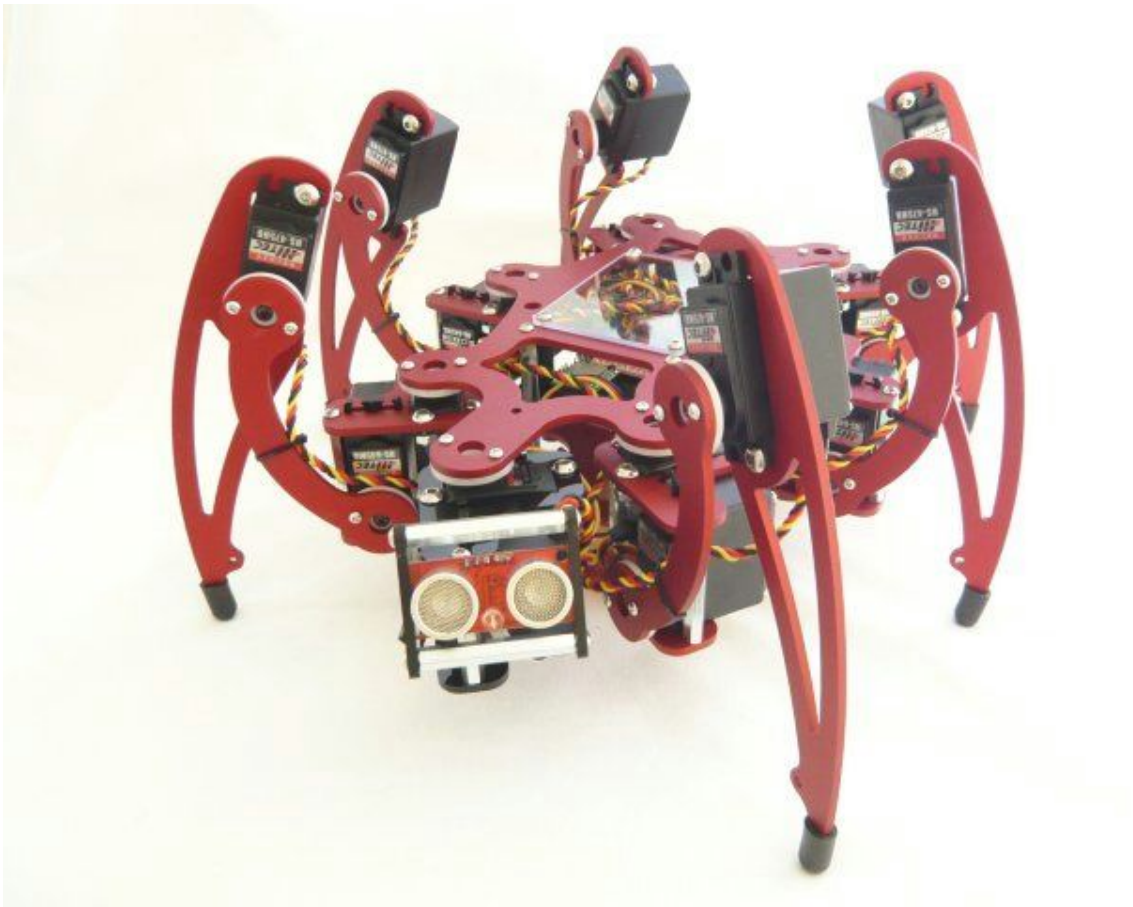
gAitano

Movimento automatizzato tramite lettura QRcode

Capparrucci Matteo, Corò Federico

aa 2014-2015



Strumenti utilizzati

- MSR-H01 Hexapod Kit, che nella relazione verrà rinominato gAitano
- Cellulare con sistema operativo Android (testato su Android M e Android 4.3)
- Libreria OpenCV per android (http://opencv.org/platforms/android.html)
- Libreria usb-serial-for-android
  https://github.com/mik3y/usb-serial-for-android)

Obiettivi

In questo progetto si avevano molteplici obiettivi, tra cui:

- gestire il movimento del robot gAitano
- Far eseguire mosse al robot tramite comandi inviati tramite lettura di un codice QR

Nell'applicazione sono inoltre presenti le seguenti funzionalità:

- Marker Detection tramite OpenCV per Android
- Gioco TicTacToe, utente contro computer, implementato con MiniMax

Movimento gAitano

La libreria da noi scritta per la gestione del movimento è composta da due classi: *Movement* e *MovementHex.*
Per “comunicare” con gAitano usiamo la porta seriale, questa è connessa al dispositivo Android tramite cavo OTG (microusb -> usb), a cui viene inserito un adattatore da usb a cavo seriale che verrà infine collegato al robot.

