Università degli Studi di Perugia

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

Tesi di Laurea

Implementazione di un sistema mobile ed autonomo per la ricerca di oggetti in base al colore

Laureando

Simone Mariotti

Relatori

Prof. Marco Baioletti

Dott. Emanuele Palazzetti

Anno Accademico 2013-2014

TODO: DEDICA

Indice

Introduzione

Obiettivi

Strumenti utilizzati

Capitolo 1

Visione Artificiale

La *visione artificiale* è l'unione dei procedimenti che permettono di creare un modello del mondo reale in tre dimensioni a partire da numerose immagini bidimensionali per cercare di emulare il processo biologico della vista. Negli umani, così come in molte altre specie, vedere non è solo scattare una fotografia bidimensionale mentale di un ambiente, è interpretare le informazioni ricevute attraverso la retina per analizzare e creare un modello 3D dell'ambiente circostante.

Un sistema di visione artificiale, per provare ad avvicinarsi alla capacità visiva e di interpretazione umana, ha bisogno di numerosi componenti di diversa natura: ottici, elettronici e meccanici per acquisire e memorizzare le immagini da elaborare. Il primo tentativo in tal senso è datato 1883 quando Paul Gottlieb Nipkow inventò il primo sistema in grado di trasformare o meglio trasdurre, informazioni visive in un segnale elettrico.

Il sistema si basa su un disco con dei fori praticati lungo una spirale che parte dal centro del disco e procede verso l'esterno. Il disco ruota ad una velocità costante mentre l'immagine inquadrata viene focalizzata da una lente

verso i fori in modo che un sensore, posto sul retro del disco, possa percepire i cambi di luminosità della porzione di scena inquadrata e convertire questa variazione in segnali elettrici.

Il ricevitore emetterà luce in base a dei segnali elettrici e attraverso un disco identico al primo e in rotazione sincrona proietterà un'immagine con tante righe quante sono i fori del disco. Questo sistema porterà alla creazione della TV meccanica nel 1925. Negli anni successivi, con l'introduzione della

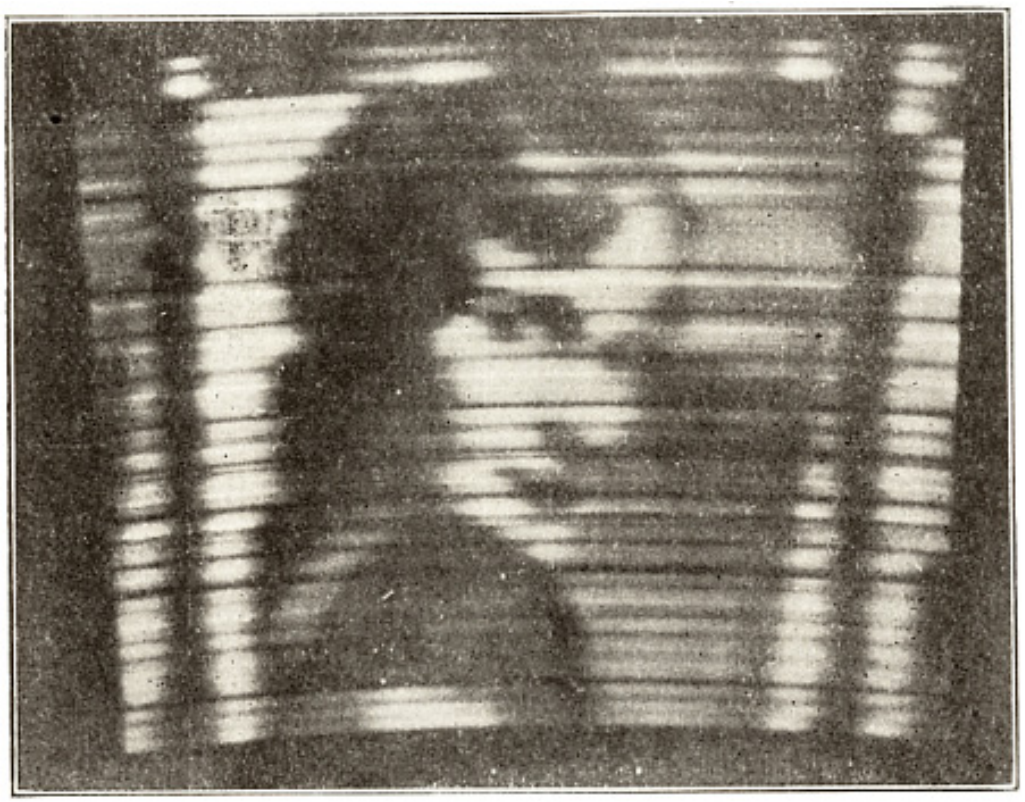

Figura 1.1: Immagine visibile su una TV meccanica del 1925

miniaturizzazione nell'elettronica, sono stati compiuti enormi progressi. Da un punto di vista teorico, gli scienziati cominciarono ad ispirarsi al corpo umano, una delle migliori "macchine" esistenti, nel tentativo di riprodurre

il suo comportamento. Analizzando da vicino l'occhio e in particolare la retina hanno scoperto che era formata da minuscoli recettori sensibili alla luce collegati, tramite il nervo ottico, al cervello. Si è così deciso di creare dei micro sensori di luce e formarne un'enorme matrice. Il primo risultato di questi studi si ebbe circa 40 anni dopo e fu il sensore CCD[1] , ideato da Willard S. Boyle e George E. Smith nel 1969 presso i Bell Laboratories, mentre dobbiamo aspettare il 1993 per vedere i primi prototipi funzionanti del sensore CMOS[2] sviluppati presso il Jet Propulsion Laboratory, entrambi hanno come elemento base il fotodiodo che equivale ad un pixel. Il CCD è un sensore analogico[3] che ha bisogno di più energia ma offre in genere una qualità superiore un rumore minore ad un costo più elevato.

Il CMOS è un sensore digitale che offre una buona qualità di immagine ad

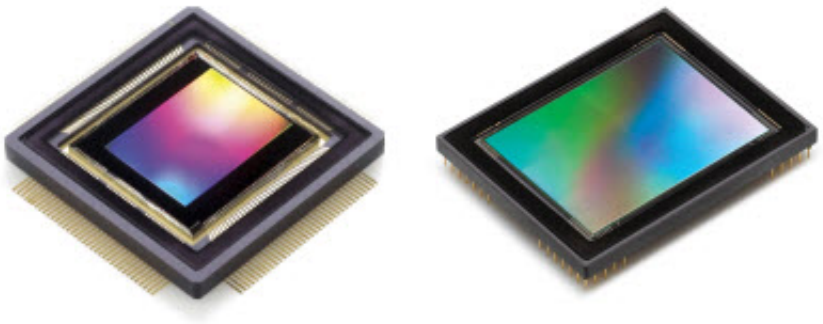

Figura 1.2: Due sensori moderni. A sinistra un sensore CMOS a destra uno CCD

costo minore; per contro l'immagine presenta un forte rumore a causa della

[1]Charge-Coupled Device

[2]Complementary Metal-Oxide-Semiconductor

[3]Il sensore più grande esistente è di 1,4 Gigapixel ed è montato sul telescopio Pan-STARRS sviluppato per l'individuazione di meteoriti in rotta di collisione con la Terra

conversione analogico-digitale.

