Università di Padova Facoltà di Ingegneria

Dipartimento di Ingegneria dell'Informazione

Laurea Magistrale in Ingegneria Informatica

Porting e integrazione di software di people tracking su piattaforma ARM U-Mobo e ambiente Linux

Laureando

Nicolò Ongaro

Relatore: Prof. Michele Moro

Anno Accademico 2014/2015

ABSTRACT

Recentemente l'hardware impiegato in alcune piattaforme embedded ha raggiunto un livello tale che è attualmente possibile utilizzare quest'ultime per svolgere operazioni di cui poco tempo fa solo computer desktop potevano occuparsi. L'utilizzo di soluzioni embedded ha sempre presentato numerosi vantaggi, fra i quali la facilità di integrazione all'interno di sistemi più grandi e il minor consumo energetico apportato. L'ampliamento della capacità di calcolo di cui prima accennato ha reso il loro possibile utilizzo ancora più allettante.
Verrà introdotta la piattaforma ARM U-Mobo e dimostrate le sue capacità in quanto sistema general purpose. In particolare, verranno prese in considerazione delle particolari applicazioni che prevedono l'utilizzo del device Microsoft Kinect per svolgere delle operazioni di people tracking. A questo scopo si percorreranno i passi necessari per il setup della piattaforma e per l'integrazione e il porting del software che si intende utilizzare.

INDICE

CAPITOLO 1

INTRODUZIONE

1.1 Scopo della tesi

Scopo della tesi è l'integrazione e il porting di un software di people tracking sulla piattaforma ARM U-Mobo. Il punto di partenza sarà un'unità U-Mobo ancora priva del software che permette il suo avvio e si procederà di conseguenza con l'installazione del bootloader U-Boot e del sistema operativo Linux. In seguito verrà integrato il framework ROS, il driver della Kinect e altri componenti richiesti dal software di people tracking. Infine verrà presentato, in qualità di esempio applicativo, lo sviluppo di un sistema di sicurezza che, sfruttando le capacità di riconoscimento, si occuperà di monitorare il flusso di persone in ingresso e in uscita da un determinato luogo.
Prima di procedere con le operazioni menzionate, verrà in questo capitolo introdotta la piattaforma U-Mobo e l'architettura ARM di cui fa utilizzo.

1.2 L'architettura ARM

L'architettura ARM (precedentemente Advanced RISC Machine, prima ancora Acorn RISC Machine) indica una famiglia di microprocessori RISC a 32-bit sviluppata da ARM Holdings, utilizzata in una moltitudine di sistemi embedded. La natura RISC di questa architettura si traduce in un numero significativamente minore di transistor rispetto a quelli presenti un tipico processore x86 CISC, presente nella maggior parte dei personal computer.

Questo approccio riduce i costi, il calore generato e il consumo energetico. Queste caratteristiche sono particolarmente importanti nei sistemi embedded. Inoltre lo sviluppo di CPU ARM multi-core si rivela così particolarmente vantaggioso.

Figura 1.1. *Processore ARM ARM7TDMI.*

Il progetto ARM è iniziato nel 1983 nella sezione ricerca e sviluppo della Acorn Computers Ltd. Il team, guidato da Sophie Wilson e Steve Furber, iniziò lo sviluppo puntando a realizzare una versione migliore del MOS Technology 6502. Acorn in quel periodo utilizzava il MOS 6502 per i suoi computer e i manager pensarono che poter utilizzare dei processori prodotti internamente dalla società avrebbe fornito un notevole vantaggio competitivo all'azienda. Il team completò lo sviluppo del prototipo chiamato semplicemente ARM1 nel 1985 e il primo processore realmente prodotto, l'ARM2, venne realizzato l'anno successivo. L'ARM2 era un processore con un bus dati a 32 bit, un bus di indirizzi a 26 bit (in modo da poter indirizzare fino a 64 Megabyte) e dei registri a 16/32 bit. L'ARM2 era probabilmente il più semplice processore a 32 bit, con i suoi 30.000 transistor era significativamente più semplice del Motorola 68000 che pur avendo 68.000 transistor forniva prestazioni paragonabili. Il processore doveva la sua semplicità alla mancanza di microcodice, e come la maggior parte delle CPU dell'epoca, non era dotato di cache. Nel 1989, il successore ARM3 venne invece dotato di 4KB di cache per migliorare le prestazioni.

Alla fine degli anni ottanta Apple iniziò a lavorare con Acorn per sviluppare una nuova versione del core ARM. Il progetto era talmente importante da spingere Acorn a spostare il team di sviluppo in una nuova compagnia chiamata Advanced RISC Machines Ltd. Per questo motivo spesso ARM viene espanso come Advanced RISC Machine invece che Acorn RISC Machine. Advanced RISC Machines divenne ARM Ltd quando la società madre ARM

Holdings decise di quotarsi alla London Stock Exchange e al NASDAQ nel 1998. I lavori del team di sviluppo portarono alla realizzazione dell'ARM6. Il primo modello venne prodotto nel 1991 e Apple utilizzò il processore ARM 610 basato su core ARM6 per l'Apple Newton. Nel 1994 Acorn utilizzò ARM610 come processore centrale del suo computer RiscPC. L'implementazione di maggior successo è sicuramente l'ARM7TDMI che viene utilizzato in console portatili, telefoni cellulari, e periferiche varie. Centinaia di milioni di esemplari di questo processore sono stati prodotti. L'architettura comunemente supportata da Windows Mobile ed Android (sistemi operativi installati su PDA, smartphone, Tablet computer e altri dispositivi portatili) è l'ARM4. ARM Holdings è attualmente responsabile dello sviluppo del set di istruzioni dell'architettura dei processori ARM, ma non li produce personalmente. L'azienda si limita a rilasciare periodici aggiornamenti al design dei suoi core.
Attualmente queste CPU supportano indirizzamento e operazioni aritmetiche a 32 bit, ARMv8-A, già annunciato nell'ottobre del 2011, estenderà queste capacità a 64 bit. Le istruzioni macchina hanno, su le versioni dell'architettura più recenti, lunghezza variabile di 32 o 16 bit per rendere più compatto il codice. Alcuni core forniscono anche la possibilità di esecuzione hardware di bytecode Java. ARM Holdings cede le licenze sul design e sul set di istruzioni a terze parti che si occupano di implementare i propri systems-on-chips (SoC) che incorporano memoria, interfacce e altro. Fra le compagnie che utilizzano CPU ARM copiano Apple, Broadcom, Freescale Semiconductor, Nvidia, Qualcomm, Samsung Electronics e Texas Instruments.
Attualmente la famiglia ARM copre il 75% del mercato mondiale dei processori a 32 bit per applicazioni embedded, ed è una delle più diffuse architetture a 32 bit del mondo. I processori ARM vengono utilizzati in cellulari, tablet, lettori multimediali, console portatili, PDA e periferiche per computer (come router, hard disk di rete ecc). Importanti rami della famiglia ARM sono i processori XScale e i processori OMAP, prodotti da Texas Instruments.

1.3 La piattaforma U-Mobo

Come già accennato, oggetto della tesi è la piattaforma ARM U-Mobo. Si tratta di un sistema high-end la cui architettura è stata sviluppata per adeguarsi ai più diversi impieghi che variano dall'utilizzo in ambito di ricerca alle applicazioni per l'industria. Presenta una struttura modulare che ricorda in qualche modo quella tipica dei PC desktop.

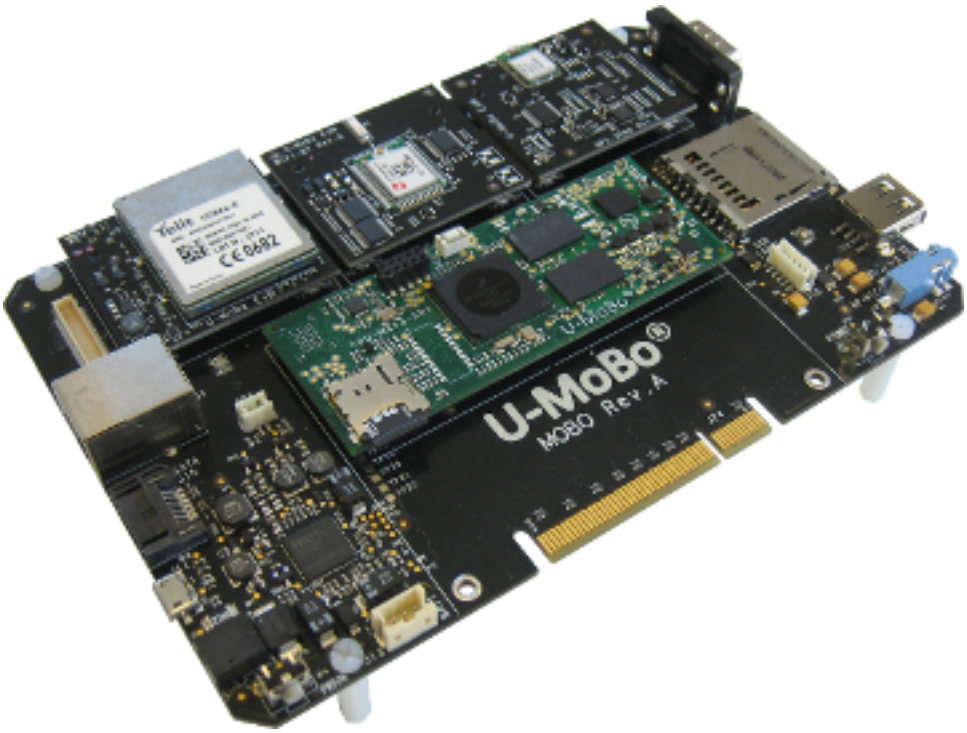

Figura 1.2. *Scheda U-Mobo completa di moduli SoM e ExM.*

U-MoBo si prefigge due obiettivi. Da una parte definire una piattaforma standard, totalmente "open", sulla quale possano convergere progettisti e produttori per facilitare la diffusione delle nuove tecnologie semplificandone l'integrazione e l'utilizzo. Dall'altra intende porsi come referente di una community che partecipi attivamente ad accrescere la disponibilità di nuovi strumenti hardware e software pronti all'utilizzo per lo sviluppo rapido di progetti ad alto valore aggiunto.

U-MoBo utilizza soluzioni basate sui processori della serie i.MX di Freescale. In pieno spirito "open", la documentazione di progetto, così come il software operativo (distribuzioni Android, Linux e Windows Embedded Compact 7 che includono OpenGL e librerie Qt), sono di pubblico dominio e permettono di sfruttare le potenzialità dei processori, delle periferiche e dei moduli di espansione disponibili. Grazie a questo è possibile, per chi lo desiderasse, sviluppare liberamente e autonomamente dei nuovi moduli per far fronte alle proprie necessità. Questa caratteristica permette sempre di ottenere le funzionalità desiderate in un'unica soluzione che rimane semplice e compatta.

L'architettura modulare prevede principalmente la presenza di quattro componenti:

- **U-Mobo base board**: si tratta della componente base del sistema. Permette l'integrazione dei vari moduli e fornisce le funzionalità e la connettività di base. Oltre allo slot per SoM e a tre slot ExM fornisce connettore Ethernet, porta seriale, slot SD/MMC, USB Host, jack audio, speaker e altro ancora.

