologie di *region-growing* su base predittiva, che classificano i pixel in regioni, risultando in una segmentazione dell'immagine basata sui differenti valori di profondità [9].

L'utilizzo di tali artifici ha un duplice vantaggio: se da un lato è sufficiente una sola immagine della scena inquadrata, contrariamente a quanto accade nel caso stereoscopico, dall'altro, sfruttando la disposizione degli speckles, viene facilitata l'operazione di riconoscimento, il che consente una riduzione globale della complessità del sistema di proiezione e favorisce la ricostruzione degli oggetti in tempo reale.

1.4 Specifiche Hardware scheda madre

Per mantenere il design piuttosto snello, non potendo agire sulla larghezza in quanto vincolati dall'attenta collocazione dei microfoni, la scheda madre si compone di tre schede impilate, nominate per facilità A, B e C. La scheda A contiene:

- un ADC stereo con preamplificatore (Wolfson Microelectronics WM 8737G)
- un *N-Channel PowerTrench MOSFET* (Fairchild Semiconductor FD S8984)
- un *hub controller USB* 2.0 hub (NEC uPD720114)

- un *SAP package* 6 mm $\times$ 4,9 mm non identificato, forse un SPI flash (H1026567 XBOX1001 X851716-005 GEPP)
- un SoC per il controllo dell'interfaccia relativa alla macchina fotografica (Marvell AP102)
- una SDRAM DDR2 512 Megabit (Hynix H5PS5162FF)

Nella scheda B sono montati:

- due amplificatori operazionali d'uscita a basso costo CMOS Rail-to-Rail (Analog Devices AD8694)
- un campionatore e convertitore A/D 8 bit a 8 canali, con interfaccia I2C (TI ADS7830I)
- uno *Stepper Motor* a basso voltaggio (Allegro Microsystems A3906)
- una memoria Flash NV da 8 Mbit, 1 Mb $\times$ 8 oppure 512 Kb $\times$ 16 (ST Microelectronics M29W800DB)
- un SoC per l'elaborazione di immagini (PrimeSense PS1080-A2)

Nella scheda C sono inseriti un *audio controller* USB frontale e centrale (TI TAS1020B) e un accelerometro (Kionix MEMS KXSD9), utilizzato per stabilire l'orientamento del sensore e il relativo inclinamento.

Occorre precisare, infine, che il cavo USB che la periferica usa per collegarsi alle normali *console* è proprietario; tuttavia, nella confezione originale viene fornito un adattatore, per questioni di retrocompatibilità con le prime versioni della Xbox 360: oltre a fornire la necessaria alimentazione (servono circa 12 Watt), ne consente la connessione a un computer.

1.5 Kinect™ for Windows® Software Development Toolkit (SDK)

Anche se in principio è stato provato l'utilizzo dei driver e delle API offerte dalla combinazione OpenNI®-NITE™, appoggiando la filosofia open source e la conseguente libertà di licenza, entrambi si sono rivelati da subito inefficaci per il riconoscimento dei microfoni come periferica di acquisizione;

di conseguenza, la scelta è naturalmente ricaduta sul software ufficiale Microsoft, il Kinect™ for Windows® Software Development Toolkit (SDK) v.1.6 (rilasciato a Ottobre 2012).

L'utilizzo dell'SDK prevede alcuni requisiti di base per garantire un corretto funzionamento di tutte le operazioni di supporto fornite per lo sviluppo di applicazioni:

- Sistema operativo: Windows 7 e successivi
- Hardware: processore dual-core (almeno 2.66 GHz), 2 GB di RAM, USB 2.0 dedicata, scheda grafica che supporti DirectX® 9.0c
- Software: Microsoft Visual Studio® 2010 e successivi, .NET Framework 4, DirectX SDK (per lavorare con le mappe di profondità), Microsoft Speech Platform SDK v.11 (per lo speech recognition)

Attraverso l'installazione dell'SDK si ottengono tutti i driver per il riconoscimento di Kinect come periferica di acquisizione e la documentazione sulle API di riferimento. Tuttavia, per poter realizzare un'applicazione di qualsiasi tipo viene richiesto di utilizzare anche il Kinect Developer Toolkit; la versione utilizzata è la 1.6 (Ottobre 2012). In questo modo si ha accesso ad alcuni esempi di applicazioni di base, corredati dei relativi codici sorgente scritti in C++, C# e Visual Basic, utili per approcciarsi alla logica di funzionamento del dispositivo e per interfacciarsi con i sensori; compreso nel pacchetto viene installato anche Kinect Studio, un utile strumento per la registrazione, l'archiviazione e la riproduzione dei dati video (colore e profondità). In appendice A è riportata brevemente la procedura di installazione.

L'architettura dell' SDK si struttura attraverso la stratificazione di cinque diversi gruppi di componenti (a tal proposito, si veda la figura 1.6) [10]

1. Componenti hardware: videocamere, microfoni, motore e hub USB.
2. Driver di supporto alle seguenti funzionalità: accesso all'array di microfoni (in modalità kernel) tramite le API standard del sistema operativo, controllo dello stream di dati audio/video, enumerazione delle periferiche (qualora fossero connesse più Kinect).
3. Componenti audio e video: NUI API, un insieme di API per la gestione dei dati provenienti dai sensori e per il controllo dell'apparecchio stesso.

4. DirectX® Media Object (DMO)[9]: oggetto che estende le funzionalità dell'array di microfoni già supportate in Windows Vista®; consente di eseguire il beamforming e la localizzazione di una sorgente sonora.

5. API standard di Windows per applicazioni audio, speech e multimediali in generale.

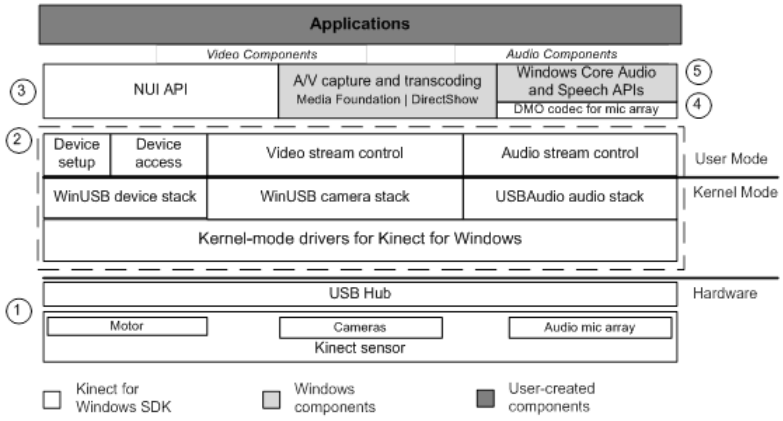

Figura 1.6: Architettura del Software Development Toolkit.

1.6 Concetti di base e principi di funzionamento

Se con Application Programming Interface (API) si intende quell'insieme di procedure, a metà tra hardware e software, necessarie per eseguire un determinato compito, tale concetto può essere allora esteso alla luce di nuove modalità di interazione, andando a definire le cosiddette NUI API. Queste ultime costituiscono il centro nevralgico di tutta la gestione di Kinect, in quanto consentono l'accesso ai sensori connessi, ai flussi di dati relativi alle immagini catturate e alle relative mappe di profondità, e pongono le basi per funzionalità più avanzate come lo skeletal tracking, lo speech recognition e il rilevamento delle espressioni del volto.

Si definisce Natural User Interface (NUI) l'interfaccia di un sistema che mette l'utente in condizioni di interagire con la macchina utilizzando un ap-

[9] Per un trattamento più completo del DMO, si rimanda al paragrafo 2.1.

proccio per l'appunto naturale, ossia consente di accedere alle funzionalità del sistema senza necessitare di dispositivi artificiali (mouse, tastiera, controller, etc.). Una NUI si basa sul riconoscimento di azioni relativamente semplici – movimenti, gesti, impiego della voce o semplice posizionamento della persona – attraverso cui controllare un applicazione del computer o manipolare contenuti sullo schermo. Tale approccio presenta l'indubbio vantaggio di essere di rapido apprendimento: sfruttando le capacità di previsione di un essere umano, risulta intuitivo e richiede minor tempo all'utente per divenire "esperto". A ciò si aggiunga l'estensione del bacino di utenza anche a individui non normodotati, essendo fruibile da persone con mobilità ridotta o portatori di handicap.

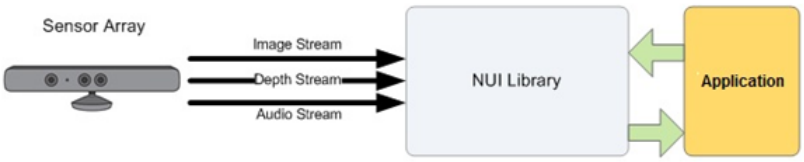

Figura 1.7: Interazione Hardware-Software con Kinect.

Uno degli aspetti più innovativi delle NUI consiste nella sua versatilità, offrendo l'opportunità di implementare sistemi all'avanguardia così come sistemi già consolidati. Tra le possibili aree di sviluppo si citano infatti:

- realtà aumentata
- ambienti intelligenti
- robotica e computational vision
- riabilitazione medica
- gesture recognition
- visualizzazione dati

Sono allora queste le basi che hanno consentito una così estesa diffusione di applicazioni sperimentali che, leggendo i dati prelevati dal set di sensori, hanno cercato di sfruttare in modo originale tutte le caratteristiche di polivalenza e raffinatezza offerte da Kinect.

Capitolo 2

Studio del comparto audio

L'array di microfoni in dotazione a Kinect consente di raggiungere un buon livello di accuratezza sia dal punto di vista della dinamica della voce che dell'adattamento alle caratteristiche dell'ambiente: a seconda della situazione, quindi, il suo utilizzo può essere applicato a diversi contesti, dall'acquisizione ad alta qualità alla localizzazione di una sorgente, dal riconoscimento di comandi vocali all'elaborazione dei dati. L'SDK fornisce due approcci per accedere al contenuto audio:

- utilizzare il DMO, che è l'elemento di base che serve principalmente per ottenere funzionalità avanzate come il beamforming o la cancellazione dell'eco
- servirsi dell'interfaccia WASAPI, che consente un'analisi più diretta e dettagliata dei dati

In generale, tutte le informazioni acquisite sono rese disponibili sotto forma di *stream* o flussi. Nel corso del presente capitolo, verranno dapprima affrontate le tematiche legate agli esempi e ne verrà data un'attenta spiegazione, per poi passare a una trattazione più teorica dei concetti sottostanti.

2.1 Analisi di un primo esempio: Audio Capture Raw

Per poter utilizzare il sensore come puro strumento di acquisizione, in fase iniziale si è scelto di dedicarsi alla semplice cattura di dati *raw*[1] o grezzi, focalizzandosi principalmente sui meccanismi di immagazzinamento e accesso ai dati. Il Developer Toolkit fornisce un esempio chiamato *Audio Capture Raw*, un'applicazione di tipo *console* che provvede alla scrittura e all'archiviazione di quanto acquisito dai singoli microfoni mediante l'interfaccia WASAPI; l'unica forma di interazione con l'utente è legata alla scelta dell'interruzione della registrazione. Dal momento che tale processo è abbastanza complesso e prevede l'intervento di diverse entità, occorre fare chiarezza su alcuni concetti di base.

Windows Audio Session API (WASAPI) Rappresenta la risorsa principale per l'acquisizione di un flusso audio: abilita le applicazioni clienti a creare e gestire tutti i flussi audio scambiati tra l'applicazione stessa e un *audio endpoint device* – astrazione che rappresenta uno qualsiasi degli apparati fisici che si trovano a una delle due estremità del percorso, potendo rappresentare dunque sia degli input (microfoni) che degli output (speaker), in modo da renderli manipolabili direttamente.

Essa è costituita a sua volta da diverse interfacce, che vengono implementate a seconda delle diverse necessità; tra queste, emergono la `IAudioClient`, che abilita un cliente a creare e inizializzare un flusso audio, e la `IAudioCaptureClient`, che permette a un cliente di leggere i dati presenti nel buffer di acquisizione di un endpoint. Per poterla definire correttamente, basta includere nel codice sorgente i file di header `Audioclient.h` e `Audiopolicy.h`.

La WASAPI fa parte delle cosiddette "Core Audio-APIs", interfacce (di basso livello) di accesso alle periferiche audio, sviluppate originariamente per il sistema operativo Windows Vista™ con lo scopo di creare diverse modalità di utilizzo, in particolare il cosiddetto *user-mode*: si consente all'applicazione di accedere alla periferica audio direttamente attraverso l'audio-API, senza dover necessariamente passare per il kernel o l'hardware vero e proprio, evitando così anche eventuali problemi di latenza. L'*audio engine* è la componente

[1] In realtà il vocabolo *raw*, oltre ad essere una parola di senso compiuto nella lingua inglese, viene talvolta considerato come l'acronimo che sta per *Read And Write*, a sottolineare l'essenzialità dei dati trattati.

attraverso cui le applicazioni condividono l'accesso a un endpoint (modalità *shared*). A seconda del tipo di funzione che si vuole svolgere, esso si occupa di trasportare i dati tra il buffer di un endpoint e l'endpoint stesso: nel caso di riproduzione, leggerà i dati scritti nel rendering buffer; nel caso di registrazione, leggerà i dati dal capture buffer. Il diagramma contenuto nella figura 2.1 mostra il percorso compiuto nel caso di riproduzione dei dati, mettendo in rilievo le Core Audio-API e le loro relazioni con le altri componenti in Windows Vista.

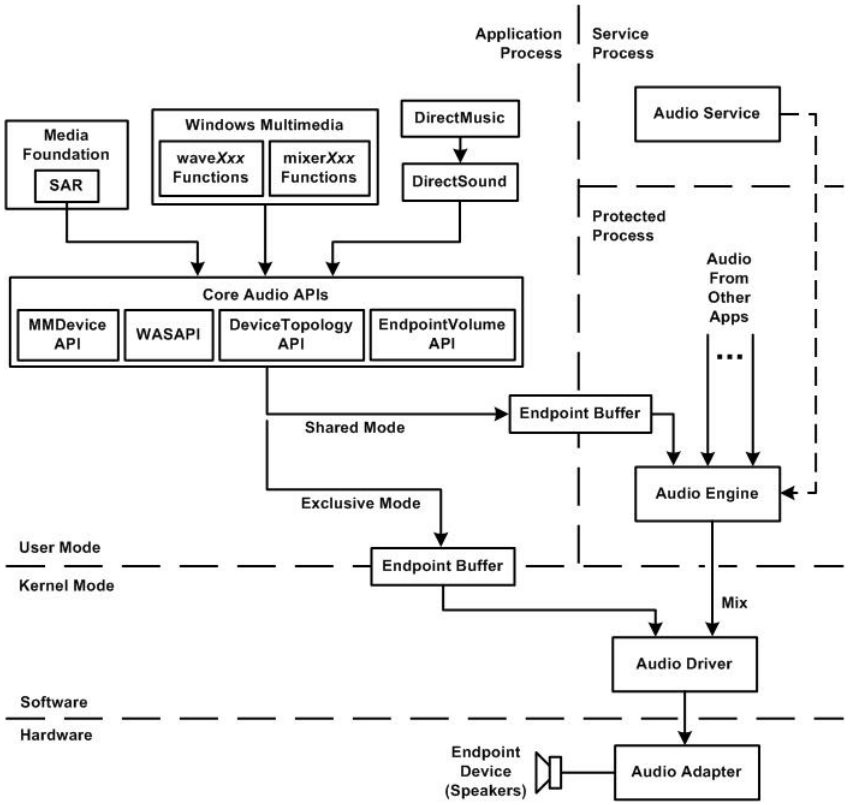

Figura 2.1: Le Core Audio API e la loro relazione con le altre componenti audio (anche se sono mostrate entrambe le modalità *shared* ed *exclusive*, un flusso può essere di un solo tipo per volta).

Sessione Audio È un gruppo di più flussi audio correlati tra loro; attraverso tale astrazione, un cliente WASAPI identifica uno stream audio come membro di una specifica sessione, e permette al sistema di gestire collettivamente tutto il gruppo come se fosse una unità singola. Affinché

ciò avvenga correttamente, il flusso va inizializzato mediante la funzione `IAudioClient::Initialize`, la quale provvede anche al suo assegnamento a una sessione unica. Lo stream, e con esso anche il relativo endpoint, rimangono appartenenti a quella sessione per tutta la durata dello scambio, finché non viene rilasciato cioè l'ultimo riferimento all'*object stream*. Tipicamente, un'applicazione assegna per default tutti i flussi alla stessa sessione, ma è comunque prevista la possibilità di assegnazione diversificata.

I flussi possono essere solo di due generi, di riproduzione o di registrazione. Ciò comporta che ciascuna sessione conterrà solo una delle due tipologie.

Una sessione, infine, è caratterizzata da uno stato che ne descrive la situazione corrente:

- attivo: la sessione contiene uno o più flussi nello stato di running
- inattivo: l'ultimo dei flussi attivi della sessione diviene stopped
- scaduto: la sessione è rimasta nello stato inattivo per un certo lasso di tempo
- terminato: la sessione non presenta più flussi (è stato cancellato l'ultimo dei flussi appartenenti alla sessione)

Component Object Model (COM) È un sistema object-oriented che consente la creazione di componenti software in grado di comunicare tra loro. Non è un linguaggio, bensì uno standard che specifica le caratteristiche dell'oggetto COM stesso e i requisiti di programmazione per abilitare gli oggetti a interagire con altri oggetti analoghi, che possono anche appartenere a processi differenti o addirittura essere scritti in altri linguaggi (in quanto tale standard viene applicato solo dopo che il programma è stato convertito in codice binario). Un oggetto COM, come unico vincolo, deve puntare alla *COM-Interface*: in questo modo, ogni cliente ha a sua disposizione un un gruppo predefinito di metodi che consentono di accedere ai dati di altri oggetti e quindi manipolarli. Per essere utilizzate, tali funzioni sono implementate da una specifica classe COM, ma non rappresentano necessariamente le uniche che la classe può supportare (a differenza di quanto avviene con una normale interfaccia), dal momento che si limitano solo a garantire certi principi basilari per l'interazione. Per poter usufruire di tutti i servizi offerti dalla libreria

COM, quest'ultima deve essere inizializzata in tutti i thread[2] in esecuzione tramite l'invocazione della funzione `CoInitialize` (oppure `CoInitializeEx`), mentre le risorse vengono liberate con `CoUninitialize`.

KinectAudio DirectX® Media Object (DMO) Rappresenta un oggetto COM standard, concepito come estensione del predecessore Windows Microphone Array DMO, di cui ha mantenuto la modalità di lavoro e l'insieme di interfacce supportate. Prevede un suo identificatore di classe *CLSID*[3], supporta l'array di microfoni di Kinect e include le funzionalità di beamforming e di localizzazione di una sorgente. In sostanza, il DMO prende i dati contenuti negli m flussi che gli vengono passati in ingresso e ritorna n flussi contenenti dei dati trasformati; per ciascuno stream, è impostato un *media type* che definisce come interpretare le relative informazioni. Dal momento che in questa sezione non interessano tali funzionalità avanzate, le modalità di utilizzo verranno approfondite nel paragrafo 2.2.3.

L'applicazione *Audio Capture Raw*, come già detto, si pone l'obiettivo di acquisire i dati tramite i microfoni, salvarli in un file di tipo WAVE e accedere ai singoli dati raw [11]. Per ottenere ciò, essa risulta composta di quattro file principali:

- `AudioCaptureRaw.cpp`, che contiene il *main()* dell'applicazione e quindi gestisce l'esecuzione generale del programma
- `WASAPICapture.cpp`, che definisce e implementa la (unica) classe `CWASAPICapture`, la quale si occupa di tutti i dettagli relativi all'acquisizione dei dati audio
- `ResamplerUtil.cpp`, che ricampiona i dati e li converte in formato PCM, in modo che possano essere codificati e archiviati come WAVE
- `stdafx.h`, che raccoglie tutte le inclusioni standard del progetto e un template per il rilascio delle interfacce

[2] Nell'ambito della programmazione, con thread si intende una suddivisione del processo principale in più parti, in modo da poter essere eseguite in parallelo, condividendo solo alcune risorse e risultando in un più efficiente utilizzo della CPU.

[3] Un CLSID è un codice a 128 bit che viene utilizzato da Windows per riconoscere come gestire un file.

La procedura si realizza attraverso una chiamata in cascata di diverse funzioni, le quali svolgono ognuna un compito predefinito in accordo con la struttura precedentemente esposta. Volendo entrare nel dettaglio, l'applicazione si suddivide logicamente in tre sezioni:

- attivazione della periferica
- preparazione delle risorse necessarie alla registrazione
- acquisizione vera e propria, scrittura e archiviazione

2.1.1 Attivazione della periferica

Il primo passo consiste nello stabilire una connessione tra il computer e il sensore. Stante la possibile presenza anche di altri sensori (diversi da quello in esame), si procede nel seguente modo:

1. `CoInitializeEx()`: inizializza la libreria necessaria per creare un oggetto COM con gestione delle chiamate in modalità *multithreaded*: i metodi creati dal thread possono essere eseguiti in qualsiasi altro thread.
2. `CreateFirstConnected()`: attraverso una successione standard di chiamate, conteggia tutti i sensori connessi al computer, li controlla e inizializza il primo che risulta disponibile.
3. `GetMatchingAudioDevice()`: seleziona, tra tutti gli apparati audio attivi, quella corrispondente a Kinect; a questo scopo, il metodo invoca a sua volta la funzione `IsMatchingAudioDevice()`, che va a controllare proprio l'ID associato al sensore.
4. `GetWaveFileName()`: ricava il nome del file destinato a contenere i dati; l'oggetto viene memorizzato per default nella forma "KinectAudio_HH_MM_SS.wav", cioè identificando ogni registrazione con l'ora, i minuti e i secondi in cui viene lanciata l'applicazione.

2.1.2 Preparazione delle risorse necessarie alla registrazione

Lo step successivo si occupa della predisposizione della connessione audio per l'acquisizione di quanto verrà captato dallo strumento. Anche in questo caso, la cascata di funzioni si articola come segue:

1. viene istanziata la classe `CWASAPICapture`.