La visione artificiale ha principalmente tre utilizzi:

- **Ricognizione:** ricercare uno o più oggetti, scelti a priori, e organizzarli in insiemi generici o classi mantenendo informazioni riguardo il loro posizionamento nella scena.

  Esempio: ricercare in un'immagine o in un video tutte le persone, le macchine o gli animali.

- **Identificazione:** identificare una istanza specifica di una classe di oggetti.

  Esempio: ricercare in un'immagine o in un video un volto, una macchina o un animale specifico..

- **Individuazione:** cercare una condizione specifica nell'immagine.

  Esempio: cercare imperfezioni nelle immagini a raggi X di superfici o materiali.

Un campo che ne fa massiccio utilizzo è la modellazione di ambienti 3D a partire da due o più immagini 2D; questo è stato reso possibile grazie all'enorme aumento della capacità di elaborazione grafica delle GPU.

La visione artificiale è stata applicata a molti altri campi nei quali si è verificata una vera e propria rivoluzione. Uno di questi è la medicina nella quale l'introduzione di questa tecnica ha portato alla realizzazione di radiografie, angiografie, tomografie e molte altre; in questo modo è possibile identificare anomalie e problemi, quali i tumori, che non sarebbero visibili all'occhio umano se non in seguito a procedure molto invasive. Un altro campo è quello del controllo di veicoli autonomi i quali stanno aumentando ad un ritmo

vertiginoso, progressivamente al migliorarsi delle tecniche di visione artificiale. Autovetture, droni, robot e carri per il rifornimento sono solo degli esempi a cosa può portare la visione artificiale applicata nei settori civili e di ricerca. La nostra tesi si concentra proprio sull'applicazione ad un robot

Figura 1.3: Drone della società statunitense Amazon. Consegnerà prodotti a domicilio in completa autonomia

autonomo dei concetti di visione artificiale appena descritti.

Capitolo 2

Componenti hardware

2.1 UDOO Quad

UDOO è un progetto tutto italiano di una piattaforma hardware destinata alla generazione dei "makers", cioè quelle persone che vogliono realizzare i proprie progetti con le tecnologia a basso costo ad oggi disponibili. La scheda ha visto la luce dopo una sorprendente campagna di crowdfunding [1] terminata l'8 Giugno 2013 con 4172 donazioni per un totale di $641.612 a fronte di $27.000 richiesti per iniziare la produzione. Per permettere l'utilizzo di librerie e applicazioni computazionalmente pesanti come openCV, PureData e altre UDOO monta un processore ARM Freescale i.MX6 Cortex-A9 Quad core 1GHz che supporta sia Android che Linux. Il tutto è completato da una GPU Vivante, 1GB di RAM DDR3, numerose porte di I/O come SATA, microfono, audio out, Ethernet, HDMI, USB, connettore per display LVDS con touch screen, connettore CSI per camera esterna e connettività bluetooth e Wi-Fi. La periferica di "boot" è una microSD il che permette

[1] dall'inglese crowd, folla e funding, finanziamento. In italiano finanziamento collettivo.

un rapido passaggio da Linux a Android e viceversa. Quello che però rende veramente unica questa piattaforma, e che ne ha fatto la nostra scelta per questo progetto di tesi, è la presenza di un Arduino DUE completamente integrato nella stessa board. E' presente una CPU Atmel SAM3X8E ARM Cortex-M3 [2] e 76 GPIO[3], di cui 62 digitali e 14 digitali/analogici, disposti per essere perfettamente compatibili con la piedinatura dell'Arduino DUE e dell'Arduino UNO Rev.3.

La presenza di un Arduino DUE all'interno della board rende UDOO una scheda di prototipazione a tutti gli effetti e apre nuovi interessanti scenari e possibilità unendo la versatilità e semplicità di Arduino, la potenza di calcolo del Freescale i.MX6 e le numerose periferiche disponibili per Linux o Android.

Essendo una piattaforma open-source è possibile accedere alla shell del sistema operativo come root tramite la porta seriale integrata e modificare a piacimento la configurazione del sistema operativo. Arduino è collegato al Freescale i.MX6 tramite un bus interno e quindi viene rilevato come una normale periferica USB da Linux; su Android la comunicazione tra i due dispositivi avviene sullo stesso bus ma usa lo standard USB OTG[4]. L'interconnessione tra l'accessorio Arduino e l'applicazione Android è realizzata tramite l'ADK[5] 2012, di cui parleremo più avanti in questo stesso capitolo, che permette l'integrazione delle più disparate periferiche a dispositivi

[2] la stessa di cui dispone l'Arduino DUE

[3] General Purpose Input/Output

[4] On-The -Go è una specifica che permettere di agire come host ad un qualsiasi dispositivo (tipicamente smartphone e tablet). A differenza dell'USB classico l'OTG è driver-less, cioè non necessita l'installazione di driver specifici per ogni dispositivo