- **SoM** (System on Module): U-Mobo supporta attualmente chip basati su Freescale i.MX. Al suo interno è integrato l'intero sistema vero e proprio (i.e., CPU, RAM etc.). Tramite l'interfaccia offerta dalla base board ottiene accesso ai bus I2C e SPI e di conseguenza alle varie periferiche e moduli ExM.

  Per le attività che verranno in seguito presentate si farà uso del SoM i.MX53. Questo integra un core ARM Cortex-A8 a 1,2 Ghz con capacità di accelerazione grafica 2D e 3D, 512 MB di RAM (DDR3 @800 Mhz), slot microSD, physical Ethernet e I/O digitali e analogici.

- **ExM** (Expansion Module): questi moduli possono virtualmente implementare qualsiasi periferica e vanno ad estendere le capacità del sistema. U-Mobo fornisce un interfaccia plug-and-play per queste estensioni, fino a tre moduli ExM sono supportati con l'utilizzo della sola base board. Fra i moduli attualmente già disponibili sono presenti gli ExM GPSense (accelerometro, giroscopio, sensore di pressione e ricevitore GPS), PSoC (I/O analogici), WiFi-BT (WiFi e bluetooth) e WAN2G3G (GSM/GPRS).

- **Schermo touchscreen**: sono disponibili una serie di schermi touchscreen, capacitivi o resistivi, che si possono collocare direttamente sulla board. Grazie a questi l'aspetto e le funzionalità diventano simili a quelle di un device handheled completamente autonomo.

CAPITOLO 2

SETUP DELL'AMBIENTE

2.1 L'U-Boot

I file e i dati necessari all'avvio vengono letti dalla microSD inserita direttamente nel SoM della U-Mobo. Come prima cosa sarà necessario dunque installarvi un adeguato bootloader. U-Boot è tipicamente quello scelto per queste applicazioni: si tratta di un bootloader che gira su numerose piattaforme hardware (fra cui ARM) ed rilasciato con licenza GNU GPL. Il progetto originale prevedeva la realizzazione di un bootloader per i PowerPC 8xx chiamato 8xxROM, il codice era stato scritto da Magnus Damm. Nel 1999 il progetto fu cambiato in PPCBoot e nel 2002 nuovamente in U-Boot (Universal Uboot) per riflettere la nuova capacità del bootloader di funzionare su architetture diverse. U-Boot-0.1.0, rilasciato nel novembre del 2002, possedeva già la compatibilità per i processori x86 e nei mesi successivi molte altre architettura si aggiunsero alla lista. La versatilità e la semplicità di questo software lo resero subito di comune utilizzo fra le svariate soluzioni embedded dei diversi produttori di hardware. Attualmente è ad esempio impiegato nelle board SAM440ep e SAM460ex prodotte da ACube Systems, nonché nei Chromebooks basati su architettura ARM e sviluppati da Google.

2.2 Compilazione del boot loader

I sorgenti della versione dell'U-Boot per i.MX specifici per la piattaforma U-Mobo sono reperibili facilmente dal repository Git online http://github.com/u-mobo/uboot-imx. Dato

che a questo punto tipicamente non si ha la possibilità di compilare nativamente i sorgenti, quello che generalmente si vuole fare è effettuare una cross-compilazione da una macchina host di architettura diversa. In questo caso sarà necessario installare l'adeguata toolchain per effettuare questa operazione. A questo scopo attualmente, su sistemi Linux, è sufficiente installare il package *gcc-4.7-arm-linux-gnueabihf*. I passi da effettuare per la cross-compilazione sono dunque i seguenti:

```
$ sudo apt-get install gcc-4.7-arm-linux-gnueabihf
$ git clone http://github.com/u-mobo/uboot-imx
$ cd uboot-imx
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bluelightning_config
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

Le configurazioni disponibili per il *make* sono *bluelightning_config* per i.MX53 e *silverbullet_*_config* per per la serie i.MX6. La compilazione produrrà come output il file binario *u-boot.imx* che dovrà essere direttamente scritto sulla microSD in una specifica locazione. Si vedrà questo passaggio più avanti.

2.3 Il sistema Linux

2.3.1 Il kernel

Il kernel è la parte principale di un sistema operativo e fornisce tutte le funzioni essenziali per il sistema, in particolare la gestione della memoria primaria, delle risorse hardware del sistema e delle periferiche, assegnandole di volta in volta ai processi in esecuzione. L'accesso diretto all'hardware può essere anche molto complesso, quindi i kernel usualmente implementano un sistema di astrazione dall'hardware. Questo permette di "nascondere" la complessità e fornire un'interfaccia pulita ed uniforme all'hardware sottostante, in modo da semplificare il lavoro degli sviluppatori.

Linux è un kernel distribuito con licenza GNU General Public License, creato da Linus Torvalds nel 1991. Torvalds scelse una struttura kernel monolitica. Questo approccio definisce un'interfaccia virtuale di alto livello sull'hardware, ovvero un set di primitive o chiamate di sistema che implementano i servizi del sistema operativo in diversi moduli eseguiti in una modalità privilegiata (modalità supervisore). L'integrazione del codice è tuttavia molto

stretta e la presenza di un errore in una qualsiasi sua parte può condizionare la funzionalità dell'intero sistema. D'altro canto quando l'implementazione è completa e sicura, un kernel monolitico può essere estremamente efficiente. Tipico svantaggio dei kernel monolitici è che non è possibile aggiungere un nuovo dispositivo hardware senza aggiungere il relativo modulo al kernel, operazione che in genere richiede la ricompilazione del kernel. Linux è invece in grado di caricare moduli in fase di esecuzione, permettendo così l'estensione del kernel quando richiesto, mantenendo al contempo le dimensioni del codice nello spazio del kernel al minimo indispensabile.

Il codice sorgente di Linux è reso disponibile a tutti ed è ampiamente personalizzabile al punto da rendere possibile, in fase di compilazione, l'esclusione di codice non strettamente indispensabile. La flessibilità di questo kernel lo rende adatto a tutte quelle tecnologie embedded emergenti oltre che a prodotti come videoregistratori digitali o telefoni cellulari.

Mentre sviluppa il proprio codice e integra le modifiche create da altri programmatori, Linus Torvalds continua a rilasciare nuove versioni del kernel di Linux. Queste sono chiamate versioni "vanilla", ad indicare che non sono state modificate da nessun altro. Molte distribuzioni Linux modificano il kernel per il proprio sistema, principalmente allo scopo di aggiungere il supporto per driver o caratteristiche che non sono state ufficialmente rilasciate come stabili, mentre altre distribuzioni usano un kernel vanilla.

Le versioni "vanilla" sono rese disponibile online all'indirizzo https://www.kernel.org. Ogni release che viene fatta rientra in una delle seguenti categorie:

- **Mainline**: si tratta della linea di sviluppo principale, gestita direttamente da Linus Torvalds. Queste versioni contengono tutte le ultime funzionalità introdotte al kernel che potrebbero però non essere ancora sufficientemente mature, motivo per il quale il loro utilizzo è scosigliato ai normali utenti.

- **Prepatch** (RC): si tratta versioni mainline candidate a "stable" ovvero che sembrano essere sufficientemente mature. Anche queste release sono supervisionate direttamente da Linus Torvalds.

- **Stable**: ogni 2 o 3 mesi viene rilasciata dalla mainline una versione considerata stabile. Queste versioni vengono modificate solo nel caso in cui dovessero emergere dei bug. I possibili bug fix vengono importati direttamente dalla mainline (backporting).

- **Longterm** (LTS): esistono un certo numero di versioni "stable" il cui supporto è garantito a lungo a termine. Queste sono le versioni che generalmente un utente desidera utilizzare, vengono aggiornate raramente e solo se strettamente necessario, tuttavia la loro bontà viene garantita fino alla loro end of life (EOL). Linus Torvalds affida generalmente la gestione di queste versioni a terzi.

Ogni versione del kernel è univocamente identificata da una serie di tre o quattro numeri che vengono indicati nel formato X.Y.Z.K. Il primo numero viene modificato raramente e solo in occasione di mutamenti radicali. L'introduzione di nuovi elementi è segnalata dall'incremento di uno dei due numeri successivi detti major e minor number, a seconda dell'entità delle novità. L'ultimo numero, se presente, indica invece che sono avvenuti dei cambiamenti minori. La presenza di alcune lettere poste alla fine può indicare una versione prepatch (suffisso "rc") oppure le iniziali della persona responsabile nel caso si tratti di una versione proveniente da un fork del progetto principale.

2.3.2 Installazione

Come già anticipato faremo uso di un sistema operativo Linux. Il primo passo consiste nel rimediare i sorgenti del kernel. Al novembre 2014, sui repository ufficiali U-Mobo, sono disponibili i sorgenti per la versione 3.0.35 del kernel Linux e per gli scopi prefissati si farà utilizzo di quest'ultimi. Al loro interno sono già inclusi tutti i driver e le modifiche per far funzionare correttamente le periferiche U-Mobo, comprese quelle rese disponibili da tutti gli ExM attualmente esistenti. Come per il bootloader, quello che si vorrà fare ora è eseguire una cross-compilazione su un sistema host differente:

```
$ git clone https://github.com/u-mobo/linux-imx
$ cd linux-imx
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- imx5_umobo_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- uImage
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- modules
```

Si otterrà così l'immagine kernel uImage e i moduli necessari. Infine servirà root filesystem (rootfs) della distribuzione Linux che si intende utilizzare. Si è scelto in questo caso Ubuntu 12.04 (attuale versione LTS). Il modo più semplice di ottenere un root filesystem è quello di

scaricarne uno già pronto, si può ad esempio utilizzare una delle release Linaro disponibili online. Un rootfs minimale di Ubuntu 12.04 con interfaccia grafica LXDE è disponibile al seguente link:

https://releases.linaro.org/12.04/ubuntu/precise-images/alip

Se si decide di procedere in questo modo si ricorda che per il primo login sarà necessario utilizzare l'account con username "linaro" e password "linaro", sarà poi possibile modificare tale impostazione in seguito. Per accedere all'utente root si dovrà invece innanzitutto dichiararne la password eseguendo il seguente comando come utente "linaro":

