POLITECNICO DI TORINO

Corso di Laurea
in Ingegneria Informatica

Tesi di Laurea Magistrale

Misura di parametri di dinamica del veicolo mediante smartphone

Relatori
prof. Antonio Servetti
Corelatori
prof. Enrico Galvagno
prof. Stefano Mauro
prof. Paolo Stefano Pastorelli

Laureando
Michelangelo Moncada

Anno Accademico 2020-2021

Alla mia famiglia e ai miei amici più cari.

Sommario

Lo studio della dinamica del veicolo si basa sull'analisi di modelli matematici che descrivono il comportamento dinamico di un autoveicolo in termini prestazionali, di stabilità e di sicurezza. Nei sistemi di controllo di stabilità del veicolo, una grandezza che entra in gioco è l'angolo di assetto beta: un parametro che descrive la stabilità del veicolo durante una curva o in situazioni di emergenza. Per ottenere un calcolo diretto del beta, solitamente, è necessario utilizzare strumenti esterni, anche molto costosi, come la reta CAN o la piattaforma inerziale IMU. L'utilizzo di uno smartphone e dei suoi sensori, permette di ridurre i costi a discapito della precisione nel calcolo dei parametri. Per ottenere una stima dell'angolo beta è necessario calcolare alcuni parametri di dinamica del veicolo ovvero l'angolo di sterzo, velocità e accelerazione longitudinali e accelerazione laterale. Lo scopo del lavoro svolto è quello di migliorare un'applicazione smartphone, già esistente, per la stima di tali parametri con particolare attenzione all'angolo volante. A tal scopo, sono stati effettuati dei miglioramenti dell'app, modificando e aggiornando l'intera struttura e i moduli che la compongono. E' stata effettuata un'analisi del flusso e dell'elaborazione dei dati adattandone la struttura per la memorizzazione interna. Si è prestata particolare attenzione al miglioramento della rilevazione e accuratezza dell'angolo volante. E' stato quindi possibile eliminare alcune delle problematiche note, nella versione precedente dell'app, come la copertura accidentale dei markers, il basso frame-rate di acquisizione dell'angolo volante e soprattutto il de-allineamento dei dati acquisiti.

Indice

Elenco delle figure

Parte I

Prima Parte

Capitolo 1

Introduzione

1.1 Principi generali

L'obiettivo del lavoro svolto è quello di studiare e migliorare la stima dell'angolo volante θ e la sua sincronizzazione con i dati acquisiti dagli altri sensori istante per istante. La correttezza dell'angolo θ, infatti, condiziona la precisione dell'angolo di assetto β, il quale rappresenta l'angolo fra direzione longitudinale del moto e la direzione della velocità V del veicolo. L'angolo di assetto β è importante in quanto fornisce informazioni sulla stabilità del veicolo nel momento in cui un veicolo sterza, o in situazioni di emergenza, deve essere corretta la traiettoria dello stesso. La misura diretta di queste grandezze richiede l'uso di dispositivi costosi e solitamente non disponibili nelle vetture di utilizzo standard. Per questo motivo si vuole ottenere una stima tramite l'utilizzo dei sensori presenti in ogni smartphone:

- Fotocamera
- Accelerometro
- Giroscopio
- GPS

L'accelerometro ed il giroscopio permettono di calcolare le componenti di accelerazione e velocità angolare mentre il gps determina una stima della traiettoria del veicolo e la sua velocità istantanea.
La fotocamera permette di ottenere una stima dell'angolo di sterzo, puntando dei markers applicati allo sterzo ed elaborandone l'immagine affinché vengano riconosciuti. Per migliorare quest'ultimo aspetto, il modulo fotocamera e gli algoritmi utilizzati saranno saranno ricostruiti e migliorati.

Capitolo 2

Problema dell'angolo volante

2.1 Descrizione del problema

L'acquisizione dell'angolo volante θ avviene a runtime tramite l'utilizzo di OpenCv, una libreria grafica di cui si parlerà successivamente. L'algoritmo precedentemente sviluppato ed i moduli che lo sfruttano sono stati rivisti e migliorati poiché, durante le fasi di test, si è notato uno sfasamento variabile (2.1) dovuto alla mancanza di un meccanismo che garantisca un clock di sistema condiviso per tutti i sensori dello smartphone. Questa mancanza viene accentuata da tre problemi principali:

- Copertura accidentale dei Markers durante i test
- Framerate in un range non costante e troppo basso
- De-allineamento dei dati durante la cattura

Queste problematiche portano ad uno sfasamento che si accentua nel tempo determinando un errore più o meno variabile.

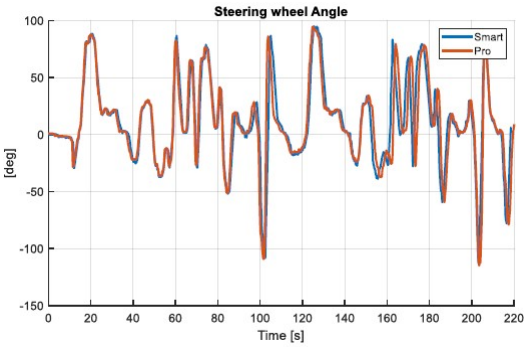


Figura 2.1: Sfasamento variabile individuato durante un test track.

Parte II

Seconda Parte

Capitolo 3

Applicazione Android: Vehicle-App

3.1 Struttura dell'applicazione

3.1.1 Android studio

Di seguito è mostrata la struttura interna dell'applicazione gestita da Android Studio. Come si può facilmente intuire la struttura rappresenta un vero e proprio albero, in cui vi è una root da cui tutto parte e da lì si diramano le varie componenti.

- **app Folder:** rappresenta la cartella root del progetto: sono esclusi solo i file relativi a librerie esterne e moduli aggiuntivi.
- **manifests Folder:** contiene il file manifest che descrive permessi e proprietà dell'app.
- **java Folder:** al suo interno troviamo tutte le classi java del progetto, inclusi i test da effettuare sulle classi. E' possibile creare delle sottocartelle per comodità, ad esempio per raggruppare classi dello stesso tipo.
- **cpp Folder:** questa è una cartella che viene creata dall'utente (nel nostro caso) per poter inserire classi in linguaggio nativo. Nel nostro caso contiene solo i file relativi ad OpenCv.
- **jniLibs Folder:** anche questa è appositamente creata dall'utente per poter incorporare librerie in linguaggio nativo esterne(nel nostro caso Open-Cv). All'interno sono presenti i file binari .a per ogni architettura supportata.
- **res Folder:** questa è la libreria delle resources, al suo interno contiene tutte le risorse che andremo ad utilizzare nel codice: animazioni, i file di layout in .xml, i file di menu in .xml, i valori delle stringhe definite e tutte le immagini (drawable) utilizzate, incluse le icone.

- **openCv Library:** rappresenta il modulo esterno importato nel progetto Android direttamente dall'IDE, si viene a creare una cartella a parte contenente pressoché la stessa struttura di un'applicazione vera e propria, contiene infatti un file manifest e una cartella java, contenente le chiamate alle classi native della libreria come spiegato nel cap. 4.

3.2 Analisi delle classi

In questa sezione si andranno ad analizzare le classi di supporto, i servizi e le strutture dati che compongono l'applicazione. In particolare troveremo 4 tipi di classi.