[5] Android Development Kit

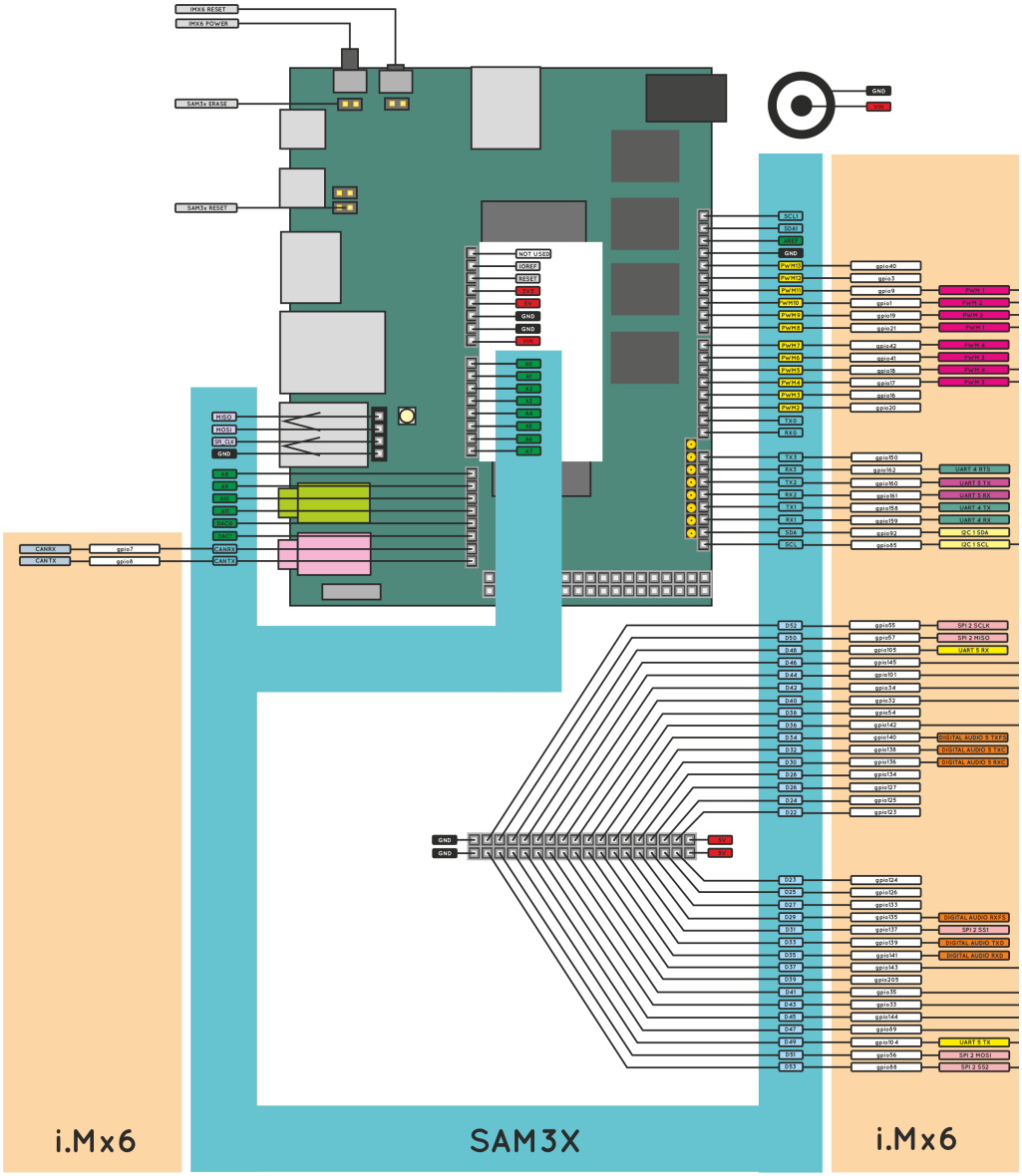

Figura 2.1: Schema piedinatura UDOO

Android tramite una connessione USB o Bluetooth.

2.2 Tank Kit

Per dare la giusta stabilità e manovrabilità al robot si è deciso di usare una locomozione a cingoli che richiede solo due motori e permette di ruotare sul posto o comunque in spazi ristretti: la nostra scelta è stata il "Multi-Chassis - Tank Version". Questa piattaforma, appositamente pensata per la realizzazione di robot multifunzione, si è rivelata la scelta perfetta in quanto possiede due potenti motori DC già forniti di riduttori 48:1 per affrontare terreni impervi e scoscesi, quattro ruote da 52 mm di diametro a cui sono applicati i due cingoli. E' presente anche un alloggiamento per un servomotore standard che nella nostra applicazione non è stato usato. L'intelaiatura, di alluminio spesso 2,5 mm, presenta numerosi fori e asole per il montaggio di accessorie quali sensori, staffe e motori. Presenta inoltre un "doppio fondo" in cui sono alloggiati i motori DC e i riduttori e in cui è possibile sistemare altri componenti che non debbano essere facilmente accessibili.

2.3 Sensori

2.3.1 Sensore di riflessività - QRD1114

Avevamo la necessità di fornire al robot un modo per rilevare eventuali sconfinamenti dall'ambiente di test che fosse il più flessibile possibile. Abbiamo optato per il sensore di riflessività QRD1114 prodotto dalla Fairchild Semiconductor: questo sensore è costituito da un LED infrarosso e un fototransistor tarato sulla luce infrarossa e con filtro per la luce solare onde evitare disturbi. Il robot era stato pensato per lavorare su un tavolo o altra

superficie con spigoli netti: per rilevare l'imminente caduta in questo tipo di ambiente sarebbe stato sufficiente un sensore di distanza puntato verso terra. Con il sensore di riflessività abbiamo reso possibile l'utilizzo in terra o comunque in ambienti estesi delimitati da un recinto spesso circa 10 cm realizzato con materiale a basse riflettività come del semplice cartoncino nero opaco. Il sensore non fa differenza tra il cartoncino nero o lo spazio a fianco di un tavolo, rileva semplicemente una riflessività vicina allo zero. Il sensore così come fornito dal produttore non è direttamente utilizzabile, per far si che Arduino potesse acquisire dal sensore valori proporzionali alla riflessività del materiale in esame abbiamo dovuto realizzare un circuito elettronico di interfaccia.

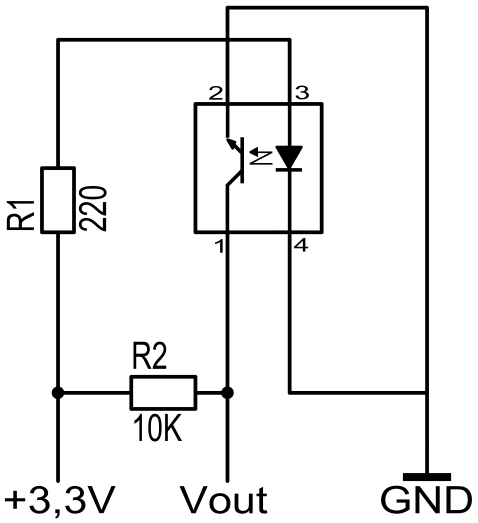

Figura 2.2: Circuito di interfaccia tra Arduino DUE e il sensore QRD1114

Il circuito alimenta il LED tramite una resistenza da 220Ω (*R1*) e colle-

ga V_{CC}[6] al collettore del fototransistor tramite una resistenza da 10 $k\Omega$ (*R2*) mentre l'emettitore è collegato a terra; il punto da cui prelevare il segnale (V_{OUT}) è tra *R2* e il collettore. Il principio alla base del circuito è semplice: il LED è sempre acceso e illumina in modo diffuso parallelamente al fototransistor. Il fototransistor in assenza di luce o, nel nostro caso, in presenza di una bassa riflessività si trova in stato di interdizione; i pin di Arduino impostati come input sono in configurazione "alta impedenza", equivalenti ad un interruttore aperto dal punto di vista circuitale, quindi non c'è passaggio di corrente né tramite il transistor né tramite la resistenza *R2* il che porta esattamente il valore di V_{CC} in ingresso ad Arduino. Quando il fototransistor è totalmente illuminato, cioè in presenza di alta riflessività, entra in stato di conduzione così che nel punto V_{OUT} si venga a trovare la massa. Ogni stato intermedio di illuminazione equivale ad una conduzione parziale del fototransistor a cui corrisponde una tensione proporzionale alla riflessività sul pin V_{OUT}. Il sensore può essere utilizzato in modalità digitale o analogica semplicemente collegando il pin V_{OUT} ad un pin digitale o analogico e cambiando la configurazione relativa all'interno della programmazione di Arduino. La configurazione digitale si è rivelata inadatta all'applicazione in quanto l'intervallo di valori che rappresenta un oggetto riflette nelle immediate vicinanze del sensore, e quindi lo stato di normale funzionamento, è troppo stretto e anche un minimo sussulto manda il robot in allarme sconfinamento. La configurazione analogica invece ci permette di avere circa 1024 valori discreti dal trasduttore, abbiamo quindi impostato una soglia oltre la quale il robot va in allarme sconfinamento; è importante notare che questa libertà nello scegliere una soglia di allarme ci

