ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA (SEDE DI BOLOGNA)

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA DI

MATTIA GIANESSI

SVOLTA PRESSO

DISI – Dipartimento Informatica Scienza e Ingegneria

In

Calcolatori Elettronici T

SVILUPPO DI UNA APP DI AUSILIO PER PERSONE NON VEDENTI PER ANDROID

CANDIDATO

Mattia Gianessi

RELATORE

Prof. Stefano Mattoccia

CORRELATORE

Dott. Matteo Poggi

Anno Accademico 2015/16

Sessione II

RINGRAZIAMENTI

Al relatore prof. Stefano Mattoccia e al correlatore dott. Matteo Poggi per avermi dato la possibilità di realizzare questo progetto. Al collega Pierpaolo per gli strumenti forniti e al collega Luca per il supporto datomi durante tutto il proseguimento del mio lavoro.

A famiglia e amici per avermi spronato a raggiungere questo risultato.

INDICE

CAPITOLO 1

INTRODUZIONE

Il tema di questa tesi riguarda lo sviluppo di un'applicazione Android di ausilio per persone non vedenti. In particolare, tale applicazione (alla quale d'ora in avanti si farà riferimento con "app") presenta due scopi principali:

1. Interagire con un sistema di ausilio per non vedenti e ipovedenti già esistente, sviluppato dal gruppo di ricerca del DISI [1,2], ricevendo informazioni da quest'ultimo (che viene definito "visore") e riproponendole all'utente sotto forma d'informazioni audio e tattili
2. Proporre una versione di Google Maps personalizzata che faciliti l'interazione dell'utente con questa tipologia di contenuti

Più in specifico, per quanto riguarda la prima parte, l'app fornisce all'utente tre diverse modalità di funzionamento, di cui successivamente verranno spiegati i dettagli:

- Ricezione singolo ostacolo
- Ricezione ostacoli multipli
- Navigazione

Mentre, per quanto riguarda la seconda parte, sono due le modalità di funzionamento offerte all'utente:

- Navigazione da un punto di partenza ad un punto di arrivo
- Ottenimento di informazioni riguardo luoghi nelle vicinanze (ristoranti, uffici postali, chiese e altro)

L'intera app è stata realizzata basandosi sul linguaggio Java e sull'ambiente di lavoro Android Studio [3]; inoltre per la prima parte sono stati utilizzati alcuni strumenti di simulazione per testare il corretto funzionamento dell'app (realizzati in linguaggio C e utilizzati in ambiente Linux), mentre per la seconda sono state utilizzate alcune Google API.

Figura 1: Logo dell'applicazione.

CAPITOLO 2

STRUMENTI UTILIZZATI

In questo breve capitolo verranno analizzati gli strumenti utilizzati per poter realizzare e testare l'app.

2.1 ANDROID STUDIO

Oggi giorno Android è il sistema operativo per dispositivi mobili più diffuso al mondo e, per far sì che l'app potesse essere utilizzata dal maggior numero di utenti possibile, è stato deciso di svilupparla proprio per tale sistema operativo.

I possibili ambienti di sviluppo in cui realizzare l'app erano molteplici e, dopo un attento esame, la scelta è ricaduta su Android Studio, un ambiente di sviluppo integrato (IDE) per la piattaforma Android, reso disponibile agli sviluppatori recentemente (dicembre 2014).

Le motivazioni che hanno portato a tale scelta sono diverse, ma le più importanti sono sicuramente il fatto che Android Studio è stato creato appositamente da Google per sviluppare app in ambiente Android (al contrario di altri strumenti come ad esempio Eclipse) e il fatto che il continuo supporto che l'azienda californiana sta fornendo a tale ambiente garantisca futuri miglioramenti e modifiche all'app rimanendo sempre all'interno dello stesso IDE.

Come qualsiasi IDE per Android, Android Studio sfrutta Java, un linguaggio di programmazione orientato agli oggetti a tipizzazione statica, progettato specificatamente per essere il più possibile indipendente dalla piattaforma di esecuzione.

2.2 VISORE

Come già anticipato nell'introduzione, parte del compito dell'app è quello di interagire con un sistema già esistente definito visore; esso è composto da una telecamera stereo con elaborazione su FPGA [4], collegata a un SBC (single-board computer) Odroid U3 sviluppato da Hardkernel [5].

Il compito di questo visore è quello di riconoscere gli ostacoli che l'utente non vedente incontra durante il percorso e comunicare tale informazione a un sistema di vibrazione oppure ad un sistema audio.

Figura 2: A sinistra i moduli che compongono il "visore" e, a destra, il visore indossato da un utente.

Non potendo però usufruire di questo visore nel corso della realizzazione dell'app, sono stati utilizzati degli strumenti di simulazione (introdotti nel paragrafo successivo) che hanno come compito quello di simulare l'incontro con ostacoli casuali e di testare il corretto funzionamento dell'app.

2.3 SIMULATORI

Gli strumenti di simulazione sono stati di fondamentale importanza per la realizzazione dell'app; essi sono realizzati in linguaggio C e utilizzati in ambiente Linux (in particolare sulla macchina virtuale WMware [6]).

Lo scopo di tali strumenti di simulazione è quello di generare ostacoli, casuali o non, da poter fornire all'app a seconda delle richieste dell'utente.

In particolare il simulatore, gestito da linea di comando, permette all'utilizzatore di generare una mappa di ostacoli "random" oppure di inserire manualmente l'insieme degli ostacoli, specificandone la posizione all'interno della mappa, la distanza e il tipo.

Una volta terminato di inserire gli ostacoli, il simulatore utilizza la libreria NetworkManager (realizzata dai miei colleghi Pierpaolo Perrozzi [7] e Luca Ranalli [8]) la quale contiene un insieme di metodi utili per la trasmissione dati; in particolare essa ha il compito di aprire due FIFO:

- Una in lettura per ricevere i dati degli ostacoli da inoltrare su socket (e quindi all'app)
- Una in scrittura usata per inoltrare ad una libreria di comando la modalità di funzione (ostacolo singolo, ostacolo multiplo o navigazione) desiderato dall'utente

Interagendo quindi con un server demone, il cui compito è quello di "smistare" le richieste ricevute o effettuate sulle *socket*, l'intero sistema viene simulato permettendo di testare l'app come se venisse utilizzato il visore vero e proprio.

2.4 GOOGLE API

Gli ultimi strumenti utilizzati sono le Google API (Application Programming Interfaces): queste sono composte da un set di procedure che Google fornisce in maniera totalmente gratuita (anche se con alcune limitazioni) e che dà la possibilità ai programmatori di comunicare con i Google Services (esempi sono Gmail, Translate e Maps) e di includerle in applicazioni terze parti. I programmatori che vogliono sfruttare queste API devono essere in possesso di una credenziale, o key, ottenibile dal Google Developers Console [9] (una piattaforma tramite la quale richiedere e gestire le varie procedure offerte) per poi includerle nel proprio software.

Come già anticipato, l'utilizzo di queste API è gratuito ma deve sottostare ad alcune restrizioni, una delle quali riguarda in maniera diretta questo progetto: [10]

> *No navigation. You will not use the Service or Content for or in connection with (a) real-time navigation or route guidance; or (b) automatic or autonomous vehicle control.*

Questa clausola sostanzialmente impedisce l'inserimento in applicazioni terze parti della classica modalità di navigazione "turn by turn" di Google Maps obbligando, come verrà illustrato meglio successivamente, l'utente a dover utilizzare proprio l'applicazione di Google nel caso in cui si desideri effettuare una navigazione in tempo reale.

Le API utilizzate nel corso della realizzazione di questa app sono:

- Google Maps Android API [11], utilizzata per ricreare una mappa in tutto e per tutto uguale a quella proposta da Google Maps
- Google Maps Directions API [12], utilizzata per ottenere informazioni sul percorso che l'utente desidera effettuare
- Google Places API Web Service [13], utilizzata per ottenere informazioni sui luoghi presenti intorno all'utente

Più avanti nella relazione tutte e 3 le API verranno esaminate maggiormente nel dettaglio.