2. `Initialize()`: inizializza il registratore. Tale operazione si ottiene creando da subito un evento di arresto – che ha lo scopo di inviare un segnale nel momento in cui l'utente, tramite *console*, vuole interrompere la registrazione – e attivando un oggetto `IAudioClient` relativo all'endpoint trovato. Attraverso tale interfaccia, il cliente può creare e inizializzare un flusso audio esistente tra l'applicazione e l'endpoint. A questo punto viene chiamata la funzione `LoadFormat()`, che si occupa di recuperare il formato interno del flusso utilizzato dall'audio engine per il trasporto dei dati tra l'endpoint e il buffer. Vengono poi invocato il metodo `InitializeAudioEngine()`, che inizializza la WASAPI e ottiene un riferimento all'interfaccia `IAudioCaptureClient`, consentendo la lettura dei dati dal capture-endpoint buffer. Siccome il formato dell'output deve essere di tipo PCM, il sistema si crea dei buffer interni da mettere in ingresso e in uscita da un resampler per trasformare i dati dal formato interno a quello supportato dall'endpoint.

2.1.3 Acquisizione, scrittura e archiviazione

Dopo aver inizializzato correttamente l'oggetto prefissato, arriva il momento di chiamare il metodo che gestisce l'acquisizione vera e propria: `CaptureAudio()`, che registra i dati audio grezzi e scrive un header (provvisorio, in quanto l'effettiva grandezza in byte non può essere nota a priori) per il file WAVE; vengono inoltre mandati su schermo alcuni messaggi con alcune brevi istruzioni da seguire e la descrizione di quello che avviene. Per ottenere ciò, sono invocate le seguenti funzioni:

1. `Start()`: crea il cosiddetto *worker thread* con cui gestire il processo di acquisizione. Dato che si tratta di un thread secondario, è necessario inizializzare un nuovo oggetto COM specifico; la registrazione avviene mediante l'utilizzo di un loop la cui logica può essere sintetizzata come segue: viene letto il primo pacchetto nel buffer, lo si ricampiona per modificarne il formato, lo si scrive su file e si rilasciano le risorse precedentemente occupate. Questa procedura continua frame dopo frame finché la periferica continua ad avere dati disponibili; nel momento in cui viene segnalato al thread primario l'evento di arresto, l'acquisizione

viene interrotta. Siccome il buffer è tarato sull'intera latenza, il loop di acquisizione aspetta la metà del tempo, in modo da non dover scartare campioni, così da evitare problemi connessi all'overload (l'acquisizione avviene in modalità *timer-driven*).

2. `IAudioClient::Start()`: inizia lo scambio di dati tra il buffer dell'endpoint e l'audio engine. Contestualmente viene fatto partire un audio-clock che inizia il conteggio a partire dalla posizione corrente.

3. `Stop()`: ferma lo streaming tra il buffer e l'audio engine attraverso l'`IAudioClient::Stop()`, congelando il clock alla posizione corrente. Tale metodo viene invocato con la semplice pressione del tasto "s" sulla tastiera.

4. `WriteWaveHeader()`, che sistema l'header del file `.wav` in base al conteggio dei byte effettivamente acquisiti.

5. `CoUninitialize()`: libera tutte le risorse allocate in fase iniziale per l'utilizzo dell'oggetto COM.

2.1.4 Accesso ai dati

Scopo dell'analisi sinora svolta sulla logica di funzionamento del processo di acquisizione era quello di poter accedere liberamente ai singoli canali relativi a ciascuno dei quattro microfoni. I dati sono immagazzinati nel formato `WAVE_FORMAT_PCM`, il quale è caratterizzato da una struttura della forma d'onda del tipo `WAVEFORMATEX`; tale struttura predefinita è costituita dai seguenti membri:

- `wFormatTag`: etichetta che indica il tipo di formato; i valori possibili, in questo caso, possono rimandare solo al `WAVE_FORMAT_PCM` oppure al `WAVE_FORMAT_IEEE_FLOAT`[4];

- `nChannels`: specifica il numero di canali; il valore è impostato a 4, in quanto corrisponde al numero di microfoni attivi durante la registrazione;

[4] La differenza tra i due formati risiede solo nella modalità di rappresentazione dei singoli dati, rispettivamente interi o in virgola mobile.

- `nSamplesPerSec`: è la frequenza di campionamento, impostata a 16 KHz. Ciò comporta che la distanza temporale tra un campione e l'altro è data dalla seguente espressione:

$$Intervallo = \frac{1}{16\,000\ Hz} = 62,5\ \mu s; \tag{2.1}$$

- `wBitsPerSample`: rappresenta la quantizzazione, espressa in bit per campione; se il formato è `WAVE_FORMAT_PCM`, va impostato a 8 o 16, mentre nel caso di `WAVE_FORMAT_IEEE_FLOAT` i bit sono 32;

- `nBlockAlign`: è la dimensione in byte dell'unità minima contenente dati, che corrisponde dunque a un singolo *audio frame*:

$$\begin{aligned} nBlockAlign &= \frac{4\ canali \cdot 32\ b/camp}{8\ b} \\ &= 16\ B/camp; \end{aligned} \tag{2.2}$$

  tale membro serve in fase di riproduzione/registrazione per evitare al sistema di partire da un punto non allineato con il blocco: in sostanza, si agisce per multipli di `nBlockAlign`, in modo da cominciare sempre dall'inizio dei dati;

- `nAvgBytesPerSec`: è la velocità di trasferimento dei dati in byte/sec; è utile per stimare la grandezza del buffer:

$$\begin{aligned} nAvgBytesPerSec &= 16\,000\ camp/s \cdot 16\ B/camp \\ &= 256\,000\ B/s = 256\ KB/s; \end{aligned} \tag{2.3}$$

- `cbSize`: indica la dimensione in byte di alcune informazioni extra, inserite alla fine della struttura, che esulano dagli scopi di questa analisi. Per questo motivo, tale valore è posto a 0.

Per conferma, tali membri sono stati resi *public*, in modo da poterne mostrare esplicitamente sulla *console*, durante l'esecuzione del programma, i relativi valori.

Oltre a questi attributi, altre quantità di interesse per il raggiungimento dell'obiettivo iniziale sono:

- `cTargetLatency`: costituisce l'intervallo di tempo accettabile tra suono prodotto e registrazione effettiva; viene utilizzato anche per scandire i ritmi di acquisizione interni dei diversi frame. È impostato a 20 ms[5];
- `_InputBufferSize` / `_OutputBufferSize`: indicano la grandezza dei buffer di input e output:

$$\begin{aligned} \_I/OBufferSize &= 20 \cdot 10^{-3}\ s \cdot 256 \cdot 10^{3}\ B/s \\ &= 5120\ Byte. \end{aligned} \tag{2.4}$$

Considerato che la cattura avviene mediante il metodo `CaptureAudio`, è proprio al suo interno che è stato inserito un ciclo *for* per scandagliare singolarmente i quattro canali, immagazzinati globalmente nell'oggetto `capturer`; quest'ultimo è un'istanza della classe `CWasapiCapture`, e i campioni relativi a ciascun canale, memorizzati nella variabile `_OutputSample`, possono essere esplicitati nel seguente modo: dal momento che ogni pacchetto è composto da 16 byte, si avranno complessivamente 4 byte per ogni canale. Ricordando che le informazioni non sono codificate consecutivamente, bensì sono alternate tramite *interleaving*, per ogni gruppo di 4 byte ci sarà solo 1 byte per canale. Si riporta di seguito un esempio di codice per ottenere le informazioni cercate:

```
1  for (size_t i=0;i<(capturer->_OutputBufferSize)/16;i+=1)
   {
3
   float *pFloatSample_1=((float*)capturer->_OutputSample) + i*4 + c1;
5  float *pFloatSample_2=((float*)capturer->_OutputSample) + i*4 + c2;
   float *pFloatSample_3=((float*)capturer->_OutputSample) + i*4 + c3;
7  float *pFloatSample_4=((float*)capturer->_OutputSample) + i*4 + c4;
   printf_s("Valore campione(ch_%d) %d: %f, Valore campione(ch_%d) %d:
9                %f \n",c1+1,i,*pFloatSample_1,c2+1,i,*pFloatSample_2);
   printf_s("Valore campione(ch_%d) %d: %f, Valore campione(ch_%d) %d:
11               %f \n",c3+1,i,*pFloatSample_3,c4+1,i,*pFloatSample_4);

13 }
```

Codice 2.1: Istruzioni per ottenere i dati relativi ai quattro canali.

[5] La scelta di tale valore non è casuale, e sarà motivato nel paragrafo 2.3.

dove con `c1`, `c2`, `c3` e `c4` si sono indicate delle semplici costanti per la selezione del canale. Naturalmente, `_OutputSample` viene sovrascritta a ogni iterazione, dunque i dati visualizzati saranno relativi all'ultimo blocco registrato; quello che interessava in questa prima fase era semplicemente trovare un modo per accedere – dall'interno dell'algoritmo – alle informazioni effettivamente registrate. Cercando una visione più d'insieme, si potrebbero recuperare i dati direttamente dall'interno del thread di acquisizione attraverso la variabile `pData`, che punta ai valori effettivamente scritti nel buffer. Tuttavia, un metodo molto più rapido è ad esempio la lettura del file con MATLAB®: per ottenere tutti i campioni, è sufficiente memorizzarli in una variabile mediante la funzione `wavread`.

2.2 Secondo esempio: Audio Basics

Il successivo esempio preso in considerazione, chiamato *Audio Basics*, pone le basi per l'utilizzo di funzionalità avanzate. La prima grande differenza che si nota rispetto al caso precedente è l'output che viene prodotto: se prima si visualizzava una semplice *console*, la situazione cambia radicalmente, in quanto si ha a che fare con una finestra vera e propria. Questo dettaglio a prima vista di poco conto, è in realtà molto rilevante perché comporta un approccio di base totalmente differente e, al tempo stesso, una diversa finalità di utilizzo. Per un trattamento più approfondito delle modalità di utilizzo dell'interfaccia grafica, si rimanda al capitolo 4.

L'esempio sostanzialmente rivela cosa si può arrivare a fare con un array di microfoni, mostrando in maniera molto semplice e intuitiva come localizzare una sorgente sonora e, al tempo stesso, avere un riscontro grafico dell'intensità della forma d'onda che si sta acquisendo.

Come anticipato nel precedente capitolo, per poter offrire queste operazioni la tecnica sottostante è il beamforming: il sensore è strutturato in modo da supportare 11 beam prefissati, ottenuti suddividendo la regione del piano orizzontale compresa tra i $-50°$ e i $50°$ (rispetto all'asse di simmetria) in porzioni da $10°$; è prevista la possibilità di selezione del beam sia automatica che manuale. È stato già detto che l'elemento chiave offerto a questo scopo è il DMO: è proprio tramite di esso che viene consentito di accedere al cosiddetto *voice-capture Digital Signal Processor (DSP)*, un oggetto nato con Windows

Vista che incapsula diverse procedure di elaborazione relative all'acquisizione della voce, in particolare in presenza di più trasduttori [12].

Audio Basics si struttura come segue:

- `AudioBascis.cpp`, che contiene il *main()* e tutti i metodi relativi sia alla gestione della finestra che all'acquisizione dei dati; il relativo file di header contiene la definizione delle classi `CStaticMediaBuffer` e `CAudioBasics`
- `AudioPanel.cpp`, che implementa i metodi definiti nella classe `AudioPanel` e consente di "disegnare" i dati audio
- `AudioBasics.rc`, `Resource.h` e `Kinect.ico`, che costituiscono rispettivamente il file di risorse della finestra – con la conseguente definizione delle costanti di controllo – e l'icona associata al file eseguibile prodotto mediante l'applicazione
- `stdafx.h`, che comprende tutte le inclusioni standard del progetto e un template per il rilascio delle interfacce

Se *Audio Capture Raw* appariva piuttosto macchinosa dal punto di vista della gestione delle funzioni da richiamare, sotto questo punto di vista il secondo esempio è più lineare, anche se la gestione della parte grafica introduce una serie di argomenti che non sono ancora stati trattati. Per affrontare con ordine la logica di progettazione dell'algoritmo, si suddivide l'applicazione in aree tematiche.

2.2.1 Gestione della finestra

La prima cosa che occorre considerare, dopo aver istanziato la classe `CAudioBasics` e aver inizializzato tutte le variabili, è la creazione della finestra di dialogo o *dialog box*. A partire dalla dichiarazione del *main()*, dunque, i vari metodi si susseguono secondo l'ordine:

1. `Run()`: si occupa di definire la finestra principale – impostandone tutte le varie opzioni di visualizzazione – registrarla, crearla e mostrarla su schermo, seguendo il modello di default offerto dalla struttura apposita `WNDCLASS`; al tempo stesso, la gestione della finestra avviene attraverso uno scambio di messaggi (di tipo `MSG`) tra l'applicazione e il sistema

operativo: di conseguenza, il metodo prevede un loop di processamento di tali messaggi. Ciò è reso possibile attraverso un'apposita *procedura*, che svolge la funzione di controllare l'elaborazione di quanto passato alla finestra.

2. `MessageRouter()`: costituisce la suddetta *procedura*: gestisce i messaggi da mandare alla finestra, recuperando le informazioni necessarie e impostando i dati utente associati. In questo contesto si inserisce anche il metodo `DlgProc()`, che per questioni di logica di funzionamento viene rimandato al paragrafo 2.2.3.

2.2.2 Preparazione delle risorse grafiche

La parte destinata alla visualizzazione dei dati è implementata interamente attraverso la classe `AudioPanel`. Per raggiungere tale obiettivo, l'applicazione si serve del cosiddetto oggetto Direct2D (D2D), che non è altro che un insieme di API grafiche che consentono il rendering di geometrie, testo e immagini bitmap di tipo bidimensionale. A tale scopo, è necessaria l'inclusione dell'header `d2d1.h`. Le funzioni sono chiamate secondo la seguente successione:

1. creazione di un *factory object*, una sorta di contenitore per l'oggetto D2D.

2. `Initialize()`: imposta la finestra su cui si andrà a disegnare successivamente.

3. `EnsureResources()`: alloca le risorse necessarie per disegnare (utile anche qualora, in caso di errore, fossero state precedentemente perdute); in particolare, va a creare un rettangolo che fa da display, delle dimensioni corrispondenti a quelle che si vogliono mostrare su schermo, e si serve di un *render target*, uno strumento che realizza le operazioni di disegno vere e proprie sulla finestra.

4. `CreateEnergyDisplay()`, `CreateBeamGauge()`, `CreateSourceGauge()`, creazione di tutte le risorse per realizzare rispettivamente l'oscilloscopio, che quantifica dal punto di vista energetico il segnale in acquisizione, l'indicatore dell'angolo del beam e il localizzatore della sorgente sonora. Quest'ultimo colloca il parlatore disegnando una banda colorata a

larghezza variabile: tale espediente esprime il livello di confidenza con cui l'algoritmo ha ricavato la posizione, avvalendosi di una metodologia che si basa sul calcolo del gradiente.

5. `CreatePanelOutline()`: genera un pannello unico su cui verrà visualizzato quanto predisposto.

6. `Draw()`: gestisce l'operazione di disegno.

7. `DiscardResources()`: rilascia le risorse allocate per poter manipolare l'oggetto D2D.

2.2.3 Inizializzazione e aggiornamento dei dati

L'operazione successiva consiste nel gestire i dati prelevati dal sensore lungo tutta la catena di acquisizione, dall'inizializzazione alla chiusura dell'applicazione. I file di header necessari sono `dmo.h`, `wmcodecdsp.h`, `mmreg.h`, `strsafe.h` e `NuiApi.h`. I metodi invocati sono:

1. `CoInitializeEx()`: come già detto sopra, tale metodo serve per inizializzare le librerie destinate alla creazione di un oggetto COM (sempre in modalità *multithreaded*).

2. `DlgProc()`: offre un metodo per l'opportuna gestione dei messaggi pervenuti alla finestra; nello specifico, esistono quattro categorie:

   - `WM_INITDIALOG`, che inizializza i controlli e tutte le le funzioni che vanno ad agire sul dialog box; oltre ad allocare le risorse grafiche, invoca `CreateFirstConnected()` per usare Kinect come oggetto di acquisizione con l'opzione `NUI_INITIALIZE_FLAG_USES_AUDIO` e imposta un timer che manda periodicamente un messaggio `WM_TIMER`;
   - `WM_TIMER`, che provvede all'aggiornamento della finestra ogni 20 ms, chiamando le funzioni `ProcessAudio()` e `Update()`;
   - `WM_CLOSE`, che corrisponde alla chiusura della finestra (quando viene cliccato sul pulsante rosso con la "X"); manda a sua volta il messaggio `WM_DESTROY`;

- `WM_DESTROY`, che avvisa la *procedura* che l'applicazione deve essere chiusa, inviando poi al sistema operativo un `WM_QUIT`.

3. `InitializeAudioSource()`: una volta che la periferica è stata attivata, si rende necessaria la creazione del DMO e la sua configurazione; tra le opzioni, infatti, è possibile scegliere se servirsi semplicemente dei microfoni (Microphone Array Processing (MAP)), del cancellatore dell'eco acustica (Acoustic Echo Cancellation (AEC)), di entrambi o della combinazione soppressione del rumore (Noise Suppression (NS)) e controllo del guadagno (Automatic Gain Control (AGC)); contestualmente, viene assegnato il formato che si vuole abbia l'output del DMO:

   - formato: `WAVE_FORMAT_PCM`
   - numero di canali del flusso generato: 1
   - frequenza di campionamento: 16 KHz
   - velocità di trasferimento dei dati: 32 KB/s
   - dimensione dell'unità dati (per la sincronizzazione in riproduzione): 2 B
   - quantizzazione: 16 b/camp

4. `ProcessAudio()`: si occupa dell'ottenimento dei nuovi dati attraverso la definizione di un ciclo *do-while*; utilizzando l'istanza della classe `CStaticMediaBuffer`, si innesca un meccanismo sincronizzato con il timer per cui i dati vengono scritti sul buffer mediante `IMediaObject::ProcessOutput()` e se ne ricavano angolo del beam e della sorgente (con il relativo livello di confidenza) utilizzando rispettivamente `INuiAudioBeam::GetBeam()` e `INuiAudioBeam::GetPosition()`[6]. Infine, per ogni gruppo di 40 campioni, si calcola l'energia media – ottenuta come logaritmo della somma dei quadrati – i cui valori sono mostrati sotto forma di oscilloscopio, in modo da fornire una descrizione quantitativa di ciò che viene captato dai microfoni.

5. `SetBeam()`, `SetSoundSource()`: provvedono alla conversione da radianti a gradi degli angoli di beam e sorgente, in modo da aggiornarne i valori da mostrare sulla finestra.

[6] A titolo puramente informativo, si sottolinea come i valori degli angoli siano sempre espressi in radianti, per cui le direzioni di beam e sorgente saranno comprese tra i valori $-0,875$ e $0,875$, equamente ripartiti con passo $0,0175$.

6. `Update()`: consente la visualizzazione delle informazioni aggiornate sia sull'energia che sulla localizzazione.

7. `CoUninitialize()`: libera tutte le risorse allocate in fase iniziale per l'utilizzo dell'oggetto COM.

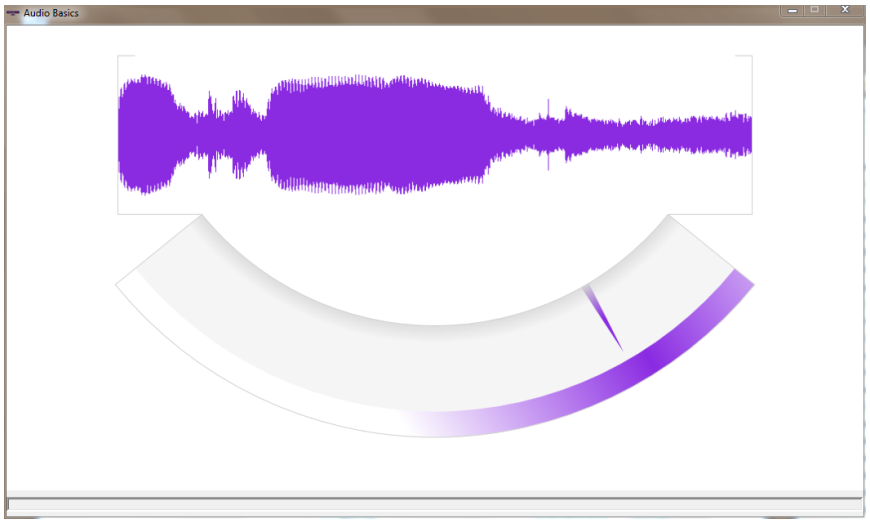

Figura 2.2: La finestra di dialogo di *Audio Basics*.

2.3 Fondamenti teorici sul beamforming

Dopo aver trattato nel dettaglio l'implementazione delle due applicazioni di base, può essere utile approfondire alcune tematiche analizzandole da un punto di vista più teorico.

Innanzitutto, la tecnica del beamforming è concettualmente meno complessa di quanto possa apparire a un primo esame. Il caso più semplice che si possa analizzare è quello in cui si hanno solo due microfoni, indicati con *m1* e *m2*, posti a una distanza d, come illustrato nella figura 2.3.
Se si prende un'onda – che per semplicità si considera piana – che giunge con una direzione perpendicolare rispetto alla linea che unisce idealmente i trasduttori, questa viene percepita da entrambi con la stessa fase. Se però l'angolo di arrivo θ è diverso da 90°, la fase non sarà più la stessa, e il sensore di sinistra (*m1*) riceverà il segnale con un leggero ritardo rispetto a *m2*, in

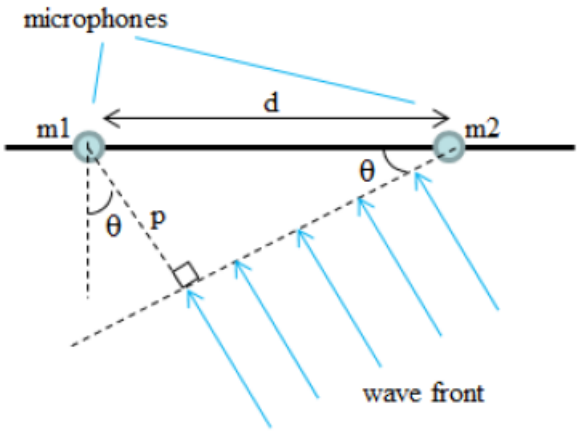

Figura 2.3: Un semplice esempio di beamforming nel caso di 2 canali.

quanto l'onda ha dovuto coprire la distanza aggiuntiva p. Questa situazione può essere sintetizzata facilmente con l'equazione:

$$\sin(\theta) = \frac{p}{d} \tag{2.5}$$

Considerando la velocità del suono c (pari a 343 m/s) in funzione di tale distanza, si ottiene la seguente espressione:

$$c = \frac{p}{\Delta t} \tag{2.6}$$

dove con Δt si intende il ritardo (in secondi) di arrivo dell'onda tra *m1* e *m2*. Combinando le due equazioni, l'angolo θ può essere allora ottenuto come

$$\theta = \arcsin\left(\frac{c \cdot \Delta t}{d}\right). \tag{2.7}$$

Partendo da questi concetti, se si collegano i segnali in uscita dai microfoni (o meglio in uscita dai rispettivi ADC) a un sommatore, nel caso la direzione di arrivo sia perfettamente perpendicolare la somma riproporrà fedelmente l'onda incidente, mentre nella seconda situazione la differente fase con cui viene visto il segnale genererà una somma attenuata rispetto alla precedente, in dipendenza proprio dall'angolo di incidenza. Sfruttando

questa proprietà, allora, l'idea è quella di introdurre in uscita da ogni singolo microfono un ritardo, così da orientare virtualmente lo schieramento senza il bisogno di muovere fisicamente qualcosa: semplicemente operando dei calcoli che simulino specifici ritardi e combinando opportunamente tali valori, le sovrapposizioni daranno luogo a interferenze di tipo costruttivo per alcune direzioni e distruttivo per altre. Dunque, tale metodologia è in grado di conferire un certo livello di direttività all'array, tanto maggiore quanto più aumenta il numero degli elementi componenti, e può essere impiegato sia in trasmissione che in ricezione, estendendo il campo delle possibili applicazioni anche a posizionamenti in tutte e tre le dimensioni.