[6] pari a 3,3 V nell'Arduino DUE

permette di effettuare una calibrazione affinata in base al materiale su cui si svolge il test per minimizzare la possibilità di falsi positivi.

2.3.2 Sensore di distanza a ultrasuoni - HC-SR04

Il robot aveva bisogno di conoscere la distanza degli oggetti che si trovavano di fronte ad esso, la scelta è ricaduta un sensore ad ultrasuoni, a discapito di uno ad infrarossi, per la totale assenza di disturbi dovuti all'illuminazione dell'ambiente di test. In particolare abbiamo scelto il sensore HC-SR04 che offre ottime prestazioni, è facile da integrare ed è di dimensioni contenute. Il sensore può rilevare oggetti distanti da 2 cm a 400 cm con una risoluzione di 0,3 cm entro un cono frontale con apertura di 60°, 30° per lato rispetto alla perpendicolare dal sensore.

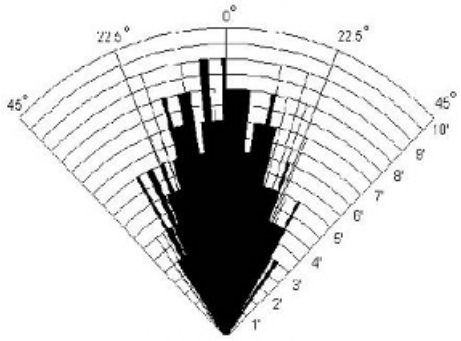

Figura 2.3: Risposta del sensore rispetto alla posizione dell'ostacolo

Il collegamento alla piattaforma di sviluppo è effettuato tramite quattro pin: $+5V$, *trig*, *echo* e *GND*. Il funzionamento è piuttosto semplice, una volta alimentato il sensore si invia sul pin *trig* un impulso di durata 10μs, il sensore invia quindi un impulso ultrasonico e restituisce sul pin *echo* un segnale positivo di durata pari al tempo di percorrenza andata e ritorno del

segnale ultrasonico. Una volta ottenuto il tempo di percorrenza si utilizza la formula del moto rettilineo uniforme:

$$v = \frac{\Delta s}{\Delta t}$$

sapendo che la velocità del suono a 20 °C è approssimabile a 344 m/s allora la durata dell'impulso di *echo* sarà compreso tra 116.3 μs e 23,26 ms rispettivamente per distanze percorse di 4 cm (2+2) e di 8 m (4+4).
Per ricavare la distanza (d) in centimetri a partire dalla durata dell'impulso notando che 344 m/s = 29,1 cm/μs e indicando con T_e la durata dell'impulso di echo usiamo:

$$d = \frac{T_e}{2 \cdot 29,1}$$

Per quanto riguarda il collegamento ad Arduino se la nostra scheda accettasse input a 5 V questo sensore non avrebbe bisogno di nessun accorgimento per essere collegato, purtroppo però, anche se la nostra UDOO e l'Arduino DUE in essa integrato possono fornire alimentazione a 5 V, non possono accettare input allo stesso voltaggio ma solo a 3,3 V.
Allo stesso modo del sensore QRD1114 ci serve un circuito di interfaccia che in questo caso sarà un semplice partitore di tensione il cui schema generico è riportato di seguito

Il partitore di tensione, a fronte di un noto valore di tensione di ingresso V_{IN} riesce a fornirci il valore di tensione desiderato in uscita V_{OUT} tramite due resistenze opportunamente dimensionate.

La formula che regola il partitore di tensione è la seguente

$$V_{OUT} = V_{IN} \cdot \frac{R_2}{R_1 + R_2}$$

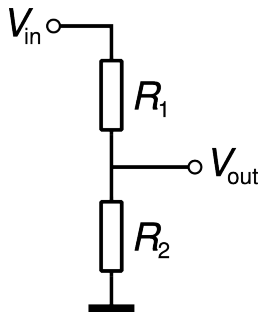

Figura 2.4: Schema generico del partitore di tensione

nel nostro caso $V_{IN} = 5$ V e $V_{OUT} = 3{,}3$ V quindi scegliendo arbitrariamente il valore di R_1 1 kΩ ricaviamo semplicemente il valore di R_2 tramite la formula inversa

$$R_2 = R_1 \cdot \frac{V_{OUT}}{V_{IN} - V_{OUT}}$$

che da come risultato 1941,48 Ω; approssimando il valore appena calcolato al più vicino taglio standard di resistenze, cioè $R_2 = 2$ kΩ, si ottiene $V_{OUT} = 3,\overline{3}$ che è assolutamente accettabile.

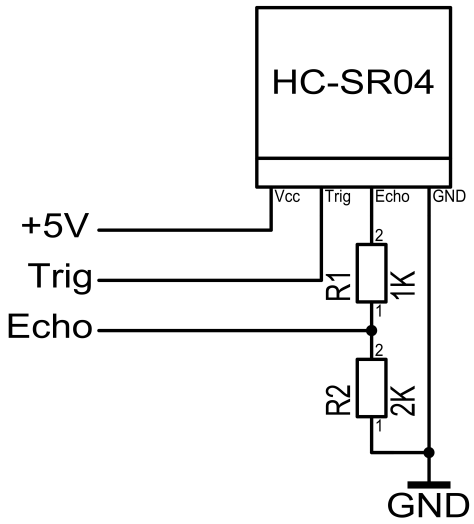

Figura 2.5: Circuito di interfaccia tra Arduino DUE e il sensore HC-SR04

Capitolo 3

Componenti software

3.1 OpenCV

Per dare al robot una visione dell'ambiente circostante ci serviva una libreria di visione artificiale. Le più famose disponibili per Java sono OpenCV e JavaCV; la nostra scelta è stata OpenCV per via della maggiore disponibilità di documentazione. OpenCV[1] è stata originariamente sviluppata da Intel che cercava di migliorare le prestazioni delle loro CPU con applicazioni computazionalmente pesanti come per esempio gli algoritmi di *ray tracing* [2] ed è ora disponibile sotto licenza BSD.