- **Base Class:** raggruppa tutte le classi che definiscono le strutture dati utilizzate per memorizzare tutte le informazioni ottenute dai sensori. Ogni classe farà da contenitore dei dati estrapolati da ogni specifico sensore.
- **Helper Class:** rappresenta le classi utili allo svolgimento di routine secondarie e predefinite come l'utilizzo di animazioni. Saranno presenti anche delle classi usate nella versione precedente per l'accesso ad un database che permette la gestione di file, di testo e CSV, utili per un'analisi esterna dei dati: le seguenti classi sono state aggiornate in *Utility Class*.
- **Utility Class:** contiene le classi per l'interfacciamento con un database interno che sfrutta **SQLlite**, una libreria che ha il compito di strutturare il database e garantirne la scrittura o la lettura. Fornisce inoltre la classe *FitHelper* per la memorizzazione dei dati di configurazione della prova ed il salvataggio in file txt.
- **Service Class:** contiene il servizio per poter utilizzare la camera in background e tutte le sue versioni precedenti.

Queste classi verranno utilizzate dalle Activity e Fragment Class che rappresentano la vera interfaccia con l'utente.

3.2.1 Classi Base

Come detto precedentemente, le classi base vengono utilizzate per contenere i dati ottenuti o per definire strutture dati statiche. Si descriveranno i tipi di classi accennando le modifiche, trattate al cap. 3.3, che sono state introdotte nella nuova versione nell'app.

Entries Classes:
Rappresentano le strutture dati per la memorizzazione dei valori acquisiti dai sensori dello smartphone e per la successiva estrapolazione. In tal modo si potranno elaborare tutte le informazioni necessarie all'algoritmo di stima per il calcolo del β. Come vedremo, rispetto alla versione precedente, tutte le entries classes sono state modificate con aggiunta di un Timestamp. L'inserimento è stato necessario per contribuire alla soluzione del problema di de-allineamento dei dati (5.4).

Vediamo la descrizione delle classi:

- **FrameEntry Class:**

```
public class FrameEntry {
    private Mat mat;
    private Long tmsp;

    public FrameEntry(Mat mat, Long tmsp) {
        this.mat = mat;
        this.tmsp = tmsp;
    }

    ...

}
```

  Questa classe viene utilizzata per contenere il Frame e il timestamp, associato alla cattura, durante l'utilizzo della fotocamera. Durante lo svolgimento di una prova viene utilizzata una lista, o meglio una **blockingQueue**, di questi oggetti che vengono quindi immagazzinati da un produttore. Un consumatore, contemporaneamente, ha il compito di estrarre un elemento alla volta ed eseguire tutti i calcoli e le manipolazioni necessarie per ottenere l'angolo volante θ associato al timestamp. Il meccanismo produttore-consumatore, utilizzato per l'acquisizione dei frame in real-time, verrà spiegato meglio nel cap. **3.2.3**.

- **StEntry Class:**

```
public class StEntry implements Parcelable {

    private double theta;
    private Long tsmp;

    public StEntry(double theta, Long tsmp) {
    this.theta = theta;
    this.tsmp = tsmp;
    }

    ...

}
```

  Permette di memorizzare il valore θ dell'angolo volante e il suo timestamp di acquisizione. La classe implementa metodi dell'interfaccia *Parcelable*[1] di Android.

- **AccelEntry Class:**

[1]Questo tipo di interfaccia è utile nel caso in cui è necessario scambiare dei dati non primitivi tra Activity o Service. La classe quindi rappresenta una struttura dati creata ad-hoc che subisce una serializzazione quando deve essere passata tramite Intent.

```
public class AccelEntry implements Parcelable {

    private double x;
    private double y;
    private double z;
    private Long tsmp;

    public AccelEntry(double x, double y, double z, Long tsmp) {
        this.x = x;
        this.y = y;
        this.z = z;
        this.tsmp = tsmp;
    }

    ...

}
```

Con questa classe è possibile memorizzare i valori delle componenti x, y, z delle accelerazioni provenienti dall'accelerometro.

- **GpsEntry Class:**

```
public class GpsEntry implements Parcelable {

    private double speed;
    private double lat;
    private double lon;
    private Long tsmp;

    public GpsEntry(double speed, double lat, double lon, Long tsmp) {
        this.speed = speed;
        this.lat = lat;
        this.lon = lon;
        this.tsmp = tsmp;
    }

    ...

}
```

Questa classe presenta le stesse funzioni di AccelEntry ma permette la memorizzazione dei valori restituiti dal sensore GPS ovvero valocità, latitudine, longitudine e timestamp di acquisizione.

- **GyroEntry Class:**

```
public class GyroEntry implements Parcelable {

    private double x;
    private double y;
    private double z;
    private Long tsmp;
```

```
    public GyroEntry(double x, double y, double z, Long tsmp) {
        this.x = x;
        this.y = y;
        this.z = z;
        this.tsmp = tsmp;
    }

    ...

}
```

Anche questa classe permette di memorizzare le componenti x, y, z e timestamp di acquisizione del sensore giroscopio.

- **CarParameters Class:**

```
public class CarParameters {

    double C1 , //[N/rad] rigidezza deriva pneumatici anteriori
           C2,      //[N/rad] rigidezza deriva pneumatici posteriori
           a1,       //[m] semipasso ant
           a2,       //[m] semipasso post
           tau,     //rapporto sterzo anteriore
           J,    //[kg*m^2] mom di inerzia cassa
           m;       //[kG] massa veicolo
    String param;   //univoco per ogni segmento di veicolo
    int img;        //Drawable associato al segmento

    ...

}
```

3.2.2 Classi Helper

AnimationHelper Class: Questa classe contiene tutti i metodi per gestire le animazioni all'interno dell'applicazione. Questi metodi si concentrano in particolare su animazioni che riguardano i *Drawable*[2] della *View*.

Una parte dei metodi della libreria non è stata più utilizzata per motivi di implementazione e di inserimento di nuovi moduli nell'app.

[2]Un Drawable è un'astrazione generale di "Qualcosa che può essere disegnata". Spesso un oggetto Drawable è trattato come una risorsa utilizzata per disegnare qualcosa sullo schermo. Sono fornite quindi delle API utilizzate per interagire con le risorse grafiche sottostanti. A differenza di una View, un Drawable non può reagire ad eventi e quindi interagire con l'utente.

3.2.3 Classi Service

Questa sezione contiene le classi *Services* ed in particolare sia quelle utilizzate nell'app che quelle sostituite, durante lo svolgimento della tesi, al quale sono state applicate delle migliorie. I *Services* vengono utilizzati in background per svolgere operazioni durature nel tempo e che non hanno bisogno di interazioni con l'utente. Nell'applicazione è stato sviluppato un service che sfrutta le API camera2 per l'acquisizione dei frame catturati dalla fotocamera. Parleremo quindi della sua implementazione.

Camera2Service Class:

```
public class Camera2Service extends Service {

//  private HardwareCamera mCamera;
    private LinkedBlockingQueue<FrameEntry> mFrameEntries = new LinkedBlockingQueue<>();
    private FrameEntry tmpFrameEntry;

    ...
```

Il service *Camera2Service* utilizza una **LinkedBlockingQueue** [1], ovvero una coda bloccante utilizzata per immagazzinare ogni frame ed il suo rispettivo timestamp in un FrameEntry (3.2.1). Questo per poter attuare il meccanismo produttore-consumatore del quale parleremo in seguito. E' possibile notare l'eliminazione dell'attributo **mCamera** utilizzato dalla prima versione delle fotocamera API per collegare il service al layer hardware sottostante e eseguire la cattura di un frame; questo perché, utilizzando camera2, il layer sottostante viene controllato automaticamente esonerando il programmatore da un controllo e link manuali.