```
$ sudo passwd root
```

Verrà chiesto all'utente di specificare la nuova password due volte per questione di sicurezza. Al termine dell'operazione sarà possibile accedere all'utente root nel modo usuale.

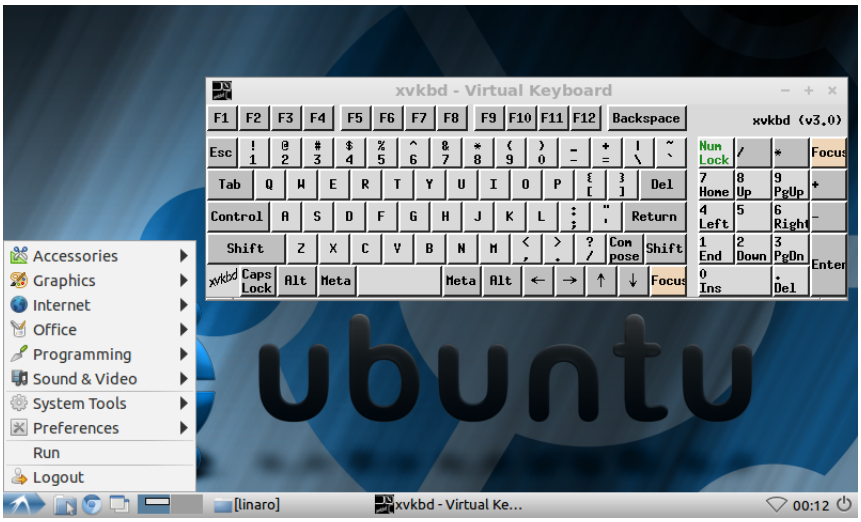

Figura 2.1. *Ambiente grafico LXDE su U-Mobo.*

2.4 Setup della microSD

Ora si procederà a mettere assieme tutti gli elementi raccolti fin'ora. E' necessario ora predisporre correttamente le partizioni sulla scheda microSD. Si dovrà creare una prima partizione FAT32 (chiamata boot) da cui il bootloader possa caricare l'immagine kernel e una seconda partizione EXT (chiamata rootfs) per ospitare il root filesystem. Per la prima

partizione saranno sufficienti 32MB mentre la seconda potrà estendersi per tutto il resto microSD. Nel fare questo sarà necessario aver cura di lasciare liberi i primi MByte della memoria (10MB saranno sufficienti) per far spazio al bootloader che dovrà essere posto proprio nelle prime locazioni di memoria, ad un offset di 1024 byte. Per creare questo layout è possibile utilizzare un qualsiasi software di partition manager, su sistemi Linux è disponibile ad esempio il programma *GParted*. La configurazione finale della microSD deve essere analoga a quella riassunta in figura 2.2.

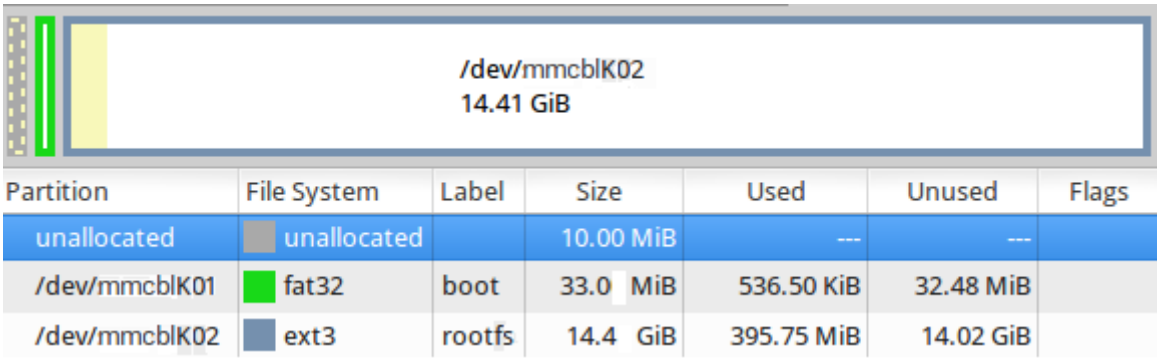

Figura 2.2. *Layout delle partizioni della microSD.*

Il passo successivo è scrivere il bootloader sulla microSD. Con i seguenti comandi si ottiene proprio questo, avendo l'accortezza di "azzerare" la parte di memoria interessata prima di sovrascriverci e inserire i dati con l'offset voluto. Si suppone di seguito che il lettore di schede SD utilizzato sia identificato dal device */dev/mmcblk0*.

```
$ sudo dd if=/dev/zero of=/dev/mmcblk0 bs=1M count=10
$ sudo dd if=u-boot.imx of=/dev/mmcblk0 bs=512 count=2
```

A questo punto è sufficiente copiare l'immagine kernel uImage nella partizione FAT32 e l'intero rootfs in quella EXT. I moduli ottenuti durante la compilazione andranno integrati all'interno del rootfs, copiando semplicemente i file all'interno della cartella */usr/modules* situata sulla microSD.

La microSD è ora pronta ad essere utilizzata. Tuttavia è necessario comunicare al bootloader dove l'immagine kernel si trova e come utilizzarla. Un modo per fare questo è accedere, tramite porta seriale, al terminale dell'U-Boot per eseguire le operazioni richieste. Per evitare questo passaggio e rendere automatico il processo di boot è possibile tuttavia creare un file

di script che il bootloader si preoccuperà di eseguire automaticamente ad ogni avvio. Per fare ciò è necessario innanzitutto creare un file (in questo caso *boot.txt*) contenente i seguenti comandi U-Boot:

```
# boot.txt - Boot script file
fatload mmc 0:1 0x71000000 uImage
setenv bootargs console=ttymxc3,115200n8 root=/dev/mmcblk0p2 rootwait
        video=mxcfb0:dev=lcd,SAMSUNG-LMS700,if=RGB24
        da9052_i2c_addr=$da9052_i2c_addr quiet splash
bootm 0x71000000
```

Con il primo comando (*fatload*) si chiede al bootloader di caricare in memoria RAM, a partire all'indirizzo specificato, l'immagine kernel che si trova nella prima partizione della microSD. Il secondo comando (*setenv*) si occupa invece di definire il valore della variabile bootargs, il contenuto di questa verrà automaticamente passata al kernel Linux come lista di argomenti per l'avvio. Infine l'ultimo comando (*bootm*) provvede ad eseguire il codice presente a partire dall'indirizzo dato, ovvero ad avviare, in questo caso, il kernel Linux.

L'ultima operazione da svolgere è convertire il file di testo appena creato nel formato riconosciuto dall'U-Boot. Questo è possibile farlo tramite il tool *mkimage* reso disponibile su sistemi Ubuntu tramite il package *uboot-mkimage*.

```
$ sudo apt-get install uboot-mkimage
$ mkimage -A arm -O linux -T script -C none -a 0 -e 0 -n 'Boot script' \
          -d boot.txt boot.scr
```

Il file *boot.scr* così creato andrà copiato nella partizione FAT32, a fianco all'immagine kernel. All'avvio il bootloader, per default, controlla l'esistenza di un file chiamato *boot.scr* all'interno della prima partizione e nel caso lo esegue come file di script.

2.5 L'ambiente di lavoro

A questo punto è possibile lavorare direttamente sulla scheda U-Mobo: con l'ausilio di un hub USB è infatti possibile collegare direttamente mouse e tastiera e operare come se

si stesse utilizzando un normale PC. La limitazione più grande si rivela essere in genere lo schermo di piccole dimensioni. Per ovviare a questo è possibile fare uso di un monitor separato collegato all'uscita video della U-Mobo oppure collegare in rete LAN la scheda a un PC e interagire da questo. Per operazioni semplici si può anche utilizzare sulla scheda una testiera su schermo come *xvkbd* e sfruttare il touchsceen. Per la connessione a internet si può invece fare affidamento ad un collegamento Ethernet o, nel caso di presenza del modulo ExM WiFi-BT, utilizzare direttamente una connessione Wi-Fi.

Collegare la scheda U-Mobo in rete LAN con un PC può essere molto utile per trasferire facilmente file e, se necessario, condividere una connessione a internet. Ora si vedrà come configurare una rete di questo genere e come poterla sfruttare per semplificare alcune operazioni e poter creare un ambiente di lavorare ideale.

2.5.1 Configurazione di una rete locale

Per configurare la rete locale facendo uso del network manager di Linux è necessario, per entrambi i sistemi, andare nel menù "Edit Connections", selezionare la connessione wired e premere "Edit". Per quanto riguarda le impostazioni del PC è sufficiente impostare il campo "Method" a "Link-local only" oppure "Shared to other computers" se si intende condividere una connessione internet e premere infine il pulsante "Save...".
Prima di procedere con la configurazione dell'U-Mobo si dovrà guardare quali parametri di rete sono stati assegnati al PC. Per fare questo basta andare su "Information" dal menù a tendina del network manager e fare riferimento ai dati indicati per la connessione wired. I valori a cui bisogna prestare attenzione sono l'indirizzo IP e la netmask, che dovrebbero avere valori analoghi ai seguenti:

Indirizzo IP: 169.254.6.108
Netmask: 255.255.0.0

Sulla scheda U-Mobo è ora possibile configurare la rete inserendo un indirizzo IP adeguato (in questo caso ad esempio 169.254.6.109) e la stessa netmask. Nel caso si intenda condividere una connessione a internet si dovrà indicare anche l'IP del PC come gateway e specificare un server DNS (ad esempio 8.8.8.8). Le figure 2.3 e 2.4 mostrano come dovrebbero apparire le finestre di configurazione al termine di queste operazioni.

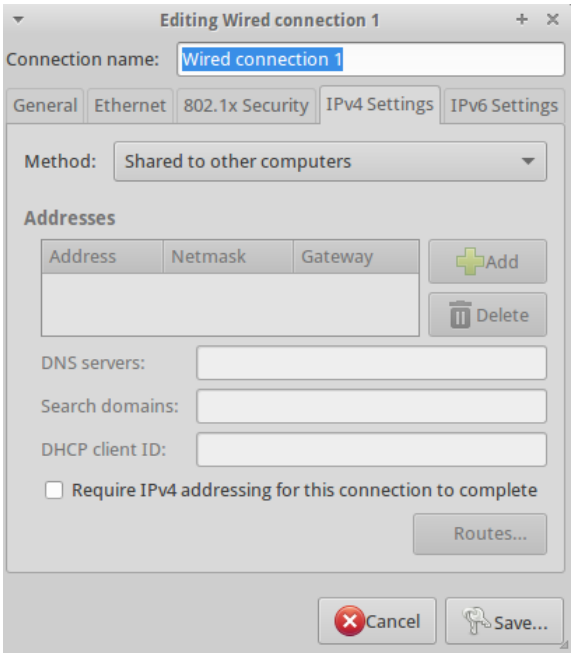

Figura 2.3. *Configurazione PC*

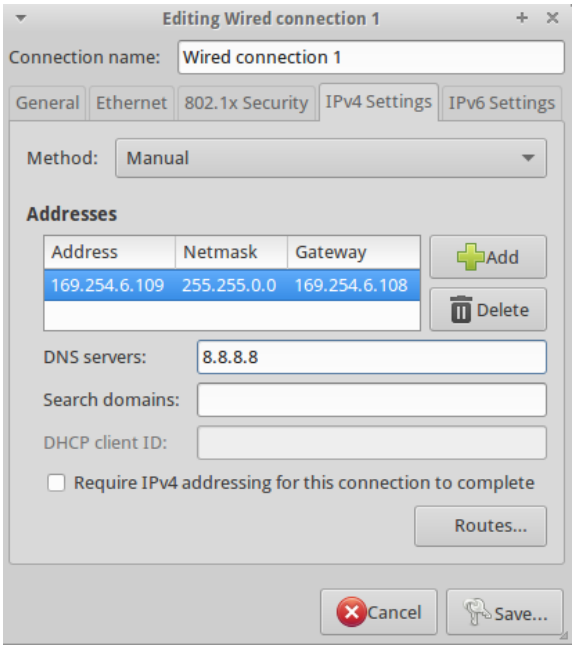

Figura 2.4. *Configurazione U-Mobo*

2.5.2 Strumenti utili

Per interagire con la scheda tramite un PC collegato in rete con quest'ultima è possibile utilizzare una serie di comodi strumenti fra i quali:

- TightVNC: questo tool permette di visualizzare su schermo il desktop di un sistema remoto e agire su di esso. Sulla U-Mobo si vorrà installare il server TightVNC e avviarlo nel seguente modo:

  ```
  $ sudo apt-get install tightvncserver
  $ sudo /usr/bin/tightvncserver
  ```

  Al primo avvio verrà chiesto di specificare una password per accedere in remoto al servizio. Fatto questo si potrà ora accedere dal PC al desktop della U-Mobo specificandone l'indirizzo IP seguito dall'identificativo dello schermo interessato (in questo caso dovrebbe necessario aggiungere semplicemente :1) in questo modo:

  ```
  $ tightvncserver 169.254.6.109:1
  ```

  In seguito a questo comando sarà semplicemente necessario inserire la password precedentemente scelta nella fase di installazione del server.

- OpenSSH: il protocollo ssh permette di aprire un terminale si un sistema su un computer remoto. Anche in questo caso il server andrà installato sulla U-Mobo:

  ```
  $ sudo apt-get install opensshd
  ```

  Dal PC sarà ora possibile collegarsi specificando l'utente remoto con il quale accedere e l'indirizzo IP della board.