La libreria per poter funzionare su un dispositivo Android ha bisogno di un'applicazione di supporto chiamata OpenCV Manager che può essere scaricata dal Play Store o installata direttamente con il pacchetto apk.[3] La

[1] Open Source Computer Vision

[2] Tecnica di geometrica ottica che analizza il percorso dei raggi di luce. In grafica 3D è un algoritmo di rendering che costruisce la scena facendo partire i raggi luminosi dalla camera (visuale del giocatore) invece che dalla sorgente luminosa.[1]

[3] Android Package, il tipo di file delle App Android simile al formato JAR

libreria viene caricata in modo asincrono all'avvio dell'App e fornisce ogni frame come una matrice dove ogni cella rappresenta un pixel; OpenCV usa il tipo di dato interno Mat per rappresentare le matrici. Tutte le operazioni effettuate su oggetti di tipo Mat usano algoritmi appositamente ottimizzati per le operazioni tra matrici.

3.2 ADK

In Android, a partire dalla versione 3.1, è supportato il protocollo Android Open Accessory (AOA), ogni accessorio sviluppato per Android deve supportare lo stesso protocollo.
A causa della limitata autonomia dei dispositivi su cui Android è maggiormente utilizzato (smarthphone e tablet) l'AOA impone che l'accessorio funga da host e alimenti quindi il bus utilizzato
Per usare Arduino come un accessorio per Android e rispettare lo standard AOA esiste la libreria *adk.h* che permette di dialogare con un dispositivo Android dopo un'accurata configurazione. Android distingue diversi accessori sulla base di informazioni che ogni accessorio fornisce all'atto della connessione, queste informazioni sono:

- Vendor
- Name
- Longname
- Version
- Url

- Serial

Il dispositivo Android accetterà la richiesta di connessione solo se le stringhe identificative in suo possesso combaciano perfettamente con quelle fornite dall'accessorio. Per ottenere effettivamente la connessione bisogna indicare ad Android quali sono gli accessori supportati e quale applicazioni è in grado di gestire un particolare accessorio. Questo si ottiene inserendo la successiva direttiva nel Manifest file dell'App

Codice 3.1: Porzione del Manifest file dell'App

```
...
<meta-data
    android:name="android.hardware.usb.action.
                                USB_ACCESSORY_ATTACHED"
    android:resource="@xml/usb_accessory_filter" />
...
```

In questo modo alla connessione di qualsiasi accessorio sarà il sistema operativo a suggerire di aprire una certa app se l'accessorio connesso è presente nel file usb_accessory_filter.xml associato.
Nel nostro caso il file usb_accessory_filter.xml si presenta così

Codice 3.2: usb_accessory_filter.xml

```
<resources>
    <usb-accessory
            version="0.1.0"
            model="Mobile-Tanker"
            manufacturer="Simone Mariotti"/>
</resources>
```

Le stesse identiche stringhe saranno impostate durante la fase di configurazione di Arduino in modo da permette l'accoppiamento.

3.3 ADK Toolkit

L'ADK toolkit è una libreria che aggiunge un grado di astrazione all'ADK, l'invio e la ricezione dei messaggi sono semplificati così come l'inizializzazione della connessione.

Il toolkit si basa su due classi principali: AdkManager e AdkMessage.

AdkManager espone metodi per la gestione della connessione e per l'invio e la ricezioni di dati. AdkMessage rappresenta il messaggio ricevuto dall'accessorio ne suo formato nativo, cioè un array di byte; tramite dei metodi ausiliari, *getString()*, *getFloat()*, *getInt()*, è possibile ottenere una versione tipizzata del messaggio e naturalmente la versione "grezza" dei dati con *getBytes()* e *getByte()*.

Grazie a questa libreria l'uso dell'ADK, originariamente poco intuitivo, diventa efficiente ed elegante

Codice 3.3: Inizializzazione della connessione con l'accessorio

```
private AdkManager mAdkManager;


@Override
protected void onCreate(Bundle savedInstanceState) {
    // ...
    mAdkManager = new AdkManager(this);
}
```

```
@Override
protected void onResume() {
    super.onResume();
    mAdkManager.open();
}
```

Codice 3.4: Lettura e scrittura dati

```
// Write
adkManager.write("Ciao da Android!");

// Read
AdkMessage response = adkManager.read();
System.out.println(response.getString());
// Could outputs: "Ciao da Arduino!"
```

Capitolo 4

Implementazione

È definito "agente" una qualunque entità in grado di percepire l'ambiente circostante attraverso dei sensori e di eseguire delle azioni attraverso degli attuatori. Nel campo dell'intelligenza artificiale è definito intelligente quell'agente che fa la cosa giusta al momento giusto. Il compito dell'agente è quello di stabilire una sequenza di azioni che portano al raggiungimento di uno stato obiettivo.[2]

4.1 Android

L'app per Android è stata sviluppata avendo come priorità la modularità. E' infatti molto semplice cambiare totalmente il comportamento dell'agente o la codifica dei messaggi sostituendo o modificando una singola classe senza coinvolgere il resto del codice.

La modularizzazione a grana più grossa è a livello di package[1]: ci sono

[1]Un contenitore che racchiude classi che svolgono compitini affini

tre package, ognuno con un compito preciso, e sono **logic**, **messaging** e **opencv**.

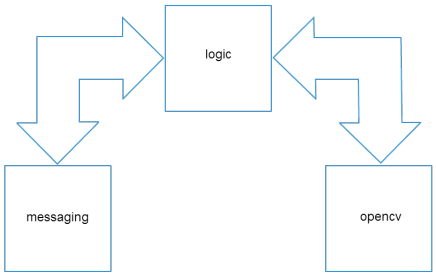

Figura 4.1: Schema di interconnessione dei package

4.1.1 Il package *logic*

Si occupa di reperire le informazioni dal mondo reale e analizzarle al fine di eseguire l'azione più adatta. Contiene tre classi: RobotActivity, RobotLogic e UpdateDirections.

La classe *RobotActivity*

Come suggerisce il nome è l'Activity vera e propria, cioè quella classe che il sistema operativo istanzia all'avvio dell'app. Essa stessa istanzia e prepara tutti gli altri oggetti per l'esecuzione. Implementa due interfacce: View.OnTouchListener e CvCameraViewListener. La prima permette di gestire gli input da touch screen senza ricorrere ad una classe esterna, la seconda è un'interfaccia presente nella libreria OpenCV e permette di "intercettare" i frame proveniente dalla camera prima che vengano renderizzati a schermo tramite l'override[2] del metodo *OnCameraFrame()*. Ogni

[2]Tecnica che permette di ridefinire il comportamento di un metodo ereditato

frame verrà elaborato e solo alla fine visualizzato a schermo. All'avvio si occupa di inizializzare OpenCV tramite la callback *BaseLoaderCallback* e ottiene il riferimento all'istanza dell'ADK tramite l'ADKToolkit, inizializza il Communicator e passa il riferimento di quest'ultimo a TargetSearch durante l'esecuzione.