CAPITOLO 3

SVILUPPO APPLICAZIONE: PRIMA PARTE (classi)

Nei capitoli successivi verrà descritto il processo di realizzazione dell'app, concentrandosi in particolar modo sulle classi utilizzate per poter gestire i dati necessari al funzionamento dell'app e sulle activity, ovvero l'output fornito direttamente all'utente.

L'analisi di questi componenti sarà suddivisa in prima parte (utilizzo visore e rilevamento ostacoli) e seconda parte (realizzazione versione personalizzata di Google Maps).

3.1 UDP Connection, TCP Connection

Il primo punto su cui mi sono focalizzato è stato il trasporto dei dati tra app e Visore (o nel mio caso specifico simulatore).

In particolare, la comunicazione tra le due entità è stata gestita tramite l'utilizzo di socket (come già anticipato nel paragrafo "Simulatori"), ovvero uno strumento di comunicazione standard per lo scambio di messaggi attraverso una rete.

Lo sviluppo del progetto ha reso necessario l'utilizzo di due diverse tipologie di socket, ovvero UDP (User Datagram Protocol) e TCP (Transmission Control Protocol). Brevemente, la prima tipologia è impiegata quando è prioritaria l'efficienza della comunicazione (in termini di risorse e tempo impiegato) mentre non lo è l'affidabilità, al contrario della seconda tipologia dove è necessario creare un canale di comunicazione fra l'entità mittente e quella ricevente garantendo un livello di affidabilità maggiore a discapito di una minore efficienza.

Nello specifico tra l'app e il server demone vengono instaurate (lato app):

- Una socket UDP di scrittura per comunicare la modalità desiderata dall'utente
- Una socket UDP di lettura per ricevere una mappa di ostacoli
- Una socket TCP di lettura per ricevere un ostacolo o ostacoli multipli o, nel caso in cui la modalità sia quella di navigazione, richiamare il metodo che si occuperà di realizzare la socket UDP sopra citata

Per poter gestire queste socket, sono state realizzate le due classi *UDPConnection* e *TCPConnection* che hanno come scopo quello di creare, mantenere ed utilizzare le socket appena descritte. In particolare, è stato necessario estendere la classe *AsyncTask* di Android che ha come scopo quello di eseguire delle elaborazioni in background e quindi mostrarne i risultati sull'UI del thread principale. Di notevole importanza è il metodo *doInBackground* che è il vero e proprio cuore di ogni elaborazione asincrona compiuta con questa classe.

Ecco un esempio di creazione e utilizzo di questo tipo di thread:

```
udp = new UDPConnection(indirizzo);
udp.execute(modalita);
```

In questo caso è creato un thread che si occupa di gestire una socket UDP (passando come parametro l'indirizzo a cui collegarsi) per poi essere attivato tramite il metodo *execute* a cui viene passata la modalità con il quale l'utente vuole utilizzare l'app.

E' importante specificare che sono state utilizzate due differenti porte di default, ovvero 5555 e 5556 (la seconda riservata per la ricezione della mappa di ostacoli).

3.2 Obstacle

Il passo successivo è stato quello inerente l'invio e ricezione degli ostacoli. Ogni ostacolo presenta 5 interi che lo caratterizzano*: left, center, right, distance_meter, distance_cm e type*.

I primi tre interi, che possono assumere valori 0 o 1, identificano la posizione dell'ostacolo rispetto all'utente:

- Sinistra (*left* = 1, *center* = 0, *right* = 0)
- Centro-Sinitra (*left* = 1, *center* = 1, *right* = 0)
- E così via per un totale di 5 differenti posizioni

Type non è altro che il tipo di ostacolo rilevato dal visore; può assumere un valore da 1 a 8 (qualsiasi altro valore è segnato come OSTACOLO generale, non riconosciuto). In particolare sono presenti i seguenti tipi:

```
private static final String ALBERO = " un albero rilevato ";
private static final String AUTOMOBILE = " un'automobile rilevata ";
private static final String PALO = " un palo rilevato ";
private static final String PERSONA = " una persona rilevata ";
private static final String MURO = " un muro rilevato ";
private static final String GRADINO = " un gradino rilevato ";
private static final String PANCHINA = " una panchina rilevata ";
private static final String CESTINO = " un cestino rilevato ";
private static final String OSTACOLO = " un ostacolo rilevato ";
```

Per poter gestire questi ostacoli, è stata realizzata la classe *Obstacle* che fornisce due modalità per costruire l'ostacolo:

- Attraverso un costruttore in cui vengono impostati i 5 parametri sopra descritti (utile per la mappa, vedi dopo)
- Attraverso il metodo *getObstacle*

In particolare, la seconda modalità risulta utile per la ricezione, da parte dell'app, di un ostacolo o di ostacoli multipli; questo metodo prende in input un *DataInputStream* (uno

stream dati che permette la lettura di tipi di dato primitivi in Java) su cui andremo a richiamare per 5 volte il metodo *readInt* che ci permetterà di ottenere i 5 interi inviati, tramite socket, all'app dal visore.

Questa classe presenta diversi metodi *xToString* (x sta per *position*, *distance* e *type*) che, richiamati insieme, permettono di ridefinire il *toString* della classe e restituire in output (in particolare al *TextToSpeech*, vedi dopo) una stringa identificante il contenuto di quell'ostacolo.

```
@Override
public String toString()
{
    return typeToString() + " "
            + positionToString() + " "
            + distanceToString(true);
}
```

3.3 Map, MapObstacle

Il prossimo passo per realizzare l'app è stato quello di gestire la ricezione delle mappe della modalità navigazione, ovvero l'insieme degli ostacoli visibili dal visore, in un dato istante, di cui poi poter eseguire una sorta di trasposizione sullo schermo dello smartphone attraverso la quale l'utente possa scorrere il dito simulando di fatto l'utilizzo di un bastone da passeggio.

Ogni mappa risulta essere composta da una serie di elementi: un *tag*, *width* (larghezza) e *height* (altezza), *numObst* (numero degli ostacoli presenti all'interno della mappa), un vettore di ostacoli obstacles ed un *maxItems* (ovvero massimo numero di oggetti che costituisce un ostacolo).

Per poter gestire correttamente le mappe, è stato necessario realizzare due differenti classi: *Map* e *MapObstacle*, la prima rappresentante la mappa nella sua interezza e la seconda rappresentante gli ostacoli contenuti all'interno della mappa.

La prima delle due classi presenta un metodo *getMap* che, similmente a quanto fatto nel *getObstacle* di *Obstacle*, permette la lettura di una mappa tramite *DataInputStream* passato come parametro, su cui vengono invocati diversi metodi come *readShort* o *readChar*. Oltre a ciò, saranno creati tanti *MapObstacle* quanti sono quelli indicati da *numObst*, e su ognuno di essi verrà invocato il metodo *getObstacle* (differente nel contenuto, ma non nel compito rispetto a quello già citato) passando come parametro il *DataInputStream* già utilizzato all'interno di *getMap*.

Ogni elemento di *MapObstacle* è di fatto composto da un certo numero di *items*, ognuno identificato sulla mappa attraverso delle particolari coordinate; in particolare, ciascun elemento presenta un *numItems*, un vettore *items* (contenente le coordinate X, Y e *width* di ogni item), *type*, *distMt*, *distCM* e un vettore *side* (simile a *left*, *center* e *right* di *Obstacle*).

Infine è presente un elemento *Obstacle* che, di fatto, rappresenta l'insieme degli *items* e verrà utilizzato, con apposito *toString*, per poter mandare in output, sotto forma di stringa, il contenuto del *MapObstacle*.