Poiché i protocolli di comunicazione tra usb e cavo seriale sono diversi, come prima cosa abbiamo utilizzato la libreria OpenSource usb-serial-for-android per riconoscere e attivare il protocollo di trasmissione su cavo seriale (di base tramite seriale si possono inviare solo array di byte).
Il nostro robot invece utilizza come metodo di comunicazione due tipi diversi di pacchetti SIM e PIP, entrambi composti da stringhe esadecimali che dovranno poi essere convertiti in byte.

Connessione alla seriale

Per recuperare la porta seriale deve essere richiamato il metodo *searchUsbSerial* che si trova sulla classe *Movement.* Questo è preferibile farlo tramite un AsyncTask poiché il processo potrebbe richiedere del tempo o più di un ciclo per trovarla. Effettivamente, nel nostro progetto, questo viene fatto dal servizio *ServiceMovimento*, il quale ha il compito di ricevere dalle altre classi i movimenti assegnati e di inviarli al robot utilizzando il metodo *executeCommand* che si trova nella classe *Movement*.

Spieghiamo ora nel dettaglio i metodi per il collegamento con la seriale e l’inizializzazione della classe *Movement*.
Nel nostro servizio abbiamo, come detto prima, un AsyncTask in cui nel metodo *doInBackground* richiamiamo la funzione statica:
Movement.searchUsbSerial(mUsbManager), questo ha il compito di recuperare i driver per le porte seriali (poiché la libreria da noi usata permette più tipi di connessione), poi tramite un ciclo for cercherà di recuperare eventuali connessioni seriali attive.
Su *onPostExecute* viene instanziata la classe *Movement* tramite l’oggetto porta trovato e poi si avvia il metodo *init_connection* che serve ad inizializzare effettivamente l’oggetto *porta* tramite i metodi *sPort.open* e *sPort.setParameters.*

Pacchetti SIM e PIP

PIP e SIM sono i pacchetti utilizzati per la comunicazione delle istruzioni eseguite da gAitano e sono composti da 4 parti:
[Header Byte] [Packet Count] [n Bytes of Data.....] [Check Sum]
Ogni singola parte viene scritta come stringa esadecimale e poi convertita in byte.
L'Header è una stringa sempre uguale (0x7e), si segue poi con un esadecimale che indica il numero di byte che seguiranno (escluso il CheckSum).
Si hanno poi i comandi veri e propri (spiegati più avanti) e infine il checkSum che è semplicemente l'esadecimale 0xff a cui bisogna sottrarre ogni altro esadecimale utilizzato nella "zona" *Bytes of Data.*
Per i movimenti base (es: svegliati, avanti, indietro, etc...) utilizziamo il pacchetto di tipo SIM, più veloce e facile da scrivere ma meno personalizzabile.
Infatti con il pacchetto di tipo PIP è possibile eseguire gli stessi comandi, ad esempio nel comando "avanti" tramite PIP è possibile scegliere di quanto viene mossa in avanti ogni zampa (per tutte uguali, non è possibile gestirle separatamente)
Inoltre si nota che ogni comando di tipo movimento è suddiviso in 4 movimenti ripetuti, ovvero per andare avanti bisognerà inviare 4 volte il comando "avanti" prima che il robot faccio effettivamente un passo avanti. Questo perchè le 6 zampe, in un movimento, sono gestite in 4 fasi.

Di seguito descriviamo un rapido esempio per far meglio comprendere la scrittura dei pacchetti da inviare.
Per il wakeUP per esempio il pacchetto sarà composto da le seguenti quattro stringhe esadecimali: "0x7e, 0x01, 0x2b, 0xd4"
Come vediamo si ha l'header, seguito da il numero di comandi (nel caso di SIM sarà sempre 1), il comando stesso, da documentazione ufficiale per "Power up hexapod" si ha il comando "+".
Si esegue quindi la conversione da ASCII a hex e si ha la stringa "0x2b"
L'ultima stringa è semplicemente "0xff - 0x2b = 0xd4"
Per inviarli devono essere concatenati, quindi la stringa ufficiale da mandare sarà: "0x7e012bd4"