  ```
  $ ssh linaro@169.254.6.109
  ```

  Verrà richiesto a questo punto di inserire la password dell'account dell'utente scelto. Una volta all'interno della shell remota, se si desidera avviare un programma che presenta un'interfaccia grafica si dovrà prima ridefinire una variabile d'ambiente per specificare il display da utilizzare. Il comando per fare questo è il seguente:

  ```
  $ export DISPLAY:=0
  ```

  In questo modo la parte grafica del programma verrà visualizzata sul server, ovvero sullo schermo della U-Mobo. Se si vuole invece vedere l'interfaccia grafica sul PC, al comando ssh per collegarsi si dovrà aggiungere il flag *-X*. In questo caso non sarà necessario modificare alcuna variabile d'ambiente.

  ```
  $ ssh -X linaro@169.254.6.109
  ```

- FTP: per trasferire dati fra i due sistemi può risultare comodo installare un server FTP sulla U-Mobo.

  ```
  $ sudo apt-get install ftpd
  ```

  Per il trasferimento dei file il modo più semplice è quello di installare sul PC un client FTP con interfaccia grafico. Un software di questo genere è, ad esempio, FileZilla. Al suo avvio, per la connessione, verranno richiesti come di consueto l'indirizzo IP della U-Mobo e username e password dell'account remoto con cui collegarsi.

CAPITOLO 3

APPLICAZIONI NEI SISTEMI ROBOT

3.1 Sistemi embedded nei robot

Disponendo di un sistema operativo perfettamente funzionante è possibile ora svolgere tutte le operazioni desiderate nativamente sulla piattaforma U-Mobo, non ci sarà più bisogno di un cross-compilatore installato su un secondo sistema. Le prestazioni della board sono tali che il semplice utilizzo di solo quest'ultima risulta in genere preferibile.
Si vuole ora illustrare una particolare applicazione alla quale un sistema di questo genere si presta. L'obiettivo sarà quello di integrare in un'unità U-Mobo il software necessario a utilizzare il sensore di movimento Kinect per svolgere un'attività di tracking e riconoscimento delle figure umane. Il software che implementa gli algoritmi di riconoscimento, sviluppato presso il laboratorio di robotica autonoma dell'Università di Padova, è stato sviluppato e funziona attualmente su PC e sistema operativo Linux. L'impiego di un sistema host più compatto potrebbe essere desiderabile per certi tipi di applicazione.
Una possibilità già stata messa in pratica è quella di fornire a un robot capace di muoversi la possibilità di riconoscere una determinata persona e identificare la sua posizione in modo da seguirla nei suoi spostamenti. Un simile sistema può essere utilizzato per un vasto numero di applicazioni pratiche. Un robot di questo genere può portare assistenza di vario genere, ad esempio trasportando per la persona interessata attrezzatura pesante o bagagli. In questo caso sarebbe sicuramente importante il tempo di autonomia e il consumo energetico dell'hardware impiegato risulta di conseguenza un fattore da tenere in considerazione.

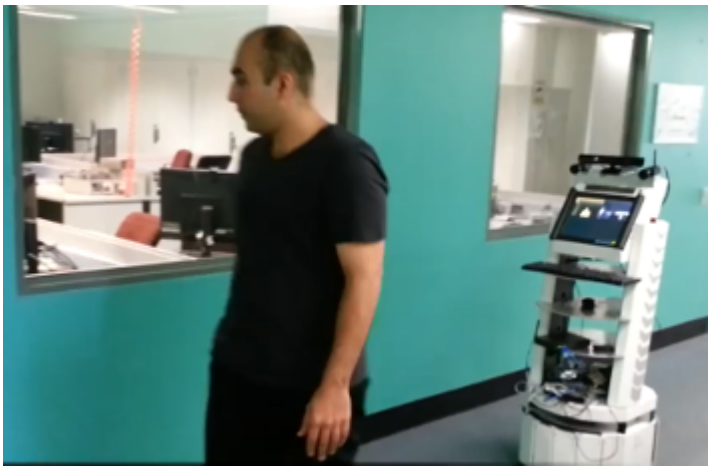

Figura 3.1. *Robot capace di seguire un taget sfruttando la Kinect.*

3.2 Integrazione del framework ROS

3.2.1 Il framework

ROS (Robot Operating System) è un framework per lo sviluppo di software robot che include una vasta gamma di software comunemente identificata con il nome di robotics middleware. Analogamente a quanto fatto da un sistema operativo, ROS fonisce astrazione dell'hardware, primitive di basso livello per il controllo dei device, implementazioni di funzionalità di comune utilizzo, message-passing fra processi, package management e molto altro. I processi in esecuzione all'interno del framework sono rappresentati da una struttura a grafo dove l'esecuzione è a carico dei nodi che hanno la possibilità di ricevere, pubblicare ed elaborare dati, eseguire operazioni di controllo, pianificare e agire tramite attuatori o l'invio di messaggi di notifica. Nonostante l'importanza dei tempi di risposta nelle applicazioni robot, ROS non presenta specifiche real-time, tuttavia è possibile integrare del codice capace di fornire garanzie di questo genere. Il software proposto da ROS può essere identificato in tre diversi gruppi:

- Una serie di tool platoform-independent utilizzati per la compilazione e la gestione del software che costituisce il framework e che è stato sviluppato al suo interno.

- Implementazioni delle più comuni librerie, adattate per le particolari necessità del framework. Esempi di queste librerie sono roscpp, rospy e roslisp.

- Package contenenti applicazioni di vario genere: questi sono spesso sviluppati dalla community ROS e poi resi pubblicamente disponibili. Si appoggiano alle funzionalità base proposte dal core software di ROS e possono a loro volta essere integrati in package più grandi per sviluppare programmi più complessi e articolati.

Le prime due categorie di software sono rilasciate sotto licenza BSD, e costituiscono dunque materiale open-source di libero utilizzo, sia per applicazioni commerciali che per scopi di ricerca. La maggior parte del software rimanente è anch'esso rilasciato come open-source, sotto varie diverse licenze.

Le principali librerie incluse in ROS (C++, Python e LISP) sono legate alla presenza di un sistema sottostante Unix-like; questo è dovuto principalmente alle dipendenze verso una larga collezione di software open-source presenti nativamente solo in questi ambienti. Per questo motivo sistemi operativi come Ubuntu Linux sono indicati come supportati, mentre altri come Fedora Linux, Mac OS X e Microsoft Windows hanno attualmente supporto solamente sperimentale. Per questi ultimi la community rimane comunque disponibile a fornire l'aiuto necessario per risolvere eventuali problemi.

Figura 3.2. *Nextage, robot basato su software ROS.*

Fa eccezione l'implementazione ROS della libreria Java, rosjava. Questa non presenta limitazioni legate al particolare sistema operativo in uso, ed ha recentemente permesso di scrivere software ROS capace di funzionare su Android OS. Inoltre è stato possibile alcune delle funzionalità del framework all'interno del software MATLAB, tramite una toolbox ufficialmente supportata che può essere utilizzato su Linux, Mac OS X e Microsoft Windows. Di particolare interesse risulta anche il package roslibjs, implementazione della libreria Java-

script, con la quale è possibile portare in esecuzione applicazioni ROS su un qualsiasi browser standard.

ROS è stato originalmente sviluppato nel 2007 dallo Stanford Artificial Intelligence Laboratory sotto il nome di Switchyard. Fra il 2008 e il 2013, lo sviluppo del software è stato nelle mani di Willow Garage, centro di ricerca per applicazioni robot. Durante questo periodo tuttavia, ricercatori provenienti da più di 20 istituti diversi hanno collaborato in diversi modi all'estensione del framework. Nel febbraio del 2013, l'amministrazione dello sviluppo di ROS passò alla Open Source Robotics Foundation.

Attualmente ROS è impiegato in un grande numero di applicazioni robot dalle più svariate capacità e utilizzi. Una lista completa di tutti questi robot è resa disponibile al seguente indirizzo:

http://wiki.ros.org/Robots

3.2.2 Installazione su sistemi ARM

Lo scopo dell'unità U-Mobo è chiaramente quello di eseguire il software di tracking. Questo è stato sviluppato utilizzando il framework ROS e di conseguenza la sua integrazione nel sistema che si andrà a creare si rivela essere un requisito necessario. Come già detto, il core di ROS è rilasciato sotto una licenza BSD open source ed è reso di conseguenza disponibile direttamente tramite repository Linux. Di seguito sono indicate le operazioni per installare la versione Hydro di ROS per Ubuntu Precise (12.04):

```
  Impostazione del locale
$ sudo update-locale LANG=C LANGUAGE=C LC_ALL=C LC_MESSAGES=POSIX
  Impostazione repository ROS
$ sudo sh -c 'echo "deb http://packages.namniart.com/repos/ros \
      precise main" > /etc/apt/sources.list.d/ros-latest.list'
$ wget http://packages.namniart.com/repos/namniart.key -O - | sudo apt-key add -
$ sudo apt-get update
  Installazione ROS e gestore pacchetti
$ sudo apt-get install ros-hydro-ros-bsde python-rosdep
$ sudo rosdep init
```

```
$ rosdep update
  Setup variabili d'ambiente
$ echo "source /opt/ros/hydro/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

ROS utilizza il file */etc/lsb-release* per avere informazioni sul sistema operativo impiegato. Dato che alcune release, come quelle offerte da Linaro, modificano questo file è necessario controllare che segua il formato sotto riportato. I repository per ROS Hydro sono disponibili oltre che per Ubuntu Precise (12.04), anche per Quantal (12.10) e Raring (13.04); per quest'ultime due versioni è necessario modificare i passaggi visti di conseguenza.