A questo punto, terminate le classi base utilizzate nel programma, vengono introdotti gli strumenti *Text to Speech* e *Speech Recognition*, fondamentali per la realizzazione dell'app.

3.3 Text to Speech, Speech Recognition

Essendo l'app destinata a utenti impossibilitati, parzialmente o totalmente, a usufruire della vista, si è cercato di ridurre al minimo l'interazione dell'utente con componenti grafici, puntando a rendere l'app il più possibile utilizzabile tramite comandi vocali. Per poter fare questo è stato necessario l'utilizzo del *Text to Speech* (TTS) [14] e dello *Speech Recognition* (SR) [15].

Il primo dei due componenti (TTS) rappresenta la funzionalità di sintesi vocale presente di base su tutti i dispositivi Android ed è una tecnologia che permette di convertire testo scritto in parlato, riproducendo in maniera artificiale la voce umana.

Per poter sfruttare questo componente, è necessario come prima cosa avviare l'*engine* relativo al nostro TTS; per farlo, viene fatto partire un *Intent* (una sorta di messaggistica gestita dal sistema operativo che permette ad un componente di richiedere l'esecuzione di un'azione da parte di un altro componente) tramite il quale viene controllata la disponibilità dei dati necessari per sfruttare appieno il TTS. In caso positivo si potrà procedere con il settaggio del TTS (lingua, velocità riproduzione e altro), mentre in caso negativo dovrà essere eseguito un'installazione di tali dati.

```
Intent checkTTSIntent = new Intent();
checkTTSIntent.setAction(TextToSpeech.Engine.ACTION_CHECK_TTS_DATA);
startActivityForResult(checkTTSIntent, TTS_DATA_CHECK_CODE);
```

Intent che avvia il controllo dei dati del TTS (*ACTION_CHECK_TTS_DATA*)

```
Intent installTTSIntent = new Intent();
installTTSIntent.setAction(TextToSpeech.Engine.ACTION_INSTALL_TTS_DATA);
startActivity(installTTSIntent);
```

Intent per installare dati mancanti (*ACTION_INSTALL_TTS_DATA*)

```
if (initStatus == TextToSpeech.SUCCESS)
{

if(myTTS.isLanguageAvailable(Locale.ITALY)==TextToSpeech.LANG_AVAILABLE)
        myTTS.setLanguage(Locale.ITALY);
```

Se tutto è corretto, è impostata la lingua del sintetizzatore (*ITALY*).

Nel caso in cui tutti i controlli siano stati eseguiti correttamente, il TTS risulta essere pronto per l'utilizzo; in particolare è stato realizzato il metodo *speakWord* attraverso il quale il sintetizzatore è in grado di pronunciare la stringa passata come argomento:

```
myTTS.speak(speech, TextToSpeech.QUEUE_FLUSH, myHashAlarm);
```

Quella sopra riportata è la riga di codice che si occupa di far "parlare" il TTS (*speech* è la stringa passata come argomento); possiamo osservare altri due parametri:

- .*QUEUE_FLUSH* che obbliga il TTS ad interrompere un'eventuale pronuncia in quanto ne è arrivata un'altra (utile nel caso di ostacoli che si avvicinano, l'altra opzione era .*QUEUE_ADD* che semplicemente mette in coda le nuove pronunce)
- *myHashAlarm*, ovvero una *HashMap<>* utile nel caso in cui, in corrispondenza di un utilizzo di TTS, sia necessario effettuare un riconoscimento vocale (SR)

Come detto, il terzo parametro è utilizzato in collaborazione con un SR, ovvero il riconoscimento vocale che Google offre per i dispositivi Android, utilizzato all'interno dell'app per evitare all'utente la maggior parte dell'interazione con componenti grafici classici di qualsiasi applicazione. In particolar modo, è necessario far capire all'app quando compiere un riconoscimento vocale dopo che la stessa ha usufruito del TTS (ovvero concedere il tempo necessario al sistema per terminare una pronuncia prima che si attivi il SR); per farlo viene appunto utilzzato il *myHashAlarm* che, collegando un apposito *utterance*, segnalerà al sistema la fine della pronuncia:

```
myHashAlarm = new HashMap<String, String>();
myHashAlarm.put(TextToSpeech.Engine.KEY_PARAM_STREAM,
String.valueOf(AudioManager.STREAM_ALARM));
myHashAlarm.put(TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID, utterance);
```

Modificando l'apposito metodo ***onUtteranceCompleted***, sarà possibile attivare l'SR solamente quando necessario.

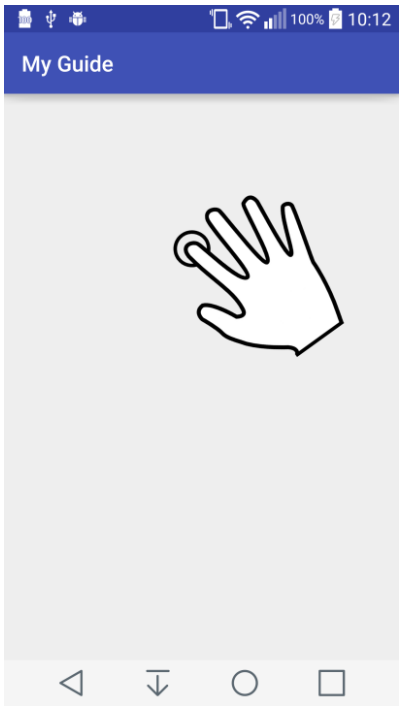
CAPITOLO 4

SVILUPPO APPLICAZIONE: PRIMA PARTE (activity)

In questo capitolo sono introdotte le tre *activity* realizzate per la prima parte di funzionamento dell'app. Un'activity non è altro che una finestra contenente l'interfaccia utente di un'applicazione, e ha come scopo quello di permettere l'interazione con gli utenti. Nel nostro particolare caso, sono 3 le activity realizzate:

- MainActivity (permette all'utente di decidere quale modalità utilizzare)
- OstacoliActivity (modalità ostacolo singolo e ostacoli multipli)
- NavigazioneActivity (modalità navigazione)

4.1 Main Activity



Questa è l'activity che si presenta all'utente al lancio dell'app. Dovendo, come già detto, ridurre al minimo l'interazione dal punto di vista grafico, questa finestra si presenta vuota; il suo compito è quello, una volta che l'utente avrà cliccato in qualsiasi suo punto, di avviare un "dialogo" con l'utente stesso al termine del quale potrà essere avviata un'altra activity oppure settato l'indirizzo IP (oltre alla possibilità di richiedere un aiuto sulle possibili modalità da utilizzare).

Una volta lanciata l'app, basta quindi cliccare la finestra per fa attivare il metodo *onTouchEvent* che, nel caso di un *ACTION_UP* (ovvero il momento in cui il dito si alza dallo schermo) attiva il metodo *speakWords* precedentemente illustrato per pronunciare la costante *INIZIO*, che rappresenta la stringa in cui il sistema richiede come l'utente vuole proseguire.

Figura 3: L'activity principale in cui l'utente può decidere quale modalità scegliere.

A questo punto si attiva l'SR e l'utente può scegliere tra diverse opzioni da pronunciare:

- Istruzioni: fa pronunciare al sistema l'elenco delle possibili modalità eseguibili dall'utente.
- Ostacolo: attiva l'activity *OstacoliActivity* per ricevere un singolo ostacolo.
- Ostacoli: attiva l'activity *OstacoliActivity* per ricevere ostacoli multipli.
- Navigazione: attiva l'activity *NavigazioneActivity* per ricevere mappe.
- Navigatore: vedi seconda parte.

- Indirizzo: l'app richiede all'utente di pronunciare l'indirizzo IP a cui è collegato il visore.
- Comando non riconosciuto: nel caso in cui l'utente si sbagli o il suo comando non sia correttamente compreso, il sistema avverte l'utente stesso dell'accaduto richiedendo di pronunciare il comando un'altra volta.