Nonostante il principio su cui si basa la tecnica sia concettualmente semplice, una sua realizzazione mediante implementazione al calcolatore è molto complessa, a causa della necessità di ottenere ritardi differenti in funzione delle varie direzioni di arrivo, delle frequenze considerate, del posizionamento e della qualità dei singoli sensori. Per un utilizzo corretto del sistema è necessario trovare una matrice di pesi da assegnare alle quantità in gioco. Pur avendo a che fare con un dispositivo le cui caratteristiche sono ben note, gli algoritmi interni per l'ottenimento del beam non sono mai stati resi noti da Microsoft, che si è sempre giustificata dicendo che se fossero stati rivelati i meccanismi interni di calcolo, non sarebbero state più garantite le prestazioni di Kinect stessa. Tramite l' SDK, infatti, non si può andare oltre l'invocazione dei metodi `GetBeam()` e `GetPosition()`, che si limitano a generare i valori che vengono poi mostrati su schermo senza fornire altri dettagli di implementazione. Quello che segue, perciò, è frutto di una documentazione personale che si ritiene abbia verosimilmente contribuito alla progettazione del dispositivo.

Gli algoritmi per la realizzazione di un beamformer erano inizialmente basati, a seconda della geometria dei sensori, sul calcolo di soluzioni parametriche che sopprimessero il rumore. Data l'elevata complessità del problema, in tempi successivi si è passati alla ricerca di soluzioni sub-ottimali (tra cui si cita il Minimum-Variance Distortionless Response (MVDR)), più efficienti dal punto di vista computazionale ma legati a situazioni di tipo statico. Per garantire prestazioni migliori in caso di sorgente in (rapido) movimento, sicuramente un approccio adattativo risulta il più indicato, anche se un possibile inconveniente potrebbe essere la lentezza di convergenza. Secondo quanto scritto da Tashev e Malvar in [4], un generico algoritmo applicabile a diverse

geometrie di sensori dovrebbe concentrarsi essenzialmente sui seguenti aspetti: calcolare il guadagno del beamformer in funzione del rumore degli strumenti, considerare la funzione di trasferimento dei trasduttori, sopprimere il rumore e semplificare il problema cercando di ridurne i gradi di libertà. Il loro modello considera M microfoni, le cui posizioni (note) sono determinate dal vettore $\vec{p}$; ciascun sensore m ha un diagramma direzionale noto $U_m(f, c)$, con $c = (\varphi, \theta, \rho)$ che rappresenta la collocazione della sorgente in coordinate sferiche. Nel dominio della frequenza, il segnale derivante da ciascun microfono è espresso dalla relazione:

$$X_m(f, p_m) = D_m \, A_m(f) \, U_m(f, c) \, S(f) \tag{2.8}$$

$$D_m(f, c) = \frac{e^{-j2\pi f \|c - p_m\|}}{\|c - p_m\|} \tag{2.9}$$

con $D_m(f, c)$ che rappresenta il decadimento dovuto alla distanza dal microfono, $A_m(f)$ che corrisponde alla risposta in frequenza della catena preamplificatore-ADC, (per semplicità considerata unitaria) e $S(f)$ che indica la sorgente.

Le possibili fonti di disturbo sono racchiuse in due categorie: rumore isotropico e rumore prodotto dagli strumenti[7]. Per quanto riguarda il modello del beamformer, la forma canonica utilizzata è la seguente:

$$Y(f) = \sum_{m=1}^{M-1} W_m(f) \, X_m(f) \tag{2.10}$$

dove $W_m(f)$ è la matrice $N \times M$ dei pesi per il microfono m-esimo (N è il numero di gruppi in cui è suddivisa la banda che tiene conto della struttura del banco filtri) e $Y(f)$ è l'uscita. Di conseguenza, il guadagno del beam $B(f, c)$ è dato da:

$$B(f, c) = \sum_{m=1}^{M-1} W_m(f) \, D_m(f, c) \, U_m(f, c). \tag{2.11}$$

[7] Il primo tiene conto della proprietà di isotropia per cui una grandezza definita nello spazio è indipendente dalla direzione; il secondo, invece, riguarda l'hardware fisico, e risulta incorrelato tra i vari canali (il suo spettro è vicino a quello del rumore bianco).

L'obiettivo è dunque quello di calcolare la matrice dei pesi $W_m(f)$ per un determinato punto focale c_T, in modo da ottenere minimizzare l'energia E_N del rumore; i vincoli da considerare per il problema di ottimizzazione sono il guadagno unitario e la fase nulla in c_T:

$$\begin{cases} |B(f, c_T)| & = 1 \\ arg(B(f, c_T)) & = 0 \end{cases} \qquad \forall f \in [f_B, f_E] \tag{2.12}$$

dove f_B ed f_E sono le frequenze che delimitano la banda di interesse. Per semplificare il calcolo dell'ottimo, piuttosto che minimizzare il rumore viene preferito sintetizzare e normalizzare l'errore mediante minimi quadrati per poi concentrarsi esclusivamente sulla ricerca della migliore larghezza δ della focalizzazione, riducendo i gradi di libertà a uno solo. Tenendo presente che il rumore era stato ridotto a due soli termini, si osserva come l'analisi diventa una questione di equilibrio tra due condizioni contrapposte: restringere la larghezza vuol dire aumentare la direttività, il che si traduce in una diminuzione del rumore d'ambiente e in un aumento del rumore strumentale; diminuire la larghezza, invece, provoca una situazione contraria. Gli autori spostano la loro ricerca al valore di δ che garantisca il minimo disturbo, con l'intento di trovare una soluzione sub-ottima. La procedura si articola attraverso le seguenti fasi:

- definizione della forma del beam in funzione dell'unico parametro δ
- creazione di un pattern di pesi iniziale, minimizzando l'errore quadratico medio
- ottimizzazione del parametro δ in modo da garantire la minimizzazione di E_N
- calcolo della matrice dei pesi W per ogni beam
- ottenimento dell'uscita mediante l'equazione (2.10) e calcolo dell'energia del segnale: tra tutti i valori ricavati, quello più alto indicherà la posizione della sorgente

Le simulazioni sono svolte utilizzando una frequenza di campionamento pari a 16 KHz, frame audio lunghi 20 ms (quantità scelta in funzione della larghezza di banda in ambito teleconferenza), il che origina un numero di gruppi

di frequenze $N = 16\,000 \cdot 0.02 = 320$; gli altri valori forniti sono $f_B = 200$ Hz, $f_E = 7\,000$ Hz, $M = 4$ microfoni a cardioide disposti linearmente lungo 190 mm e operanti nel range $\pm 50°$. Non a caso, tutti i dati rispecchiano le caratteristiche strutturali di Kinect, a conferma che tale metodologia possa verosimilmente essere stata utilizzata in fase di progettazione.

Tra i vantaggi connessi a quanto proposto, sicuramente si ottiene una riduzione del Signal-to-Noise Ratio (SNR) che oscilla tra i 10 e i 15 dB, cui si aggiunge un modesto costo in termini di risorse occupate al calcolatore, il che lo rende di facile implementazione in tempo reale.

Un'ulteriore aspetto da sottolineare è legato al fatto che gli algoritmi che operano con degli array si basano sempre sull'assunzione che i vari canali siano perfettamente corrispondenti. Nella realtà i microfoni, anche se teoricamente identici, presentano presentano sempre delle sottili differenze, legate sia a tolleranze di costruzione, sia all'azione di agenti esterni come pressione, temperatura, umidità, etc. che possono influire sulla cascata sensore-preamplificatore-ADC. Per ovviare a questo inconveniente, in [13] viene suggerito un processo di autocalibrazione dei guadagni, sviluppabile in tempo reale, basata sulla proiezione delle coordinate $\{x, y, z\}$ del sensore lungo la direzione individuata dalla Direction of Arrival (DOA), così da interpolare i livelli di energia relativi a ciascun canale con un'unica retta che rende uniforme il calcolo dei guadagni.

Capitolo 3

Studio del comparto video

Gli argomenti finora affrontati si sono concentrati essenzialmente sui segnali prelevati dai microfoni, sulla loro elaborazione e sui possibili utilizzi. In questo capitolo si intende completare l'analisi delle componenti hardware ponendo l'attenzione su quanto Kinect è in grado di rilevare tramite gli altri sensori: proiettore e ricevitore IR e telecamera RGB; l'intento è quello di ricavare una visione accurata delle potenzialità offerte dal dispositivo nella sua totalità. È necessario allora fornire dapprima una base teorica sulle informazioni di profondità, per comprenderne a fondo il significato e il modo in cui sono state efficientemente utilizzate, per poi passare a una trattazione di come tali studi trovino applicazione concreta nell'esempio *Skeletal Viewer*.

3.1 Dall'immagine di profondità all'individuazione delle articolazioni

L'interazione uomo-macchina è una disciplina che si occupa dello studio e dello sviluppo di sistemi usabili e affidabili che facilitino le attività umane. Tale materia copre diversi ambiti che vanno oltre l'informatica, comprendendo anche la psicologia e la comunicazione. Un concetto molto importante in questo senso è il *body tracking*, cioè il tracciamento del corpo umano nella sua libertà di movimento; le sue applicazioni trovano ormai impiego nei settori più disparati: oltre all'ambito dei videogiochi, esso può essere utilizzato anche nel settore della sicurezza o in medicina. Fino a qualche anno fa, i limiti sullo sviluppo di algoritmi pensati per il raggiungimento di questa finalità erano

legati alla velocità di esecuzione, dovendo essere necessariamente utilizzati in tempo reale, e alla robustezza delle procedure di ri-inizializzazione del modello.

L'avvento di Kinect ha portato delle piccole novità in questo campo: in base a quanto affermato da Shotton *et al.* in [14][15], utilizzando le informazioni di profondità ricavate tramite il sensore e applicandogli un approccio di tipo object-recognition, l'idea è quella di segmentare l'immagine avvalendosi di una distribuzione di probabilità delle diverse parti del corpo, centrate intorno alle articolazioni principali, generando così una rappresentazione intermedia dell'individuo.

Basandosi sugli studi disponibili in letteratura, le principali difficoltà riscontrate erano legate alla variabilità delle caratteristiche della sagoma della persona, in particolare per quanto riguarda la forma del corpo e del vestiario; a ciò si aggiungeva una problematica sulla qualità della realizzazione: pur avendo a disposizione delle immagini dettagliate ottenute mediante tecniche di MOtion CAPture (MOCAP), appariva quasi impossibile riuscire a riprodurre in maniera esaustiva la complessità dei movimenti di cui è capace l'essere umano. Per cercare di ovviare a tali criticità sono stati innanzitutto sfruttati i progressi tecnologici raggiunti in ambito *depth imaging*, in modo da trarre vantaggio utilizzando hardware in grado sia di lavorare sotto diverse condizioni di illuminazione, senza essere influenzati da colori e consistenza delle superfici, sia di semplificare l'estrazione degli sfondi e la generazione di immagini sintetiche. In secondo luogo, essendo Kinect stata pensata come una periferica da utilizzare in ambito videoludico, il database di riferimento è stato circoscritto ai gesti che un individuo poteva compiere in situazioni di gioco: guidare, calciare, ballare, etc. Concentrandosi solo su pose di tipo statico, in modo tale da scartare molte immagini che apparivano ridondanti, si voleva cercare di sfruttare la proprietà di generalizzazione dell'algoritmo di apprendimento, operando una predizione di tutto ciò che non veniva acquisito dal sensore.

Per semplificare la realizzazione della mappa intermedia del corpo, la segmentazione è stata affrontata come un problema di classificazione dei pixel. Le parti del corpo sono state raggruppate in 31 categorie, in modo che fossero abbastanza piccole da garantire una localizzazione piuttosto accurata delle articolazioni ma al tempo stesso non troppo piccole da appesantire le prestazioni dell'algoritmo. Per l'addestramento del sistema, sono state

usate immagini artificiali che riproducessero delle pose il più possibile fedeli ad atteggiamenti reali, cercando di spaziare per forma e dimensione. La classificazione del singolo pixel si realizza in base alla parte del corpo a cui dovrebbe appartenere; l'estrazione delle caratteristiche f_θ deriva dal confronto con le immagini campione:

$$f_\theta(I, x) = d_I\left(x + \frac{u}{d_I(x)}\right) - d_I\left(x + \frac{v}{d_I(x)}\right) \tag{3.1}$$

dove $d_I(x)$ indica la profondità relativa al pixel x dell'immagine I, e $\theta = (u, v)$ racchiude gli scarti u e v normalizzati in modo che le caratteristiche siano invarianti lungo l'asse Z. A supporto di quanto ricavato, viene impiegato il modello della *foresta di decisione*. Tale classificatore è costituito da un certo numero di alberi di decisione, ciascuno dei quali si compone a sua volta di archi, foglie e *split* o nodi di biforcazione. A ogni split sono associati una caratteristica f_θ e una soglia τ; per valutare il singolo pixel x appartenente all'immagine I, si procede come segue: a partire dalla radice dell'albero, si sceglie se procedere verso destra o verso sinistra in base al valore che assume f_θ, in accordo all'equazione (3.1), confrontandolo di volta in volta con quello di τ. Alla fine, raggiunta una certa foglia, si ottiene una distribuzione "addestrata" relativa a una specifica parte del corpo.

In base a quanto descritto dagli autori, i test sull'accuratezza sono stati svolti considerando i seguenti parametri:

- 3 alberi di decisione, composti da 20 nodi
- 300 000 immagini, sia reali che sintetiche, variabili rispetto a posa, forma e dimensione del corpo, profondità della scena
- 2 000 pixel estratti da ciascuna immagine per l'addestramento
- 2 000 caratteristiche
- 50 soglie

ottenendo dei buoni risultati in termini sia di classificazione che di predizione delle articolazioni. L'esito positivo ha consentito di garantire che la combinazione hardware-software trovata fosse rapida e al tempo stesso robusta, per cui la progettazione delle varie componenti di Kinect si è servita proprio di tali scoperte.

3.2 Generazione dei dati

In generale, Kinect produce tre tipi di dati, relativi al colore, alla profondità e allo scheletro; per ciascuno di essi, risulta definito un relativo spazio di coordinate.

- Nello spazio dei colori, il sensore acquisisce un'immagine bidimensionale RGB contenente tutto ciò che entra a far parte del campo visivo inquadrato dalla telecamera. In accordo alla risoluzione scelta, ciascun frame presenta un certo numero di pixel: ognuno di essi indica la quantità di rosso, verde e blu del corrispondente punto nel piano (x,y). Con l'ultima release dell'SDK sono stati resi disponibili anche i formati *YUV* e *Bayer*, consentendo il bilanciamento di luminosità, contrasto, saturazione, etc. dell'immagine.

- Discorso analogo può essere fatto per lo spazio della profondità, in cui l'immagine bidimensionale è in scala di grigi. Anche in questo caso, il frame è costituito da pixel, ma le informazioni riguardano la distanza cartesiana tra l'oggetto che contiene quel punto e il piano del sensore, espressa in millimetri (il concetto è chiarito nella figura 3.1). È importante sottolineare che le coordinate (x,y) vanno interpretate in funzione dell'immagine acquisita, e non all'ambiente inquadrato. Le distanze di utilizzo garantite possono variare nell'intervallo $0.8 - 4$ m oppure $0.4 - 3$ m, a seconda che si stia lavorando in modalità di default o in modalità *near*; ciò premesso, le specifiche indicano che possono essere captati oggetti fino a 8 m, anche se la degradazione dell'informazione rischia di generare un output più confuso. In aggiunta, i dati sono elaborati in modo da costituire una *mappa di segmentazione*: in ogni pixel viene memorizzato l'indice della persona che è più vicina al sensore (in relazione a quella posizione). Tale indicizzazione consente di individuare fino a sei persone distinte (nel campo di vista) e agevola lo *skeletal tracking*.

- Per quanto riguarda la terza categoria, i dati di tipo *skeleton* sono ricavati a partire dall'immagine di profondità: ciascun frame contiene le coordinate tridimensionali (x,y,z) relative alle 20 articolazioni individuate, espresse questa volta in metri; inoltre, viene fornita una quarta coordinata w, che deriva da una stima della distanza dal pavimento

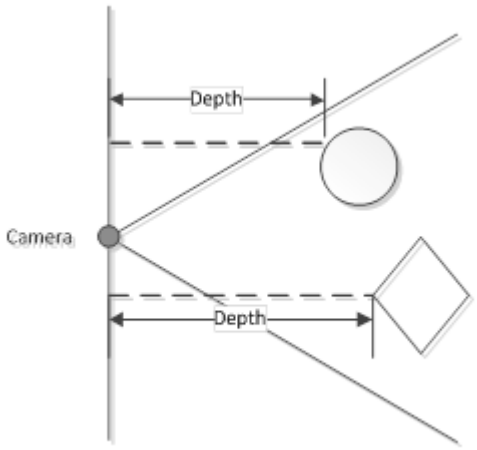

Figura 3.1: Modalità di ottenimento dei valori relativi al flusso di profondità.

ottenuta mediante la logica del *clipping plane*: il piano viene usato per dividere in due l'inquadratura, ciò che è dentro e ciò che è fuori da esso. Tramite questo espediente, se si omette tutto ciò che è fuori si risparmia tempo di calcolo, rendendo più agevole la rimozione dello sfondo e la segmentazione dell'utente. Di default, il sensore è posto sempre nella posizione (0,0,0,1). Si noti che lo scheletro, essendo destinato a una visualizzazione su schermo, viene disegnato in accordo a un "effetto specchio" (altrimenti ci si troverebbe di fronte ad una vista *di spalle*).

3.3 L'esempio Skeletal Viewer

A partire da quanto sinora illustrato, il terzo esempio studiato è stato lo *Skeletal Viewer* [16], che racchiude al suo interno alcuni spunti interessanti riguardo la visualizzazione di diverse tipologie di informazione, cui si aggiunge la possibilità da parte dell'utente di selezionare alcune opzioni sul tracciamento. L'applicazione produce una finestra di dialogo contenente tre riquadri distinti:

- un'immagine in scala di grigi, che fornisce le informazioni di profondità e, una volta riconosciuta una persona, ne colora la sagoma;

- un pannello a sfondo nero su cui vengono riconosciuti e tracciati i movimenti di tutto il corpo, sotto forma di uno scheletro;
- un'immagine a colori, che ha il solo scopo di mostrare quanto inquadrato.

Inoltre è presente un riquadro che permette di vedere la velocità di acquisizione, ossia l'effettiva frequenza di cattura dei fotogrammi che compongono il video visualizzato, espressa in frame per second.

Il meccanismo di riconoscimento delle persone impone un limite sul numero di soggetti rilevabili, pari al più sei individui; tra questi, lo skeletal tracking può essere effettuato su un massimo di due figure. Il progetto che realizza le finalità preposte risulta articolato nel seguente modo:

- `DrawDevice.cpp`, che implementa i metodi (definiti nel corrispondente file di header all'interno della classe `DrawDevice`) per realizzare un'immagine bitmap: imposta la finestra e il formato video, per poi provvedere all'operazione di disegno
- `SkeletalViewer.cpp`, che contiene il *main()* e implementa le funzioni membro della classe `CSkeletalViewerApp`
- `NuiImpl.cpp`, che si occupa dei metodi che hanno a che fare con il *NUI processing*, ossia gestisce i dati prelevati da Kinect
- `SkeletalViewer.rc`, `Resource.h`, `SkeletalViewer.ico`, che costituiscono il file di risorse della finestra, la definizione delle relative costanti di controllo e l'icona associata all'eseguibile prodotto dall'applicazione
- `stdafx.h`, che raccoglie tutte le inclusioni standard del progetto e un template per il rilascio delle interfacce

Analogamente a quanto già visto con *Audio Basics*, l'interfaccia grafica è rappresentata da un dialog box, per cui valgono le stesse considerazioni fatte nel capitolo 2: si procede con la descrizione dell'applicazione, raggruppata per aree tematiche.

3.3.1 Gestione della finestra

Seguendo l'impostazione illustrata nei casi precedenti, dopo aver creato la variabile globale `g_skeletalViewerApp`, la prima azione da intraprendere

è predisporre correttamente la finestra. Di seguito si elenca la successione delle chiamate:

1. `Main()`: anche se non si può propriamente affermare che venga invocato – il *main()* costituisce il punto di ingresso per l'intera applicazione – definizione, registrazione e creazione della finestra avvengono proprio all'interno di tale funzione, a differenza di quanto accadeva con gli altri esempi. La struttura di riferimento è sempre di tipo `WNDCLASS`; i messaggi scambiati durante l'esecuzione vengono gestiti mediante un apposito ciclo *while*. Viene inoltre creato un *mutex*, cioè un oggetto che provvede alla coordinazione dell'accesso alle risorse da parte di thread e processi, sincronizzando così l'ordine di chiamata delle varie funzioni.

2. `MessageRouter()`: è la *procedura* che gestisce i messaggi da mandare alla finestra. Invoca a sua volta la funzione `WndProc()`, affrontata nel paragrafo 3.3.3.

3. `ClearKinectComboBox()`: considerato che nella finestra di dialogo è presente un *combo box*[1] contenente l'elenco dei sensori Kinect che sono connessi, tale metodo serve a rimuovere dalla lista tutti i dispositivi che non risultano correntemente attivi.