Vediamo, adesso, come è stato effettuato l'Override dei metodi del service modificati per implementare le nuove migliorie:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Log.i(TAG, "onStartCommand flags " + flags + " startId " + startId);

    if (COUNTER) {
        frameCounter = 0;
        lastNanoTime = System.nanoTime();
    }

    markers = (HashMap<String, Double>) intent.getSerializableExtra("markers");
//  zoomLevel = intent.getIntExtra("zoom", 0);
    initDelta = intent.getDoubleExtra("currentValue", 0);
    values = new DescriptiveStatistics(5);
//    Handler h = new Handler();
//    h.postDelayed(new Runnable() {
//        @Override
```

```
//          public void run() {
//              int camera = Camera.CameraInfo.CAMERA_FACING_BACK;
//              //int camera =
//              if (PreferenceManager
//                      .getDefaultSharedPreferences(getApplicationContext())
//                      .getBoolean(SettingsFragment.KEY_PREF_CAMERA, false))
//                  camera = Camera.CameraInfo.CAMERA_FACING_FRONT;
//              mCameraId = camera;
//              mCamera = new HardwareCamera(CameraService.this, CameraService.this, mCameraId);
//              mCamera.setZoomLevel(zoomLevel);
//              mCamera.connect();
//              (TAG, "Service started!");
//              mThread = new Thread(new BackgroundCamera());
//          }
//      }, 500);
        mThread = new Thread(new BackgroundCamera());
        mRgb = new Mat();
        cameraMatrix = new Mat(3, 3, CvType.CV_32F);
        coeffs  = new Mat(5, 1, CvType.CV_32F);

        /**Add init for ChArUco board**/

        dictionary  = Aruco.getPredefinedDictionary(Aruco.DICT_6X6_250);
        board = CharucoBoard.create(5, 7, (float) 0.04,  (float) 0.02,  dictionary);
        params = DetectorParameters.create();

        /*****************************/

        cameraMatrix.put(0, 0, 1024.3446);
        cameraMatrix.put(0, 1, 0);
        cameraMatrix.put(0, 2, 640.0);
        cameraMatrix.put(1, 0, 0);
        cameraMatrix.put(1, 1, 1024.3446);
        cameraMatrix.put(1, 2, 360);
        cameraMatrix.put(2, 0, 0);
        cameraMatrix.put(2, 1, 0);
        cameraMatrix.put(2, 2, 1);

        coeffs.put(0,  0,  0.08816299);
        coeffs.put(1,  0,  -0.21675685);
        coeffs.put(2,  0,  0);
        coeffs.put(3,  0,  0);
        coeffs.put(4,  0,  0);

        readyCamera();

        return super.onStartCommand(intent, flags, startId);
    }
```

Nella prima parte di codice: vengono inizializzati i parametri per ottenere il conteggio degli fps e si ottiene il valore iniziale dei markers ovvero l'ultima posizione utile dallo start della prova. E' possibile notare (nella parte commentata) come non sia più necessario il collegamento manuale al layer sottostante per il controllo dell'hardware della fotocamera.

Inoltre, adesso non è più necessario creare un thread secondario con ritardo di 500ms per evitare sovrapposizioni. Una volta inizializzati tutti i parametri per la detection della *CharucoBoard* (4.3) ed il calcolo dell'angolo, viene lanciato il metodo *readyCamera()*:

```
public void readyCamera() {
    CameraManager manager = (CameraManager) getSystemService(CAMERA_SERVICE);
    try {
        String pickedCamera = getCamera(manager);

        manager.openCamera(pickedCamera, cameraStateCallback, null);

        imageReader = ImageReader.newInstance(320, 240, ImageFormat.YUV_420_888, 2);
        imageReader.setOnImageAvailableListener(onImageAvailableListener, null);

        Log.i(TAG, "imageReader created");
    } catch (CameraAccessException e){
        Log.e(TAG, e.getMessage());
    }
}
```

In questo metodo viene resa operativa la fotocamera e viene stanziato un *Imagereader*[3] nel quale viene passato il listener **onImageAvailableListener**:

```
protected ImageReader.OnImageAvailableListener onImageAvailableListener =
                                = new ImageReader.OnImageAvailableListener() {
    @Override
    public void onImageAvailable(ImageReader reader) {
        Log.i(TAG, "onImageAvailable");
        Image image = reader.acquireLatestImage();
        if (image != null) {

            if (COUNTER) {
                frameCounter++;
                if (frameCounter >= 30) {
                    final int fps = (int) (frameCounter * 1e9 /
                                                (System.nanoTime() - lastNanoTime));
                    Log.i(TAG, "drawFrame() FPS: "+fps);
                    frameCounter = 0;
                    lastNanoTime = System.nanoTime();
                }
            }

            // sanity checks - 3 planes
            Image.Plane[] planes = image.getPlanes();
            assert (planes.length == 3);
            assert (image.getFormat() == mPreviewFormat);
```

[3]Permette l'accesso diretto a un'immagine ottenuta dalla Surface del display

```
                processImage(image);
                image.close();
            }
        }
};
```

Il listener in questione è molto importante poiché, dopo aver effettuato un conteggio degli fps, fornisce un frame, ovvero un'immagine che viene acquisita e che, dopo un sanity check, viene passata al metodo **processImage**:

```
private void processImage(Image image){
        //Process image data
        Long timestamp = System.currentTimeMillis();
        try {
            Mat m = new Mat();
            JavaCamera2ToFrame tempFrame = new JavaCamera2ToFrame(image);
            Imgproc.cvtColor(tempFrame.rgba(), m, Imgproc.COLOR_BGRA2BGR);
            mFrameEntries.put(new FrameEntry(m, timestamp));
            tempFrame.release();


        } catch (Exception e) {
            e.printStackTrace();
        }
        if (!isRunning) {
            mThread.start();
        }
}
```

In questo metodo l'immagine viene trasformata in un formato comprensibile alle ChArUco API ed inserita, insieme al timestamp corrispondete, all'interno del costrutto *mFrameEntries*:

```
        mFrameEntries.put(new FrameEntry(m, timestamp));
```

Il costrutto viene stanziato come **LinkedBlockingQueue<FrameEntry>** ovvero un costrutto FIFO usato nel caso di condivisione dati tra 2 thread diversi, identificati come:

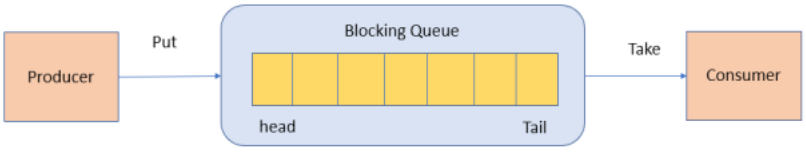


Figura 3.1: Costrutto BlockingQueue.

- **Producer Thread**: nel nostro caso rappresentato dal listener *onImageAvailableListener* che raccoglie il frame, lo elabora e lo inserisce nella coda.

- **Consumer Thread**: invocato tramite il metodo *mThread.start()* preleva il dato dalla coda e, come vedremo successivamente, lo elabora. Il thread in questione è rappresentato dalla classe BackgroudCamera che estende vari metodi *Runnable*[4].