E' buona norma interrompere le connessioni e liberare il canale usato nel momento in cui un'app viene chiusa. Per questo nel metodo *onDestroy()* viene chiuso il canale usato per comunicare con Arduino tramite il metodo *close()* fornito dall'ADKToolkit e viene fermato lo scheduler che esegue il Communicator

In aggiunta alle normali funzioni di un'Activity RobotActivity integra un listner per gli eventi touch verificatesi all'interno della porzione di interfaccia utente che mostra il video proveniente dalla camera. Al verificarsi di un tale evento viene invocato il metodo *onTouch()* ereditato da View.OnTouchListener che esegue la media dei colori presenti in una regione di 8x8px intorno alle coordinate in cui è avvenuto il tocco da parte dell'utente. Fornisce il colore così calcolato a alla classe TargetSearch del package **opencv** tramite il metodo pubblico di quest'ultima *setTargetHsvColor()*. Questo sarà il colore che l'agente cercherà nella scena, il suo obbiettivo.

La classe *UpdateDirections*

Questa classe è un "singleton"[3] e si occupa di mostrare all'utente tramite immagini e testi quello che l'agente sta facendo o quale sarà la sua prossima

[3]E' un design pattern descritto dalla cosiddetta "Gang of four" nel libro "Design Patterns". Permette la creazione di una sola istanza della classe e ne regola l'accesso.

mossa. Dovendo agire sull'UI[4] deve essere eseguita sul thread che si occupa dell'UI, per questo implementa l'interfaccia *Runnable* e ogni sua esecuzione è lanciata tramite *runOnUiThread()*. Le informazioni visualizzabili sono limitate e ben distinte, ad ognuna corrisponde un metodo da invocare per visualizzare quella data informazione a schermo. I metodi disponibili sono:

- *left()*: indica che l'obiettivo è visibile è stato individuato e si trova alla sinistra dell'agente.
- *right()*: indica che l'obiettivo è visibile è stato individuato e si trova alla destra dell'agente.
- *aimed()*: indica che l'obiettivo è visibile è stato individuato e si trova esattamente di fronte all'agente che si muoverà in quella direzione.
- *search()*: indica che l'obiettivo non è visibile, l'agente si muoverà secondo l'algoritmo di ricerca.
- *found()*: indica che l'obiettivo è visibile e l'agente si trova a meno di 30 centrimetri.
- *avoidingLeft()*: indica che è presente un ostacolo sul percorso dell'agente a meno di 30 centimetri. L'agente lo aggirerà verso sinistra.
- *avoidingRight()*: indica che è presente un ostacolo sul percorso dell'agente a meno di 30 centimetri. L'agente lo aggirerà verso destra.
- *chooseColor()*: indica che l'agente è in attesa che venga impostato il colore da cercare.

[4]User Interface, interfaccia utente

La classe espone anche altri metodi che mostrano (*show()*) o nascondono (*hide()*) la porzione di interfaccia che visualizza le indicazioni, oppure bloccano (*lock()*) e sbloccano (*unlock()*) la possibilità di cambiare le indicazioni visualizzate.

La classe astratta *BaseAi* e l'interfaccia *IBaseAi*

Usando l'ereditarietà forniscono la base di partenza per la classe RobotLogic.

L'interfaccia ha due metodi: *targetPosition()* e *think()*. BaseAi sfrutta l'interfaccia IBaseAi e implementa il metodo *targetPosition()*. Tramite questo metodo TargetSearch fornisce all'AI la larghezza del frame, la posizione dell'obbiettivo e il buonding rect dell'obbiettivo; a partire da questi verranno calcolati altri parametri che saranno utilizzati in *think()*.

Con questa organizzazione si fa si che i comportamenti e i dati indispensabili per una classe di AI siano definiti in BaseAI e quindi ogni nuova classe nata per sostituire o migliorare la logica attuale dovrà estendere BaseAi e implementare *think()*

La classe *RobotLogic*

La classe *RobotLogic* è il vero "cervello" di tutta l'app.

Viene chiamata in causa ogni volta che arriva un nuovo messaggio da Arduino. Per far questo si è usato un altro famoso design pattern, quello di "Observable" e "Observer" in cui l'"Observer" è questa stessa classe e l'"Observable" è la classe IncomingMessage del pacchetto **messaging**.

Alla ricezione di ogni messaggio viene invocato il metodo *getIncoming()* del-

la classe IncomingMessage che fornisce un'istanza di ArduinoMessage che contiene la decodifica del messaggio appena ricevuto. Se il messaggio contiene informazioni sulla distanza del primo oggetto nella direzione dell'agente allora tale distanza viene salvata nell'istanza in modo che successivamente si possa usare per valutare in modo più preciso la situazione.
Il metodo principale della classe è *think()* ed è invocato ogni volta che dal package opencv, e in particolare dalla classe TargetSearch, giunge un aggiornamento sull'obbiettivo, la sua eventuale presenza in scena e la sua posizione. *think()* viene effettivamente eseguito solo se sull'interfaccia utente è stato premuto Start e per prima cosa stabilisce in che fase l'iterazione precedente ha portato il sistema.
Le possibilità sono:

- "Cheer Phase": indica che l'obbiettivo è stato trovato e l'agente sta girando su se stesso per segnalare la fine della ricerca.
- "Avoiding Phase": indica che sulla traiettoria dell'agente si è presentato un ostacolo e lo sta aggirando. Si compone di tre sottofasi:
  - Fase 1: scegliere in modo casuale una direzione tra destra e sinistra e ruota di circa 90° in quella direzione.
  - Fase 2: si muove in avanti per un tempo prestabilito.
  - Fase 3: ruota di 90° nella direzione opposta a quella scelta nella fase 1.
- "Search Phase": indica che l'obbiettivo non è stato ancora trovato e l'agente si sta muovendo per trovarlo.