4. `UpdateKinectComboBox()`: consente, ogni volta che avviene un aggiornamento dello stato dei sensori, di scegliere quale tra questi utilizzare.

3.3.2 Preparazione delle risorse grafiche e realizzazione delle operazioni di disegno

Per la visualizzazione dei dati, anche in questo caso l'applicazione si serve degli oggetti D2D. È necessaria l'inclusione del file `d2d1.h` e `d2d1helper.h`, per accedere alle API grafiche. Dal momento che la gestione avviene secondo due modalità differenti, se ne separa la trattazione.

Per ottenere lo scheletro, la procedura è la seguente:

1. creazione del *factory object* che conterrà l'oggetto D2D (in modalità *single threaded*).

[1] Con *combo box* si intendono genericamente delle liste contenenti una serie di opzioni (mutuamente esclusive) che l'utente può scegliere; un utilizzo tipico si osserva nella comune barra degli strumenti delle finestre.

2. `EnsureDirect2DResources()`: alloca le risorse necessarie per disegnare, generando fisicamente il display attraverso la creazione di un rettangolo delle dimensioni desiderate; per mostrare il disegno nella finestra, si serve di un *render target*. Vengono infine scelti i colori da utilizzare per lo scheletro: a seconda che il movimento del corpo sia stato seguito correttamente o sia frutto di deduzioni (derivanti da dati precedenti, assunzioni di natura geometrica, etc.), sono usate delle diverse opzioni. Nel primo caso vengono impiegati il verde per le "ossa" e il verde chiaro per le "articolazioni", mentre nel secondo i colori sono rispettivamente il grigio e il giallo, cui si aggiunge l'uso di uno spessore più sottile.

3. `Nui_DrawBone()`: dati due punti, la funzione si occupa di disegnare fisicamente una linea retta compresa tra di essi; ciascuna linea costituirà un osso dello scheletro. Quest'ultimo viene etichettato come `tracked` solo se ne sono state seguite entrambe le articolazioni che lo comprendono, altrimenti è considerato `inferred`. Si sottolinea che in generale un osso viene disegnato solo se entrambe le giunture che lo racchiudono sono `tracked`: in tutte le altre situazioni, la funzione non ritorna niente.

4. `Nui_DrawSkeleton()`: si occupa di disegnare una linea per ciascuna coppia di articolazioni, in modo da ottenere lo scheletro completo della persona tracciata. Il numero totale di giunture è posto a 20, in accordo a quanto definito in `NUI_SKELETON_POSITION_INDEX`.

5. `SkeletonToScreen()`: disegna su schermo le informazioni pervenute dal sensore, convertendole in coordinate conformi alla risoluzione del depth stream.

6. `DiscardDirect2DResources()`: rilascia il *render target* e le altre risorse impegnate.

Per quanto riguarda gli altri due riquadri, i relativi video sono realizzati come delle immagini bitmap, in accordo a quanto dichiarato nella classe `DrawDevice`:

1. `Initialize()`: imposta la finestra su cui disegnare e il formato video.

2. `EnsureResources()`: alloca le risorse, crea il render target e l'immagine bitmap.

3. `Draw()`: gestisce tutte le operazioni di disegno, utilizzando 32 *bit per pixel*.

4. `DiscardResources()`: libera tutte le risorse che erano state impegnate in precedenza.

3.3.3 Inizializzazione e manipolazione dei dati

Tutto ciò che viene prelevato dal sensore è reso disponibile sotto forma di frame. Per potervi accedere, è necessario impostare un meccanismo di recupero delle informazioni che preveda l'impiego di un apposito buffer, stante che il dispositivo opera in tempo reale e rende quindi disponibili i frame non più di una volta; il rischio che può capitare è che se ne richieda l'accesso quando non risultano ancora pronti. A questo proposito, la logica di acquisizione è di tipo "a eventi" (*event model*): ogni volta che il frame di dati (colore o profondità) è completo, viene notificato un apposito evento che per così dire avverte gli eventuali thread in attesa che le informazioni sono disponibili[2].

I file di header necessari sono `NuiApi.h`, che aggrega tutti gli header relativi alle API NUI, e `strsafe.h`, per manipolare correttamente i vari buffer. I metodi interessati sono allora:

1. `Nui_Zero()`: azzera tutte le variabili che servono per la visualizzazione dei dati.

2. `WndProc()`: offre un metodo di gestione dei messaggi pervenuti alla finestra; le tipologie possibili sono elencate di seguito:

   - `WM_INITDIALOG`, che inizializza tutti i controlli e le funzioni che modificano l'aspetto del dialog box;
   - `WM_SHOWWINDOW`, che viene mandato ogni volta che la finestra deve essere mostrata o nascosta; invoca il metodo `Nui_Init()`;

[2] Al modello a eventi si contrappone quello del *polling*, in cui l'applicazione, nell'accedere a un frame, stabilisce la durata di un intervallo di tempo durante il quale rimarrà in uno stato di attesa; in questo modo, viene generato un risultato o quando il frame è pronto oppure quando è scaduto il *time-out*; tale metodo è più semplice da utilizzare, ma risulta meno flessibile e accurato.

- `WM_USER_UPDATE_FPS`, che serve per l'aggiornamento della velocità di acquisizione video;
- `WM_USER_UPDATE_COMBO`, che segnala l'aggiornamento del combo box;
- `WM_COMMAND`, che notifica la selezione di un opzione da parte dell'utente; dato che sono presenti tre combo box attraverso cui comunicare con l'applicazione, le alternative possibili riguardano la distanza di azione (default o near), la modalità di tracciamento (in piedi o seduto) e il criterio di scelta dello scheletro (quello/i più vicini, quello/i tracciati per primi o selezione automatica di default);
- `WM_CLOSE`, che corrisponde alla chiusura della finestra (quando viene cliccato sul pulsante rosso con la "X"); manda a sua volta il messaggio `WM_DESTROY`;
- `WM_DESTROY`, che avvisa la *procedura* che l'applicazione deve essere chiusa, inviando poi al sistema operativo un `WM_QUIT`.

3. `Nui_StatusProcThunk()` e `Nui_StatusProc()`: costituiscono la funzione di *callback* che gestisce i cambi di stato di Kinect, quando cioè viene connessa/disconnessa.

4. `Nui_Init()`: innanzitutto si occupa della procedura di inizializzazione del sensore secondo lo standard già visto nei precedenti esempi; poi crea i suddetti eventi per la gestione dei dati e richiama le funzioni destinate all'allocazione delle risorse grafiche; successivamente invoca `INuiSensor::NuiInitialize()`, segnalando le tre modalità di utilizzo del sensore: `NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX`, `NUI_INITIALIZE_FLAG_USES_SKELETON` e infine `NUI_INITIALIZE_FLAG_USES_COLOR`; dopodiché abilita lo skeletal tracking e apre i flussi relativi ai dati di tipo colore e profondità, specificandone anche la risoluzione; in ultima istanza crea un evento di stop per l'elaborazione e genera il thread destinato all'acquisizione vera e propria.

5. `Nui_ProcessThread()`: rappresenta il cuore del processo di acquisizione ed elaborazione dei dati: invoca le funzioni relative ai vari flussi, gestisce i display e calcola la velocità in fps. Le operazioni sono governate da un

ciclo *while* che, attraverso il richiamo delle funzioni standard `WaitForMultipleObjects()` e `WaitForSingleObject()` impostate per operare in accordo al modello a eventi, utilizza i metodi:

- `Nui_GotDepthAlert()`: gestisce le nuove informazioni relative alla profondità
- `Nui_GotColorAlert()`: gestisce le nuove informazioni ottenute sul colore
- `Nui_GotSkeletonAlert()`: gestisce le nuove informazioni ricavate sullo scheletro e provvede al suo disegno

6. `UpdateTrackedSkeletonSelection()`, `UpdateTrackingMode()`, e `UpdateRange()`: servono a rendere effettive le scelte effettuate dall'utente tramite i combo box.

7. `MessageBoxResource()`: manda un *message box*, un oggetto preimpostato che rientra nella categoria dei dialog box e serve per mostrare delle stringhe di testo caricate in precedenza, indicanti in questo caso errori.

8. `Nui_UnInit()`: ferma il processo di acquisizione, chiude il thread e rilascia le risorse impegnate.

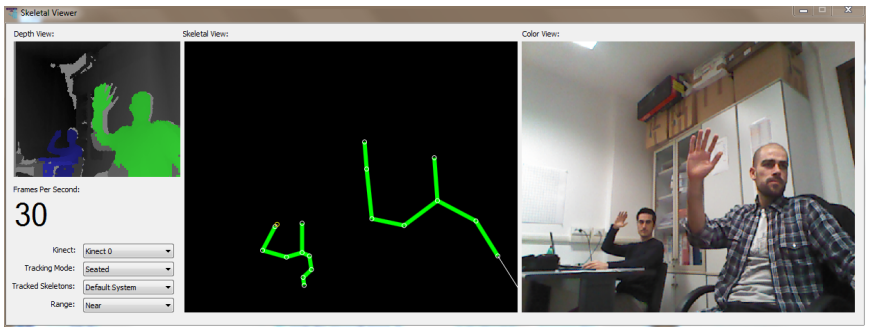

Figura 3.2: La finestra di dialogo di *Skeletal Viewer*.

Si vuole porre l'attenzione su un particolare aspetto di natura concettuale che contribuisce a migliorare la resa visiva dell'algoritmo; per ridurre il *jitter* tra un frame e il successivo, e quindi rendere il video meno

rumoroso, all'interno di `Nui_GotSkeletonAlert()` è invocata la funzione `INuiSensor::NuiTransformSmooth()`: i fotogrammi – rappresentati dalla variabile `SkeletonFrame` – vengono elaborati attraverso un filtro a doppio smorzamento esponenziale (basato sul metodo di Holt[3]), su cui è possibile agire modificando alcuni parametri come il numero di frame da predire, il fattore di correzione e il raggio di riduzione del jitter: in questo modo, si offre la possibilità di garantire che la rilevazione delle articolazioni sia accurata e al tempo stesso precisa[4] per diversi scenari di utilizzo, lasciando al progettista la libertà di distribuire appropriatamente le risorse in favore della fluidità dell'immagine o della latenza.

[3] Tale filtraggio è molto diffuso sia in campo statistico sia in campo economico nell'ambito dell'analisi di serie storiche: nel caso in questione, il principale vantaggio per cui viene usato è che offre una bassa latenza, rendendo l'operazione di *smoothing* applicabile in tempo reale.

[4] In ambito scientifico, con *accuaratezza* si indica la capacità di una misurazione di mantenersi il più vicina possibile al valore reale assunto da una grandezza, mentre con *precisione* si intende il grado con cui le ripetizioni dello stesso esperimento portano allo stesso risultato.

Capitolo 4

Sviluppo del software

Dopo aver affrontato nel dettaglio la ripartizione dei vari sensori, il modo corretto di accedervi e la conseguente manipolazione dei relativi dati, si vuole ora illustrare quanto sviluppato durante il lavoro di tesi. Le intenzioni che ci si era posti erano quelle di realizzare un software che permettesse l'accesso contemporaneamente a tutti i flussi, e che fosse in grado di sfruttare l'efficienza di Kinect per operare in tempo reale, con l'obiettivo di localizzare e inseguire un parlatore all'interno di uno spazio chiuso limitato.

Per il raggiungimento degli scopi stabiliti, lo sviluppo dell'applicazione si fonda sulle procedure di base fornite dagli esempi *Audio Basics* e *Skeletal Viewer*; in realtà in un primo momento, per mantenere un certo grado di semplicità di sviluppo e concentrarsi maggiormente sui contenuti, si era pensato di riferirsi a *Skeleton Basics*. Tuttavia questa strada si è rivelata riduttiva: l'interfaccia grafica si limitava a mostrare solo lo scheletro, senza possibilità di ottenere più informazioni nella stessa finestra o di effettuare delle scelte sulle opzioni di interazione offerte, per cui si è giunti alla conclusione di servirsi dell'altro modello.

Per i riferimenti dettagliati sulle varie modifiche apportate al codice e sull'implementazione dei metodi sviluppati, si rimanda all'appendice B.

4.1 Descrizione dell'applicazione

Delineati i contorni di quello che si voleva ottenere, si vuole fornire una breve descrizione sulle modalità con cui sono stati perseguiti i risultati pre-

fissati. La questione può essere schematizzata per grandi linee attraverso la cooperazione di diversi blocchi funzionali, come rappresentato nella figura 4.1. Da una parte c'è la coppia proiettore-ricevitore IR, che provvede all'elaborazione delle mappe di disparità per generare un'immagine in scala di grigi che descriva la profondità della scena; su questa base viene poi effettuata una segmentazione della figura che porta al riconoscimento della persona e al conseguente disegno dello scheletro. Dall'altra c'è l'array di microfoni, che, tramite il beamforming, consente di individuare la direzione di provenienza di un'onda sonora e "ascoltare" in corrispondenza di quella determinata angolazione. Per riuscire a mettere in contatto le due funzionalità si è scelto di utilizzare un approccio di tipo *gesture recognition*[1]: l'idea era quella di per così dire attirare l'attenzione del dispositivo, in modo da venire riconosciuti tramite l'esecuzione di un gesto chiave che garantisse l'individuazione esclusiva del soggetto agente. Tale concetto è stato concretizzato proprio in virtù del tracciamento delle articolazioni.

Lo studio si è protratto nella ricerca di un gesto che fosse semplice, naturale (in accordo alla filosofia delle NUI) e al tempo stesso non fosse fonte di ambiguità. Alla fine la scelta è ricaduta su quello che è comunemente seguito come modello di comportamento per prendere la parola quando ci sono più persone: l'alzata di mano. Oltre ad essere una consuetudine consolidata in situazioni collettive, tale gesto è unico, facile da svolgere, non richiede posture forzate che possano risultare scomode da mantenere e non richiede la memorizzazione di una sequenza di azioni.

Le modalità di realizzazione dello stesso saranno discusse più approfonditamente nel paragrafo 4.5.1. Il risultato che interessa è che l'individuazione dell'utente consente di estendere il contesto applicativo: l'inseguimento riguarda sia i movimenti del corpo che la direzione di provenienza della voce.

[1] Si precisa fin da ora che si dovrebbe parlare più precisamente di *pose recognition*, in quanto un gesto in senso proprio comporta il mantenimento di una *posa* per un certo lasso di tempo, mentre il modello utilizzato non prevede l'estrapolazione di alcun tipo di informazioni temporali. Ad ogni modo, per questioni di fluidità dei contenuti, nel resto della trattazione ci si riferirà comunque al gesto, alla luce però delle considerazioni appena fatte.

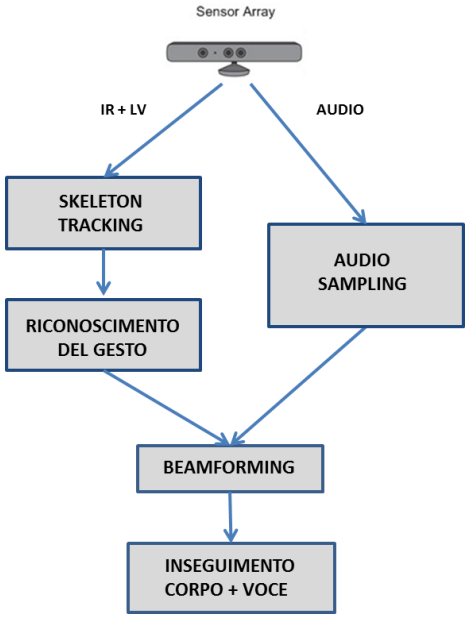

Figura 4.1: Descrizione funzionale dell'applicazione.

4.2 Struttura del progetto

In base a quanto richiesto dalle specifiche di programmazione previste dall'SDK, il progetto è rappresentato da una soluzione di Visual Studio (un file `.sln`) in C++, ed è composto dai seguenti file:

- `ClasseAudio.cpp`, che contiene i metodi – dichiarati nel relativo file di header – relativi alle classi `CStaticMediaBuffer`, `AudioPanel` e `CAudio` e si occupa della gestione dei dati di tipo audio, sia per quanto riguarda l'acquisizione che la visualizzazione.

- `DisegnoImmagine.cpp`, che implementa le funzioni della classe `DrawDevice` attraverso cui vengono rappresentate le informazioni di profondità e colore; imposta la finestra e il formato del video, e provvede a disegnare le immagini di tipo bitmap che compongono i due filmati da visualizzare.

- `Main.cpp`, che rappresenta il punto di accesso dell'intera applicazione; provvede a creare e a controllare la finestra, congiuntamente a fornire l'ossatura sovrastante per l'interfaccia grafica, attraverso cui si inseriscono i metodi che di volta in volta assolvono i vari compiti.

Nel corrispondente header, viene dichiarata la classe `CSkeletal` per la manipolazione dei dati relativi alla figura umana.

- `MetodiNui.cpp`, che riguarda tutte le funzioni che hanno a che fare con il NUI processing, ossia i dati di tipo colore, profondità, *skeleton* e audio.

- `SkeletalViewer.ico`, `SkeletalViewer.rc` e `Resource.h`, che rappresentano rispettivamente l'icona associata all'eseguibile generato, il file di risorse della finestra e la relativa definizione delle costanti di controllo.

- `WaveWriter.cpp`, che implementa i metodi contenuti nella classe `WaveWriter` (dichiarata nel corrispondente header) con lo scopo di scrivere un file di tipo WAVE contenente una registrazione dei dati audio.

- `stdafx.h`, che comprende tutte le inclusioni standard del progetto e un template per il rilascio delle interfacce.

Quello che si vuole produrre come output è, sulla falsariga di quanto si ottiene attraverso *Skeletal Viewer*, un'unica finestra di dialogo in cui poter avere un riscontro grafico delle diverse tipologie di dati. In particolare, come mostrato nella figura 4.2, ci sono un riquadro dedicato per ciascuna delle informazioni di tipo profondità, colore, *skeleton*, la frequenza di acquisizione, un indicatore della posizione della sorgente, un oscilloscopio, un pulsante attraverso cui registrare il file audio relativo a quanto captato dai microfoni, più altre stringhe di controllo (su cui ci soffermerà in maniera più approfondita nel paragrafo successivo).

Si procede allora con l'elenco di tutte le funzioni che entrano a far parte del esecuzione del programma.

4.3 Progettazione della finestra di dialogo

Entrando più nel dettaglio, dopo l'inclusione dei vari header presenti nel progetto e la creazione della variabile globale `g_skeletalApp`, le prime funzioni ad essere chiamate sono, nell'ordine:

1. `Main()`: definisce, registra e crea la finestra di dialogo, in accordo alla struttura di riferimento `WNDCLASS`; i messaggi scambiati durante

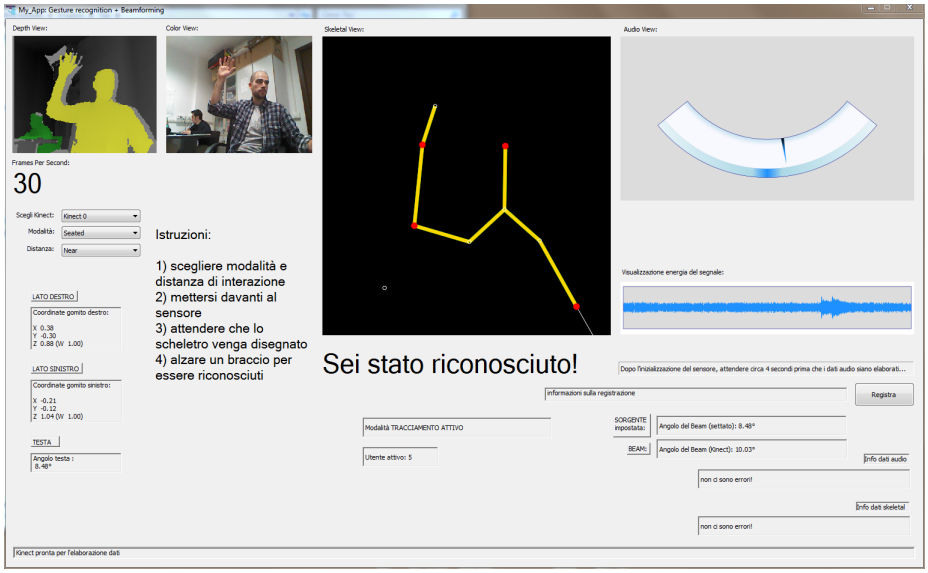

Figura 4.2: Visualizzazione dell'iterfaccia grafica.

l'esecuzione vengono gestiti mediante un apposito loop (un ciclo *while*), che li indirizza direttamente alla *procedura*. Anche in questo caso, avendo a che fare con più thread, l'accesso alle risorse è coordinato da un mutex.

2. `MessageRouter()`: è la *procedura* che gestisce i messaggi da mandare alla finestra. Invoca a sua volta la funzione `WndProc()`, affrontata nel paragrafo 4.5.

3. `ClearKinectComboBox()`: rimuove dal combo box contenente la lista delle Kinect tutti i dispositivi che non risultano correntemente attivi.

4. `UpdateKinectComboBox()`: consente, ogni volta che avviene un aggiornamento dello stato dei sensori (nel caso in cui l'utente connetta o disconnetta un dispositivo), di scegliere quale tra questi utilizzare.

Per agire sulla finestra di dialogo bisogna aprire il relativo editor, accessibile tramite il file `SkeletalViewer.rc`: nella cartella chiamata *Dialog* è contenuto il layout del dialog box, attraverso cui è possibile modificare dimensioni e contenuti di quello che sarà l'output prodotto dall'applicazione. Innanzitutto, per gestire i riquadri su cui vengono mostrati i relativi video, si deve utilizzare un *picture control*, che ritaglia uno spazio (tipicamente di forma rettangolare) sulla finestra e lo identifica attraverso l'assegnazione di

uno specifico ID, in modo da poterne disporre liberamente nel codice; tale ID sarà poi automaticamente inserito all'interno del file `Resource.h`. Discorso analogo si può fare per il *combo box control*, con cui predisporre la collocazione degli elementi raggruppati sotto forma di lista; in aggiunta, si può scegliere una tra le modalità *semplice*, *a cascata con input di testo* e *a cascata senza input*: la scelta è ricaduta sull'ultima opzione.

Un altro strumento importante è il *text control*, che, sempre tramite identificazione numerica di ciascun riquadro aggiunto, ne permette il controllo in maniera univoca attraverso l'utilizzo della relativa costante. In questo caso, è possibile anche impostare alcune opzioni di natura grafica, come ad esempio la presenza di un bordo, l'allineamento e il troncamento del testo. Il text control serve per creare delle etichette o per mostrare delle stringhe impostate dal programmatore, come messaggi di errore o feedback sul corretto funzionamento di alcune parti dell'algoritmo.