```
private class BackgroundCamera implements Runnable {

    @Override
    public void run() {
        isRunning = true;
        Mat mFrame;
        double prevTheta = Double.MAX_VALUE;
        double theta = 0;
        SynchronizedDescriptiveStatistics mean = new SynchronizedDescriptiveStatistics(WINDOW_SIZE);
        while (isRunning) {
            try {
                if (mFrameEntries.isEmpty()) continue;
                tmpFrameEntry = mFrameEntries.take();
                mRgb = tmpFrameEntry.getMat();
                if (mRgb == null) continue;
                if (resetState) {
                    resetState = false;
                    continue;
                }

            ...

        final Intent i = new Intent("angle_update");
        i.putExtra("theta", theta);
        i.putExtra("thetaTime", tmpFrameEntry.getTmsp());
        sendBroadcast(i);
}
```

Il seguente consumer thread effettua un check all'interno della coda condivisa tramite il metodo *mFrameEntries.take();* e se è presente un oggetto *FrameEntry* lo estrae eliminandolo dalla coda. Il procedimento, naturamente, viene eseguito in modalità *Thread-Safe*[5]. Dal *FrameEntry* viene estratto il frame e salvato nella variabile *mRgb* per poi essere elaborato dai metodi della libreria *OpenCv* (4). Se invece, in coda non c'è nessun elemento, il thread rimmarà in attesa evitando il *polling*[6]. Il thread quindi va in *sleep* e viene risvegliato solo quando è disponibile un nuovo dato nella coda.
Una volta identificato e calcolato l'angolo θ rispettivo al frame, il valore viene inserito

[4]E' un'interfaccia che viene implementata in quelle classi le quali istanze vogliono essere eseguite da un thread.

[5]Meccanismo utilizzato per sincronizzare gli accessi di uno o più thread ad una sezione di codice. Avremo quindi un accesso in mutua esclusione.

[6]Quando viene effettuato un continuo controllo ad ogni ciclo. Questo meccanismo è sconsigliato quando si vogliono ottenere alte prestazioni poiché vengono consumati cicli inutili di CPU dal processore.

in un *Intent*[7] ed inviato in *broadcast* per poi essere catturato da un *Broadcast Receiver* (3.7.5).

3.2.4 Classi Utilities

In questa sezione troviamo varie tipologie di classi utilizzate per la gestione del database, il salvataggio di alcuni parametri necessari per la fase di inizializzazione di una prova, la gestione dei file Raw e il calcolo dell'angolo di assetto β.

DatabaseUtilities Class:
La seguente classe permette la gestione di tutte le operazioni in lettura e scrittura sul database interno. Stanziando l'oggetto **DatabaseUtilities** viene creata la struttura del database con le *tables* contenenti i dati per ogni sensore o parametro elaborato. Per garantire il corretto salvataggio dei dati, la classe è stata modificata sostituendo le funzioni deprecate con quelle attualmente utilizzate con *SQLlite*. Anche la struttura della *table* viene modificata: vengono aggiunti nuovi attributi per l'inserimento di nuovi dati provvisti di *timestamp* associato.

StorageUtility Class:
Questa classe viene utilizzata per la lettura e la scrittura dei valori di configurazione iniziale della prova. Le operazioni avvengono tramite 2 funzioni *readFileAccs()* e *writeFileAccs()* utilizzando il costrutto *InputStream* che, come descritto dal nome, permette la lettura di uno stream di dati. La configurazione iniziale, eseguita dopo la Camera2Tracker Activity (3.4.2), è caratterizzata da una prima fase statica e da una seconda di accelerazione gestite tramite la classe **fitHelper**.
Mentre nella precedente versione dell'app il salvataggio su file era stato disabilitato, adesso, alla termine di ogni prova, vengono creati una serie di file testuali sia per i dati raw di ogni sensore che per quelli elaborati.

3.2.5 Classi Room

La libreria Room [2] fornisce un layer d'astrazione di SQLite semplificando l'accesso e la gestione del database. In particolare, si ottengono i seguenti benefici:

- Verifica delle queries SQL nel tempo di compilazione.
- Utilizzo di annotazioni java per minimizzare la scrittura di codice ripetitivo (boilerplate).
- Gestione semplificata dei path del database.

[7]Struttura dati che contiene una descrizione astratta di un'operazione che deve essere eseguita. Ad esempio può essere usato per lanciare una nuova *Activity* oppure può essere inviato in broadcast come BroadcastIntent per passare delle strutture dati tra componenti di un'applicazione. Il componente ricevente i dati, utilizza un *Broadcast Receiver*