Se si trova nella "Cheer Phase" ignora ogni comunicazione proveniente da Arduino, finisce la rotazione di segnalazione e invoca il metodo *reset()* di RobotActivity per preparare il sistema all'inserimento di un nuovo obbiettivo da parte dell'utente.
Se si trova nella "Avoiding Phase" esegue in successione le tre fasi. Si può interrompere solo se durante le manovre di aggiramento dell'ostacolo l'obbiettivo compare nella scena.
Se si trova nella "Search Phase" controlla se l'obbiettivo è presente in scena e si trova a meno di 30 centimetri, in caso di risposta affermativa ferma l'agente, interrompe l'esecuzione di *think()* e attiva la "Cheer Phase". Se l'obbiettivo non è in vista ma c'è un oggetto a meno di 30 centimetri ferma l'agente e attiva la "Avoiding Phase".
Se l'obbiettivo non è visibile rimane in "Search Phase" e si muove in avanti.
Se l'obbiettivo è visibile ma si trova a più di 30 centimetri effettua le correzioni di rotta necessarie per portarlo in direzione di marcia.
Se l'obbiettivo è precisamente di fronte all'agente si muoverà in quella direzione.
La rotazione necessaria per portare l'obbiettivo esattamente di fronte all'agente implica movimenti lenti e precisi che è difficile ottenere con i motori DC di cui è fornito l'agente. Lo scenario tipico è vedere i motori sforzarsi di muovere l'agente con un certo quantitativo di energia, aumentare quell'energia di un'unità e vedere l'agente iniziare a ruotare velocemente. Purtroppo l'energia richiesta per mettere in rotazione l'agente cambiava in funzione di troppe variabili: la direzione di rotazione, l'uso di entrambi i cingoli o solamente uno, la presenza di piccoli ostacoli sotto l'agente. Per ovviare

a questo problema che impediva una corretta movimentazione dell'agente quando erano necessari movimenti precisi si è usato una soluzione adattiva che permette all'agente di trovare sempre la minima energia necessaria per muoversi. Per far questo l'agente prende come riferimento la posizione dell'obbiettivo e tenta di ruotare, se all'iterazione successiva non nota uno spostamento relativamente all'obbiettivo allora aumenta l'energia inviata ad i motori. Appena si muove l'energia cessa di essere aumentata e l'agente ruota in modo fluido e controllato.

4.1.2 Il package *opencv*

Il package **opencv** è costituito di due sole classi. Si occupa di analizzare i frame proveniente dalla camera alla ricerca dell'obbiettivo, comunica poi la posizione dell'obbiettivo o l'assenza dello stesso alla classe RobotLogic del package **logic**.

La classe *ColorBlobDetector*

ColorBlobDetector è la classe che effettivamente cerca l'obbiettivo nei frame proveniente dalla camera. L'obbiettivo gli viene fornito da RobotActivity tramite *setTargetHsvColor()* nel momento in cui l'utente lo indica sull'interfaccia utente. I colori in OpenCV vengono rappresentati con un tipo di dato interno detto Scalar che può essere assimilato ad un array. Il metodo che esegue l'operazione di ricerca è *process()*. Per prima cosa converte lo schema

di colori del frame sotto analisi da RGBa[5]. a HSV[6] e filtra l'immagine in modo che rimangano solo i pixel che hanno un colore che si trova entro un certo intorno del colore cercato. Usa poi la funzione fornita da OpenCV *findContours()* che cerca i contorni delle zone con il colore di nostro interesse e li restituisce come una un ArrayList di MatOfPoint; MatOfPoint è un tipo di dato di OpenCV che è usato per memorizzare una quantità variabile di punti che sono in relazione tra di loro, in questo caso dei punti che formano un contorno. In seguito cerca il contorno che ha area maggiore e scarta tutti quelli che sono più piccoli del limite impostato ad un un'ordine di grandezza in meno. I contorni rimasti saranno quelli analizzati da TargetSearch.

La classe *TargetSearch*

Questa classe per prima cosa invoca *process()* di ColorBlobDetector passandogli il frame da analizzare e raccoglie i risultati tramite *getContours()* che restituisce i contorni individuati da *process()*.
Scorre poi uno ad uno i contorni così ricevuti e trova per ognuno un "bounding rectangle", cioè il rettangolo di area minima che contiene per interno il contorno corrente, scartando tutti i rettangoli minori di un certo valore e mantenendo un riferimento a quello di area maggiore trovato fino a quel momento. I rettangoli saranno tutti colorati di blu tranne quello di area maggiore che sarà colorato di rosso. Sarà proprio questo rettangolo il nostro

[5]Ogni colore è rappresentato da quattro valori compresi tra 0 e 255 per le immagini CV_8U, cioè le immagini dove i colori sono memorizzati in un byte. I quattro canali sono R (red), G (green), B (blue) e a (alpha)

[6]Ogni colore è rappresentato da quattro valori compresi tra 0 e 255 per le immagini CV_8U, cioè le immagini dove i colori sono memorizzati in un byte. I quattro canali sono H (hue), S (saturation), V (value) e a (alpha)

obbiettivo.
Fornisce un feedback all'utente sul colore scelto e la gamma di colori simili che sono stati cercati.
Solo tre dati forniti da questa classe servono alla RobotLogic per valutare l'azione da intraprendere e sono: la posizione dell'obbiettivo, la larghezza del frame e il bounding rectangle che racchiude l'obbiettivo.
Le posizioni sono sempre riferite all'asse x in quanto l'agente non ha capacità di movimento sull'asse y e possono essere:

- TARGET_POSITION_LEFT: il centro dell'obbiettivo si trova a sinistra del centro del frame.
- TARGET_POSITION_RIGHT: il centro dell'obbiettivo si trova a destra del centro del frame.
- TARGET_POSITION_FRONT: il centro dell'obbiettivo e il centro del frame coincidono.
- TARGET_POSITION_NONE: l'obbiettivo non è visibile.

Comanda ad UpdateDirection del pacchetto **logic** di mostrare le informazioni corrispondenti alla posizione dell'obbiettivo.

4.1.3 Il package *messaging*

Questo package è un modulo completamente a se stante, è così possibile cambiare il protocollo di comunicazione senza alterare la logica dell'agente. Le stringhe sono tutte codificate e decodificate all'interno di questo package,

non c'è nulla di "hard coded"[7].

Nel nostro caso i messaggi sono formati da tre interi da un byte, così che il range possibile è 0-255, separati da virgole. I messaggi diretti da Arduino ad Android hanno questa forma:

MessageType,Message,Data

dove MessageType può assumere due valori: INFO, indica che il messaggio trasporta un dato informativo, o STATE, indica che il messaggio comunica lo stato in cui si trova Arduino.

Se il MessageType è INFO allora il Message può essere di due tipi:

- DISTANCE: sulla porzione Data del messaggio sarà presente la distanza del primo oggetto di fronte all'agente.
- TERMINATE: indica l'intenzione di Arduino di chiudere l'app Android.

Se il MessageType è STATE allora il Message può essere di quattro tipi:

- IDLE: l'agente non sta eseguendo alcuna azione.
- SEARCHING: l'agente si sta muovendo in cerca dell'obbiettivo.
- HUNTING: l'agente si sta muovendo verso l'obbiettivo.
- EMERGENCY: l'agente è in emergenza, non eseguirà nessun comando fino al termine della situazione di emergenza.

I messaggi diretti da Android ad Arduino hanno invece questa forma:

[7]Una pratica considerata da evitare che prevede stringhe di testo scritte direttamente nel codice

Command,Parameter1,Parameter2

Command identifica l'azione da eseguire, Parameter1 e Parameter2 sono due parametri opzionali che modificano il comportamento standard dell'azione richiesta.