Infine, per inserire un bottone che serva ad avviare una qualche specifica operazione (in questo caso, la registrazione) si può ricorrere al *button control*.

In questa fase è anche possibile regolare le dimensioni del pannello e delle sue sottofinestre, sceglierne la disposizione e aggiungere eventuali informazioni da mostrare di default.

4.3.1 Cenni sull'utilizzo delle stringhe

Si vogliono ora passare brevemente in rassegna le parti del programma che fanno uso di stringhe, per capire a cosa possono servire e quale procedura seguire per crearle.

Il primo strumento su cui soffermarsi è il combo box, che rappresenta un utile strumento di interazione tra l'utente e l'applicazione, e può contribuire notevolmente all'aumento dell'usabilità complessiva. Nel caso sviluppato, i combo box vengono utilizzati per mostrare i sensori connessi, le modalità di tracciamento e la distanza di interazione. Il caricamento delle stringhe avviene nel seguente modo: una volta riservato l'apposito spazio nella finestra di dialogo, ottenuto l'ID e definite le dimensioni appropriate, è necessario scegliere le opzioni che si intendono mostrare all'utente, caricando le corrispondenti stringhe attraverso la funzione `LoadStringW()`. Successivamente, mediante `SendDlgItemMessageW()` viene mandato un messaggio di `CB_ADDSTRING`, che segnala alla procedura l'intenzione di aggiungere la stringa alla lista.

Per terminare la fase di predisposizione, viene mandato un messaggio di `CB_SETCURSEL`, sempre tramite `SendDlgItemMessageW()`. A questo punto, ogni volta che l'utente selezionerà una delle alternative proposte, il messaggio mandato al relativo controllo sarà di tipo `CB_GETCURSEL`.

Un altro ambito interessante è quello del testo statico: inizializzando le opportune risorse, si possono comporre delle frasi più o meno brevi da mostrare durante lo svolgimento delle azioni. Nello specifico, l'utilizzo che ne se ne è fatto serve a illustrare una breve procedura da seguire per il corretto funzionamento dell'applicazione (delle "istruzioni per l'uso"). Il raggiungimento di tale proposito è abbastanza agevole: si crea una variabile di tipo `LOGFONT`, si recupera un *handle* al font da utilizzare, in modo da poterne impostare appropriatamente gli attributi, si memorizza la stringa in una variabile e si utilizza la funzione `SendDlgItemMessageW()` per mandare i messaggi di `WM_SETFONT` e `WM_SETTEXT` al riquadro predisposto nell'editor delle risorse.

Si vuole evidenziare un ultimo caso, che ha svolto un ruolo molto importante in fase di progettazione: quello dei messaggi di conferma o di errore. Il programma, essendo molto articolato, presenta diverse funzioni, spesso annidate; per rendere tutto perfettamente funzionante, la maggior parte dei metodi sono stati definiti come `HRESULT`, un tipo di ritorno che descrive mediante un codice sintetico se lo svolgimento è andato a buon fine oppure no. Impostare dei riquadri su cui visualizzare il corretto funzionamento dei punti più delicati si è rivelato fondamentale per monitorare quello che succedeva, spesso più dal punto di vista semantico che sintattico. Anche in quest'ultima situazione, il procedimento è semplice: basta ritagliarsi lo spazio appropriato sulla finestra, memorizzare la stringa di controllo in una variabile `WCHAR` e mandare un messaggio di `WM_SETTEXT`.

```
1  LOGFONT lf_istr;
   GetObject((HFONT)GetStockObject(DEFAULT_GUI_FONT),sizeof(lf_istr),
3                                                      &lf_istr);
   lf_istr.lfHeight *= 2;
5  m_hFontFPS_istr = CreateFontIndirect(&lf_istr);
   WCHAR * szMessIstr = L" Istruzioni:\n\n";
7  SendDlgItemMessageW(hWnd,IDC_FPS3,WM_SETFONT,(WPARAM)m_hFontFPS_istr,
                                                    (LPARAM)szMessIstr);
9  SendDlgItemMessageW(hWnd,IDC_FPS3,WM_SETTEXT,0,(LPARAM)szMessIstr);
```

Codice 4.1: Esempio di visualizzazione di una stringa sulla finestra di dialogo.

4.4 Preparazione delle risorse grafiche e realizzazione delle operazioni di disegno

Avendo ormai acquisito una certa familiarità con gli oggetti D2D, la scelta non poteva che ricadere su di loro per quanto concerne il disegno dei fotogrammi da visualizzare. Viene creato un unico *factory object* in modalità *multithreaded*, dal momento che si ha a che fare con più oggetti D2D, ciascuno trattato da un diverso thread. Per utilizzare le relative API, i file di header da includere sono `d2d1.h` e `d2d1helper.h`. La gestione delle immagini avviene in maniera separata: profondità, colore e energia del segnale vocale sono immagini di tipo bitmap, mentre lo scheletro e il localizzatore sono ottenuti per via geometrica, mediante linee, archi ed ellissi. Nel primo caso la successione segue quanto dichiarato nella classe `DrawDevice`:

1. `Initialize()`: imposta la finestra su cui disegnare e il formato video.
2. `EnsureResources()`: alloca le risorse, crea il render target e l'immagine bitmap.
3. `Draw()`: gestisce tutte le operazioni di disegno, utilizzando 32 *bit per pixel*.
4. `DiscardResources()`: libera tutte le risorse che erano state impegnate in precedenza.

Per quanto riguarda l'oscilloscopio, la struttura generale è la stessa, ma i metodi fanno parte della classe `AudioPanel`:

1. `InitializeEnergy()`: inizializzazione
2. `EnsureEnergyResources()`: allocazione delle risorse
3. `CreateEnergyDisplay()`: creazione del display
4. `CreateEnergyPanelOutline()`: posizionamento del display all'interno (attraverso la scelta delle coordinate)
5. `UpdateEnergy()`: aggiornamento dei valori da mostrare
6. `DrawEnergy()`: disegno dell'energia come immagine bitmap
7. `DiscardEnergyResources()`: deallocazione delle risorse

Nel caso geometrico, invece, con riferimento alla classe `CSkeletal` la procedura è la seguente:

1. `EnsureDirect2DResources()`: alloca le risorse necessarie per poter disegnare, generando fisicamente il display attraverso la creazione di un rettangolo delle dimensioni desiderate; per mostrare il disegno nella finestra, si serve di un *render target*. A questo punto, si interviene sulla scelta dei colori da utilizzare per lo scheletro: mantenendo la convenzione già vista nel caso *Skeletal Viewer* che distingue le parti dedotte da quelle effettivamente tracciate, si è scelto di utilizzare un colore a parte per testa, gomiti e polsi, dal momento che rappresentano delle parti di riferimento per l'algoritmo (il loro ruolo apparirà più chiaro quando si analizzerà il riconoscimento del gesto nel successivo paragrafo 4.5.1). Inoltre, le ossa appartenenti alla persona riconosciuta saranno contraddistinte da una tinta giallo oro.

2. `Nui_DrawBone()`: dal momento che la funzione si occupa di tracciare un singolo osso, rappresentato da una linea retta, è qui che si effettua un primo intervento per contraddistinguere i segmenti facenti parte di una persona riconosciuta: oltre a `tracked` e `inferred`, viene prevista anche l'etichetta `Recog`, con colore relativo.

3. `Nui_DrawSkeleton()`: anche in questa funzione sono state effettuate delle modifiche; stante che essa provvede all'ottenimento dello scheletro completo relativo al soggetto tracciato, si è deciso di ottenere esplicitamente le coordinate rilevate da Kinect riguardo la posizione dei gomiti, cui si ha accesso attraverso l'attributo `SkeletonPositions[NUI_SKELETON_POSITION_ELBOW_RIGHT/LEFT]`. La scelta ha un duplice scopo: traccia un'articolazione importante ai fini del riconoscimento e fornisce un riferimento esplicito sul posizionamento della giuntura nello spazio del sensore.

4. `SkeletonToScreen()`: disegna su schermo le informazioni pervenute dal sensore, convertendole in coordinate conformi alla risoluzione del depth stream.

5. `DiscardDirect2DResources()`: rilascia il *render target* e le altre risorse impegnate.

Per ciò che concerne il localizzatore, la situazione ricalca quanto appena visto, utilizzando lo stesso *factory object* già creato. Naturalmente le funzioni sono state create a parte e sono contenute nella classe `CAudio`:

1. `InitializeGauge()`: inizializzazione
2. `EnsureGaugeResources()`: allocazione delle risorse e dimensionamento del pannello
3. `CreateBeamGauge()`, `CreateBeamGaugeNeedle()`: creazione del misuratore
4. `CreateGaugePanelOutline()`: scelta delle coordinate posizionamento del display all'interno del riquadro
5. `DrawGauge()`: disegno del pannello mediante linee ed archi
6. `DiscardGaugeResources()`: deallocazione delle risorse

4.5 Gestione delle informazioni e data fusion

Avendo a che fare con informazioni gestite in modo separato, una prima difficoltà è stata riscontrata nel mettere d'accordo la diversa logica sottostante la gestione dei dati: da una parte il modello *a eventi* descritto nel paragrafo 3.3.3, dall'altro l'acquisizione su base temporale mediante segnalazione periodica da parte di un timer. In entrambe le situazioni vengono creati degli appositi frame in cui è immagazzinato quanto captato dai sensori, il cui accesso avviene tramite degli appositi buffer. La questione assume dunque una certa rilevanza in quanto la lettura dei dati, per andare a buon fine, deve essere regolamentata in maniera corretta.

Il primo passo che è stato fatto per venire incontro a questa esigenza è stato creare delle apposite classi che racchiudessero tutte le proprietà specifiche dei vari oggetti, sia a livello di dati che a livello grafico, in modo da separarne l'elaborazione.

Un'altra scelta progettuale è stata la creazione di un doppio thread, così da scindere il controllo dei dati video e da quelli audio; le due operazioni, svolgendosi in parallelo, accedono alle risorse del calcolatore in maniera

efficiente in accordo alle diverse modalità previste, consentendo di operare in tempo reale senza il vincolo di latenze o ritardi di sincronizzazione.

Si indicano di seguito i file di header necessari al raggiungimento di tali scopi: `NuiApi.h`, per aggregare tutti gli header relativi alle API NUI, `strsafe.h`, per manipolare correttamente i vari buffer; relativamente alla parte audio servono `propsys.h` per accedere all'interfaccia `IPropertyStore`, `dmo.h` per avere a disposizione `IMediaObject`, `wmcodecdsp.h` per configurare le proprietà del DMO, `mmreg.h` per aver accesso alla struttura `WAVEFORMATEX` e `uuids.h` per poter utilizzare la suddetta struttura; infine viene incluso `math.h` per poter svolgere alcune operazioni di tipo matematico (per il calcolo dell'energia). I metodi interessati sono:

1. `Nui_Zero()`: azzera tutte le variabili che servono per la visualizzazione dei dati.

2. `WndProc()`: offre un metodo di gestione dei messaggi pervenuti alla finestra; le tipologie possibili sono elencate di seguito:

   - `WM_INITDIALOG`, che inizializza tutti i controlli e le funzioni che modificano l'aspetto del dialog box, imposta i caratteri scelti per mostrare la frequenza di acquisizione e le istruzioni, e infine riempie i combo box con le relative opzioni;
   - `WM_SHOWWINDOW`, che viene mandato ogni volta che la finestra deve essere mostrata o nascosta; invoca il metodo `Nui_Init()`;
   - `WM_TIMER`, che governa l'acquisizione dei dati audio su base temporale, a intervalli di $20\,ms$;
   - `WM_USER_UPDATE_FPS`, che serve per l'aggiornamento della velocità di acquisizione video;
   - `WM_USER_UPDATE_COMBO`, che segnala l'aggiornamento del combo box;
   - `WM_COMMAND`, che notifica la selezione di un opzione da parte dell'utente. Dato che sono presenti due combo box attraverso cui comunicare con l'applicazione, le alternative possibili riguardano la distanza di azione (di default o ravvicinata) e la modalità di tracciamento (in piedi o seduto); viene inoltre segnalato se è stato premuto il pulsante relativo alla registrazione;

- `WM_CLOSE`, che corrisponde alla chiusura della finestra (quando viene cliccato sul pulsante rosso con la "X"); manda a sua volta il messaggio `WM_DESTROY`;
- `WM_DESTROY`, che avvisa la *procedura* che l'applicazione deve essere chiusa, inviando poi al sistema operativo un `WM_QUIT`.

3. `Nui_StatusProcThunk()` e `Nui_StatusProc()`: costituiscono la funzione di *callback* che gestisce i cambi di stato di Kinect, quando cioè viene connessa/disconnessa.

4. `Nui_Init()`: si occupa della procedura di inizializzazione del sensore secondo lo standard già visto nei precedenti esempi: bisogna subito chiarire che, nonostante la gestione dei dati sia separata, in generale l'accesso a Kinect è consentito solo a un'applicazione per volta, per cui la procedura di inizializzazione deve essere unica e comune. Il metodo crea gli eventi necessari per la gestione dei dati di tipo profondità, colore e skeleton e in seguito richiama le funzioni destinate all'allocazione delle risorse grafiche; successivamente invoca `INuiSensor::NuiInitialize()`, segnalando le quattro modalità di utilizzo del sensore: `NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX`, `NUI_INITIALIZE_FLAG_USES_SKELETON`, `NUI_INITIALIZE_FLAG_USES_COLOR` e infine `NUI_INITIALIZE_FLAG_USES_AUDIO`; dopodiché abilita lo skeletal tracking e apre i flussi relativi ai dati di tipo colore e profondità, specificandone anche la risoluzione; in ultima istanza crea un evento di stop per l'elaborazione e genera i due thread che governano l'acquisizione vera e propria.

5. `Nui_ProcessThread()`: rappresenta il punto centrale del processo di acquisizione ed elaborazione dei dati provenienti dai sensori video: invoca le funzioni relative ai vari flussi, gestisce i relativi display e calcola la velocità in fps. Le operazioni sono governate da un ciclo *while* che, attraverso il richiamo delle funzioni standard `WaitForMultipleObjects()` e `WaitForSingleObject()` impostate per operare in accordo al modello a eventi, utilizza i metodi:

   - `Nui_GotDepthAlert()`: gestisce le nuove informazioni relative alla profondità;
   - `Nui_GotColorAlert()`: gestisce le nuove informazioni ottenute sul colore;

- `Nui_GotSkeletonAlert()`: gestisce le nuove informazioni ricavate sullo scheletro e provvede al suo disegno. È all'interno di questo metodo che si è intervenuti in maniera significativa, inserendo la definizione e il controllo del gesto, con i relativi messaggi da mostrare su schermo nel momento in cui l'individuo viene identificato e riconosciuto.

6. `UpdateTrackingMode()` e `UpdateRange()`: servono a rendere effettive le scelte effettuate dall'utente tramite i combo box.

7. `UpdateTrackedSkeletonSelection()`: aggiorna la modalità dello skeletal tracking, che può essere di due tipi: *default*, cioè il classico tracciamento già visto, oppure *attivo*, che è lo status che contraddistingue una persona che si è fatta individuare mediante il gesto di riconoscimento. Nel codice 4.5 si riporta uno stralcio del codice utilizzato.

```
1  void CSkeletal::UpdateTrackedSkeletonSelection(int mode)
   {  ...
3     if (m_TrackedSkeletons==MODO_DEFAULT)
      {
5       WCHAR * szMess = L"Tornato in modalità di DEFAULT";
        SendDlgItemMessageW(m_hWnd,IDC_STATUS12,WM_SETTEXT,0,(LPARAM)
7                                                              szMess);
        WCHAR * szMessPr = L"Nessun utente attivo...";
9       SendDlgItemMessageW(m_hWnd,IDC_FPS2,WM_SETFONT,(WPARAM)
                              m_hFontFPS2,(LPARAM)szMessPr);
11      SendDlgItemMessageW(m_hWnd,IDC_FPS2,WM_SETTEXT,0,
                                   (LPARAM)szMessPr);
13      appoggio.m_BeamAngle = NULL;
      }
15
      else if (m_TrackedSkeletons==MODO_ATTIVO)
17    {
        WCHAR * szMess2 = L"Modalità TRACCIAMENTO ATTIVO";
19      SendDlgItemMessageW(m_hWnd,IDC_STATUS12,WM_SETTEXT,0,
                                          (LPARAM)szMess2);
21      WCHAR * szMessPr2 = L"Sei stato riconosciuto!";
        SendDlgItemMessageW(m_hWnd,IDC_FPS2,WM_SETFONT,(WPARAM)
23                            m_hFontFPS2,(LPARAM)szMessPr2);
        SendDlgItemMessageW(m_hWnd,IDC_FPS2,WM_SETTEXT,0,(LPARAM)
25                                                    szMessPr2);
      } ...
27 }
```

Codice 4.2: Visualizzazione dell'avvenuta identificazione di un utente.

8. `Nui_AudioThread()`: è il nodo centrale attorno a cui ruota l'intera gestione dei dati audio. Non si preoccupa dell'attivazione di Kinect in quanto tale operazione è stata precedentemente effettuata all'interno dell'altro thread, quindi provvede direttamente ad allocare le librerie e predisporre le risorse grafiche necessarie per il localizzatore e l'oscilloscopio. Facendo appoggio sulla variabile globale `appoggio` di classe `CAudio`, richiama a sua volta le seguenti funzioni:

   - `CoInitializeEx()`: alloca la libreria COM in modalità *multi-threaded*.
   - `InitializeAudioSource()`: crea, inizializza e configura l'oggetto DMO destinato all'acquisizione e al controllo dei dati audio (modalità MAP); poi predispone il formato del flusso in uscita:
     - √ formato: `WAVE_FORMAT_PCM`
     - √ numero di canali del flusso generato: 1
     - √ frequenza di campionamento: 16 KHz
     - √ velocità di trasferimento dei dati: 32 KB/s
     - √ dimensione dell'unità dati (per la sincronizzazione in riproduzione): 2 $Byte$
     - √ quantizzazione: 16 $bit/camp$

     Come già detto nel capitolo 2, la funzionalità di localizzazione della sorgente attraverso il beamforming si ottiene esclusivamente ricorrendo al DMO; si è intervenuti allora all'interno di questo metodo per operare tutte le modifiche necessarie, per le quali si rimanda al successivo paragrafo 4.5.2.
   - `SetTimer()`: manda periodicamente il messaggio `WM_TIMER`, attivando la ricezione dei dati in base a quanto stabilito in fase di progetto.

9. `Nui_ProcessAudio()`: si occupa di ottenere i nuovi dati attraverso la definizione di un ciclo *do-while* che scandisce tutto ciò che è memorizzato nel buffer (istanza della classe `CStaticMediaBuffer`). A questo punto, dopo aver scritto i dati mediante `IMediaObject::ProcessOutput()`, si può impostare l'angolo del beam in modo che segua il soggetto individuato, utilizzando la funzione `INuiAudioBeam::SetBeam()`. Infine

viene calcolata l'energia media, i cui valori costituiscono l'informazione mostrata dall'oscilloscopio.

10. `SetBeam()`: questa funzione di classe `AudioPanel` provvede ad aggiornare l'angolo che sarà disegnato sulla finestra.

11. `Nui_Update()`: consente la visualizzazione delle informazioni aggiornate sia sull'energia che sulla localizzazione.

12. `MessageBoxResource()`: manda un *message box*, un oggetto preimpostato che rientra nella categoria dei dialog box e serve per mostrare delle stringhe di testo caricate in precedenza, indicanti in questo caso errori.

13. `SetStatusMessage()`: manda un messaggio alla finestra con la segnalazione di eventuali errori; a differenza della funzione precedente, la notifica avviene esplicitamente mediante visualizzazione diretta su schermo in un'apposita barra di stato.

14. `Nui_UnInit()`: ferma il processo di acquisizione, azzera il timer, chiude i due thread, rilascia le risorse impegnate, libera le interfacce mediante `SafeRelease()` e richiama `INuiSensor::NuiShutdown()` per lo spegnimento di Kinect. Al suo interno è stata inserita anche l'invocazione di `CoUninitialize()`, per la deallocazione della libreria COM.

4.5.1 Il gesto di riconoscimento: individuazione dell'utente

Per la traduzione del gesto all'interno dell'applicazione, il modo più semplice di ottenimento è rappresentato da un ragionamento per soglie: stabiliti i punti mobili caratterizzanti, il passo seguente è prendere un riferimento superato il quale si possa considerare eseguito il gesto. Le articolazioni a disposizione sono elencate nella struttura `NUI_SKELETON_POSITION_INDEX` (si veda la tabella 4.1).

Articolazione	ID
NUI_SKELETON_POSITION_HIP_CENTER	0
NUI_SKELETON_POSITION_SPINE	1
NUI_SKELETON_POSITION_SHOULDER_CENTER	2
NUI_SKELETON_POSITION_HEAD	3
NUI_SKELETON_POSITION_SHOULDER_LEFT	4
NUI_SKELETON_POSITION_ELBOW_LEFT	5
NUI_SKELETON_POSITION_WRIST_LEFT	6
NUI_SKELETON_POSITION_HAND_LEFT	7
NUI_SKELETON_POSITION_SHOULDER_RIGHT	8
NUI_SKELETON_POSITION_ELBOW_RIGHT	9
NUI_SKELETON_POSITION_WRIST_RIGHT	10
NUI_SKELETON_POSITION_HAND_RIGHT	11
NUI_SKELETON_POSITION_HIP_LEFT	12
NUI_SKELETON_POSITION_KNEE_LEFT	13
NUI_SKELETON_POSITION_ANKLE_LEFT	14
NUI_SKELETON_POSITION_FOOT_LEFT	15
NUI_SKELETON_POSITION_HIP_RIGHT	16
NUI_SKELETON_POSITION_KNEE_RIGHT	17
NUI_SKELETON_POSITION_ANKLE_RIGHT	18
NUI_SKELETON_POSITION_FOOT_RIGHT	19

Tabella 4.1: Elenco delle articolazioni previste da Kinect con il relativo codice di identificazione.