Nel caso in cui si debbano lanciare nuove activity, è creato un Intent collegato all'activity richiesta in cui sono inseriti uno o due parametri (indirizzo IP e modalità, la seconda solo per *OstacoliActivity*) e successivamente avviata la nuova activity grazie all'intent appena creato.

```
case "ostacolo":
{
    speakWords(PRONUNCIATO + comando, false, "Fine Speech");
    Intent intentOstacolo = new Intent(this, OstacoliActivity.class);
    intentOstacolo.putExtra("MODALITA", "1");
    intentOstacolo.putExtra("IP", indirizzo);
    startActivity(intentOstacolo);
    break;
}
```

MODALITA non è altro che la distinzione tra ostacolo singolo e ostacoli multipli (1 o 2).

4.2 Ostacoli Activity

Questa è l'activity che si presenta all'utente nel caso in cui quest'ultimo abbia pronunciato le modalità "ostacolo" o "ostacoli"; come nel caso precedente, anche questa risulta essere totalmente vuota, per ridurre al minimo l'interazione dell'utente con i classici componenti grafici di Android.

Come prima cosa, sono recuperati i dati passati dalla *MainActivity* grazie ad un intent; in questo modo, oltre a ricevere l'eventuale indirizzo IP impostato, si verifica la volontà dell'utente di ricevere un singolo ostacolo o ostacoli multipli; questa viene poi usata nel momento dell'utilizzo della *UDPConnection* dove, passando come parametro di esecuzione proprio la modalità, è impostato il comportamento del programma.

```
Intent intent = getIntent();
modalita = intent.getStringExtra(MainActivity.MODALITA);
indirizzo = intent.getStringExtra(MainActivity.IP);
```

Dopo aver utilizzato la *UDPConnection*, è creata ed eseguita una *TCPConnection*, per predisporre l'activity a ricevere l'ostacolo o gli ostacoli a seconda della modalità scelta. Ogni qualvolta sarà rilevato un nuovo ostacolo, il suo contenuto (ottenuto con i già citati *toString*) sarà pronunciato tramite l'apposito TTS, in modo tale da avvertire l'utente della presenza di tale ostacolo.

Figura 4: L'activity ostacoli con la quale l'utente è in grado di rilevare ostacoli singoli o multipli.

4.3 Navigazione Activity

Infine è introdotta l'activity più interessante, ovvero quella della modalità navigazione, attivabile dall'utente pronunciando, nella *MainActivity*, l'apposito comando "navigazione".

Dopo aver ottenuto l'indirizzo IP tramite intent, come già visto nell'activity precedente, sono introdotte due novità: la prima è il componente *Vibrator* il quale, come suggerisce il nome, permette all'app di far vibrare lo smartphone; la seconda invece è il componente *RelativeLayout* il quale viene utilizzato per poter contenere i *button* creati in base alla mappa di ostacoli ricevuti.

```
rlButtons = new RelativeLayout(this);
RelativeLayout.LayoutParams rlButtonsParam = new
RelativeLayout.LayoutParams(
        ViewGroup.LayoutParams.FILL_PARENT,
        ViewGroup.LayoutParams.FILL_PARENT);
setContentView(rlButtons, rlButtonsParam);

mVibrator = (Vibrator) getSystemService(Context.VIBRATOR_SERVICE);
```

A questo punto, dopo che l'utente clicca la finestra, si attiva *l'onTouchEvent* che permette all'activity di creare le due *UDPConnection* e la *TCPConnection* necessarie per poter ricevere la mappa dal visore; una volta ricevuta, viene invocato il metodo più importante di questa activity, ovvero il *createButton* a cui viene passato come parametro la mappa appena ottenuta.

Il compito di questo metodo consiste nel creare i *button* rappresentanti la mappa di ostacoli, assegnando a ciascuno di essi la possibilità di effettuare lo swipe in/swipe out (tramite *onTouchListener*) e la pressione lunga (tramite *onLongClickListener*). Per poter creare i *button*, è stato sviluppato un algoritmo tramite il quale si va a riprodurre la mappa ricevuta scalandola a seconda delle dimensioni dello schermo del proprio smartphone (*xScale* e *yScale*) e impostando le coordinate dei *button* (*setX***,** *setY*) e le loro dimensioni in termini di larghezza e altezza (*setLayoutParams*) proprio ridimensionandoli a seconda della scala ottenuta. In particolar modo si va a suddividere lo schermo in N blocchi aventi tutti la stessa altezza, all'interno del quale andare ad inserire i vari button aggiunti, infine, al già citato *RelativeLayout***.**

```
int xScale = rlButtons.getWidth() / map.width;
int yScale = rlButtons.getHeight() / map.height;

for(int i = 0; i < map.numObst; i++)
{
    final MapObstacle obs = map.obstacles[i];
    for(int j = 0; j < obs.numItems; j++)
    {
        final Button btnOst = new Button(this);
        btnOst.setBackgroundColor(Color.BLACK);
        btnOst.setX(obs.items[j][0] * xScale);
        btnOst.setY(obs.items[j][1] * yScale);
        btnOst.setLayoutParams(new
RelativeLayout.LayoutParams(obs.items[j][2] * xScale, yScale));
        rlButtons.addView(btnOst);

        buttons.add(btnOst);
```

Come già anticipato, sono settati due metodi su ogni singolo *button* per poter eseguire lo swipe in / swipe out e la pressione lunga.

Nel primo caso per poter verificare che il dito dell'utente, nel momento dello "swipe", si trovi sopra un particolare *button* oppure fuori da tutti questi, vengono creati dei *Rect* (rettangoli) ottenibili dalle coordinate dei *button*; se l'evento che ha scaturito *l'onTouch* si trova all'interno di uno di questi *rect*, allora si attiverà la vibrazione; altrimenti, in caso contrario, non succederà nulla fino a quando il dito non si ritroverà su uno dei svariati *button* presenti.

```
Rect rect = new Rect(
        (int)btnOst.getX(),
        (int)btnOst.getY(),
        (int)btnOst.getX() + obs.items[j][2] * xScale,
        (int)btnOst.getY() + yScale);

rects.add(rect);
```

Dopo che l'utente ha "scoperto" un ostacolo sullo schermo, è in grado di ottenere delle informazioni da esso semplicemente tramite il secondo caso, ovvero mantenendo premuto il dito su uno dei button che compongono il *MapObstacle* si attiverà *l'onLongClickListener* che, tramite TTS e l'apposito *toString*, pronuncerà le informazioni desiderate.

Figura 5: L'activity navigazione con la quale l'utente è in grado di rilevare una mappa di ostacoli.

CAPITOLO 5

SVILUPPO APPLICAZIONE: SECONDA PARTE (classi)

Per quanto riguarda la seconda parte di funzionamento dell'app, come già anticipato l'intenzione era quella di realizzare una sorta di versione "personalizzata" di Google Maps, dando la possibilità all'utente di utilizzare tutte le funzionalità offerte dalla famosa applicazione proprietaria dell'azienda californiana in maniera facilitata, combinando il tutto con la rilevazione degli ostacoli descritta nella prima parte in modo da essere usufruita dall'utilizzatore in maniera comoda e sicura. L'obiettivo è stato raggiunto ma in maniera differente rispetto a quella prefissata in quanto le già citate politiche di privacy adottate da Google permettono l'utilizzo parziale delle API proprietarie, impedendo la classica modalità di navigazione "turn by turn" offerta da Maps. Nel corso della relazione verrà esposto la soluzione adottata a questo problema.

Da notare che parte del codice proposto in seguito è stato preso e rielaborato da siti specializzati nell'utilizzo di queste API, i cui link verranno riportati nella bibliografia.

5.1 DirectionFinder

La prima delle classi introdotte è colei che si occupa di offrire all'utente la possibilità di inserire un punto di partenza, una destinazione e di ottenere informazioni riguardo al percorso da effettuare.