All'interno del progetto, nei nuovi folder **repository** e **data**, vengono implementati i 3 principali componenti Room. Come descritto dal nome, il folder *repository* contiene una classe repository che fa da wrapper al primo componente Room ovvero **Database class**: contiene il database e ne gestisce l'accesso principale. Il folder *data* contiene le altre 2 componenti Room:

- **Data entities:** rappresentano le tabelle del database creato.
- **Data access object (Dao):** fornisce i metodi per eseguire un insert, un update, un delete o una query al database.

La classe database fornisce all'applicazione delle istanze dei **DAO** associati al database. Successivamente, l'applicazione può usare le istanze per eseguire delle operazioni sul database recuperando i dati come oggetti **Data entities** associati.
L'utilizzo delle Room API è stato parziale così da non stravolgere totalmente la struttura dell'app. In futuro, come detto nel capitolo finale, si potrebbe effettuare un porting totale adeguato a tutti i moduli dell'applicazione.

3.3 Modifiche della struttura principale

In questa sezione e nelle successive sezioni viene descritto il ripristino dell'applicazione ormai datata aggiornando o sostituendo librerie di codice e parti dell'applicazione così da renderla più performante ed adeguarla ai pattern odierni di programmazione.

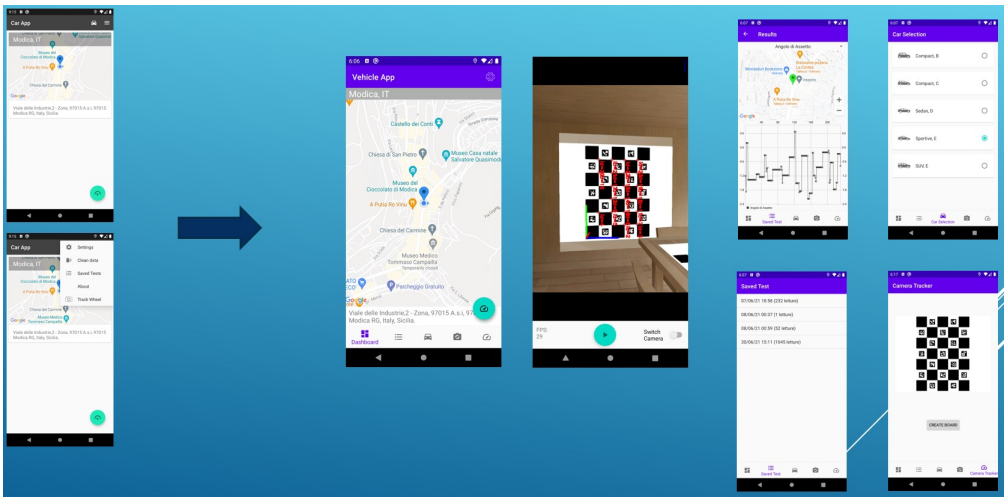


Figura 3.2: Vista generale dell'upgrade UI.

3.3.1 Utilizzo dei fragment

I Fragment [3] permettono una gestione migliore della *user interface* introducendo modularità e riusabilità degli elementi grafici di un'*Activity* dividendo così la UI in blocchi più

piccoli. L'aumento di modularità è dovuto al fatto che ogni activity complessa, può essere divisa in più fragment ottenendo una migliore organizzazione e mantenimento. Inoltre, lo stesso fragment, può essere riutilizzato da più activity o apparire più volte nella stessa.

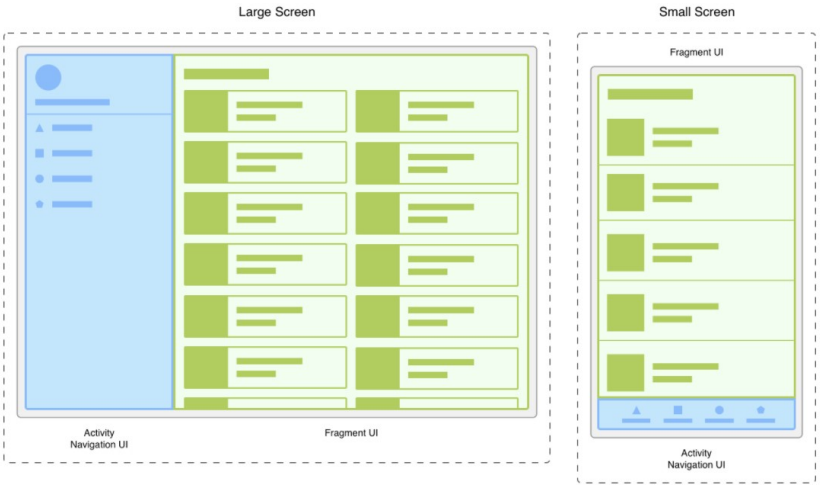


Figura 3.3: Due versioni dello stesso schermo con grandezze differenti.

Come vediamo nella figura 3.2, a sinistra viene visualizzato uno schermo più grande contenente un *Navigation Drawer*[8] controllato dall'activity ed una *Grid List*[9] controllata dal fragment. A destra invece abbiamo una *Bottom Navigation Bar* controllata dall'activity ed una *Linear List*[10] controllata dal fragment. I fragment, quindi, possono incapsulare un determinato numero di componenti in modo riutilizzabile. Un fragment è un oggetto che, concettualmente, può essere considerato una via di mezzo tra un'Activity ed una *View*[11]. Da un lato, come un'Activity, possiede un complesso life cycle gestito da numerose callback e permette l'interazione da l'utente e l'app. Dall'altro, possede una gerarchia di views che possono entrare a far parte dell'*Activity Visual Tree*[12].
La gestione dei fragments, ovvero la loro sostituzione, aggiunta o rimozione dal *Back Stack*[13], è garantita dal FragmentMenager.

[8]Un pannello UI laterale che mostra un menù per la navigazione principale dell'appicazione.

[9]Mostra gli elementi in una griglia bidimensionale di righe e colonne.

[10]Mostra gli elementi disponendoli in una lista.

[11]Classe alla base della creazione dei componenti per la user interface. Una View occupa un'area rettangolare dello schermo ed è responsabile della creazione degli elementi grafici e della gestione di eventi.

[12]Insieme di view dell'activity disposte secondo un ordine gerarchico.

[13]Struttura dati usata per diversi contesti, le cui modalita d'accesso ai dati seguono la madalità LIFO(Last In Fist Out).

Figura 3.4: Due esempi di UI layout con le rispettive relazioni tra i frameents e le loro host activities.

Considerando l'esempio della Figura 3.4 in *Example 1* abbiamo un Host Fragment che fa da host a 2 Child Fragment creando uno split-view screen; nell'*Example 2* invece fa da host ad un solo child fragment costituendo una *swipe view*[14].

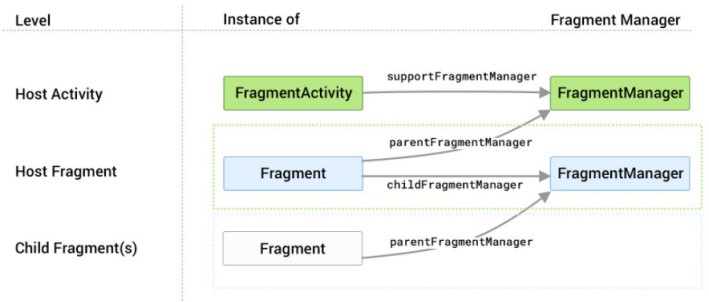


Figura 3.5: Ogni host ha il suo FragmentManager.

Dato questo setup possiamo a un *FragmentManager* come un oggetto associato ad ogni host di un determinato livello che gestisce i suoi fragment figli. Le funzioni utilizzate per il *FragmentManager* associato al livello sono *supportFragmentManager*, *parentFragmentManager* e *childFragmentManager*.
Per effettuare una *FragmentTransaction* transizione da un fragment all'altro, vengono utilizzate le funzioni add() o replace().