Occorre a questo punto fare una piccola riflessione: ogni persona tende ad adattare le azioni che compie a proprio modo, anche quando si tratta di comportamenti molto comuni come ad esempio la stretta di mano, o il saluto. Anche per la semplice alzata di mano sono state osservati diversi modi di esecuzione, in particolare per quanto riguarda l'estensione del braccio (teso verso l'alto o rilassato e allargato), la posizione della mano e l'altezza del polso. Per rendere l'implementazione più efficace e al tempo stesso non stabilire una regola di realizzazione del gesto troppo vincolante, si è scelto di seguire un meccanismo basato su una doppia soglia: prendendo come giunture di riferimento quelle identificate dalle costanti 2 e 3, il gesto è riconosciuto come tale quando uno dei due gomiti oltrepassa il livello del centro delle

spalle oppure quando uno dei polsi supera l'altezza della testa (o meglio, il suo centro): per questo motivo, tali giunture sono state disegnate in rosso, in modo da risultare più evidenti nella rappresentazione video.

A questo punto, viene cambiata la variabile booleana `personaRiconosciuta` e viene invocato il metodo `UpdateTrackedSkeletonSelection()`, che contrassegna la modalità di tracciamento con l'etichetta `MODO_ATTIVO`, indicando che la persona è stata identificata. Posto che, per default, ogni individuo che entra nel campo visivo di Kinect viene contraddistinto da un ID progressivo, e che le persone possono arrivare fino a un numero massimo di sei, si è deciso di stampare su schermo le relative costanti, in modo da ottenere un identificazione univoca di tutte le figure rilevate. Si precisa da subito che, se una persona sparisce dall'inquadratura e poi ricompare, gli sarà assegnato un nuovo identificativo.

```
bool CSkeletal::Nui_GotSkeletonAlert( )
{
NUI_SKELETON_FRAME SkelFrame = {0};
for ( int i = 0 ; i < NUI_SKELETON_COUNT ; i++ )
{
 if ( trackingState == NUI_SKELETON_TRACKED )
 {
   personaRiconosciuta = false;
   utenteAttivo[i]=i+1;
   if(SkelFrame.SkeletonData[i].SkeletonPositions[...ELBOW_LEFT].y
                                >
      SkelFrame.SkeletonData[i].SkeletonPositions[...SHOULD_CENTER].y
                               ||
      SkelFrame.SkeletonData[i].SkeletonPositions[...WRIST_LEFT].y
                                >
      SkelFrame.SkeletonData[i].SkeletonPositions[...HEAD].y
                               ||
      SkelFrame.SkeletonData[i].SkeletonPositions[..ELBOW_RIGHT].y
                                >
      SkelFrame.SkeletonData[i].SkeletonPositions[...SHOULD_CENTER].y
                               ||
      SkelFrame.SkeletonData[i].SkelPositions[...WRIST_RIGHT].y
                                >
      SkelFrame.SkeletonData[i].SkeletonPositions[...HEAD].y)
    {
      personaRiconosciuta = true;
    }
    Nui_DrawSkeleton(SkelFrame.SkeletonData[i],width,height);
    if (personaRiconosciuta == true)
    {
      UpdateTrackedSkeletonSelection( MODO_ATTIVO );
```

```
        ...
33      HRESULT hr=m_pNuiSensor->NuiSkeletonSetTrackedSkeletons(
            memoriaID);
    }
35  }
}
37    ...
}
```

Codice 4.3: Frammento sul riconoscimento del gesto.

A seguito dell'attivazione di un individuo, non vengono più presi in considerazione i dati relativi all'altra persona tracciata: il gesto funziona come un vero e proprio switch, per cui l'eventuale altro scheletro presente sul monitor viene cancellato e rimane solo un pallino che ne traccia la posizione; di conseguenza, tutte le informazioni visualizzate, come ad esempio le coordinate dei gomiti o l'angolo della testa, saranno sempre riferite correttamente allo scheletro riconosciuto. Per ottenere questa funzionalità, bisogna abilitare nuovamente lo skeletal tracking invocando la funzione `INuiSensor::NuiSkeletonTrackingEnable` con il nuovo flag `NUI_SKELETON_TRACKING_FLAG_TITLE_SETS_TRACKED_SKELETONS`. Successivamente, si memorizza l'ID della persona che ha effettuato il gesto nel vettore `memoriaID` e si ricorre alla funzione `INuiSensor::NuiSkeletonSetTrackedSkeletons`, che rende effettivo quanto detto. In questo modo si garantisce che l'utente attivo verrà inseguito per tutto il tempo che rimarrà all'interno del campo visivo delle telecamere.

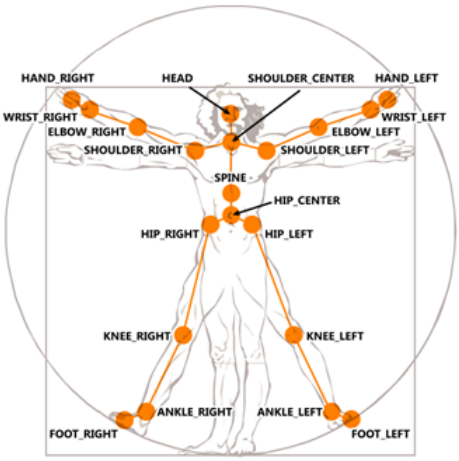

Figura 4.3: Distribuzione delle 20 articolazioni lungo il corpo.

Un ulteriore accorgimento è stato preso riguardo l'evento di reimpostazione delle condizioni di default. Si preferisce chiarire il tutto con un esempio pratico. Si consideri il caso in cui siano presenti tre persone che partecipano all'azione: in base al funzionamento generale, alle prime due che sono identificate come tali da Kinect sarà associato uno scheletro, mentre della terza verranno tracciati solo gli spostamenti del relativo centro di massa. Nel momento in cui uno dei due scheletri alza la mano, verrà contraddistinto come utente attivo, facendo sparire il secondo scheletro. Il problema che si pone è come ristabilire le condizioni iniziali, così da rendere attivabili gli altri individui rilevati quando il primo ha terminato. La scelta è ricaduta sull'uscita dall'inquadratura da parte del soggetto attivo: una volta che la persona ha finito l'interazione, basta che esca dal campo visivo per ristabilire la situazione di partenza, così che si dia la possibilità alle altre figure presenti di diventare attive a loro volta.

4.5.2 Impostazione del beam

Una volta che il gesto è stato effettuato e il parlatore è stato correttamente riconosciuto, è necessario che i microfoni inseguano la persona a seconda dei movimenti compiuti. Attraverso un orientamento virtuale dei trasduttori, si vuole infatti cercare di escludere tutte le possibili fonti di interferenza provenienti dalle altre voci. Per poter impostare correttamente il beam in corrispondenza della posizione in cui si trova il soggetto attivo, è stato necessario effettuare alcune variazioni al DMO, in modo da sfruttare adeguatamente le funzionalità offerte dal voice-capture DSP (che è già stato presentato nel paragrafo 2.2). La funzione di interesse in questo caso è `InitializeAudioSource()`. L'obiettivo si raggiunge utilizzando la struttura `PROPVARIANT`, attraverso cui si può accedere ad alcuni attributi di interesse; in particolare, dopo aver specificato di voler utilizzare tutti e quattro gli elementi dell'array tramite `MFPKEY_WMAAECMA_SYSTEM_MODE`, è bastato cambiare le proprietà `MFPKEY_WMAAECMA_FEATURE_MODE` e `MFPKEY_WMAAECMA_FEATR_MICARR_MODE`, impostandone i valori rispettivamente a `VARIANT_TRUE` e `MICARRAY_EXTERN_BEAM`, abilitando così la correzione dei parametri. Si riporta di seguito il frammento di codice che opera le suddette modifiche.

```
PROPVARIANT pvSysMode, pvMicFeature, pvMicArrMode;
PropVariantInit(&pvSysMode);     // inizializzo 'pvSysMode'
pvSysMode.vt = VT_I4;
pvSysMode.lVal = (LONG)(2);
hr=m_pPropertyStore->SetValue(MFPKEY_WMAAECMA_SYSTEM_MODE,pvSysMode);
PropVariantClear(&pvSysMode);

PropVariantInit(&pvMicFeature);     // inizializzo 'pvMicFeature'
pvMicFeature.vt = VT_BOOL;
pvMicFeature.boolVal = VARIANT_TRUE;
hr=m_pPropertyStore->SetValue(MFPKEY_WMAAECMA_FEATURE_MODE,
                                          pvMicFeature);
PropVariantClear(&pvMicFeature);

PropVariantInit(&pvMicArrMode);     // inizializzo 'pvMicArrMode'
pvMicArrMode.vt = VT_I4;
pvMicArrMode.lVal = (LONG) MICARRAY_EXTERN_BEAM;
hr=m_pPropertyStore->SetValue(MFPKEY_WMAAECMA_FEATR_MICARR_MODE,
                                              pvMicArrMode);
PropVariantClear(&pvMicArrMode);
```

Codice 4.4: Modifica delle proprietà del DMO.

Dopo aver impostato correttamente i vari attributi, non rimane che stabilire quale sia il punto migliore da prendere come riferimento per l'indirizzamento del beamformer. Dato che la voce proviene dalla bocca, la scelta è naturalmente ricaduta sulla giuntura più vicina ad essa, vale a dire la testa. Di conseguenza, l'ultima operazione da svolgere consiste nel ricavare l'angolo della testa del parlatore. Si è creata allora un'apposita funzione chiamata `CalcolaHeadAngle()` che accede alle relative coordinate nel piano (x,z) e ne calcola l'angolo corrispondente. Tale valore viene poi convertito in gradi e passato alla funzione `INuiAudioBeam::SetBeam()`.

4.6 La funzione di registrazione

La codifica WAVE rappresenta uno dei più importanti formati attualmente disponibili per file audio. Sviluppato in collaborazione da Microsoft e IBM, si basa sul più generico Interchange File Format (IFF) – un formato nato a metà degli anni ottanta per immagazzinare file di diverso tipo in modo univoco, così che potessero essere riutilizzati su più piattaforme – in particolare per quanto riguarda la suddivisione del file in blocchi (*chunk*). I dati sono memorizzati secondo la notazione *little endian*, procedendo dal byte meno significativo

a quello più significativo, e la struttura prevede la presenza di almeno tre blocchi: *RIFF*, *fmt* e *DATA* (più altri eventuali blocchi opzionali); il primo è uno standard che deriva direttamente dall'IFF e identifica il file come WAVE, il secondo contiene i parametri di codifica che descrivono la forma d'onda (frequenza di campionamento, quantizzazione, etc.), mentre il terzo racchiude i dati veri e propri.

Per consentire la registrazione di quanto prelevato dai microfoni in formato WAVE, si è scelto di utilizzare la classe definita nell'esempio *Audio Explorer*, servendosi delle semplici funzioni scritte al suo interno. Non sono state apportate modifiche sostanziali al codice in questione, per cui i relativi file `WaveWriter.cpp` e `WaveWriter.h` ricalcano i nomi originali. Si riporta di seguito la composizione ordinata della struttura:

- 8 *byte* di RIFF header
- 4 *byte* di WAVE header
- 4 *byte* di fmt header
- 4 *byte* di fmt size
- 8 *byte* di DATA header
- *x byte* di dati WAVE

Dal punto di vista grafico, si è scelto di ricorrere all'utilizzo di un pulsante che svolgesse questo compito, in quanto sembrava l'opzione più adeguata da offrire all'utente. La pressione del bottone provoca l'invocazione del metodo `RecordAudio()`, che avvia la registrazione servendosi a sua volta di `RecordAudioWave()`: innanzitutto viene ricavato il nome del file, secondo la convenzione "RegistrazioneKinectAudio-HH-MM-SS.wav"; in secondo luogo si scrive l'header del file in modo che venga indicato il corretto punto da cui cominciare; dopodiché inizia l'acquisizione vera e propria; le ultime operazioni consistono nel cambiare il valore della variabile `m_fileRecording` e modificare quanto scritto sul pulsante, da *Registra* a *Stop recording*, in modo da poterne utilizzare uno solo per entrambe le azioni. Una seconda pressione genera la chiamata delle funzioni `StopRecording()` e `StopRecordingAudioWave()`, cui sono affidate l'interruzione della registrazione, la correzione dell'header in base all'effettiva dimensione del file e il ritorno alla condizione di default.

4.7 Considerazioni pratiche

Lo skeletal tracking consente in maniera piuttosto semplice e automatica il riconoscimento e l'inseguimento delle persone: attraverso il sensore IR, l'applicazione localizza le articolazioni principali dell'utente nello spazio di interazione e ne traccia il movimento nel tempo. Non è necessario intraprendere nessuna azione di calibrazione o fare alcun gesto particolare per essere riconosciuti, basta posizionarsi davanti al sensore cercando di essere visibili almeno per quanto riguarda la testa e la parte superiore del corpo.

Nel caso di utilizzo in modalità "seduto", invece, potrebbe convenire muoversi con il busto leggermente in avanti o favorire il riconoscimento con un leggero movimento delle braccia, in quanto si presuppone che lo schienale della sedia sia molto vicina all'utente, e ciò potrebbe costituire motivo di confusione durante l'operazione di tracciamento.

Qualche problema potrebbe verificarsi se il soggetto si trova messo di profilo, o comunque ruotato rispetto al piano parallelo all'inquadratura, in quanto questa situazione potrebbe creare qualche difficoltà nella predizione delle parti del corpo che non risultano completamente inquadrate. È prevista una procedura per il calcolo dell'orientamento corrispondente alle varie ossa, basato su una classificazione delle articolazioni che incomincia con le anche e finisce con le mai, con l'assunto che ciascun osso risulta definito in relazione alle due giunture circostanti, una di ordine superiore e una di ordine inferiore. La rotazione corrispondente è dunque ricavata considerando l'articolazione gerarchicamente più importante, che viene presa come riferimento per gli assi cartesiani a partire dal quale viene effettuato il calcolo. Tale metodo è stato pensato principalmente per applicazioni di computer graphics (come ad esempio l'*avateering*).

Per quanto riguarda la distanza di interazione, i range previsti sono il *default mode* e il *near mode*; l'azienda produttrice dichiara i seguenti spazi:

- modalità di default: $0,8 - 4,0\,m$
- modalità "ravvicinata": $0,4 - 3,0\,m$

Stante che l'utente dovrà utilizzare le proprie braccia, si consiglia un utilizzo nell'ambito di $1,2 - 3,5\,m$. Non si sono riscontrati particolari limiti invece per quanto riguarda la distanza minima a cui si devono trovare due o più persone per essere distinte.

Anche per ciò che concerne la tipologia di tracciamento, le modalità offerte sono di due tipi, *default mode* e *seated mode*:

- modalità di default: viene tracciato lo scheletro in relazione a tutte e 20 le articolazioni previste; corrisponde a un utilizzo in posizione eretta
- modalità "seduto": il numero di articolazioni viene dimezzato, in quanto viene rappresentata solo la parte superiore del corpo (spalle, braccia e testa). Dal momento che Kinect è stata progettata per effettuare un riconoscimento di tipo *full-body*, la modalità "seduto" opera di conseguenza, comportando un maggiore impegno a livello di risorse del calcolatore.

Ad ogni modo, per rendere più intuitivo l'utilizzo dell'applicazione senza dover consultare la scheda tecnica del sensore o dover studiare nel dettaglio lo schema di funzionamento dell'algoritmo, sono stati presi degli accorgimenti per cercare di migliorarne l'usabilità generale. Innanzitutto la finestra mostrata durante l'esecuzione del programma è stata concepita cercando di dare maggior rilievo al riquadro in cui appare lo scheletro, aumentandone le dimensioni rispetto agli altri riquadri presenti.

In secondo luogo, per rendere agevole l'individuazione da parte dell'utente dell'avvenuta identificazione, si è scelto di utilizzare dei meccanismi di feedback visivo: una volta riconosciuti, infatti, viene cambiato colore allo scheletro, che da verde diviene dorato; in aggiunta, viene mandato un messaggio che conferma l'avvenuto riconoscimento.

Successivamente, una parte della finestra è stata riservata alla visualizzazione di alcune istruzioni sul funzionamento dell'applicazione. A ciò si aggiunga la presenza di numerose aree in cui viene segnalato se tutte le funzioni stanno agendo correttamente o se si sono verificati dei problemi.

Infine, il pulsante per incominciare e terminare la registrazione è unico, e cambia la sua etichetta dopo che viene schiacciato; di fianco al bottone è stato inserito un rettangolo in cui vengono fornite anche delle informazioni indicative dell'esito positivo della stessa.

In conclusione, la speranza è che queste piccole accortezze possano risultare in un utilizzo agevole e allo stesso tempo intuitivo, cercando di estendere quanto più possibile il bacino di utenza dell'applicazione, anche nell'ottica di eventuali sviluppi futuri.

4.7.1 Calcolo del campo visivo di Kinect

Per avere un'idea dettagliata dell'ambiente di interazione e delle relative dimensioni, è necessario calcolare le lo spazio più grande all'interno del quale il dispositivo riesce a tracciare per intero lo scheletro dei soggetti partecipanti.

Secondo quanto fornito dalla società di Redmond, l'altezza ideale di posizionamento del dispositivo varia tra i 60 e i 180 cm. Si precisa fin da ora che tale intervallo è solo indicativo per un uso ottimale del sensore, in particolare per un utilizzo in ambito di videogioco. Bisogna inoltre ricordare che Kinect è dotata di un motore interno che consente di regolarne "manualmente" l'inclinazione, accessibile via software, cosicché ne risulti ampliato l'angolo visuale, consentendo una maggiore libertà di collocazione.

Ricordando quanto affermato durante la trattazione delle specifiche tecniche, l'ampiezza del campo visivo in orizzontale è di $57,5°$, mentre in verticale è di $43,5°$: ne segue che, considerando il triangolo isoscele in cui gli angoli alla base valgono $61,25°$ ciascuno, la larghezza massima raggiungibile per garantire il riconoscimento (in via teorica) è di $4,38\,m$ per la modalità di default e $3,29\,m$ per quella ravvicinata. Discorso analogo si può fare per l'altra dimensione (gli angoli alla base varranno questa volta $68,25°$): l'altezza massima è, rispettivamente, di $3,19\,m$ e $2,39\,m$. Si considerano solo i limiti sul campo del visivo e non quelli dell'uditivo in quanto questi ultimi riguardano solo il piano orizzontale e risultano meno stringenti (si ricorda che l'ampiezza complessiva nominale è di $100°$).

Dal momento che il gesto prevede l'alzata della mano, si vuole anche caratterizzare la distanza minima di interazione. Fermo restando che non ci si può avvicinare oltre $0,40\,m$, per garantire che un individuo, alto mediamente $1,80\,m$, cui si aggiungono circa $0,5\,m$ per il braccio, venga seguito correttamente, posizionando Kinect esattamente a metà altezza (circa $1,15\,m$) la distanza minima è di $2,88\,m$. Naturalmente questi dati sono solo indicativi, in quanto spostando il sensore più in alto o in basso si può cambiare notevolmente il campo visivo, e sono relativi alla visualizzazione completa dello scheletro, mentre non ci sono distanza minime in caso di utilizzo da seduti in modalità ravvicinata. Comunque sia, per limitare i vincoli dovuti all'altezza, è già stato discusso riguardo il meccanismo di riconoscimento del gesto, in base al quale basta raggiungere il livello della testa con la mano per attivare il tracciamento, venendo così incontro a quest'ultima esigenza (la distanza minima diventa di $2,26\,m$).

4.8 Misurazioni e test di funzionamento

La parte finale del capitolo è dedicata alla presentazione dei test svolti per verificare i limiti di funzionamento e l'efficacia di quanto finora descritto, in modo da offrire una visione più critica della questione e al tempo stesso focalizzarsi su quali siano le reali potenzialità del lavoro svolto. Per quanto i criteri di valutazione utilizzati non siano estremamente rigorosi e formali, si è cercato comunque di offrire un approccio che fosse il più vicino possibile al metodo scientifico tradizionale.

Per quanto riguarda le misurazioni, è stata concessa la possibilità di utilizzare uno dei laboratori dell'Istituto Superiore delle Comunicazioni e delle Tecnologie dell'Informazione (ISCTI), situato presso il Dipartimento per le Comunicazioni del MINISTERO DELLO SVILUPPO ECONOMICO. In particolare, è stata utilizzata la camera semi anecoica a loro disposizione. Come sorgenti sono state impiegate due registrazioni di un segnale vocale, una maschile (29 anni di età) e una femminile (26 anni di età). L'acquisizione è avvenuta con il microfono integrato in una cuffia per videoconferenze Plantronics mod. Audio 625. Come programma di acquisizione è stato utilizzato il software open source Audacity®; la registrazione è avvenuta utilizzando la massima frequenza di campionamento possibile ($48\,000\,Hz$), $16\,bit$ di quantizzazione, un singolo canale, agendo manualmente sui volumi di ingresso (livello fissato al 68%), in modo da sfruttarne la sensibilità cercando di limitarne la saturazione e il *clipping*. I due segnali sono poi stati convertiti in WAVE, in modo da non subire perdite di informazione durante la codifica. Per mantenere una certa fluidità e limitare le possibili pause dovute all'improvvisazione, la voce femminile legge un estratto da un libro, mentre quella maschile ripete per i primi 12 secondi una stessa parola, per poi leggere un articolo di giornale: l'intento era quello di cercare di ricreare delle caratteristiche di periodicità per valutare le effettive capacità del beamforming in relazione a due situazioni diverse, una meno spontanea e l'altra più naturale. Le registrazioni sono poi state normalizzate con il Root Mean Square (RMS) e sono state inserite nei canali destro e sinistro di un unico file stereo, in modo da garantire uno stesso volume di riproduzione. Gli andamenti dei due segnali sono riportati nella figura 4.4.

L'esperimento si è strutturato nel seguente modo: le sorgenti sono state posizionate su una base di appoggio di altezza pari a circa $90\,cm$, e sono state

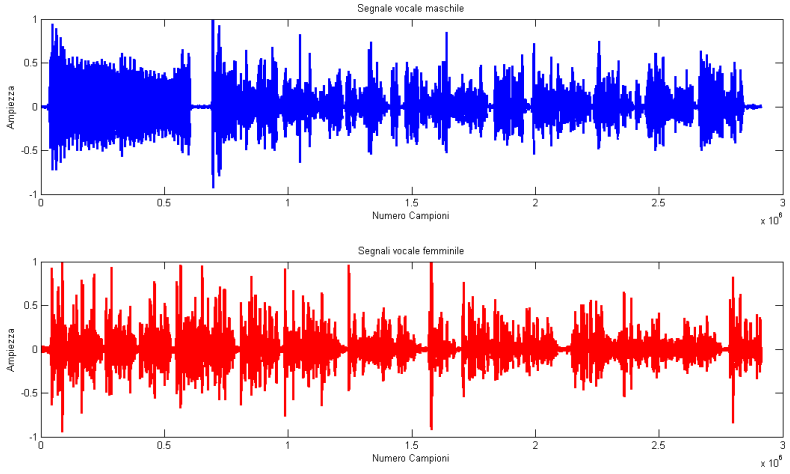

Figura 4.4: Segnali vocali utilizzati come test. Il blu è usato per la voce maschile, il rosso per quella femminile.

collocate in modo che fossero equidistanti rispetto a Kinect. Le misurazioni effettuate sono state quattro:

- prova 1: $1,50\,m$ di distanza, sorgente maschile a $-30°$, sorgente femminile a $+20°$
- prova 2: $1,50\,m$ di distanza, sorgente maschile a $-10°$, sorgente femminile a $+10°$
- prova 3: $2,80\,m$ di distanza, sorgente maschile a $-20°$, sorgente femminile a $+10°$
- prova 4: $2,80\,m$ di distanza, sorgente maschile a $-10°$, sorgente femminile a $+10°$

Si precisa che nella prova 3 non si è potuta ripetere la separazione impostata nella prova 1 a causa della limitata lunghezza del cavo di collegamento tra i due altoparlanti.