5.1.1 Google Maps Directions API

Questa API permette di calcolare il o i percorsi che collegano due punti sulla mappa, il tutto tramite una semplice richiesta HTTP (Hyper Text Transfer Protocol); specificando infatti pochi parametri, come ad esempio origine e destinazione (in formato Latitudine/Longitudine oppure indicando la via) la modalità di viaggio (automobile, mezzi pubblici o a piedi) e altri parametri opzionali come orario di partenza e di arrivo, traffico, percorso con minor tratti a piedi e molto altro ancora.

```
https://maps.googleapis.com/maps/api/directions/json?origin=Brooklyn&
destination=Queens&mode=transit&key=YOUR_API_KEY
```

Quello sopra riportato è un semplice esempio di richiesta dove l'origine è Brooklyn, la destinazione è Queens e la modalità di viaggio è *transit* (mezzi pubblici).

L'ultimo fattore da analizzare è la risposta: essa può essere fornita sia in formato JSON (Java Script Object Notation) sia in formato XML (eXtensibile Markup Language). Entrambi i formati sono corretti, ma per l'app è stato utilizzato il primo dei due in quanto JSON garantisce una rappresentazione più compatta, in termini di byte, e una velocità di riposta maggiore rispetto a XML.

```
"distance" : {
                  "text" : "10,9 mi",
                  "value" : 17560
               },
               "duration" : {
                  "text" : "50 min",
                  "value" : 2980
               },
               "end_address" : "Queens, New York, Stati Uniti",
               "end_location" : {
                  "lat" : 40.7282259,
                  "lng" : -73.7948332
               },
               "start_address" : "Brooklyn, New York, Stati Uniti",
               "start_location" : {
                  "lat" : 40.678183,
                  "lng" : -73.94416129999999
```

Questa è una piccola parte di risposta alla richiesta HTTP prima riportata come esempio.

5.1.2 DirectionFinder.java

Questa classe estende la già citata *AsyncTask*, dando quindi la possibilità di calcolare in background la riposta alla richiesta HTTP inviata. La prima cosa da realizzare è proprio l'URL, simile all'esempio precedentemente visto, che compone la richiesta HTTP.

```
return DIRECTION_URL_API + "origin=" + urlOrigin
        + "&destination=" + urlDestination
        + "&mode=" + mode
        +  "&key=" + GOOGLE_API_KEY;
```

La variabile *mode* indica la modalità di viaggio desiderata dall'utente il quale può scegliere tra mezzi pubblici (*transit*) o a piedi (*walking*). Il passo successivo è quello di aprire una connessione con la risorsa a cui fa riferimento l'URL sopra costruito e leggere la risposta fornita dal server di Google.

```
URL url = new URL(link);
InputStream is = url.openConnection().getInputStream();
StringBuffer buffer = new StringBuffer();
BufferedReader reader = new BufferedReader(new InputStreamReader(is));

String line;
while ((line = reader.readLine()) != null) {
    buffer.append(line + "\n");
}
```

Il contenuto della risposta, che risiede nella variabile *buffer*, risulta illeggibile allo stato attuale e quindi deve subire un'operazione di *parsing*: si va cioè a estrapolare le informazioni necessarie per poter fornire all'utente una risposta adeguata. Questo procedimento viene effettuato dall'apposito metodo *parseJSON* che restituirà all'activity che gestisce la seconda parte di questa app l'insieme dei percorsi possibili e le relative informazioni.

```
JSONObject overview polylineJson =
jsonRoute.getJSONObject("overview_polyline");
JSONArray jsonLegs = jsonRoute.getJSONArray("legs");
JSONObject jsonLeg = jsonLegs.getJSONObject(0);
JSONObject jsonDistance = jsonLeg.getJSONObject("distance");
JSONObject jsonDuration = jsonLeg.getJSONObject("duration");
JSONObject jsonEndLocation = jsonLeg.getJSONObject("end_location");
JSONObject jsonStartLocation = jsonLeg.getJSONObject("start_location");
```

Sono creati quindi una serie di *JSONObject* contenenti diverse informazioni, ottenibili tramite l'apposito metodo *getJSONObject* indicando, come stringa, il nome a cui fanno riferimento tale informazioni: *distance* per la distanza dal punto di arrivo, *duration* per il tempo di percorrenza e *end_location* e *start_location* per indicare punto di arrivo e punto di partenza di un singolo tratto di percorso mentre *legs* rappresenta l'insieme di questi tratti. Un discorso più delicato va fatto per la componente *overview_polyline* che sostanzialmente descrive un insieme di punti tramite i quali è possibile disegnare il percorso ottenuto sulla mappa. Questi punti vanno decodificati e per farlo è stato necessario utilizzare un algoritmo [16] già esistente e utilizzato da molti programmatori del settore.

```
do
{
    b = poly.charAt(index++) - 63;
    result |= (b & 0x1f) << shift;
    shift += 5;
} while (b >= 0x20);
int dlat =
        ((result & 1) != 0 ? ~(result >> 1) : (result >> 1));
lat += dlat;
```

Questa è una piccola parte dell'algoritmo di decodifica sopra citato.

```
"overview_polyline" : {
            "points" :
"s}gwF~eibMTB_@fPMAC`AGbCCz@N{GXwMp@k\\n@eYbBox@n@iY\\eOb@uQpAsm@p@c]PaDZ
}EXaF@aDMyM]gLUiFq@}JoA_NgAgJuAoJ}@gFyDmSiFyXuP_}@{XkzAeGk[qF}YiNut@}Mqs@
yIwd@[cA[cB{FyZiBqHsA}E_AoEg@iEIk@OkCQqBeBoJGBqDaCIDUkB]yB[mCcAmF[cB_AmHm
@}DEi@qAgHWqBM{A[wCA[w@gFs@mEYyAo@iEiAgDm@aB_@yAmAsEI[uGjDUBMm@GYi@kCaBwI
_NvGkHnD|@jDH\\y@BiBM}@BkAJcAXu@VaBn@}@L]@yAQ]MmBaAwEcCu@WgESy@M_@My@]a@Y
i@e@i@{@OWAUOa@u@_@oBk@[I_@MUYo@O_FA}@@gDB_BF}AHm@RcBdAw@b@_B|@kBv@SH{Bh@
yDlAaIfCAK@Jv@WDTJRd@rD^rDc@D"
         },
```

Questo invece è un esempio di come l'informazione di *polyline* si presenti nella risposta in formato JSON.

Nel caso in cui l'utente decida di utilizzare i mezzi pubblici, è necessario considerare ulteriori informazioni riguardanti il tipo del mezzo, la linea e le fermate che esso effettua.

```
JSONObject jsonDetails =  jsonStep.getJSONObject("transit_details");
int nStops = jsonDetails.getInt("num_stops");
JSONObject jsonLine = jsonDetails.getJSONObject("line");
String short_name = jsonLine.getString("short_name");
JSONObject jsonVehicle = jsonLine.getJSONObject("vehicle");
String name = jsonVehicle.getString("name");
```

La Directions API fornisce la gestione di una vasta gamma di tipologie di mezzi, come metropolitane, taxi e traghetti ma al momento l'app è in grado di riportare correttamente solo informazioni per bus e treni.

5.1.3 Route.java

Per poter fornire una migliore rappresentazione delle informazioni, espandibile in futuro, è stato deciso di realizzare una classe a parte, *Route.java*, che potesse racchiudere tutte i dati raccolti dalla *DirectionFinder.java* e fornirli in maniera semplice all'activity esterna.

```
route.distance = new Distance(jsonDistance.getString("text"),
jsonDistance.getInt("value"));
route.duration = new Duration(jsonDuration.getString("text"),
jsonDuration.getInt("value"));
route.endAddress = jsonLeg.getString("end_address");
route.startAddress = jsonLeg.getString("start_address");
route.startLocation = new LatLng(jsonStartLocation.getDouble("lat"),
jsonStartLocation.getDouble("lng"));
route.endLocation = new LatLng(jsonEndLocation.getDouble("lat"),
jsonEndLocation.getDouble("lng"));
route.points = decodePolyLine(overview_polylineJson.getString("points"));
```

Tra i vari dati gestiti sono presenti anche *lat* e *lng* che rappresentano rispettivamente latitudine e longitudine.