 # /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=12.04
DISTRIB_CODENAME=precise
DISTRIB_DESCRIPTION="Ubuntu 12.0"

3.3 L'ambiente di sviluppo

Gli strumenti e il software all'interno del framework ROS sono organizzati in pacchetti. Per sviluppare un proprio pacchetto ROS è possibile fare uso di due diversi ambienti di lavoro: rosbuild e catkin. Più avanti ci si troverà a dover utilizzare entrambi questi ambienti, in particolare il software di tracking è stato sviluppato tramite rosbuild mentre per la compilazione di altre dipendenze si dovrà far uso di catkin. Di seguito si vedrà come inizializzare e utilizzare questi ambienti di sviluppo:

- **L'ambinte rosbuild**: per inizializzare un ambiente di questo tipo è sufficiente creare una cartella e informare ROS che si intende utilizzare quest'ultima come root per i propri pacchetti. Volendo creare questa cartella all'interno della propria home i passaggi sono i seguenti:

  ```
  $ mkdir ~/rosbuild_ws
  ```

```
$ cd ~/rosbuild_ws
$ rosws init . /opt/ros/hydro
$ echo "source ~/rosbuild_ws/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Il comando rosws init inizializza all'interno della cartella rosbuild_ws alcuni file necessari all'utilizzo dell'ambiente ed in particolare crea il file *setup.bash* di cui si dovrà fare il source. Ipotizzando di possedere la cartella dei sorgenti di un pacchetto che si vuole integrare in ROS, è sufficiente spostare quest'ultima direttamente in rosbuild_ws, registrare il pacchetto e compilare il tutto tramite il comando rosmake.

```
$ cd ~/rosbuild_ws
$ rosws set <nome-pacchetto>
$ roscd <nome-pacchetto>
$ rosmake
```

Tramite il comando rosws set si inizializza il nuovo pacchetto. Fatto questo è possibile spostarsi all'interno della cartella contenente i sorgenti utilizzando il pratico comando roscd.

- **L'ambinte catkin**: per l'inizializzazione i passi da fare sono molto simili. I comandi di seguito riportati, creano un ambiente catkin all'interno della cartella catkin_ws:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

Il comando catkin_make, invocato all'interno della cartella *catkin_ws*, compila tutti i sorgenti dei pacchetti presenti in catkin_ws/src e pone i file di output in *catkin_ws/build*. In questo caso sarà necessario fare il source del file *catkin_ws/devel/setup.bash*.

In entrambi i casi, a compilazione terminata con successo, il software viene automaticamente integrato all'interno del framework, funzionalità, strumenti e librerie così introdotti vengono resi disponibili immediatamente agli altri pacchetti. Se il pacchetto include degli eseguibili è possibile avviarli con l'aiuto del comando *rosrun* nel seguente modo:

```
$ rosrun <nome-pacchetto> <nome-eseguibile>
```

Spesso inoltre i pacchetti includono dei speciali file XML con estensione *.launch* che permettono di eseguire con i parametri specificati all'interno più programmi contemporaneamente in una sola volta. Questi hanno una funzione simile a quella di uno script che permette di automatizzare una serie di operazioni. Per avviare un launch file si utilizza invece il tool *roslaunch* in questo modo:

```
$ rosrun <nome-pacchetto> <nome-launch-file>
```

CAPITOLO 4

IL SOFTWARE DI TRACKING

4.1 La Kinect

Kinect (inizialmente conosciuto con il nome Project Natal) è un device pensato inizialmente per le console Microsoft Xbox 360, ideato per permettere agli utenti di interagire con i videogame senza l'utilizzo del controller. Tramite una serie di sensori e telecamere è capace di rilevare ed interpretare i movimenti del corpo umano e i comandi vocali. La prima versione della Kinect è stata rilasciata nel novembre 2010 con l'obbiettivo di rivoluzionare completamente l'esperienza di gioco offerta dalla console.

Figura 4.1. *Sensore Microsoft Kinect.*

Il device è formato da una barra orizzontale collegata a una piccola base motorizzata lungo l'asse verticale, permettendo di seguire i movimenti dei giocatori, orientandosi nella

posizione migliore per il riconoscimento dei movimenti. Il device incorpora una telecamera RGB di risoluzione 640x480, un sensore di profondità e un array di microfoni, utilizzato dal sistema per la calibrazione dell'ambiente in cui ci si trova, mediante l'analisi della riflessione del suono sulle pareti e sull'arredamento. Il sensore di profondità è formato da uno scanner laser a infrarossi e da una telecamera sensibile alla stessa banda, garantendo così una visione 3D dell'ambiente circostante che permette cattura dei gesti, riconoscimento facciale e molto altro.

Kinect è in grado di gestire simultaneamente fino a sei persone diverse, di cui però solamente due realmente "attive". Da queste vengono estratte e monitorate fino a 20 feature diverse costituenti le posizioni delle principali articolazioni del corpo. Dato che la porta USB della Xbox 360 non è in grado di fornire la potenza necessaria alla Kinect, è stato impiegato un particolare connettore proprietario che permette, oltre che il passaggio dei dati, l'utilizzo di un'alimentazione esterna. E' tuttavia disponibile un cavo che da questo connettore fornisce separatamente connessione all'alimentazione e uscita USB, in modo da permettere l'utilizzo del device anche su altri sistemi. Kinect è da ritenersi il primo device del suo genere pensato per poter essere prodotto in massa e distribuito al pubblico. Questo ha permesso alla Kinect di mantenere un costo molto basso considerando la strumentazione di cui è dotata, al suo lancio è stata infatti resa disponibile al pubblico a solo 149,99$. Ben presto è stata anche acquistata da molte università e centri di ricerca che hanno così potuto accedere a tecnologie altrimenti non facilmente accessibili.

A poca distanza dalla sua uscita Adafruit Industries si offrì di mettere in palio una ricompensa a chi avrebbe sviluppato un driver open-source per la Kinect. Héctor Martín, vincitore del contest, riuscì in breve tempo a creare un primo driver Linux che permetteva l'utilizzo sia della telecamera che del sensore di profondità.

4.2 Funzionamento del software

Il problema affrontato dal software people tracker è quello del video tracking. Il video tracking è il processo di individuazione di un oggetto o di una figura (o più di una) che si muove nel tempo, utilizzando una telecamera o sensori di altro genere. Un algoritmo analizza i frame del video e dà in uscita la posizione degli oggetti bersaglio. La difficoltà principale nel video tracking è quella di catturare la giusta posizione dei bersagli in fotogrammi consecutivi,

soprattutto quando gli oggetti si muovono velocemente rispetto al frame rate. I sistemi di video tracking utilizzano quindi spesso un modello del moto che descrive come l'immagine del bersaglio potrebbe cambiare per diversi possibili movimenti dell'oggetto da monitorare. Lo scopo dell'algoritmo di tracking è dunque quello di analizzare i fotogrammi del video al fine di stimare i parametri di movimento. Questi parametri caratterizzano in ogni istante la posizione del bersaglio.
Il software people tracking, in particolare, si occupa del riconoscimento delle figure umane. Il sistema è composto principalmente da due moduli: il detector e il tracker. Il detector prende in input i dati RGB-D (Red-Green-Blue-Depth) provenienti dai sensori della Kinect. Questi dati sono rappresentati da un insieme di punti (pointcloud) e il detector ha il compito di analizzarli al fine di rilevare la presenza di persone nella scena catturata. Per raggiungere questo scopo svolge le seguenti operazioni:

- Innanzitutto applica all'insieme dei dati RGB-D un filtro voxel grid. Questo filtro suddivide in porzioni la pointcloud e sostituisce tutti i punti all'interno di ognuna di queste con un unico punto rappresentativo. Questo permette di diminuire la quantità di dati senza perdere troppa informazione e di rendere la pointcloud più uniformemente distribuita nello spazio. In questo modo inoltre i passi successivi dell'algoritmo risultano computazionalmente meno pesanti.

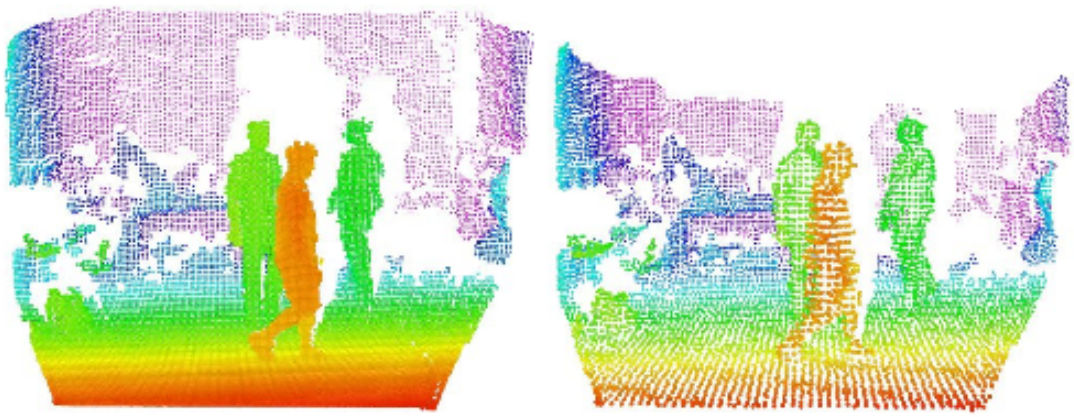

Figura 4.2. *Pointclaud prima e dopo l'applicazione del filtro voxel grid.*

- La parte dei dati facente parte del pavimento viene rimossa. Le porzioni della pointcloud rimanenti risultano in questo modo disgiunte e possono essere raggruppate tramite un

processo di clustering.

- Per ogni cluster di dati si svolge l'operazione di people detection tramite l'identificazione e la valutazione di feature HOG (Histogram of Oriented Gradients). In particolare viene prestata particolare attenzione al rilevamento della testa delle persone che risulta spesso utile per individuare situazioni in cui due persone diverse sono l'una davanti all'altra.

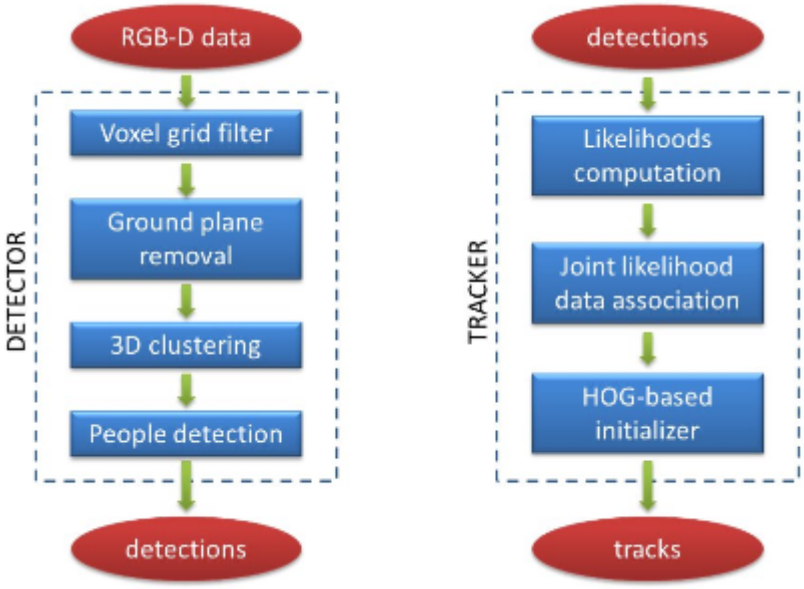

Figura 4.3. *Operazioni svolte dal software people tracker.*

Il risultato di queste operazioni è una serie di detections. Queste vengono prese in input dal tracker che ha il compito di stabilire le corrispondenze fra le rilevazioni successive, in modo da poter identificare gli spostamenti dei target. L'algoritmo riesce ad ottenere questo sfruttando un metodo basato sul calcolo di una funzione di verosomiglianza. Questa si basa, fra gli altri fattori, sulla previsione dei possibili spostamenti che una persona potrebbe fare. La ricerca delle associazioni viene fatta utilizzando un approccio del tipo Global Nearest Neighbor, utilizzando l'algoritmo Munkres. Se, per una data detection, non viene trovata un'associazione avente un livello di confidenza accettabile, viene creato un nuovo "track" ovvero si riconosce la presenza di una nuova persona. L'interfaccia grafica del software di tracking mostra costantemente le immagini della scena catturate dalla telecamera RGB della Kinect. Al riconoscimento di una persona, viene visualizzato a video un parallelepipedo che ne delimita lo spazio occupato.

4.3 Il driver Kinect

Ora sarà necessario installare il driver per la Kinect. Esiste un pacchetto ROS che include questo driver ma si dovrà provvedere a compilarlo esternamente al framework: è necessario infatti far uso di una versione del driver appositamente modificata per funzionare sull'architettura ARM. Si tratta di un fork dell'attuale progetto open-source che attualmente include moduli basati sulla libreria OpenNI per diversi tipi di sensori. Al momento è necessario fare uso della versione unstable di questo software che richiede a sua volta la versione unstable di OpenNI.

Per prima cosa bisogna accertarsi che tutte le dipendenze siano risolte installando i seguenti software:

```
$ sudo apt-get install gcc-multilib libusb-1.0.0-dev git-core build-essential
$ sudo apt-get install doxygen graphviz default-jdk freeglut3-dev libopencv-dev
```

Il codice sorgente della versione unstable di OpenNI è reperibile da repository git nel seguente modo:

```
$ cd ~
$ git clone git://github.com/OpenNI/OpenNI.git
$ cd OpenNI
$ git checkout unstable
```

A questo punto è necessario modificare il file *OpenNI/Platform/Linux/Build/Common/-Platform.Arm*, e portare la seguente modifica:

```
- "CFLAGS += -march=armv7-a -mtune=cortex-a8 -mfpu=neon -mfloat-abi=softfp
        # -mcpu=cortex-a8"
+ "CFLAGS += -march=armv7-a -mtune=cortex-a8 -mfpu=neon #-mcpu=cortex-a8"
```

Si dovrà cioè eliminare il flag gcc *-mfloat-abi=softfp* che abilita la generazione di codice hardware floating point. In questo modo il flag assumerà il valore di default che dipenderà automaticamente dalla configurazione del sistema target. Per compilare e installare si procede in questo modo:

```
$ cd ~/kinect/OpenNI/Platform/Linux/CreateRedist
$ sudo ./RedistMaker.Arm
$ cd ~/kinect/OpenNI/Platform/Linux/Redist/OpenNI-Bin-Dev-Linux-Arm-v1.5.4.0
$ sudo ./install.sh
```

In modo analogo a quanto fatto per OpenNI è ora possibile installare il driver vero e proprio.

```
$ cd ~
$ git clone git://github.com/avin2/SensorKinect.git
$ cd SensorKinect
$ git checkout unstable
```

Anche in questo caso bisogna modificare il file *SensorKinect/Platform/Linux/Build /Common/Platform.Arm*, ovvero si dovrà nuovamente eliminare la definizione del flag *-mfloat-abi=softfp* dalla stessa linea vista in precedenza. Fatto questo è possibile procedere alla compilazione:

```
$ cd ~/kinect/SensorKinect/Platform/Linux/CreateRedist
$ sudo ./RedistMaker
```

Un'ultima modifica, specifica per i sistemi ARM, va ora fatta al file *SensorKinect/Platform/Linux/ Redist/Sensor-Bin-Linux-Arm-v5.1.2.1/Config/GlobalDefaultsKinect.ini*. La seguente linea deve essere modificata in questo modo:

```
- ;UsbInterface=2
+ UsbInterface=1
```

Infine è possibile installare il software:

```
$ cd ~/kinect/SensorKinect/Platform/Linux/Redist/Sensor-Bin-Linux-Arm-v5.1.2.1
$ sudo ./install.sh
```

Per collaudare il driver è possibile avviare un semplice programma di "sample" che dovrebbe essere in grado di leggere dei dati direttamente dal sensore Kinect.

```
$ cd ~/kinect/OpenNI/Platform/Linux/Redist/OpenNI-Bin-Dev-Linux-Arm-v1.5.4.0
      /Samples/Bin/Arm-Release
$ ./Sample-NiSimpleRead
```

Nel caso in cui tutto sia andato nel verso giusto, l'output del programma dovrebbe essere simile a quello mostrato in figura 4.4.

```
Frame 73 Middle point is: 910. FPS: 30.807238
Frame 74 Middle point is: 910. FPS: 30.810444
Frame 75 Middle point is: 913. FPS: 30.813171
Frame 76 Middle point is: 913. FPS: 30.772820
Frame 77 Middle point is: 913. FPS: 30.778591
Frame 78 Middle point is: 913. FPS: 30.781240
Frame 79 Middle point is: 913. FPS: 30.744371
Frame 80 Middle point is: 910. FPS: 30.748817
Frame 81 Middle point is: 913. FPS: 30.750669
Frame 82 Middle point is: 913. FPS: 30.716995
Frame 83 Middle point is: 915. FPS: 30.721954
Frame 84 Middle point is: 913. FPS: 30.724741
Frame 85 Middle point is: 910. FPS: 30.691582
Frame 86 Middle point is: 913. FPS: 30.695990
Frame 87 Middle point is: 913. FPS: 30.699308
Frame 88 Middle point is: 915. FPS: 30.668604
Frame 89 Middle point is: 913. FPS: 30.671824
Frame 90 Middle point is: 908. FPS: 30.676571
Frame 91 Middle point is: 913. FPS: 30.644972
Frame 92 Middle point is: 913. FPS: 30.734095
Frame 93 Middle point is: 915. FPS: 30.737803
Frame 94 Middle point is: 913. FPS: 30.707314
Frame 95 Middle point is: 913. FPS: 30.749260
```

Figura 4.4. *Output per Sample-NiSimpleRead*

4.4 I pacchetti ROS

I pacchetti ROS sono installabili direttamente da terminale tramite l'ausilio del programma *apt-get* e sono riconoscibili dal prefisso comune ros-hydro-. Per avere una lista completa dei pacchetti ROS disponibili è dunque sufficiente utilizzare il seguente comando:

```
$ sudo apt-cache search ros-hydro-
```

Per installare un particolare pacchetto si procederà invece in questo modo:

```
$ sudo apt-get install ros-hydro-<nome-pacchetto-ROS>
```

E' necessario ora fornire al framework ROS tutti gli strumenti necessari per la gestione delle immagini e dei dati provenienti dalla Kinect. Al momento, sul repository ARM per ROS hydro, è presente un conflitto fra due pacchetti Linux chiamati *libopenni-dev* e *openni-dev*. Sebbene questi sembrino essere del tutto equivalenti, alcune componenti necessarie risultano dipendere dal primo mentre altre dal secondo. Non potendo coesistere entrambe le versioni della libreria nello stesso sistema, non risulta possibile fare completamente affidamento al programma apt-get che richiederebbe l'eliminazione del primo dei due nel momento in cui si cercasse di installare il secondo.

La scelta è stata quella di installare normalmente i pacchetti che dipendono da *openni-dev* e compilare a partire dai sorgenti tutti gli altri. Di seguito sono riportati i pacchetti ROS che sarà necessario ottenere tramite compilazione e i relativi repository Git per accedere ai sorgenti:

geometry, https://github.com/ros/geometry.git
image pipeline, https://github.com/ros-perception/image_pipeline.git
openni camera, https://github.com/ros-drivers/openni_camera.git
openni launch, https://github.com/ros-drivers/openni_launch.git

Questi sorgenti sono basati sull'ambiente di sviluppo catkin di ROS. Utilizzando quanto fatto precedentemente per predisporre questo ambiente, è necessario ora collocare il codice sorgente nella cartella /catkin_ws/src ricordandosi di utilizzare il branch corretto per la versione di ROS che si sta utilizzando. Per il package *geometry*, ad esempio, si dovranno eseguire quindi i seguenti passi:

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ros/geometry.git
$ cd geometry
$ git checkout hydro-devel
```

Sarà necessario procedere analogamente per tutti i pacchetti sopra indicati. Al termine di queste operazioni si potrà procedere al controllo delle dipendenze necessarie alla compilazione. Per fare questo è possibile consultare i file XML package.xml presenti all'interno della cartella principale dei file sorgente. In questi file sono indicati racchiusi dai tag ⟨run_depend ⟩ e

‹build_ depend › tutti i pacchetti necessari. Di questi si dovrà avere l'accortezza di ignorare libopenni-dev, poiché rimpiazzato da *openni-dev*, oltre a tutti i pacchetti che sono già stati scaricati sotto forma di sorgenti. Per le rimanenti dipendenze si potrà fare uso di *apt-get* senza alcun problema. Nel caso in cui dovessero mancare altre componenti necessarie, la compilazione dovrebbe fallire lasciando indicazioni piuttosto chiare riguardo le dipendenze non ancora risolte.

Prima di procedere con la compilazione si dovrà prima però portare qualche modifica al file *CMakeLists.txt* del package *openni_camera*. In pratica si vorrà fare in modo che non venga fatto il controllo della presenza di libopenni-dev e che venga utilizzata la versione di openNI precedentemente compilata manualmente. A questo scopo si sostituiscono la cartella degli header file e quella delle librerie utilizzate di default da ROS con quelle desiderate.

```
– pkg_check_modules(PC_LIBOPENNI-DEV REQUIRED libopenni-dev)
      ...
– ${libopenni-dev_INCLUDE_DIRS}
+ /usr/include/ni
      ...
– ${OPENNI_LIBRARIES}
+ /usr/lib/libOpenNI.jni.so
+ /usr/lib/libnimMockNodes.so
+ /usr/lib/libnimRecorder.so
+ /usr/lib/libOpenNI.so
+ /usr/lib/libnimMockNodes.so
```

Fatto questo è possibile procedere alla compilazione:

```
$ cd ~/catkin_ws/src
$ catkin_make
```

Un'ulteriore modifica va fatta inoltre a uno dei launch file inclusi nel package *openni_launch*. In particolare si dovrà modificare il file *openni_launch/launch/openni.launch* nel seguente modo per abilitare di default l'ingresso dei dati relativi alla profondità:

```
- ⟨arg name="depth_registration" default="false" /⟩
+ ⟨arg name="depth_registration" default="true" /⟩
```

4.5 Test della funzionalità

All'interno del framework ROS, a ogni flusso di dati viene associato un topic. In ogni istante è possibile avere la lista dei topic pubblicati dalle varie applicazioni tramite il seguente comando:

```
$ rostopic list
```

Per verificare che tutto funzioni correttamente fino a questo punto, è possibile fare la seguente prova con la Kinect collegata alla scheda U-Mobo. Per prima cosa si avvia lo script che inizializza la connessione con il device e pubblica i topic ROS relativi ai flussi dati.

```
$ roslaunch openni_launch openni.launch
```

Dall'output di questo comando dovrebbe risultare chiaro che la Kinect è stata riconosciuta correttamente. Infine per essere sicuri che i dati giungano correttamente alla scheda e possibile provare a visualizzare a schermo le immagini catturate dalla telecamera della Kinect. Per fare questo è necessario lasciare in esecuzione il processo appena avviato e con l'aiuto di un secondo terminale si può dunque utilizzare il seguente comando:

```
$ rosrun image_view image_view image:=/camera/rgb/image_color
```

Il programma *imageview* si occupa di prelevare i dati dal topic ROS */camera/rgb/image_color* è visualizzarli come immagini video. Per verificare la funzionalità degli altri topic relativi alla Kinect, è possibile ad esempio chiedere a ROS il rate con il quale i dati arrivano. Con il seguente comando si vuole sapere il rate del topic */camera/depth/points* relativo ai valori di profondità rilevati dalla Kincet.

```
$ rostopic hz /camera/depth/points
```

I dati che giungono al topic */camera/depth/points* sono strutturati come un'insieme di punti (pointcloud) e non sono quindi riconducibili immediatamente a un'immagine da visualizzare tramite il programma *imageview*. Si può tuttavia utilizzare un secondo tool grafico chiamato *rviz* che dà la possibilità di avere una rappresentazione grafica di topic ROS di varia natura.