Per la riproduzione sono stati utilizzati due speaker Creative mod. SBS 270. L'esperimento si è svolto usando un notebook HP 550 con processore Intel Celeron ($2.00\,GHz$), $2\,GB$ di RAM e sistema operativo Windows 7 a $32\,bit$. Essendo il processore un pò debole per la gestione completa dell'applicazione,

durante la simulazione è stata esclusa la parte video per limitare il consumo di risorse.

In base a quanto dichiarato ufficialmente[2], il guadagno dell'array è subottimo, con particolare rilievo per le distanze ravvicinate: anche per le registrazioni con Kinect, allora, è stato fissato un livello per i microfoni pari al 68%.

Per la valutazione dei risultati ci si è affidati a due criteri: uno più soggettivo, l'altro basato sulla quantificazione di una grandezza effettiva. Il primo metodo prende spunto dalla tecnica usata in ambito telefonico nota come Perceptual Evaluation of Speech Quality (PESQ): il giudizio su un modello è espresso sulla base della qualità *percepita* dall'orecchio di una persona. Sono state scelte le due registrazioni ritenute più significative, la prova 1 e la prova 3, e sono state fatte ascoltare da 8 soggetti maschi adulti, chiedendo loro di giudicare la bontà della riduzione della voce disturbante a favore di quella principale, secondo una scala che va da 1 (pessimo) a 5 (ottimo). I giudizi espressi sono riportati in tabella 4.2.

Individuo	Valutazione (1-5)
soggetto 1	3
soggetto 2	3
soggetto 3	4
soggetto 4	3
soggetto 5	2
soggetto 6	3
soggetto 7	4
soggetto 8	2

(a) Distanza di 1,5 metri

Individuo	Valutazione (1-5)
soggetto 1	3
soggetto 2	2
soggetto 3	2
soggetto 4	3
soggetto 5	2
soggetto 6	3
soggetto 7	1
soggetto 8	2

(b) Distanza di 2,8 metri

Tabella 4.2: Valutazione dell'applicazione su base percettiva.

Per il secondo criterio, si è mutuato dalle valutazioni in ambito comunicazioni wireless il concetto di Signal-to-Interference Ratio (SIR), definito come rapporto tra le energie del segnale principale e di quello interferente, espresso in *decibel*. Utilizzando Matlab per effettuare le relative valutazioni, si riportano i risultati in tabella 4.3. Per cercare di dare un raffronto con

[2] Si veda la pagina `http://msdn.microsoft.com/en-us/library/jj663798.aspx`.

una situazione al di fuori della camera semi anecoica e avvicinarsi a un caso più reale, gli esperimenti 1 e 2 sono stati ripetuti all'interno del Laboratorio di Multimedialità presente al Dipartimento di Ingegneria dell'Informazione, Elettronica e Telecomunicazioni (DIET), situato presso la Facoltà di Ingegneria (Sapienza – Università di Roma).

Prova	SIR_u [dB]	SIR_d [dB]
prova 1	16.5696	19.2335
prova 2	22.3394	18.4586
prova 3	11.1185	11.8210
prova 4	14.0753	12.6510
prova 5	10.9676	9.7590
prova 6	13.8922	11.3002

Tabella 4.3: Valutazione mediante Signal-to-Interference Ratio.

Entrambi i criteri mostrano dei risultati che mediamente possono essere ritenuti soddisfacenti per quanto riguarda la distanza minore, in particolare quando le sorgenti sono maggiormente separate. La valutazione soggettiva mostra in generale dei giudizi positivi sulla qualità dell'operato, mentre i valori ottenuti mediante il SIR, pur mantenendo un trend decrescente con la distanza, mostrano risultati discordanti rispetto al caso percettivo, tendendo a favore del caso di voce maschile nelle situazioni di sorgenti vicine, a favore della voce femminile quando le sorgenti sono più separate. Ciò premesso, le misurazioni si attestano su valori piuttosto bassi, di poco superiori ai $10\,dB$.

Come secondo test si è misurato il tempo reale impiegato dal calcolatore per riconoscere il gesto. Per questa simulazione e per la successiva, si è usata una macchina dotata di processore Intel Core i7-2600 ($3.40\,GHz$), $16\,GB$ di RAM e sistema operativo Windows 7 a $64\,bit$. Come prima cosa, sono state definite due variabili di tipo `clock_t`, una indicante l'inizio e l'altra la fine dell'intervallo; entrambe sono poi state adoperate per calcolare il tempo necessario all'algoritmo – una volta che la persona viene visualizzata sullo schermo – affinché alzare la mano venga riconosciuto come il gesto di attivazione. Per far ciò, è stata invocata la funzione `clock()` all'interno del metodo `Nui_GotSkeletonAlert()`, immediatamente dopo che viene disegnato lo scheletro. Ciò che viene fatto non è altro che la quantificazione del

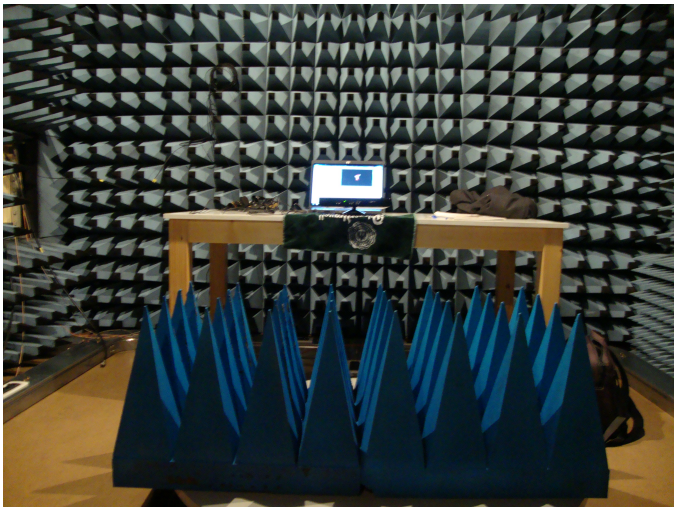

Figura 4.5: L'interno della camera semi anecoica.

tempo trascorso dall'inizio del processo. La misurazione è stata ripetuta diverse volte, stampando su schermo i risultati. I valori osservati oscillano tra 1 e 4 millisecondi (essendo molto brevi, è difficile spesso analizzare nel dettaglio le variazioni), sia in modalità normale che in modalità seduto. I tempi sembrano estremamente bassi, motivo per cui non si ritiene che ciò possa influenzare in qualche modo le prestazioni dell'applicazione. Non sono stati misurati i tempi delle funzioni più generali di tracciamento in quanto non sono state effettuate importanti modifiche su di esse.

Il terzo test è stato effettuato per valutare la capacità di identificazione del gesto, con l'obiettivo di mettere in evidenza quante volte l'operazione avvenisse con successo. Essendo il gesto molto semplice – si dovrebbe più propriamente parlare di una posa o di un semplice movimento – i risultati sono molto soddisfacenti. Il gesto è stato ripetuto 50 volte alle distanze di 1, 2 e 3 m (da seduto). La qualità del riconoscimento è eccellente in tutti e 3 i casi se il movimento è effettuato in maniera completa e in posizione eretta, con la schiena staccata dall'eventuale schienale della sedia; la quantità di tempo che si impiega per compierlo non è rilevante. L'unica fonte di confusione traspare nel caso in cui la postura sia meno corretta, magari inclinata o sbilanciata verso un lato: solo in questa situazione, l'algoritmo mostra qualche occasionale forma di decadimento delle prestazioni. Una circostanza – indipendente dal lavoro svolto – che può creare difficoltà è l'eventuale latenza che si può

presentare nell'inseguimento di una delle giunture tra testa, polsi e gomiti: se queste vengono tracciate con ritardo, oppure in presenza di disturbi interframe, si può correre il rischio che si perda il momento in cui è stato effettuato il movimento chiave. Ciò comporta che, inoltre, all'aumentare della distanza si corre il rischio che lo skeletal tracking vada in confusione: fortunatamente, una volta tracciato lo scheletro, anche se alcune ossa in qualche frame vengono dedotte, l'algoritmo opera molto velocemente nel recupero delle informazioni più importanti.

Conclusioni e sviluppi futuri

Il lavoro svolto è stato effettuato con lo scopo di progettare e produrre un'applicazione che potesse essere utilizzata su un qualsiasi personal computer. L'installazione dei driver ufficiali Microsoft ha consentito l'accesso a tutti i sensori che compongono Kinect, in maniera piuttosto rapida e funzionale. Gli esempi a corredo sono stati un ottimo punto di partenza per capire a fondo come accedere ai vari flussi di dati, come manipolarli e che tipo di operazioni svolgere. A questa parte si aggiunga l'immensa quantità di contenuti che offre la rete in merito a possibili utilizzi del dispositivo: video dimostrativi, forum di discussione e documentazioni più o meno ufficiali hanno offerto uno spunto di riflessione su quello che concretamente volesse dire ragionare in termini di Natural User Interface. Su queste basi, allora, è stato pensato un possibile impiego delle informazioni prelevate, scegliendo di orientarsi su una situazione di tipo videoconferenza.

Gli scenari di utilizzo ideali potrebbero essere tutti quelli in cui è prevista la presenza di più persone contemporaneamente, in ambiente più o meno rumoroso: oltre al già citato impiego per videoconferenze, si potrebbe pensare a una funzione di realtà aumentata all'interno di un museo: si potrebbe posizionare Kinect sopra un quadro o una scultura per fornire ad esempio una descrizione dell'opera, oppure un breve riassunto della corrente artistica di riferimento dell'autore. Ancora meglio, si potrebbe pensare a un'interazione più creativa, in cui il visitatore interagisce attivamente con l'opera d'arte e la modifica, avvicinandosi molto al concetto di video installazione. Un altro ambito molto interessante potrebbe essere quello medico. Ormai tutte le sale operatorie sono dotate di computer e proiettori, ma si pensi all'importanza di mantenere un ambiente totalmente asettico; anche la semplice pressione di un tasto del mouse o della tastiera potrebbe compromettere la sterilità del luogo. Con Kinect, l'interazione potrebbe avvenire sfruttando semplicemente le

proprie parti del corpo e la propria voce, senza necessità di toccare alcunché.

Il codice prodotto è corposo, e le funzioni che compongono il programma sono abbastanza numerose: fortunatamente è stato possibile riutilizzare molti dei metodi forniti dai vari esempi, anche se l'originalità del lavoro svolto è risieduta proprio nel riuscire a creare una struttura unica che richiamasse in maniera organica le varie funzioni, data la mancanza progetti del genere.

I risultati finali sono molto soddisfacenti nel complesso: l'algoritmo funziona in maniera efficiente e non sembrano visibili particolari intoppi. Sicuramente alcune considerazioni vanno fatte sul numero di individui che possono partecipare all'azione: due scheletri tracciabili sono pochi, ma il solo fatto di averne più di uno garantisce di per sé un meccanismo di discriminazione delle persone. Va poi precisato che, dovendo seguire i movimenti di ogni articolazione di un corpo e, successivamente, dovendone tracciare le relative ossa, quest'operazione richiede l'impegno di molte risorse del calcolatore, quindi non si esclude che, con l'evolversi della tecnologia, a breve sarà possibile tracciare molte più figure contemporaneamente, aprendo nuovi sviluppi in questo senso. Si vuole inoltre sottolineare come i limiti attuali derivino più dal numero di individui che fisicamente possono entrare nel campo visivo che da aspetti di natura tecnica.

Un limite di natura più concettuale è costituito dal fatto che, nel cercare di offrire un certo numero di opzioni per consentire l'adattamento a diversi scenari di interazione, non si è potuto tenere in considerazione che ogni scelta opera come un interruttore (*switch*). La principale conseguenza è che viene consentita la selezione di una sola modalità di tracciamento per volta, il che comporta che tutte le figure potenzialmente attive siano dello stesso tipo (tutte in piedi o tutte sedute). Questo potrebbe generare difficoltà ad esempio in caso di presenza di utenti su sedia a rotelle. Una possibile soluzione allora potrebbe essere l'utilizzo di 2 Kinect, ciascuna impostata su una ben precisa modalità, aprendo la porta a nuovi scenari di utilizzo anche per il beamforming, dal punto di vista dell'estensione del campo di azione dei microfoni oppure del passaggio ad un dominio bidimensionale.

Tra gli sviluppi futuri, sicuramente si può pensare di affiancare un algoritmo di separazione delle sorgenti o Blind Source Separation (BSS)(come suggerito ad esempio in [17]), adoperando ad esempio un'analisi delle componenti indipendenti (Independent Component Analysis (ICA)) o un meccanismo di studio della DOA del suono. Oppure si potrebbe implementare da zero

un'altra forma di beamforming, andandosi a calcolare una nuova matrice di pesi con cui gestire i ritardi dei singoli microfoni.

Comunque sia, l'avvento di Kinect ha prodotto una piccola rivoluzione nel settore, e il processo di diffusione di nuovi prototipi progettati sui suoi fondamenti è in continuo aumento. La speranza è quella di riuscire ad approfondire l'argomento e continuare nello sviluppo in questa direzione, contribuendo attivamente alla crescita di questi nuovi scenari.

Riferimenti bibliografici

[1] A. Davison, "Using the kinect's microphone array," Marzo 2012.

[2] MicrosoftCorporation, "Human interface guidelines," 2012.

[3] MicrosoftCorporation, "Microphone array support in windows vista," Agosto 2005.

[4] I. Tashev and H. S. Malvar, "A new beamformer design algorithm for microphone arrays," *ICASSP*, 2005.

[5] MicrosoftCorporation, "How to build and use microphone arrays for windows vista," Febbraio 2012.

[6] Z. Zalevsky, A. Shpunt, A. Maizels, and J. Garcia, "Method and system for object reconstruction," *Patent by PrimeSense Ltd.*, Aprile 2007.

[7] J. Garcia and Z. Zalevsky, "Range mapping using speckle decorrelation," *Patent by PrimeSense Ltd.*, Ottobre 2008.

[8] B. Freedman, A. Shpunt, M. Machline, and Y. Arieli, "Depth mapping using projected patterns," *Patent by PrimeSense Ltd.*, Maggio 2010.

[9] M. Maisto, "Sviluppo di algoritmi per il riconoscimento di movimenti gestuali fini mediante tecnologia kinect," Luglio 2012.

[10] MicrosoftCorporation, "Programming guide – getting started with the kinect for windows sdk from microsoft research," Luglio 2011.

[11] MicrosoftCorporation, "Audiocaptureraw walkthrough: C++ (capturing the raw audio stream)," 2011.

[12] MicrosoftCorporation, "Micarrayechocancellation walkthrough: C++ (capturing audio streams with acoustic echo cancellation and beamforming)," 2011.

[13] I. Tashev, "Gain self-calibration procedure for microphone arrays," *ICME*, 2004.

[14] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake, "Real-time human pose recognition in parts from single depths images," 2005.

[15] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake, "Real-time human pose recognition in parts from single depths images: Supplementary material," 2005.

[16] MicrosoftCorporation, "Skeletalviewer walkthrough: C++ and c# (capturing data with the nui api)," 2011.

[17] L.-H. Kim, I. Tashev, and A. Acero, "Reverberated speech signal separation based on regularized subband feedforward ica and instantaneous direction of arrival," *ICASSP*, 2010.

[18] H. Schildt, *La guida completa C++*. 1999.

[19] KinectForWindows, "http://www.microsoft.com/en-us/kinectforwindows." [Dicembre 2012].

[20] MicrosoftDeveloperNetwork, "http://msdn.microsoft.com/en-us/library/hh855347.aspx." [Gennaio 2013].

[21] KinectForWindowsSDKForums, "http://social.msdn.microsoft.com/forums/en-us/category/kinectsdk." [Dicembre 2012].

[22] Channel9, "http://channel9.msdn.com/series/kinectsdkquickstarts/audio-fundamentals." [Ottobre 2012].

[23] ISCTI, "http://www.isticom.it." [Gennaio 2013].

[24] cplusplus.com, "http://www.cplusplus.com/doc/tutorial." [Dicembre 2012].

Appendice A

Guida all'installazione

Anche se l'installazione dell'SDK risulta abbastanza automatizzata, di seguito se ne riporta una breve guida che illustra la corretta procedura.

A.1 Installazione dei driver ufficiali

Una volta scaricati i file necessari dal sito `http://www.microsoft.com/en-us/kinectforwindows/develop/developer-downloads.aspx`, si può poi provvedere all'installazione, nell'ordine, prima dell'SDK e poi del Developer Toolkit. A conferma del fatto che l'installazione è avvenuta con successo, basta controllare che nella finestra *Gestione Dispositivi* di Windows 7 compaiano: sotto la voce "Microsoft Kinect" gli elementi *Audio Array Control*, *Camera*, e *Security Control*, sotto la voce "Controller audio, video e giochi" *Kinect Audio USB* e sotto "Controller USB" *Dispositivo USB composito*, come appare evidente nella figura A.1.

A.2 Modifiche al progetto Visual Studio

Per poter scrivere una qualsiasi forma di software e quindi sfruttare appieno tutte le caratteristiche offerte da Kinect tramite l'utilizzo di un linguaggio di programmazione, è previsto l'utilizzo di Microsoft Visual Studio 2010, l'ambiente di sviluppo ufficiale o Integrated Development Environment (IDE) dell'azienda di Redmond.

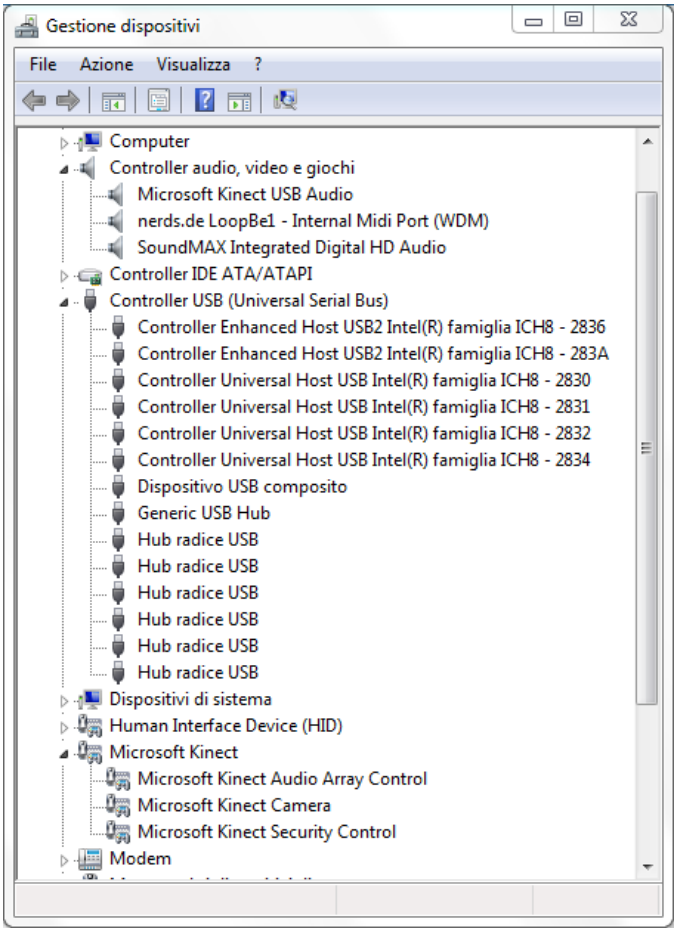

Figura A.1: Elementi visualizzati a seguito dell'installazione dell'SDK.

È opportuno mettere in rilievo che, per poter importare correttamente le librerie e quindi accedere alle varie funzionalità dell'API, è necessario impostare alcune modifiche prima di procedere con la compilazione del progetto VS2010. La procedura viene riportata di seguito: si seleziona il menù "Progetto→Proprietà", si va poi nella sezione "Proprietà di configurazione→Directory di VC++" e si inseriscono `$(KINECTSDK10_DIR)\inc` alla voce "Directory di inclusione" e `$(KINECTSDK10_DIR)\lib\x86` alla voce "Directory librerie"; infine, si aggiungono le librerie `Kinect10.lib`, `Msdmo.lib`, `dmoguids.lib`, `amstrmid.lib`, `gdi32.lib` e `gdiplus.lib` alle "Dipendenze aggiuntive" del *Linker* (sotto la voce Input).