Sono state create inoltre le classi *Distance.java*, *Duration.java* e *Transit.java* che forniscano una rappresentazione facilitata delle informazioni rispettivamente della distanza, durata e del mezzo nel caso in cui venga scelta l'opzione mezzi pubblici.

5.2 NearbyPlacesData

La seconda classe che deve essere analizzata in maniera approfondita è la *NearbyPlacesData.java* responsabile di fornire all'utente i dettagli dei luoghi che lo circondano.

5.2.1 Google Places API Web Service

Questa API permette di ottenere informazioni riguardanti i luoghi presenti nei dintorni con relativi dettagli come valutazione, orario di apertura, foto e altro oltre a fornire un servizio di auto completamento di nomi o indirizzi parzialmente digitati dall'utente. Il suo funzionamento, come per la *Google Maps Directions API* (paragrafo 5.1.1), si basa su una richiesta HTTP e su una riposta in formato JSON.

In particolare, la richiesta che l'app effettua durante il suo funzionamento è la *Nearby Search Request*, ovvero la specifica richiesta che permette di ottenere i luoghi nelle vicinanze. Oltre ai parametri obbligatori come *location* (latitudine/longitudine del posto in cui ci si trova) e *radius* (la distanza in metri entro cui cercare, fino ad un massimo di 50.000) è possibile effettuare richieste più specifiche inserendo prezzo minimo e massimo desiderato (*minprice*, *maxprice*), il tipo di luogo ricercato e altro ancora.

```
https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=-
33.8670522,151.1957362&radius=500&type=restaurant&name=cruise&key=YOUR_API_KEY
```

```
"place_id" : "ChIJLfySpTOuEmsRPCRKrzl8ZEY",
         "reference" : "CoQBewAAAK1YkcM-
s_17ppv1w5NnWhubCozDIJ_ET2HXMtfj371Hf0yBxoH7ADGRnEr_ad_AGOiln_ZuE9ZoT3_18
Cb_bXtStt3r1xfIFc5vUMdyux2gR91lpRhvNkR2K8dl49yXHWCOX6qbAVLIo0-
MH9r5QngMCBSPZPdFEXq4IEU4GqpwEhD8lABntorqxX54LaBl3-AVGhQSt6-
DIsV77hFlRmgC6Xs-Z6o_rA",
         "scope" : "GOOGLE",
         "types" : [
            "night_club",
            "bar",
            "restaurant",
            "food",
            "point_of_interest",
            "establishment"
         ],
         "vicinity" : "37 Bank Street, Pyrmont"
```

L'esempio sopra riportato indica la richiesta di ristoranti in un punto particolare a Sydney, Australia, nel raggio di 500 metri e che nel loro nome riportano la parola "cruise", con relativa risposta.

5.2.2 NearbyPlacesData.java

Anche questa classe come la *DirectionFinder.java* (paragrafo 5.1.2) lavora in modalità background estendendo la *AsyncTask*. Come prima è necessario costruire l'URL che eseguirà la richiesta HTTP e successivamente leggere la risposta in formato JSON.

```
StringBuilder googlePlacesUrl = new
StringBuilder("https://maps.googleapis.com/maps/api/place/nearbysearch/js
on?");
googlePlacesUrl.append("location=" + latitude + "," + longitude);
googlePlacesUrl.append("&radius=" + PROXIMITY_RADIUS);
googlePlacesUrl.append("&type=" + nearbyPlace);
googlePlacesUrl.append("&sensor=true");
googlePlacesUrl.append("&key=" + GOOGLE_API_KEY);
```

Il codice riguardante l'ottenimento della risposta non è riportato in quanto identico a quello già analizzato nel paragrafo 5.1.2. Una volta ottenuto, il contenuto della risposta deve subire un processo di parsing realizzato dalla apposita classe *DataParser.java*; come nel caso precedente vengono selezionati solo i dati a cui siamo interessati tra cui il nome, la posizione e la distanza dal punto ricercato del luogo trovato.

```
if (!googlePlaceJson.isNull("name")) {
    placeName = googlePlaceJson.getString("name");
}
if (!googlePlaceJson.isNull("vicinity")) {
    vicinity = googlePlaceJson.getString("vicinity");
}
latitude =
googlePlaceJson.getJSONObject("geometry").getJSONObject("location").getSt
ring("lat");
longitude =
googlePlaceJson.getJSONObject("geometry").getJSONObject("location").getSt
ring("lng");
reference = googlePlaceJson.getString("reference");

if(!googlePlaceJson.isNull("rating"))
    rating = googlePlaceJson.getString("rating");

googlePlaceMap.put("place_name", placeName);
googlePlaceMap.put("vicinity", vicinity);
googlePlaceMap.put("lat", latitude);
googlePlaceMap.put("lng", longitude);
googlePlaceMap.put("reference", reference);
googlePlaceMap.put("rating", rating);
```

E' necessario soffermarsi sul parametro *rating*, che rappresenta la valutazione media che la "comunità" ha dato al luogo trovato. Per poter facilitare l'utilizzo dell'app, è stato deciso di fornire in automatico, tramite informazioni audio, la descrizione del luogo avente una valutazione maggiore tra tutti i luoghi ricercati; un'altra opzione percorribile sarebbe stata

quella di fornire informazioni sul luogo più vicino (facilmente ottenibile sfruttando il parametro *vicinity*).

Come ultimo passo si è deciso di evidenziare l'insieme dei luoghi trovati sulla mappa presente nell'activiy sotto forma di marker, le classiche icone di Google Maps, riportanti il nome, l'indirizzo e la valutazione del luogo indicato, per consentire ad un eventuale accompagnatore di visualizzare tutte le possibili alternative disponibili.

```
MarkerOptions markerOptions = new MarkerOptions();
HashMap<String, String> googlePlace = nearbyPlacesList.get(i);
double lat = Double.parseDouble(googlePlace.get("lat"));
double lng = Double.parseDouble(googlePlace.get("lng"));
String placeName = googlePlace.get("place_name");
String vicinity = googlePlace.get("vicinity");
double rating  = Double.parseDouble(googlePlace.get("rating"));
LatLng latLng = new LatLng(lat, lng);
markerOptions.position(latLng);
markerOptions.title(placeName + " : " + vicinity + " valutazione:" +
rating);
markerOptions.icon(BitmapDescriptorFactory.defaultMarker(
        BitmapDescriptorFactory.HUE_RED));

mMap.addMarker(markerOptions);
```

Figura 6: Il marker caratteristico di Google Maps.

La variabile *mMap* non è altro che la mappa presente nell'activity esterna mentre MarkerOptions rappresenta il marker da posizionare sulla mappa con relative informazioni come titolo e simbolo utilizzato.

CAPITOLO 6

SVILUPPO APPLICAZIONE: SECONDA PARTE (activity)

In questo capitolo è analizzata l'unica activity che caratterizza la seconda parte dello sviluppo di questa app; come detto precedentemente, in questo caso è stato deciso di lasciare tutti i componenti grafici presenti in Google Maps, in modo da aiutare un eventuale accompagnatore ad orientarsi in caso di necessità.

6.1 Google Maps Activity

Questa finestra apparirà all'utente nel caso in cui lo stesso abbia pronunciato il comando "Navigatore" nella *main activity* (vedi capitolo 4).