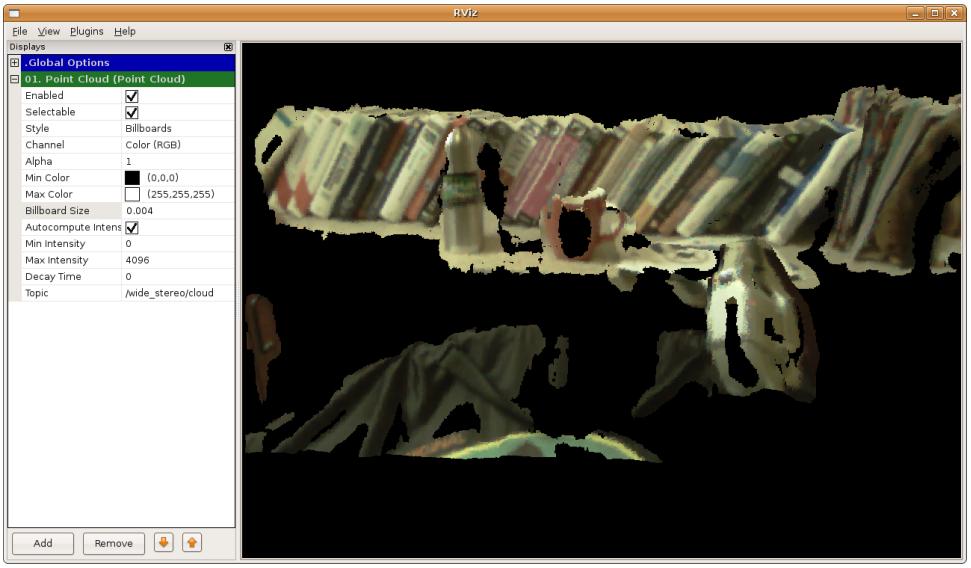

Figura 4.5. *People tracker su U-Mobo*

Per installare questo software sarà necessario compilarne i sorgenti poiché, anche in questo caso, si dovrà apportare a questi una modifica specifica per i sistemi ARM. Il repository interessato è il seguente:

https://github.com/ros-visualization/rviz.git

Per la compilazione si procede come visto in precedenza per i package catkin. Prima di invocare il comando *catkin_make* si dovranno però aggiungere le seguenti righe di codice al file sorgente *rviz/src/rviz/mesh_loader.cpp*, subito dopo la lista degli statement *#include* già presenti.

```
# ifdef __arm__
#include ⟨strings.h ⟩
bool Assimp::IOSystem::ComparePaths(const char *p1, const char *p2) const {
```

```
    return !::strcasecmp(p1, p2);
}
# endif
```

A compilazione terminata è possibile avviare il programma tramite rosrun:

```
$ rosrun rviz rviz
```

Per visualizzare una rappresentazione 3D dei valori di profondità è sufficiente ora selezionare dall'interfaccia grafica */camera_depth_optical_frame* come fixed frame, aggiungere un display di tipo *PointCloud2* tramite il pulsante "Add" in basso a sinistra e impostarne il topic correttamente come */camera/depth/points*. Si ricorda che, affinché i dati vengano raccolti, il launch file *openni.launch* deve essere avviato tramite *roslaunch* e lasciato in esecuzione separatamente.

4.6 Il software people tracker

Il software di tracking è composto da due pacchetti: *people_tracker* e *utils*. Entrambi sono stati sviluppati con l'ausilio dell'ambiente di sviluppo *rosbuild*. I sorgenti vanno quindi posti all'interno della cartella rosbuild_ws. Per la lista completa delle dipendenze per la compilazione è possibile consultare i file XML manifest.xml presenti nei sorgenti dei due pacchetti e fare riferimento a quanto riportato fra i tag $\langle depend \rangle$. In questo caso non sarà necessario fare alcuna modifica e si potrà procedere con la compilazione tramite *catkin_make*. Il launch file da utilizzare per avviare il software è in questo caso *one_kinect.launch*. Questo si occuperà anche di lanciare automaticamente *openni_launch* che quindi non dovrà essere più eseguito separatamente.

```
$ roslaunch people_tracker one_kinect.launch
```

La prima finestra che si aprirà sarà il ground detector. Verrà mostrato il video preso dalla telecamera della Kinect, su questo si dovranno indicare con il click sinistro del mouse tre punti a piacimento che si trovano sul pavimento per permettere la calibrazione del software. Con un click destro del mouse si conferma la scelta. Si avvieranno ora il detector e il tracker che si

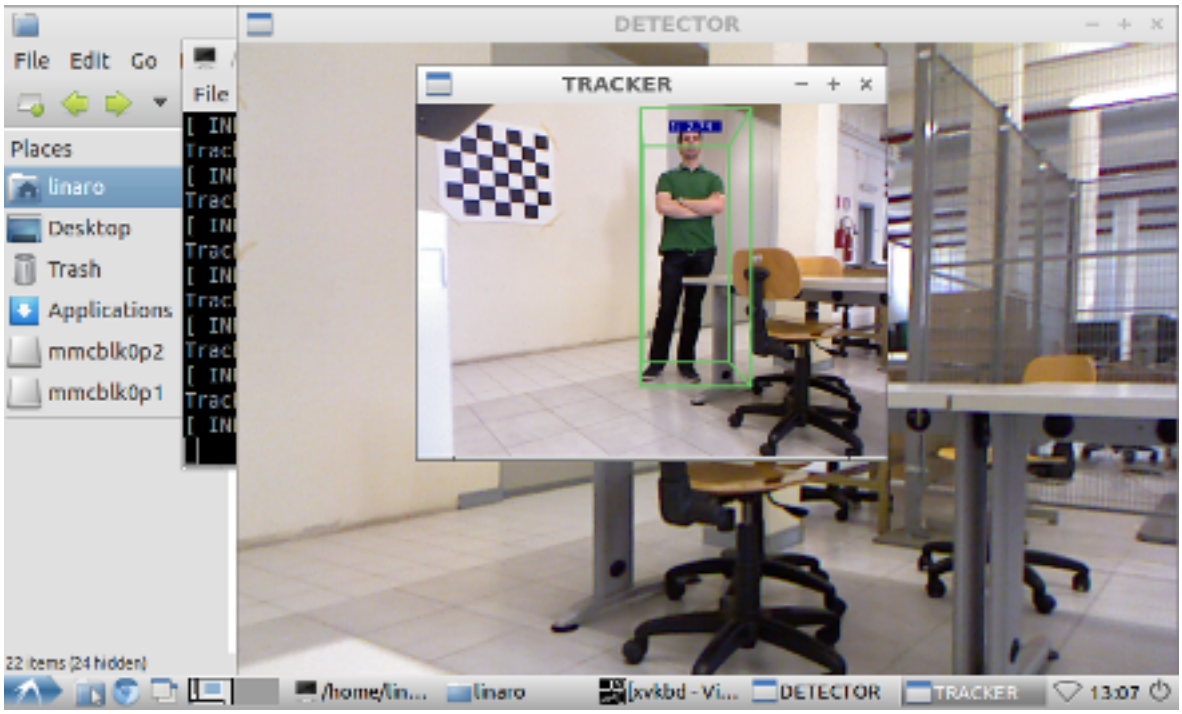

Figura 4.6. *People tracker su U-Mobo*

occuperanno del riconoscimento e del tracciamento delle persone. Il software completamente in funzione sulla scheda U-Mobo è mostrato in figura 4.6.

CAPITOLO 5

APPLICAZIONI DEL SISTEMA

5.1 Struttura del codice

A questo punto si dispone di un sistema compatto capace di svolgere delle operazioni di tracking delle persone. Il software installato rende disponibili, grazie al sensore Kinect, tutte le informazioni riguardanti le persone che vengono identificate. E' possibile sfruttare queste informazioni per una serie di applicazioni di interesse pratico. Il modo più semplice di fare questo è quello di aggiungere il codice opportuno direttamente fra il codice sorgente del software people tracker. Di seguito si presenteranno ora le due principali classi C++ del software a cui si vorrà fare particolare attenzione per l'estrazione dei dati necessari e per l'inserimento di eventuali nuove funzioni.

5.1.1 La classe Target

Come già accennato, il software, ad ogni frame, identifica le figure umane presenti nell'obiettivo della Kinect. Per ognuna di queste rilevazioni viene creato un oggetto C++ di tipo *people tracking::Detection* (definito all'interno dei file sorgente *Detection.cpp* e *Detection.h*) che incapsula tutti i dati necessari all'identificazione della figura. A ogni rilevazione riconosciuta come valida, viene poi assegnato un target che rappresenta uno specifico individuo. Analogamente i target sono rappresentati, all'interno dei sorgenti, dalla classe *people_tracking::Target* (*Target.cpp* e *Target.h*). Per ogni oggetto *Detection* quindi, vengono svolte le operazioni necessarie per stabilire se la relativa rilevazione è associabile a un target

già conosciuto. Se il target era già stato rilevato in passato, i dati (posizione, stato ecc.) all'interno della relativa classe *Target* vengono aggiornati, altrimenti viene creata una nuova istanza *Target*. Per i scopi prefissati i dati contenuti nelle singole istanze *Detection* non sono di particolare interesse. Può risultare invece utile poter accedere alle informazioni disponibili sui target individuati e per far questo è necessario prestare attenzione ad alcuni metodi e variabili della classe *Target*. Di seguito sono riportate le principali informazioni che è possibile ottenere da questa classe:

- **Id**: a ogni target viene associato univocamente un numero identificativo. Questo dato è memorizzato nella variabile d'istanza *_id*, accessibile tramite il metodo pubblico *Target::getId()*.

- **Visibilità**: in ogni istante è possibile sapere se il target è presente nella scena catturata dalla telecamera. Il metodo pubblico *Target::getVisibility()* restituisce un valore di tipo *Visibility* definito all'interno del file *Target.h* come enum C++. I possibili valori di questo dato sono tre: *VISIBLE*, *OCCLUDED* e *NOT_VISIBLE*. Come si può immaginare, lo stato "occluded" sta ad indicare che il target è attualmente presente ma solo parzialmente visibile.