Appendice B

Codice sorgente: i metodi principali

B.1 Gestione dei messaggi mandati alla finestra

```
LRESULT CALLBACK CSkeletal::WndProc( HWND hWnd, UINT message, WPARAM
    wParam, LPARAM lParam )
{
  switch(message)
  {
    case WM_INITDIALOG:
    {
      Nui_Zero();     // azzero/annullo le variabili relative alla
          visualizzazione

      m_hWnd = hWnd;     // unisco l'handle con la finestra

      LOGFONT lf;
      GetObject( (HFONT)GetStockObject(DEFAULT_GUI_FONT), sizeof(lf),
          &lf );
      lf.lfHeight *= 4;
      m_hFontFPS = CreateFontIndirect(&lf);
      SendDlgItemMessageW(hWnd, IDC_FPS, WM_SETFONT, (WPARAM)
          m_hFontFPS, 0);
      LOGFONT lf_istr;
          GetObject( (HFONT)GetStockObject(DEFAULT_GUI_FONT), sizeof(
              lf_istr), &lf_istr );
          lf_istr.lfHeight *= 2;
      m_hFontFPS_istr = CreateFontIndirect(&lf_istr);
          WCHAR * szMessageIstr = L" Istruzioni:\n\n 1) scegliere
              modalità e\n distanza di interazione\n 2) mettersi
              davanti al\n sensore\n 3) attendere che lo\n scheletro
              venga disegnato\n 4) alzare un braccio per\n essere
```

```
            riconosciuti";
        SendDlgItemMessageW( hWnd, IDC_FPS3, WM_SETFONT, (WPARAM)
            m_hFontFPS_istr, (LPARAM)szMessageIstr );
        SendDlgItemMessageW( hWnd, IDC_FPS3, WM_SETTEXT, 0, (LPARAM)
            szMessageIstr );

    UpdateKinectComboBox();     // aggiorno la selezione di quale
        Kinect utilizzare
    SendDlgItemMessageW(m_hWnd, IDC_CAMERAS, CB_SETCURSEL, 0, 0);
    TCHAR szComboText[512] = { 0 };
       // Modalità di default
    LoadStringW(m_hInstance, IDS_TRACKINGMODE_DEFAULT, szComboText,
        _countof(szComboText));
    SendDlgItemMessageW(m_hWnd, IDC_TRACKINGMODE, CB_ADDSTRING, 0,
        reinterpret_cast<LPARAM>(szComboText));
       // Modalità seduto
    LoadStringW(m_hInstance, IDS_TRACKINGMODE_SEATED, szComboText,
        _countof(szComboText));
    SendDlgItemMessageW(m_hWnd, IDC_TRACKINGMODE, CB_ADDSTRING, 0,
        reinterpret_cast<LPARAM>(szComboText));

        SendDlgItemMessageW(m_hWnd, IDC_TRACKINGMODE, CB_SETCURSEL,
            0, 0);
       // Modalità di default
    LoadStringW(m_hInstance, IDS_RANGE_DEFAULT, szComboText, _countof
        (szComboText));
    SendDlgItemMessageW(m_hWnd, IDC_RANGE, CB_ADDSTRING, 0,
        reinterpret_cast<LPARAM>(szComboText));
       // Modalità ravvicinata
    LoadStringW(m_hInstance, IDS_RANGE_NEAR, szComboText, _countof(
        szComboText));
    SendDlgItemMessageW(m_hWnd, IDC_RANGE, CB_ADDSTRING, 0,
        reinterpret_cast<LPARAM>(szComboText));

    SendDlgItemMessageW(m_hWnd, IDC_RANGE, CB_SETCURSEL, 0, 0);
    }
    break;

    case WM_SHOWWINDOW:
    {
      Nui_Init();     // Inizializzo e faccio partire il NUI
          processing
    }
    break;

    case WM_TIMER:     // il timer parte ogni 20 msec
        if (wParam == TimerId)
    {
      HRESULT hr = Nui_ProcessAudio();
      if (FAILED(hr))
          {
```

```
            WCHAR * szMess = L" Nui_ProcessAudio() non va.";
            SendDlgItemMessageW(m_hWnd, IDC_STATUS2, WM_SETTEXT, 0, (
                LPARAM)szMess);
          }

          hr = Nui_Update();
          if (FAILED(hr))
          {
            WCHAR * szMess2 = L" Nui_Update() non va.";
            SendDlgItemMessageW(m_hWnd, IDC_STATUS2, WM_SETTEXT, 0, (
                LPARAM)szMess2);
           }
        }
        break;

        case WM_USER_UPDATE_FPS:
        {
          ::SetDlgItemInt( m_hWnd, static_cast<int>(wParam), static_cast<
              int>(lParam), FALSE );
        }
        break;

        case WM_USER_UPDATE_COMBO:
        {
          UpdateKinectComboBox();
        }
        break;

        case WM_COMMAND:
        switch ( LOWORD(wParam) )
        {
          case IDC_CAMERAS:
              if ( HIWORD(wParam) == CBN_SELCHANGE )
          {
            LRESULT index = ::SendDlgItemMessageW(m_hWnd, IDC_CAMERAS,
                CB_GETCURSEL, 0, 0);
            if ( !m_fUpdatingUi )
            {
              Nui_UnInit();
              Nui_Zero();
              Nui_Init(reinterpret_cast<BSTR>(::SendDlgItemMessageW(
                  m_hWnd, IDC_CAMERAS, CB_GETITEMDATA, index, 0)));
            }
          }
          break;

          case IDC_TRACKINGMODE:
          if ( HIWORD(wParam) == CBN_SELCHANGE )
          {
            LRESULT index = ::SendDlgItemMessageW(m_hWnd,
                IDC_TRACKINGMODE, CB_GETCURSEL, 0, 0);
```

```
            UpdateTrackingMode( static_cast<int>(index) );
          }
          break;

          case IDC_RANGE:
          if ( HIWORD(wParam) == CBN_SELCHANGE )
          {
            LRESULT index = ::SendDlgItemMessageW(m_hWnd, IDC_RANGE,
                CB_GETCURSEL, 0, 0);
            UpdateRange( static_cast<int>(index) );
          }
          break;

          case IDC_BUTTON1:     // pulsante che attiva/interrompe la
              registrazione
          {
            switch( HIWORD(wParam) )
            {
              case BN_CLICKED:
              if (m_fRecording==FALSE)
              {
                HRESULT hr = RecordAudio();
                if (hr == S_OK)
                {
                  WCHAR * szMessage2 = L" Registrazione in corso...";
                  SendDlgItemMessageW(m_hWnd, IDC_STATUS21, WM_SETTEXT,
                      0, (LPARAM)szMessage2);
                }
                else
                {
                  WCHAR * szMessage1 = L" ERRORE: registrazione fallita.
                      ";
                  SendDlgItemMessageW(m_hWnd, IDC_STATUS21, WM_SETTEXT,
                      0, (LPARAM)szMessage1);
                }
              }
              else if (m_fRecording==TRUE)
              {
                HRESULT hr = StopRecording();
                if (hr == S_OK)
              }
                WCHAR * szMessage3 = L" Scrittura avvenuta con successo!
                    ";
                SendDlgItemMessageW(m_hWnd, IDC_STATUS21, WM_SETTEXT, 0,
                     (LPARAM)szMessage3);
             }
            }
            break;
           }
          }
        }
```

```
      break;

      case WM_CLOSE:
        DestroyWindow(hWnd);
        break;

      case WM_DESTROY:
        Nui_UnInit();
        ClearKinectComboBox();     // rilascio altre risorse
        DeleteObject(m_hFontFPS);
        DeleteObject(m_hFontFPS2);
        DeleteObject(m_hFontFPS_istr);
        PostQuitMessage(0);
        break;
    }
    return FALSE;
}
```

B.2 Creazione del thread per i dati audio

```
DWORD WINAPI CSkeletal::Nui_AudioThread(LPVOID pParam)
{
  CSkeletal *pthis = (CSkeletal *)pParam;
  return pthis->Nui_AudioThread( );
}

DWORD WINAPI CSkeletal::Nui_AudioThread( )
{
  HRESULT hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);
  if (hr == S_OK)
  {
    WCHAR * szMess4 = L" COM Inizializzato.";
    SendDlgItemMessageW(m_hWnd, IDC_STATUS, WM_SETTEXT, 0, (LPARAM)
        szMess4);
    WCHAR * szMess = L" Dopo l'inizializzazione del sensore, attendere
         circa 4 secondi prima che i dati audio siano elaborati...";
    SendDlgItemMessageW(m_hWnd, IDC_STATUS4, WM_SETTEXT, 0, (LPARAM)
        szMess);
    appoggio.m_pAudioPanel = new AudioPanel( );
    hr = appoggio.m_pAudioPanel->InitializeEnergy(m_hWnd,
        m_pD2DFactory, EnergySamplesToDisplay);
    appoggio.m_pAudioPanelGauge = new AudioPanel();
    hr = appoggio.m_pAudioPanelGauge->InitializeGauge(m_hWnd,
        m_pD2DFactory);
    if (hr == E_INVALIDARG)
    {
      WCHAR * szMess7 = L" Attenzione: pD2DFactory e' vuoto!";
      SendDlgItemMessageW(m_hWnd, IDC_STATUS2, WM_SETTEXT, 0, (LPARAM)
          szMess7);
```

```
      return hr;
    }
    if (hr == S_OK)
    {
      WCHAR * szMess2 = L" Finestra audio inizializzata correttamente.
          ";
      SendDlgItemMessageW(m_hWnd, IDC_STATUS, WM_SETTEXT, 0, (LPARAM)
          szMess2);
      hr = appoggio.InitializeAudioSource(m_hWnd, m_pNuiSensor);
      if (hr == S_OK)
      {
        WCHAR * szMess5 = L" Sorgente audio inizializzata";
        SendDlgItemMessageW(m_hWnd, IDC_STATUS, WM_SETTEXT, 0, (LPARAM
            )szMess5);
        SetTimer(m_hWnd, TimerId, TimerInterval, NULL);
      }
      else
      {
        WCHAR * szMess6 = L" Problemi durante l'inizializzazione della
             sorgente audio.";
        SendDlgItemMessageW(m_hWnd, IDC_STATUS, WM_SETTEXT, 0, (LPARAM
            )szMess6);
      }
    }
  }
  return 0;
}
```

B.3 Altre funzioni di supporto per i dati audio

```
HRESULT CSkeletal::Nui_ProcessAudio()
{
  HRESULT hr;
  hr = appoggio.ProcessAudio();
  if (hr == S_OK)
  {
        WCHAR * szMess2 = L" Kinect pronta per l'elaborazione dati";
        SendDlgItemMessageW(m_hWnd, IDC_STATUS, WM_SETTEXT, 0, (LPARAM
            )szMess2);
  }
  else if (hr == S_FALSE)
  {
    WCHAR * szMess3 = L" Non ho generato dati!";
    SendDlgItemMessageW(m_hWnd, IDC_STATUS2, WM_SETTEXT, 0, (LPARAM)
        szMess3);
  }
  return hr;
}
```

```
HRESULT CSkeletal::RecordAudio()
{
   HRESULT hr;
   hr = appoggio.RecordAudioWave();
   m_fRecording = TRUE;
   return hr;
}

HRESULT CSkeletal::StopRecording()
{
   HRESULT hr;
   hr = appoggio.StopRecordingAudioWave();
   m_fRecording = FALSE;
   return hr;
}

HRESULT CSkeletal::Nui_Update()
{
   HRESULT hr;
   hr = appoggio.Update();
   if (hr == S_FALSE)
   {
     WCHAR * szMess2 = L" Update() di classe CAudio non sta andando.";
     SendDlgItemMessageW(m_hWnd, IDC_STATUS2, WM_SETTEXT, 0, (LPARAM)
         szMess2);
   }
   return hr;
}
```

B.4 Inizializzazione dell'acquisizione audio

```
HRESULT CAudio::InitializeAudioSource(HWND hWnd, INuiSensor*
    pNuiSensor)
{
  ...
  PROPVARIANT pvSysMode, pvMicFeature,pvMicArrMode,disableAGC;
  PropVariantInit(&pvSysMode);     // inizializzo la variabile '
      pvSysMode'
    pvSysMode.vt = VT_I4;
    pvSysMode.lVal = (LONG)(2);
    hr=m_pPropertyStore->SetValue(MFPKEY_WMAAECMA_SYSTEM_MODE,
        pvSysMode);
    if (FAILED(hr))
    {
       WCHAR * szMessage0 = L" SetValue() Sys Mode fallito.";
       SendDlgItemMessageW(m_hWnd,IDC_STATUS2,WM_SETTEXT,0,(LPARAM)
           szMessage0);
       return hr;
    }
```

```
    PropVariantClear(&pvSysMode);

        PropVariantInit(&pvMicFeature);     // inizializzo la
            variable 'pvMicFeature'
        pvMicFeature.vt = VT_BOOL;
        pvMicFeature.boolVal = VARIANT_TRUE;
        hr=m_pPropertyStore->SetValue(MFPKEY_WMAAECMA_FEATURE_MODE,
            pvMicFeature);
        if(FAILED(hr))
        {
       WCHAR * szMessage1 = L" SetValue() Feature Mode fallito.";
       SendDlgItemMessageW(m_hWnd, IDC_STATUS2, WM_SETTEXT, 0, (LPARAM
           )szMessage1);
       return hr;
        }
    PropVariantClear(&pvMicFeature);

    PropVariantInit(&disableAGC);     // inizializzo la variabile '
        disableAGC'
    disableAGC.vt = VT_BOOL;
    disableAGC.boolVal = VARIANT_FALSE;
    hr=m_pPropertyStore->SetValue(MFPKEY_WMAAECMA_FEATR_AGC,disableAGC
        );
    if(FAILED(hr))
    {
        SetStatusMessageA(m_hWnd, L" SetValue() AGC fallito.");
        return hr;
    }
    PropVariantClear(&disableAGC);

    PropVariantInit(&pvMicArrMode);     // inizializzo la variabile '
        pvMicArrMode'
    pvMicArrMode.vt = VT_I4;
    pvMicArrMode.lVal = (LONG) MICARRAY_EXTERN_BEAM;
    hr = m_pPropertyStore->SetValue(MFPKEY_WMAAECMA_FEATR_MICARR_MODE,
        pvMicArrMode);
    if(FAILED(hr))
    {
        SetStatusMessageA(m_hWnd, L"SetValue() Mic Mode fallito.");
        return hr;
    }
    PropVariantClear(&pvMicArrMode);
        ...
}
```

B.5 Impostazione del beam

```
HRESULT CAudio::ProcessAudio()
{
```

```
const float cEnergyNoiseFloor = 0.2f;
ULONG cbProduced = 0;      // dimensione dei dati
BYTE *pProduced = NULL;      // puntatore al buffer
DWORD dwStatus = 0;
DMO_OUTPUT_DATA_BUFFER outputBuffer = {0};
outputBuffer.pBuffer = &m_captureBuffer;
HRESULT hr = S_OK;
do
{
  hr = m_captureBuffer.Init(0);
      outputBuffer.dwStatus = 0;
      hr = m_pDMO->ProcessOutput(0, 1, &outputBuffer, &dwStatus);

  if (FAILED(hr))
  {SetStatusMessageA(m_hWnd, L" ProcessOutput() fallito -
      Elaborazione dell'output audio non andata a buon fine.");
      break;}
  if (hr == S_FALSE)
  {
        cbProduced = 0;
        WCHAR * szMess2 = L" ProcessOutput() non andato.";
        SendDlgItemMessageW(m_hWnd,IDC_STATUS,WM_SETTEXT,0,(LPARAM)
            szMess2);
      }

  hr = m_captureBuffer.GetBufferAndLength(&pProduced, &cbProduced);

  if (cbProduced > 0)
  {
    double beamAngle, sourceAngle, sourceConfidence;
    hr = m_pNuiAudioSource->SetBeam(m_BeamAngle);
    if (hr == S_OK)
    {
      WCHAR angoloBeam[500];
      StringCbPrintfW(angoloBeam, _countof(angoloBeam), L" Angolo
          del Beam (settato): %0.2f°", static_cast<float>((180.0 *
          m_BeamAngle)/ M_PI) );
      SendDlgItemMessageW( m_hWnd,IDC_STATUS3,WM_SETTEXT,0,(LPARAM)(
          angoloBeam) );
        }

    hr = m_pNuiAudioSource->GetBeam(&beamAngle);
    if (FAILED(hr))
    {
      SetStatusMessageA(m_hWnd, L" Metodo GetBeam() non andato.");
      return hr;
    }
    else if (hr == S_OK)
    {
      WCHAR angoloB[500];
```

```
        StringCbPrintfW(angoloB, _countof(angoloB), L" Angolo del Beam
            (Kinect): %0.2f°", static_cast<float>((180.0 * beamAngle)
            / M_PI) );
        SendDlgItemMessageW( m_hWnd, IDC_STATUS5, WM_SETTEXT, 0, (
            LPARAM)(angoloB) );
      }
      hr = m_pNuiAudioSource->GetPosition(&sourceAngle, &
          sourceConfidence);
    if (FAILED(hr))
    {
      SetStatusMessageA(m_hWnd, L" Metodo GetPosition() non andato."
          );
      return hr;
    }
    else if (hr == S_OK)
    {
      hr=m_pAudioPanelGauge->SetBeam(static_cast<float>((180.0 *
          m_BeamAngle) / M_PI));
    } while (outputBuffer.dwStatus &
        DMO_OUTPUT_DATA_BUFFERF_INCOMPLETE);
    return hr;
}
```

B.6 Calcolo dell'angolo della testa

```
float CSkeletal::CalcolaHeadAngle(float componente_X, float
    componente_Z)
{
  float testa_x = componente_X;
  float testa_x2 = pow(testa_x, 2);
  float testa_z = componente_Z;
  float testa_z2 = pow(testa_z, 2);
  float modulo = sqrt(testa_x2 + testa_z2);
  float angoloRadianti = asin(testa_x/modulo);
  float angolo = 0.0;
  angolo = static_cast<float>((180.0 * angoloRadianti) / M_PI);
  return angolo;
}
```

B.7 Attivazione tramite gesto

```
bool CSkeletal::Nui_GotSkeletonAlert( )
{
  NUI_SKELETON_FRAME SkeletonFrame = {0};
  bool foundSkeleton = false;
  bool personaRiconosciuta;
```

```
DWORD memoriaID [2] = {0,0};
int utenteAttivo [6] = {0,0,0,0,0,0};
if ( SUCCEEDED(m_pNuiSensor->NuiSkeletonGetNextFrame( 0, &
    SkeletonFrame )) )
{
for ( int i = 0 ; i < NUI_SKELETON_COUNT ; i++ )
{
  NUI_SKELETON_TRACKING_STATE trackingState = SkeletonFrame.
      SkeletonData[i].eTrackingState;
  if ( trackingState == NUI_SKELETON_TRACKED || trackingState ==
      NUI_SKELETON_POSITION_ONLY )
  { foundSkeleton = true; }
 }
}
if( !foundSkeleton )
{  return true;  }
HRESULT hr = m_pNuiSensor->NuiTransformSmooth(&SkeletonFrame,NULL);
if ( FAILED(hr) )
{
  return false;
}
m_bScreenBlanked = false;
m_LastSkeletonFoundTime = timeGetTime( );
hr = EnsureDirect2DResources( );
if ( FAILED( hr ) )
{
 return false;
}
m_pRenderTarget->BeginDraw();
m_pRenderTarget->Clear( );

RECT rct;
GetClientRect( GetDlgItem( m_hWnd, IDC_SKELETALVIEW ), &rct);
int width = rct.right;     // ascissa dell'angolo in basso a destra
int height = rct.bottom;     // ordinata dell'angolo in basso a
    destra
for ( int i = 0 ; i < NUI_SKELETON_COUNT; i++ )     // rilevo (di
    nuovo) fino a 6 persone
{
      NUI_SKELETON_TRACKING_STATE trackingState = SkeletonFrame.
          SkeletonData[i].eTrackingState;
  if ( trackingState == NUI_SKELETON_TRACKED )
  {
    start = clock();
    personaRiconosciuta = false;
    utenteAttivo[i]=i+1;
    if (SkeletonFrame.SkeletonData[i].SkeletonPositions[
        NUI_SKELETON_POSITION_ELBOW_LEFT].y  > SkeletonFrame.
        SkeletonData[i].SkeletonPositions[
        NUI_SKELETON_POSITION_SHOULDER_CENTER].y ||
```

```
            SkeletonFrame.SkeletonData[i].SkeletonPositions[
                NUI_SKELETON_POSITION_WRIST_LEFT].y  > SkeletonFrame.
                SkeletonData[i].SkeletonPositions[
                NUI_SKELETON_POSITION_HEAD].y ||
48          SkeletonFrame.SkeletonData[i].SkeletonPositions[
                NUI_SKELETON_POSITION_ELBOW_RIGHT].y > SkeletonFrame.
                SkeletonData[i].SkeletonPositions[
                NUI_SKELETON_POSITION_SHOULDER_CENTER].y ||
            SkeletonFrame.SkeletonData[i].SkeletonPositions[
                NUI_SKELETON_POSITION_WRIST_RIGHT].y > SkeletonFrame.
                SkeletonData[i].SkeletonPositions[
                NUI_SKELETON_POSITION_HEAD].y )
            {  personaRiconosciuta = true;  }
        Nui_DrawSkeleton(SkeletonFrame.SkeletonData[i],width,height );
            if (personaRiconosciuta == true)
53          {
          UpdateTrackedSkeletonSelection( MODO_ATTIVO );
          memoriaID[0] = SkeletonFrame.SkeletonData[i].dwTrackingID;
          WCHAR ID_y[600];
          StringCbPrintfW(ID_y, _countof(ID_y), L" Valore di
              personaRiconosciuta: %d\n", personaRiconosciuta);
58        SendDlgItemMessageW( m_hWnd, IDC_STATUS15, WM_SETTEXT,0,(
              LPARAM)(ID_y));
          HRESULT hr = m_pNuiSensor->NuiSkeletonSetTrackedSkeletons(
              memoriaID);
          if (hr == S_OK)
          {
            WCHAR ID_u[600];
63          StringCbPrintfW(ID_u, _countof(ID_u), L" Utente attivo: %d\n
                ", utenteAttivo[i]);
            SendDlgItemMessageW( m_hWnd, IDC_STATUS15, WM_SETTEXT, 0, (
                LPARAM)(ID_u));
            end = clock();
            tempo = ( (double)(end - start) ) / CLOCKS_PER_SEC;
            WCHAR tt[600];
68          StringCbPrintfW(tt, _countof(tt), L" Tempo necessario per il
                 riconoscimento: %f secondi", tempo);
            SendDlgItemMessageW( m_hWnd, IDC_STATUS24, WM_SETTEXT, 0, (
                LPARAM)(tt) );
           }
         }
        }
73      else if ( trackingState == NUI_SKELETON_POSITION_ONLY )
        {
          D2D1_ELLIPSE ellipse = D2D1::Ellipse(SkeletonToScreen(
              SkeletonFrame.SkeletonData[i].Position, width, height ),
              g_JointThickness, g_JointThickness);

          m_pRenderTarget->DrawEllipse(ellipse, m_pBrushJointTracked);
78      }
      }
```

```
    if (m_TrackedSkeletons == MODO_ATTIVO)
    {
       appoggio.m_BeamAngle = (m_HeadAngle * 2 * M_PI) / 360;
83  }
    if (utenteAttivo[0] == 0 && utenteAttivo[1] == 0 && utenteAttivo
        [2] == 0 && utenteAttivo[3] == 0 && utenteAttivo[4] == 0 &&
        utenteAttivo[5] == 0)
    { UpdateTrackedSkeletonSelection(MODO_DEFAULT);}
     hr = m_pRenderTarget->EndDraw();
    UpdateTrackedSkeletons( SkeletonFrame );
88
    if ( hr == D2DERR_RECREATE_TARGET )
    {
      hr = S_OK;
      DiscardDirect2DResources();
93    return false;
    }
    return true;
}
```