6.1.1 Google Maps Android API

Questa API è utilizzata per differenti scopi come mostrare una mappa in una località specifica e con un certo livello di zoom, utilizzare la Google Street View (la visuale in prima persona utilizzata in Google Maps), ricerca di un luogo in particolare e molto altro, il tutto sfruttando il meccanismo degli intent già ampiamente discusso. Ogni "intent request" necessita di tre parametri fondamentali: *Action* (*ACTION_VIEW* per Google Maps), *URI* (stringhe codificate che esprimo l'azione che si vuole eseguire) e infine *Package* che sostanzialmente è un parametro di controllo che permette di far gestire queste richieste a Google Maps o, nel caso in cui essa sia assente nel dispositivo su cui si sta lavorando, all'applicazione maggiormente indicata per questo scopo.

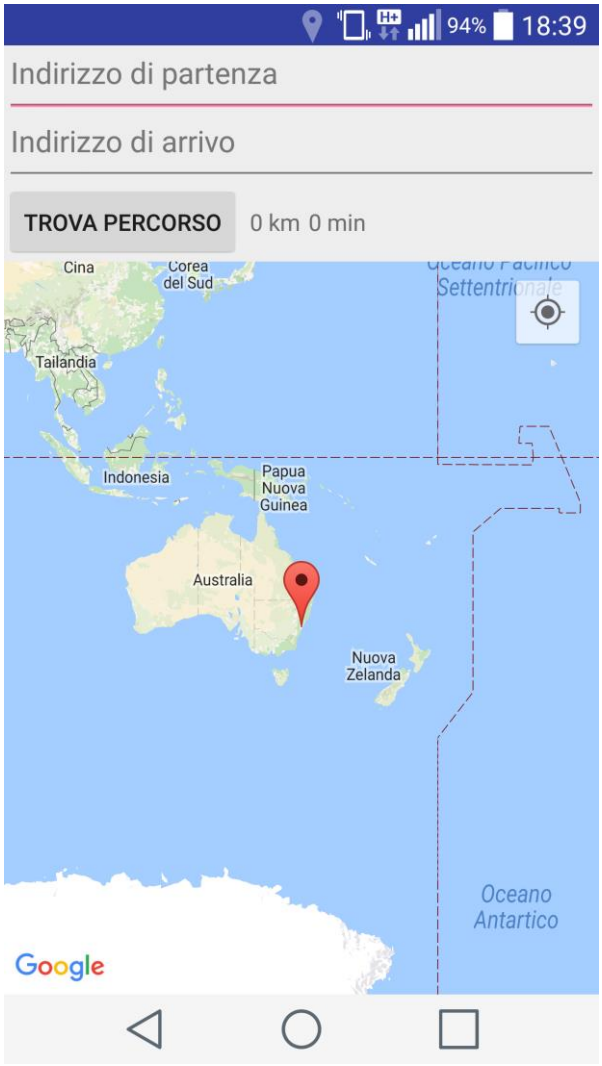
Per poter effettuare la decodifica degli URI basta utilizzare la funzione già disponibile in Android Studio ovvero *parse* di *android.net.Uri.* Ecco quindi un esempio di utilizzo:

```
// Create a Uri from an intent string. Use the result to create an
Intent.
Uri gmmIntentUri =
Uri.parse("google.streetview:cbll=46.414382,10.013988");

// Create an Intent from gmmIntentUri. Set the action to ACTION_VIEW
Intent mapIntent = new Intent(Intent.ACTION_VIEW, gmmIntentUri);
// Make the Intent explicit by setting the Google Maps package
mapIntent.setPackage("com.google.android.apps.maps");

// Attempt to start an activity that can handle the Intent
startActivity(mapIntent);
```

6.1.2 GoogleMapsActivity.java



Ecco come si presenta l'activity dedicata a questa seconda parte del progetto; in alto sono presenti due caselle di testo per indicare il punto di partenza e il punto di arrivo richiesti, un *button* per effettuare la richiesta e due *label* per riportare la distanza in km e il tempo di percorrenza in ore/minuti.

A questo punto l'utente ha la possibilità di effettuare due operazioni: attraverso un'azione di *LongPressClick* (ovvero pressione prolungata con il dito dello schermo) si attiverà la procedura per richiedere la prima delle due modalità di funzionamento (informazioni sul percorso che collegano i punti di partenza e di arrivo), mentre con un'azione di *DoubleClick* (doppio click sullo schermo) si attiverà la seconda delle due modalità (informazioni sui luoghi che circondando l'utente).

Ovviamente tutto questo sarà gestibile anche tramite comandi vocali.

Figura 7: Schermata che presenta la Google Maps Activity

```
. //LONG CLICK
mMap.setOnMapLongClickListener(new GoogleMap.OnMapLongClickListener()
{
    @Override
    public void onMapLongClick(LatLng latLng)
    {
        speakWords(PARTENZA, true, "Partenza");
    }
});
```

Nel primo dei due casi (vedi sopra per gestione LongClick) l'app richiederà, tramite il meccanismo già spiegato nei capitoli precedenti con il metodo *speakWord* e la gestione di *utterance*, l'indirizzo di partenza, quello di arrivo e se si desidera calcolare il percorso nel caso di mezzi pubblici o no.

```
if (resultCode == RESULT_OK && data != null)
{
    ArrayList<String> partenza =
            data.getStringArrayListExtra(RecognizerIntent.EXTRA_RESULTS);
    etOrigin.setText(partenza.get(0));
    speakWords(ARRIVO, true, "Arrivo");
}
```

Come prima cosa è settato il punto di partenza (che è riportato nell'apposita casella di testo sullo schermo) e successivamente parte la procedura per richiedere il punto di arrivo, del tutto uguale all'esempio sopra riportato per la partenza. Ottenuto anche il secondo dato richiesto, l'utente inserirà la modalità di viaggio desiderata.

```
if (resultCode == RESULT_OK && data != null)
{
    ArrayList<String> modeRichiesta =
            data.getStringArrayListExtra(RecognizerIntent.EXTRA_RESULTS);
    if(modeRichiesta.get(0).contains("piedi"))
        mode = "walking";
    else
        mode = "transit";
    btnFindPath.performClick();
}
```

Ovviamente i parametri inseriti nelle richieste HTTP devono essere in lingua inglese, per questo nel caso in cui l'utente richieda la modalità "a piedi" essa è impostata come *walking*, mentre nel caso di "mezzi pubblici" come *transit*. Una volta ottenuti tutti i parametri, si procede con il simulare la pressione del *button* "trova percorso" in quanto all'interno del suo metodo *OnClick* si attiva un altro metodo, *sendRequest*, che si occupa di inviare la richiesta HTTP.

Una volta ottenuta la risposta (elaborata dalla già citata *DirectionFinder.java*) bisogna riportare sulla mappa il percorso e i due *marker* rappresentanti il punto di partenza e il punto di arrivo richiesti.

```
originMarkers.add(mMap.addMarker(new MarkerOptions()
        .icon(BitmapDescriptorFactory.defaultMarker())
        .title(route.startAddress)
        .position(route.startLocation)));
destinationMarkers.add(mMap.addMarker(new MarkerOptions()
        .icon(BitmapDescriptorFactory.defaultMarker())
        .title(route.endAddress)
        .position(route.endLocation)));

PolylineOptions polylineOptions = new PolylineOptions().
        geodesic(true).
        color(Color.BLUE).
        width(10);

for (int i = 0; i < route.points.size(); i++)
    polylineOptions.add(route.points.get(i));

polylinePaths.add(mMap.addPolyline(polylineOptions));
```

Per entrambi i marker sono settati icona, titolo e posizione, mentre il percorso è ottenuto andando ad "unire i punti" che compongono lo stesso fino a formare il *polylinePath* riportato sulla mappa.