- **Tempi**: i due metodi *Target::getSecFromFirstDetection(ros::Time currentTime)* e *Target::getSecFromLastDetection(ros::Time currentTime)* forniscono rispettivamente il tempo al quale il target è stato rilevato per la prima volta e per l'ultima. Restituiscono dei valori float che rappresentano i secondi trascorsi fra l'avvio del programma e l'evento interessato. Entrambi i metodi richiedono come parametro un riferimento temporale di tipo *ros::Time* relativo al tempo corrente. Tale riferimento è ottenibile tramite la funzione *ros::Time::now()*.

- **Posizione**: il modo più semplice per ottenere le coordinate della posizione attuale del target è quello di utilizzare il metodo *Target::getPointXYZRGB(pcl::PointXYZRGB& p)*. Restituisce un valore booleano false se il target non è visibile in quel momento, true altrimenti. In quest'ultimo caso i dati interessati vengono passati tramite la variabile *pcl::PointXYZRGB* passata come argomento per riferimento. Le coordinate x, y e z sono accessibili da una oggetto *pcl::PointXYZRGB p* tramite le sue omonime variabili d'istanza pubbliche (i.e. *p.x*, *p.y* e *p.z*). Il punto esattamente davanti alla Kinect

ha coordinate (0, 0, 0) quindi i valori x e y possono assumere valori sia positivi che negativi. Il campo visivo del sensore è rappresentabile da un cono centrato sull'obiettivo di apertura di circa 90 gradi.

5.1.2 La classe Tracker

La classe *Tracker* (*Tracker.cpp* e *Tracker.h*) si occupa della gestione dei target. Di particolare interesse sono i metodi di questa classe che si occupano di manipolare e creare le istanze *Target*, da questi è possibile agganciarsi con codice supplementare per gestire eventi di vario genere. Di seguito gli elementi della classe *Tracker* a cui può essere necessario fare attenzione:

- **Lista dei target**: *std::list ⟨people_tracking::Target* ⟩ _targets* rappresenta la lista di tutti i target identificati durante il tempo di esecuzione del programma. Si tratta di un membro di classe dichiarato *protected* perciò, se si richiede di accedere a questo dato dall'esterno, sarà necessario ridefinirlo come *public* oppure definire un nuovo metodo pubblico che lo restituisca.

- **createNewTarget**: il metodo *Tracker::createNewTrack()* viene evocato ogni qualvolta una valida detection non trova un'associazione con un target già esistente ed è necessario inizializzarne uno nuovo. Inserendosi all'interno di questo metodo si ha la possibilità di gestire l'ingresso in scena di nuove persone. In un sistema di sicurezza in cui si possiedono i dati riguardanti le persone autorizzate al passaggio, si può ad esempio prevedere lo scattare di un allarme alla rilevazione di figure estranee. Allo stesso tempo "l'intruso" può essere identificato e una sua foto può essere automaticamente salvata.

- **updateDetectedTracks**: il metodo *Tracker::updateDetectedTracks()* è forse, fra tutti, quello di maggior interesse. In questo metodo vengono aggiornati i dati relativi ai target conosciuti. Se si è interessati agli spostamenti di determinati target o all'uscita e all'ingresso di questi dal campo visivo, risulta facile sfruttare le chiamate a questo metodo per accedere alle informazioni aggiornate di volta in volta.

5.2 Esempio: customer tracker

Ora si presenterà un esempio di sviluppo di un'applicazione pratica del software di tracking. Si vuole creare un sistema capace di monitorare il flusso di clienti in ingresso e in uscita da un negozio o da un qualsiasi altro luogo. Per questo esempio si suppone banalmente che esista un unico ingresso e che il sistema fornito del device Kinect si trovi in una zona di pre-ingresso. Il sensore vi è posizionato in modo tale che, nella visuale della telecamera, l'ingresso si trovi sulla destra e l'uscita sulla sinistra. Si vuole che il sistema, in ogni istante, tenga traccia delle persone che hanno visitato il negozio, distinguendo per ciascuna se questa si trovi ancora all'interno o sia già uscita. Altre informazioni che si possono fornire sono ad esempio il numero totale di visitatori unici e, per il singolo cliente, il numero di volte che ha visitato il negozio e il tempo dell'ultima azione di ingresso o uscita.

5.2.1 Interfaccia grafica

Si vuole fornire una semplice interfaccia grafica che sintetizzi le informazioni raccolte. Per fare questo si utilizzerà la libreria *Qt*. Per rendere raggiungibile dall'interno del framework ROS una libreria esterna è necessario aggiungere le seguenti voci al file *CMakeLists.txt*:

```
find_package(Qt4 REQUIRED)
include_directories(/usr/include/qt4)
target_link_libraries(tracker $QT_QTCORE_LIBRARY $QT_QTGUI_LIBRARY)
```

In questo modo il compilatore verrà informato su dove si trovano gli header file aggiuntivi e della necessità di fare il linking delle librerie *Qt* per l'eseguibile *tracker*. Una semplice interfaccia grafica come quella mostrata in figura 5.1 è ottenibile tramite il seguente codice:

```
QApplication a(argc, argv);
QVBoxLayout *mainLayout = new QVBoxLayout;
QGroupBox *gr = new QGroupBox();
gr- ›setWindowTitle("Customer Tracking");
QTableWidget *tableWidget = new QTableWidget(0, 0);
```

```
tableWidget->resize(450, 300);
QStringList labels;
labels << "ID" << "Stato" << "Num. visite" << "Tempo" ;
tableWidget->setHorizontalHeaderLabels(labels);
QLabel *label = new QLabel();
label->setText("Clienti all'interno: 0 Totale clienti: 0");
label->setAlignment(Qt::AlignBottom, Qt::AlignLeft);
mainLayout->addWidget(tableWidget);
mainLayout->addWidget(label);
gr->setLayout(mainLayout);
gr->setContentsMargins (0, 0, 0, 0);
gr->resize(450, 300);
gr->show();
```

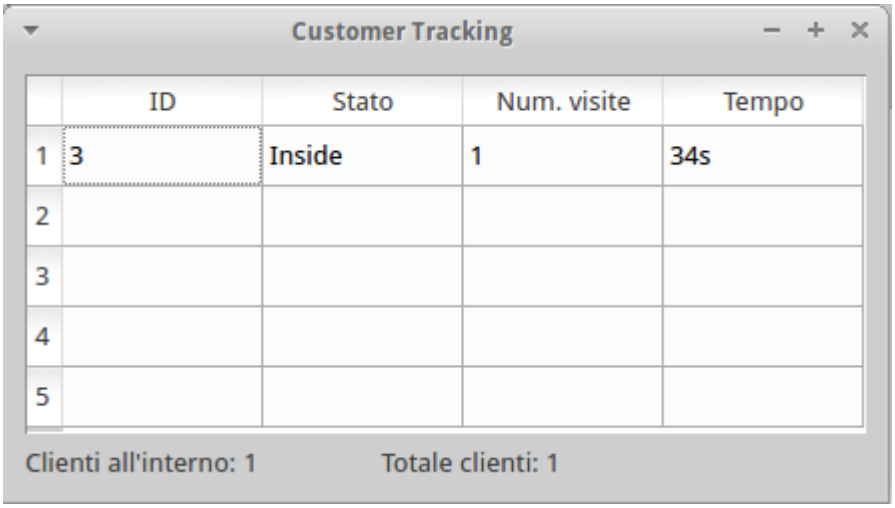

Figura 5.1. *Esempio di una semplice interfaccia Qt*

Questo codice crea la finestra desiderata. La creazione dell'oggetto *QApplication* va tipicamente fatta all'interno del metodo *main*, questo giustifica la presenza degli argomenti *argc* e*argv* nel suo costruttore. Per aggiungere una riga alla tabella e inserire del testo all'interno di una sua casella si procede come di seguito:

```
tableWidget->insertRow(numRow);
QTableWidgetItem *newItem = new QTableWidgetItem("Text");
tableWidget->setItem(numRow, numCol, newItem);
```

Per aggiornare i dati presenti nelle caselle è sufficiente utilizzare nuovamente il metodo *setItem*. Analogamente per cambiare il testo di una *QLabel* si può riutilizzare il suo metodo *setText*.

5.2.2 Implementazione

Per implementare il sistema di customer tracking si andrà a modificare per prima cosa la classe Target, alla quale si andrà ad aggiungere due nuove variabili:

```
public CustomerStatus cStatus
public int numVisits
```

Dove il tipo di dato *CustomerStatus* è definito nel seguente modo:

```
enum CustomerStatus {
    INSIDE, OUTSIDE
};
```

In questo modo si terrà traccia di dove si trova ciascun target e di quante volte ha visitato il negozio. Considerando che un nuovo target non può che arrivare dall'esterno e che non ha mai fatto visita prima, *CustomerStatus* verrà inizializzato a *OUTSIDE* mentre *numVisits* al valore 0.

Per quanto riguarda la classe Tracker, si avrà bisogno di nuove variabili integer per tenere il conto del numero totale dei clienti riconosciuti e il numero di questi attualmente all'interno del negozio.

```
public int numCustomers
public int numCustomersInside
```

numCostumers verrà semplicemente fatto aumentare di un'unità ad ogni chiamata del me-

todo *Tracker::createNewTarget()*. Si ricorda ora che si era ipotizzato che rispetto alla visuale della Kinect, l'ingresso si trova sulla destra mentre l'uscita sulla sinistra. Una persona che entra o esce dal negozio passa, in entrambi i casi, dallo stato "visibile" a quello "non visibile". Se, prima di sparire dall'obiettivo, l'ultima posizione di un target era a destra della Kinect (coordinata x positiva) allora si suppone che sia uscito dal negozio, altrimenti (coordinata x negativa) che sia entrato. Ad ogni chiamata del metodo *Tracker::updateDetectedTracks()* è possibile identificare i target che passano allo stato "non visibile" e valutarne la coordinata x della loro posizione per sapere se stanno entrando o uscendo. A seconda dello stato del target si hanno quattro possibilità:

- **target OUTSIDE esce**: questo è il caso in cui un cliente entra dall'esterno nel pre-ingresso ma poi esce nuovamente senza entrare nel negozio. Lo stato del target rimane *OUTSIDE*.
- **target OUTSIDE entra**: un cliente entra nel negozio. *numCustomersInside* viene incrementato, lo stato del target passa a *INSIDE*.
- **target INSIDE esce**: un cliente entra nel negozio. *numCustomersInside* viene decrementato, lo stato del target passa a *OUTSIDE*.
- **target INSIDE entra**: un cliente all'interno esce nel pre-ingresso del negozio ma poi rientra. Lo stato del target rimane *INSIDE*.

BIBLIOGRAFIA

[1] *http://www.u-mobo.com/index.php*, Sito ufficiale della piattaforma U-Mobo.

[2] *https://github.com/u-mobo/*, Repositoty ufficiali U-Mobo per sorgenti Linux e U-Boot.

[3] *http://www.denx.de/wiki/U-Boot/Documentation*, Documentazione ufficiale U-Boot.

[4] *http://www.ros.org/*, Sito ufficiale ROS.

[5] *http://wiki.ros.org/it*, Documentazione ufficiale ROS.

[6] *http://www.xbox.com/it-IT/Kinect*, Sito di riferimento Microsoft Kinect.

[7] Matteo Munaro, Filippo Basso, Emanuele Menegatti. *Tracking people within groups with RGB-D data*,
http://robotics.dei.unipd.it/images/Papers/Conferences/munaro_iros12