La classe *IncomingMessage*

Questa classe è un "singleton"[8] estende la classe Observable e insieme alla classe RobotLogic del package **logic** implementa il design pattern "Observer-Observable".
Accoglie il messaggio ricevuto da Arduino, crea un oggetto di tipo ArduinoMessage per decodificare e custodire il messaggio ricevuto e informa gli Observer della presenza di un nuovo messaggio tramite *triggerObservers()*. Su richiesta fornisce l'ultimo messaggio ricevuto tramite *getIncoming()*.

La classe *ArduinoMessage*

In questa classe sono definite tutte le costanti necessarie per la decodifica delle stringhe ricevute da Arduino. Decodifica la stringa passatagli come parametro in fase di istanzazione ed espone diversi metodi che consentono di fare un controllo veloce sul contenuto del messaggio. Un esempio della versatilità di questi metodi pubblici è:

Codice 4.1: Metodo *update()* di RobotLogic del pacchetto **logic**

```
@Override
public void update(Observable observable, Object data) {
```

[8] E' un design pattern descritto dalla cosiddetta "Gang of four" nel libro "Design Patterns". Permette la creazione di una sola istanza della classe e ne regola l'accesso.

```
    ArduinoMessage incomingMessage =
        IncomingMessage.getInstance().getIncoming();
    if (!incomingMessage.hasError()) {
        if (incomingMessage.isInfoMessage() &&
            incomingMessage.hasDistance()) {
            mDistance = incomingMessage.getData();
        }
        if (incomingMessage.isTerminateCommand()) {
            mRobotActivity.finish();
        }
    }
}
```

La classe *MessageEncoder*

In questa classe sono definite tutte le costanti necessarie per la codifica delle stringhe da inviare Arduino ed espone numerosi metodi statici per la codifica vera e propria:

- *moveForward(int motorLeft, int motorRight)*: comanda di muoversi in avanti e applicare ai motori le potenze indicate da motorLeft e motorRight
- *moveForward()*: comanda di muoversi in avanti e applicare ai motori potenza pari a DEFAULT_VELOCITY ad entrambi i motori.
- *moveBackward(int motorLeft, int motorRight)*: comanda di muoversi in indietro e applicare ai motori le potenze indicate da motorLeft e motorRight

- *moveBackward()*: comanda di muoversi in indietro e applicare ai motori potenza pari a DEFAULT_VELOCITY ad entrambi i motori.
- *stop()*: comanda di fermarsi.
- *turnLeft(int velocity, int mode)*: comanda di ruotare verso sinistra e applicare ai motori potenza pari a *velocity*.
- *turnLeft(int mode)*: comanda di ruotare verso sinistra e applicare ai motori potenza pari a DEFAULT_VELOCITY.
- *turnRight(int velocity, int mode)*: comanda di ruotare verso destra e applicare ai motori potenza pari a *velocity*.
- *turnRight(int mode)*: comanda di ruotare verso destra e applicare ai motori potenza pari a DEFAULT_VELOCITY.
- *search()*: comanda di cercare l'obbiettivo.

Nei metodi *turnLeft* e *turnRight*, *mode* può assumere il valore TURN_ON_SPOT o TURN_NORMALLY che fanno girare l'agente rispettivamente utilizzando entrambi i cingoli o solo il cingolo opposto alla direzione di rotazione.
Grazie a questi metodi pubblici la parte di logica dell'app invia messaggi ad Arduino senza avere la minima idea della stringa che effettivamente sarà inviata.
Tutto quello che si limita a fare per comandare, per esempio, ad Arduino di girare sul posto verso destro e con una velocità di 200 è:

Codice 4.2: Esempio di codifica e invio di un comando ad Arduino

```
Communicator.setOutgoing(MessageEncoder.turnRight(200,
    MessageEncoderDecoder.TURN_ON_SPOT));
```

La classe *Communicator*

Questa classe usa la tecnica degli "short-lived thread" per implementare la comunicazione da e verso Arduino. Questa tecnica è un'alternativa efficiente all'utilizzo di un thread in background come potrebbe essere un'AsyncTask nel caso di Android.
Necessita di una classe che estende Thread o di una che implementa Runnable, questa sarà la classe che verrà lanciata ad intervalli prestabiliti per l'esecuzione in background. Nel nostro caso tale classe è CommunicatorThread nel cui metodo *run()* avviene la vera e propria comunicazione con Arduino. Ad ogni esecuzione di *run()* invia un messaggio ad Arduino, se non è presente un nuovo messaggio reitera quello precedente.
Si mette poi in attesa di ricevere un messaggio da Arduino, se quest'ultimo è diverso dal precedente messaggio allora lo passa a IncomingMessage che lo decodificherà e notificherà i propri Observer della presenza di un nuovo messaggio, altrimenti lo ignora.
Il messaggio da inviare è memorizzato nel campo privato mOutgoing che viene di volta in volta impostato da RobotLogic tramite il "setter"[9] *setOutgoing()*. L'uso della tecnica degli "short-lived thread" in Java è piuttosto semplice:

Codice 4.3: Metodo di inizializzazzione degli short-lived thread in Communicator

```
private ScheduledExecutorService mScheduler;
...
public void start() {
```

[9]Metodo che fornisce l'accesso in scrittura ad un campo privato di un oggetto

```
    CommunicationThread thread = new CommunicationThread();
    mScheduler = Executors.newSingleThreadScheduledExecutor();
    mScheduler.scheduleAtFixedRate(thread, 0,
        READING_POLLING_TIME, TimeUnit.MILLISECONDS);
}
```

Ottenuto da Java il riferimento allo ScheduledExecutorService si imposta l'esecuzione ad intervalli regolari di READING_POLLING_TIME millisecondi del nostro CommunicatorThread.
E' bene notare che la *read()* fornita dall'ADK è bloccante, quindi finché Arduino non invia un messaggio, e questo è disponibile sul buffer di lettura, l'esecuzione di *run()* è interrotta. Questo comportamento potrebbe portare all'indesiderata situazione in cui un messaggio in attesa di essere inviato ad Arduino venga sovrascritto dal successivo messaggio elaborato da RobotLogic. Per evitare questo scenario Arduino invia ad ogni iterazione del proprio ciclo *loop()* almeno un messaggio ad Android, che, in assenza di situazioni particolari, sarà la lettura proveniente dal sensore di distanza; in questo modo oltre ad evitare il blocco di *run()* si invia ad Arduino un'informazione comunque utile per la navigazione.

4.2 Arduino

Conclusioni e sviluppi futuri

Bibliografia

[1] Marco Ambroglio. Ray Tracing GPU-based - http://www.unibg.it/dati/corsi/38001/45691-lezione_06052011_optix.pdf

[2] Federico Dolce. Soluzione di problemi di direction finding con l'utilizzo di un algoritmo di posizionamento ad ancora singola. Pag. 16 - Anno 2013

Elenco delle immagini

Appendice