Come ultimo passo l'app chiede all'utente se vuole effettivamente andare alla destinazione: in caso affermativo, l'app stessa si occuperà di attivare Google Maps passando come parametro la destinazione desiderata.

```
if (resultCode == RESULT_OK && data != null)
{
    ArrayList<String> comando =
            data.getStringArrayListExtra(RecognizerIntent.EXTRA_RESULTS);
    if(comando.get(0).equals("vai"))
    {
        Uri navigatore = Uri.parse("google.navigation:q=" +
                etDestination.getText());
        Intent navigatoreMaps = new Intent(
                Intent.ACTION_VIEW, navigatore);
        navigatoreMaps.setPackage("com.google.android.apps.maps");
        startActivity(navigatoreMaps);
    }
}
```

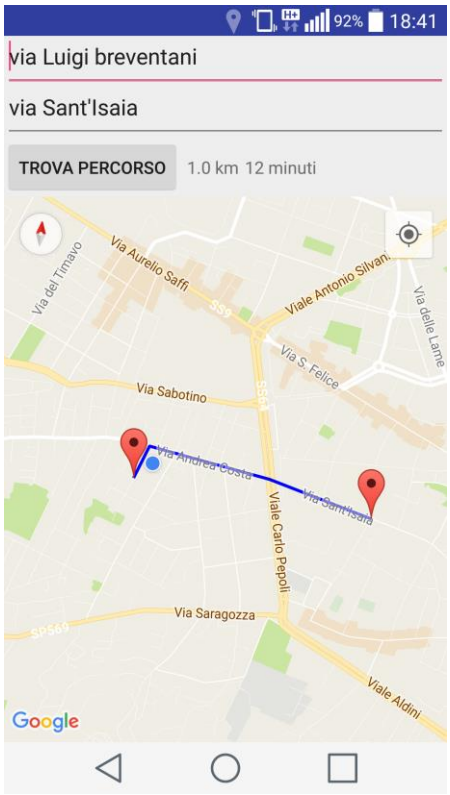


Figura 8: Richiesta per il calcolo del percorso da "Via Luigi Breventani" a "Via Sant'Isaia"

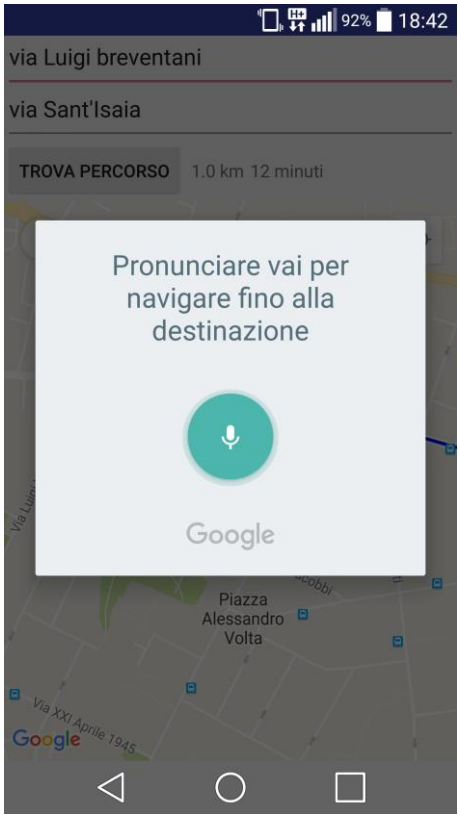


Figura 9: In caso affermativo, l'app si occupa di aprire Google Maps in automatico.

Nel secondo caso invece, ovvero doppio click, l'app richiederà all'utente quale tra le tante tipologie disponibili, banche, benzinai, parchi e tanto altro, desidera ricercare; come nel caso dei mezzi pubblici però solo alcune categorie specifiche sono gestite, ovvero ristoranti, uffici postali e chiese, scelte per poter offrire una gamma più eterogenea possibile di luoghi.

```
if(ricerca.equals("ristorante") || ricerca.equals("ristoranti"))
    return "restaurant";
else if(ricerca.equals("ufficio postale"))
    return "post_office";
else if(ricerca.equals("chiesa") || ricerca.equals("chiese"))
    return "church";
```

Una volta che l'app ha formulato la richiesta HTTP e ottenuto la risposta sfruttando la NearbyPlacesData.java (paragrafo 5.2.2) è in grado di mostrare sulla mappa tutti i luoghi trovati tramite gli appositi marker e di riferire all'utente quello con la migliore valutazione.

Figura 10: Il risultato di una ricerca di ristoranti in un punto di Bologna.

Anche in questo caso l'app chiede all'utente se vuole raggiungere il luogo ricercato con migliore valutazione, attivando come prima Google Maps in maniera automatica.

CAPITOLO 7

CONCLUSIONI

Gli obiettivi inizialmente stabiliti sono stati raggiunti tutti in maniera soddisfacente, alcuni con piccole variazioni ma pur sempre in maniera coerente con quello richiesto. L'app presenta grande margini di miglioramento in futuro, sia per quanto riguarda la parte relativa alla rilevazione degli ostacoli sia per quanto riguarda quella relativa a Maps; grazie all'utilizzo di molte altre API fornite da Google infatti l'app potrebbe offrire un numero sconfinato di nuove funzionalità (oltre a migliorare quelle già presenti aumentando il numero di luoghi ricercabili ad esempio), rimanendo comunque fedele a quello che è la linea guida di questo progetto, ovvero offrire il tutto in maniera che un utente ipovedente possa utilizzarla senza problemi.

Oltre a ciò è necessario ribadire ancora una volta che l'applicazione è stata progettata per il sistema operativo Android, rendendolo quindi accessibile alla quasi totalità degli smartphone presenti sul mercato.

Infine vorrei sottolineare quanto il lavoro da me svolto, ma soprattutto quello svolto dai componenti più esperti dal DISI, possa portare in futuro ad importanti passi avanti nello sviluppo di tecnologie che possano garantire, a persone affette da cecità, un miglioramento nella propria quotidianità.

BIBLIOGRAFIA

[1] M. Poggi, S. Mattoccia, “A wearable mobility aid for the visually Impaired based on embedded 3D vision and deep learning”, First IEEE Workshop on ICT Solutions for eHealth (IEEE ICTS4eHealth 2016)

[2] S. Mattoccia, P. Macrì, “3D glasses as mobility aid for visually impaired people”, Second Workshop on Assistive Computer Vision and Robotics (ACVR2014), ECCV Worskhop, September 12, 2014, Zürich, Switzerland

[3] Google, Android Studio https://developer.android.com/studio/index.html

[4] S. Mattoccia, M. Poggi, “A passive RGBD sensor for accurate and real-time depth sensing self-contained into an FPGA”, 9th ACM/SIGBED International Conference on Distributed Smart Cameras (ICDSC 2015), September 8-11, 2015, Seville, Spain

[5] Hardkernel. Odroid U3. http://www.hardkernel.com/main/main.php.

[6] WMWARE http://www.vmware.com/it.html

[7] Pierpaolo Perrozzi, “Progetto di un dispositivo wireless di feedback tattile per un Sistema di ausilio per non vedenti o ipovedenti”

[8] Luca Ranalli, “Sviluppo di metodologie per l’interazione tra un sistema di ausilio a ipovedenti e dispositive IOS”

[9] Google Developers Console https://console.developers.google.com/apis/library?hl=IT

[10] Google APIs Terms of Service https://developers.google.com/terms/

[11] Google Maps Android API

https://developers.google.com/maps/documentation/android-api/intro

[12] Google Maps Direction API

https://developers.google.com/maps/documentation/directions/intro

[13] Google Places API Web Service

https://developers.google.com/places/web-service/intro

[14] Android Developers, TextToSpeech

https://developer.android.com/reference/android/speech/tts/TextToSpeech.html

[15] Android Developers, SpeechRecognizer

https://developer.android.com/reference/android/speech/SpeechRecognizer.html

[16] Decode Polyline

http://wptrafficanalyzer.in/blog/drawing-driving-route-directions-between-two-locations-using-google-directions-in-google-map-android-api-v2/