```
FragmentManager fragmentManager = getSupportFragmentManager();
fragmentManager.beginTransaction()
    .replace(R.id.fragment_container, TestFragment.class, null)
```

[14]Classe che permette di navigare attraverso schermate adiacenti come le *tabs* con una gesture.

```
    .setReorderingAllowed(true)
    .addToBackStack("name") // name can be null
    .commit();
```

In questo esempio, il FragmentManager sostituisce qualsiasi fragment sia presente nel *fragment_container* con il *TestFragment* e lo aggiunge quindi al Back Stack.
Questi elementi appena descritti sono molto importanti poiché sono sfruttati internamente e quindi alla base delle API della *BottomNavigationView* che descriveremo del sotto-paragrafo seguente.

3.3.2 Navigation component

Il termine *Navigation* si riferisce all'interazione che permette all'utente di navigare attraverso, all'interno o tornare indietro, tra le varie parti del un contenuto di un'applicazione. Il **Navigation component** [4] di *Android Jetpack*[15], permette di implementare varie tipologie di navigazione, che vanno dal click di un semplice bottone a pattern più complessi come la **Bottom app bar** o il **Navigation drawer**[16]. L'obiettivo è quello di assicurare una **user experience** intuitiva, consistente e prevedibile.

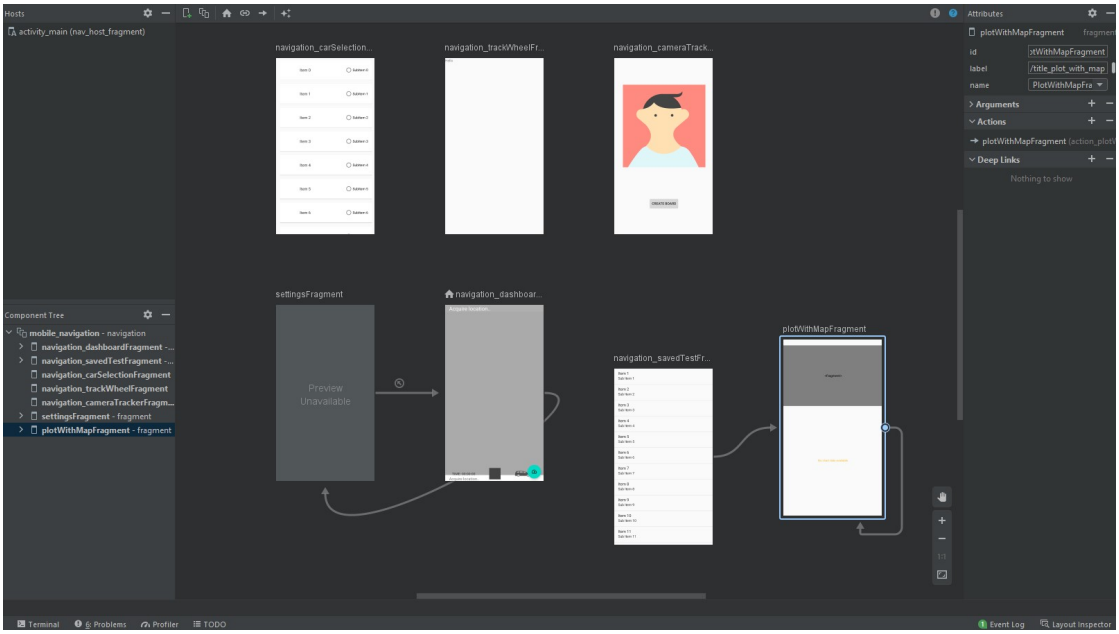


Figura 3.6: Screenshot del Navigator Graph.

[15] Insieme di librerie create per aiutare lo sviluppatore nel seguire le best practise, ridurre il boilerplate code e scrivere codice stabile tra le versioni Android.

[16] Un pannello grafico che mostra lateralmente il menù per la navigazione principale.

- **Bottom Navigation View**

  All'interno dell'applicazione, la navigazione tra i contenuti è gestita da un barra di selezione posta nella parte inferiore dello schermo. La sua visualizzazione e la disposizione degli elementi della barra è coordinata dalla *Bottom Navigation View* [5]. Quest'ultima ha anche il compito di definire la navigazione tra i contenuti sfruttando un **Navigator Graph** rappresentato in Figura 3.6: la disposizione dei contenuti e la struttura della navigazione viene fatta con un editor grafico di *Android Studio*. Un volta implementati tutti i componenti per la navigazione, cliccando su un elemento della bottom bar viene visualizzato il contenuto corrispondente descritto dal Navigator Graph.

3.4 Le classi Activity

In questa sezione vedremo le activity che costituiscono l'applicazione e attenzioneremo le loro caratteristiche principali.

3.4.1 Main Activity

Figura 3.7: Screenshot della Main activity con Dashboard Fragment.