Metodi per invio comando

Il compito di inviare i comandi al robot, nel nostro progetto, è assegnato al servizio *ServiceMovimento* il quale possiede un istanza della classe *Movement* dalla quale richiamerà il metodo *ExecuteCommand* che, come prima cosa controlla se si è effettivamente connessi alla seriale (giusto per sicurezza) e poi chiama il metodo *sPort.write(returnCommand(comando), 200).*
sPort è l'oggetto di collegamento con la seriale,da cui si richiama quindi il metodo *write* che prende come parametri un array di byte e un intero che rappresenta i millisecondi da aspettare prima dell'invio del messaggio.
Il metodo *returnCommand(comando),* in cui la variabile *comando* è una stringa equivalente al comando da eseguire, richiama a sua volta *hexStringToByteArray(map.get(command))*
La *map* è una HashMap in cui abbiamo mappato ogni comando (stringa) con la sua effettiva stringa *esadecimale* corrispondente.
Il metodo *hexStringToByteArray* come da nome converte una stringa esadecimale in un array di byte, abbiamo dovuto riscriverla poichè la funzione builtin di java non funzionava correttamente (non abbiamo capito effettivamente il perchè).

QRCodeProject

Per quanto riguarda la lettura dei QRcode abbiamo utilizzato un intent che richiama il client dell'applicazione zxing, che impiega l'omonima libreria in modo efficace e performante, per la lettura di QRcode. Il client zxing, se presente sul device, viene richiamato in modo trasparente all'utente, in caso contrario viene scaricato dal marker al link:

"market://details?id=com.google.zxing.client.android".

Più nello specifico l'activity QRcode al click sul bottone richiamerà la funzione *qrCerca()* che inizializza con i parametri necessari l'intent del client zxing, il quale, non appena individua un QRcode si chiuderà restituendo la stringa corrispondente grazie a

data.getStringExtra("SCAN_RESULT").

Viene inviato un Bundle al servizio *serviceMovimento* contenente una stringa preliminare "gAitano" e "QRProject", viene poi aggiunta la stringa contenente i comandi.
Il servizio, riconosciuta la prima stringa "QR" si occuperà della gestione della stringa di comandi veri e propri.

Dopo ogni sequenza di riconoscimento verrà lanciata nuovamente la funzione *qrCerca()*, garantendo così la possibilità di inviare uno flusso continuo di istruzioni a gAitano.

Marker Detector

All'interno del progetto è inoltre disponibile l'implementazione di una libreria, che utilizza a sua volta le funzioni di OpenCV, per il riconoscimento degli standard marker.
Questa, a partire dalla selezione di un frame della fotocamera, esegue per prima cosa una serie di operazioni preliminari sullo stesso, cambiandone il tipo da rgb a scala di grigi (*Imgproc.cvtColor(in, grey, Imgproc.COLOR_RGB2GRAY)*) e quindi eseguendo una serie di operazioni atte ad evidenziare le differenze di gradienti al fine di ottenere dei contorni evidenti e di disegnare questi ultimi come insiemi di punti.
Quindi presi questi insiemi di punti, attravero il metodo *Imgproc.approxPolyDP()*, vengono approssimati (ove all'interno di una certa soia) ai soli punti significativi e di questi nuovi insiemi di punti vengono selezionati i gruppi composti da quattro. Individuati quindi i nuovi gruppi, questi vengono ulteriormente scremati elimando quelli il cui perimetro è troppo piccolo e quelli troppo vicini tra loro e infine vengono riportati alla forma di quadrati per il riconoscimento vero e proprio tramite il metodo *Imgproc.getPerspectiveTransform()*.
Una volta individuati i quadrati questi vengono quindi suddivisi in una sotto-matrice a blocchi nei quali sulla base del colore (in scala di grigio) generale del blocco viene assegnato rispettivamente 0 o 1. La matrice di bit così modificata viene quindi confrontata (ruotata in tutte le sue 4 possibili configurazioni) con il/i marker campionati e quindi riconosciuta.
Nell'attivity *MarcherDetectorActivity*, oltre al riconoscimento del marker di prova è implementata una semplice funzione che, se in caso di riconoscimento del marker, richiama tramite intent il servizio serviceMovimento con la quale permette all'esapod di eseguire il comando di wakeUP.

Riferimenti

- http://www.hexapodrobot.com/Files/HexEngine/pBrain-HexEngine%20PIP%20v1.2.pdf
    - Qui si trova la documentazione per la scrittura di pacchetti SIM e PIP