La *Main Activity* rappresenta la parte principale dell'applicazione poiché ha il compito di creare e gestire 80% delle parti che la costituiscono. Si occupa infatti della creazione e della gestione di tutti i **Fragments** (3.5) garantendone la coordinazione tramite la *Bottom Navigation* (3.3.2) e di tutti i processi che questi le delegano, ad esempio, l'attivazione e la gestione dei sensori dello smartphone, l'invio e la ricezione dei dati con altre activity o fragment, l'accesso al database o la memorizzazione delle impostazioni. Permette la realizzazione di un nuovo pattern scalare e stabile dal punto di vista del mantenimento del codice.

3.4.2 Camera2Tracker Activity

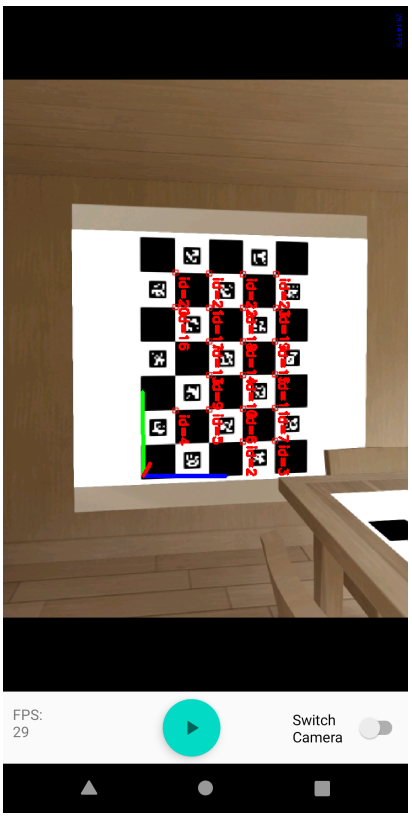


Figura 3.8: Screenshot della Camera Tracker Activity.

La classe presentata costituisce il punto di partenza per lo start di una prova. L'interfaccia semplificata permette di inquadrare i markers e, una volta rilevati, acquisire l'angolo volante di partenza. Come supporto all'utente viene mostrato il framerate, tramite un contatore di fps, per dare un feedback sulla qualità della scena. Poiché le condizioni di luce influiscono molto sulla velocità di cattura dei fotogrammi (5.3) è stato inserito un pulsante per abilitare il flash della fotocamera durante la prova.

3.5 I fragment

In questa sezione verranno descritti i fragment che costituiscono l'applicazione e permettono l'interazione dell'utente con tutte le funzionalità.

3.5.1 Dashboard Fragment

Figura 3.9: Screenshot del Dashboard Fragment.

Rappresenta la schermata principale dell'applicazione: all'interno di questo fragment è contenuta la mappa con la posizione corrente dell'utente [6] e il suo percorso sulla mappa nel caso questo di muova [7]. Tramite il pulsante in basso a destra è possibile accedere alla

CameraTrackerActivity che abilita la fotocamera e nel caso di rilevamento della *charuco board* permette l'inizio della prova. In questa sezione è possibile accedere ai settings per regolare le impostazioni dell'applicazione.

3.5.2 SavedTests Fragment

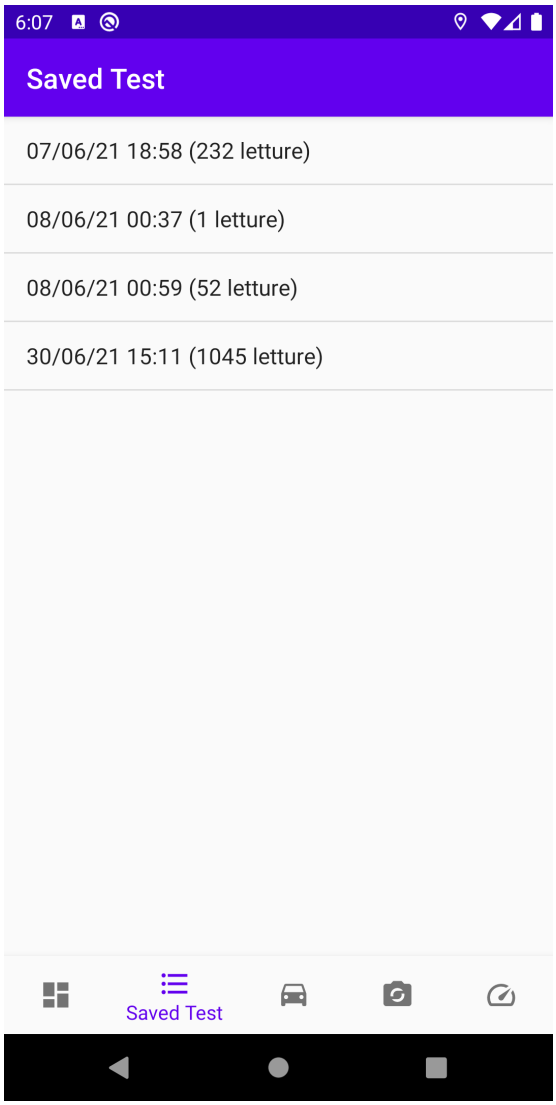


Figura 3.10: Screenshot del Saved Test Fragment.

Vengono raggruppati in una lista tutte le prove ordinate cronologicamente. Permette il passaggio al *Result Fragment* (3.5.3) con i dati della prova selezionata.

3.5.3 Result Fragment

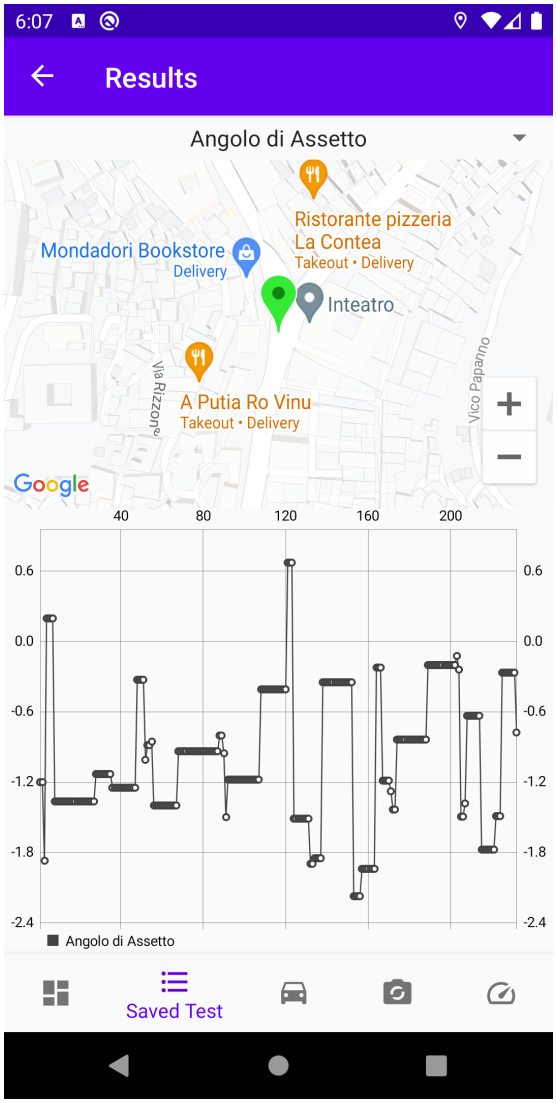


Figura 3.11: Screenshot del Result Test Fragment.

In questa sezione vengono mostrati i dati acquisiti. In particolare, in alto è possibile selezionare i segmenti di prova con una mappa che visualizza e traccia il percorso effettuato durante la prova. In basso vengono plottati i dati del segmento di prova selezionato, permettendo un'analisi dei dati più dettagliata.

3.5.4 Car Selection Fragment

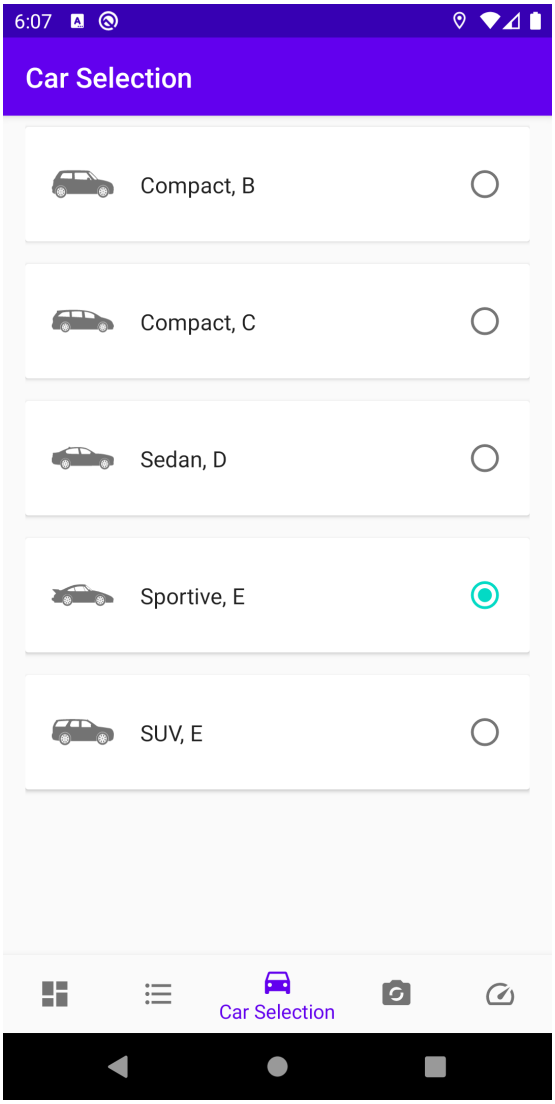


Figura 3.12: Screenshot del Car Selection Fragment.

Fornisce un apposito menù dal quale è possibile selezionare il *segmento* di veicolo utilizzato durante la prova.

3.5.5 Board Generator Fragment

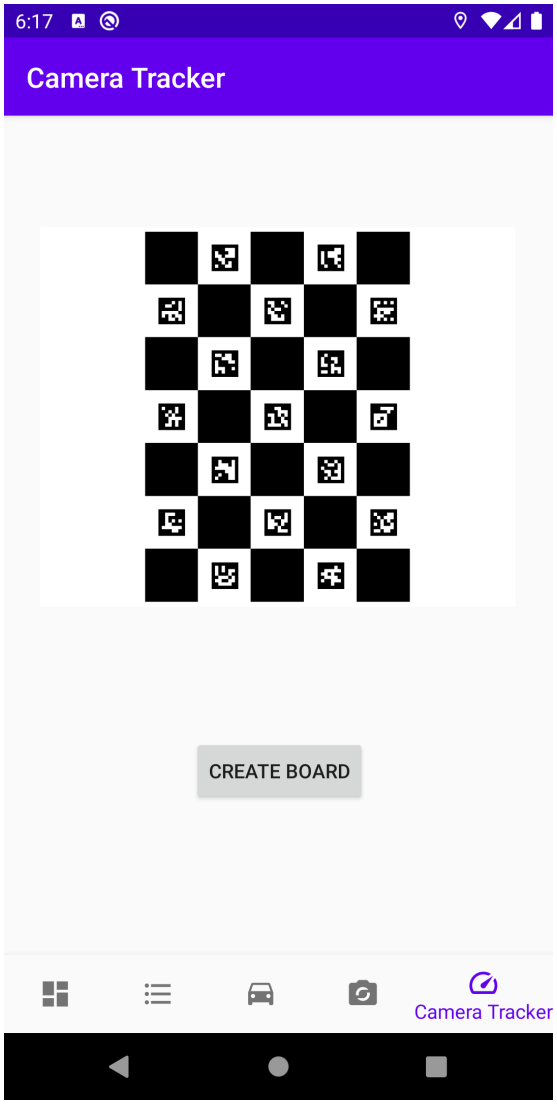


Figura 3.13: Screenshot del Board Generator Fragment.

Permette la creazione ed il salvataggio della **board** (4.3) all'interno del dispositivo. In questo modo è possibile recuperare la board e stamparla per poter eseguire le prove.

3.5.6 Settings Fragment

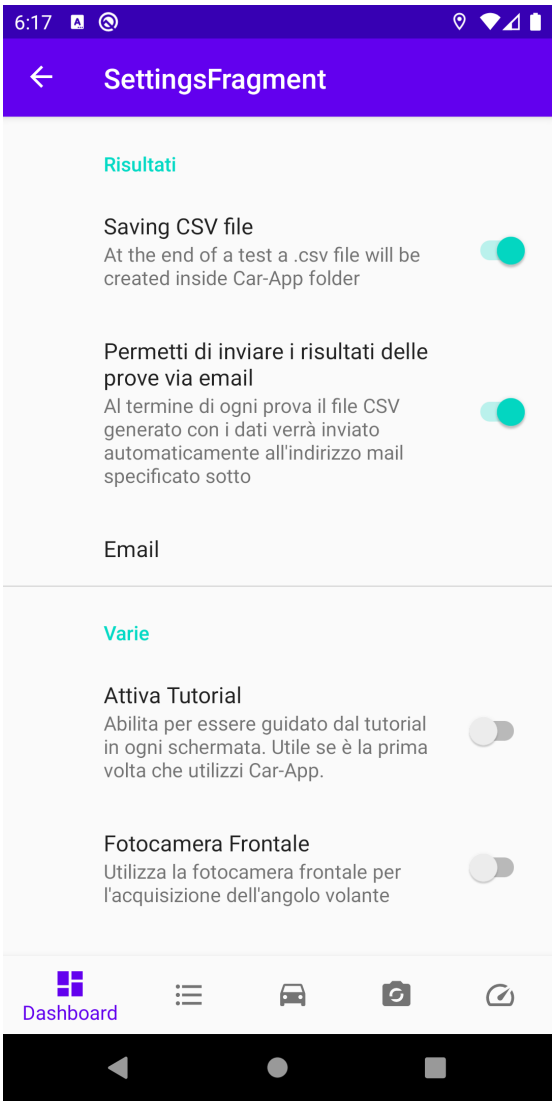


Figura 3.14: Screenshot del Settings Fragment.

In questa sezione sono contenute le impostazioni dell'applicazione: Si può abilitare il **salvataggio** dei dati in formato CSV e testo(.txt). Un campo apposito permette l'inserimento della mail utilizzata per inviare i dati raccolti dalle prove così da poterli manipolare tramite pc. In questo modo si ottiene un trasferimento veloce dei dati, da un dispositivo all'altro, senza dover utilizzare un dispositivo di memoria esterno. Nel caso l'impostazione sia disabilitata, i dati vendono comunque salvati nella memoria interna dello smartphone. Inoltre un'impostazione secondaria permette di abilitare un tutorial per spiegare le funzionalità della schermata principale (*Dashboard Fragment* 3.9).

3.6 Elementi deprecati

3.6.1 HardwareCamera Class

Questa classe permetteva di gestire il layer hardware della fotocamera e poter sfruttare le funzioni principale. E' stata sostituita in quanto la gestione del codice ed il collegamento ai layer superiori era onerosa per il programmatore e non molto stabile. Portava anche delle limitazioni: mancanza di supporto di molte funzioni come regolazione del framerate, pre-frame control e stabile integrazione e collegamento con la libreria OpenCv. Per questi motivi è stata sostituita con le nuove **Camera2 API** (4.4).

3.7 Analisi dell'esecuzione e del flusso dati

Uno degli obiettivi è stato quello di analizzare tutto il **flusso** di scambio e acquisizione dati per trovare alcune criticità e risolverle. Quest'analisi ha permesso di ottenere una vista d'insieme delle informazioni scambiate e di come i vari moduli interagiscano tra loro. Questo punto è stato essenziale per capire come approcciarsi ad ogni singolo modulo e cosa modificare per garantire performance più alte e integrità delle informazioni durante l'invio o la ricezione dei dati. Prima di focalizzarsi sulle varie interazioni, che vedremo nelle successive sottosezioni, si riporta lo **schema complessivo dell'analisi** e la sua descrizione:

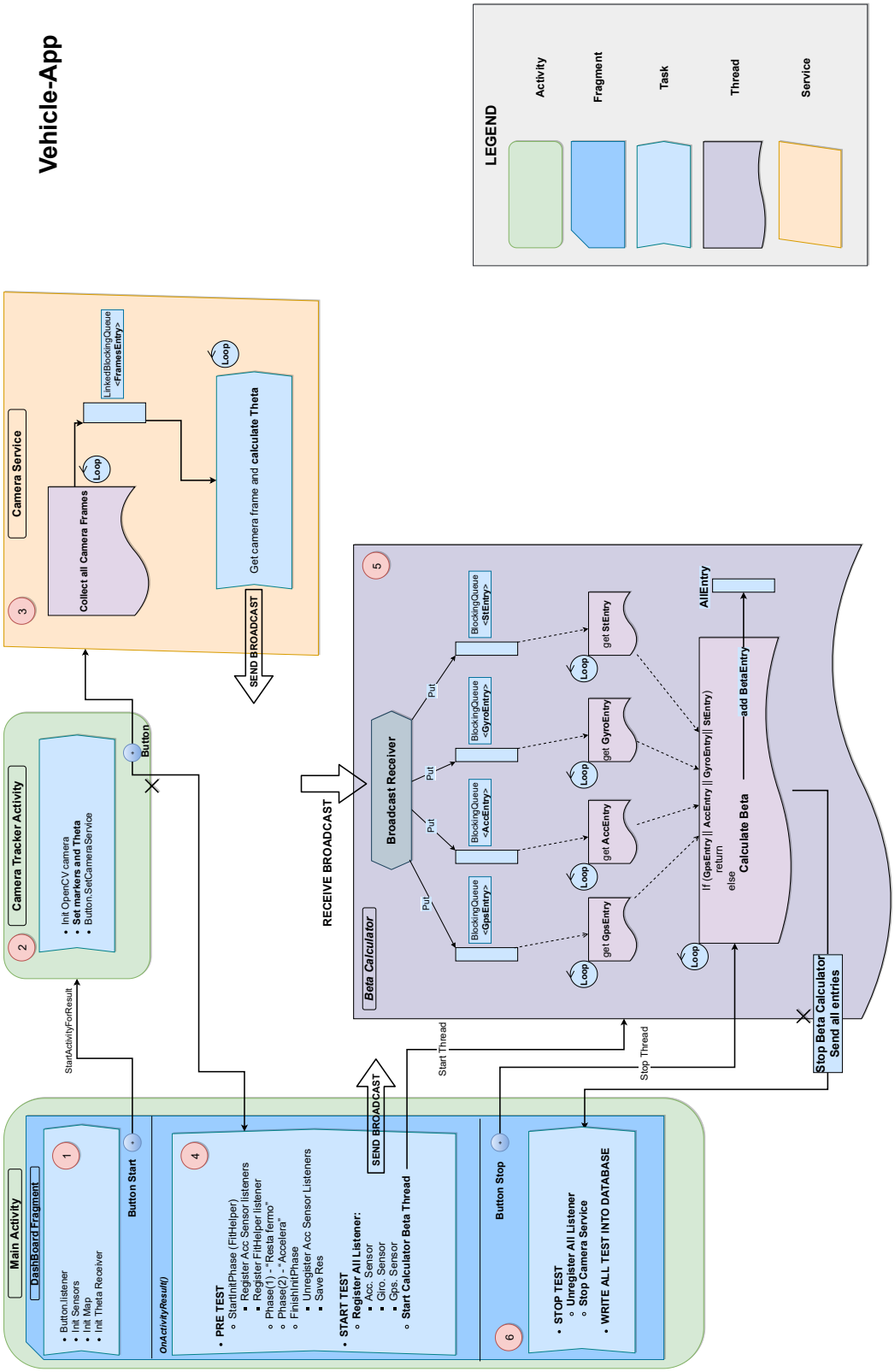


Figura 3.15: Vehicle-App: struttura e moduli.

3.7.1 Parte 1

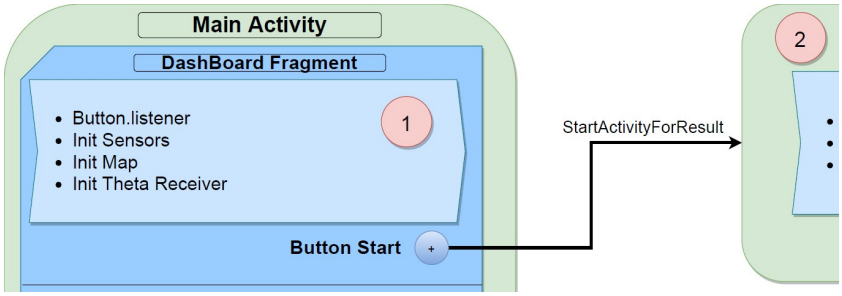


Figura 3.16: Prima parte dell'analisi del flusso dati.

All'avvio dell'applicazione, vengono inizializzati tutti i fragments gli elementi grafici, ad esempio la mappa, e di background come i sensori e le variabili per l'invio e la recezione dei dati. Il *Button* della vista principale (3.9) permette di accedere alla fotocamera (3.8).

3.7.2 Parte 2

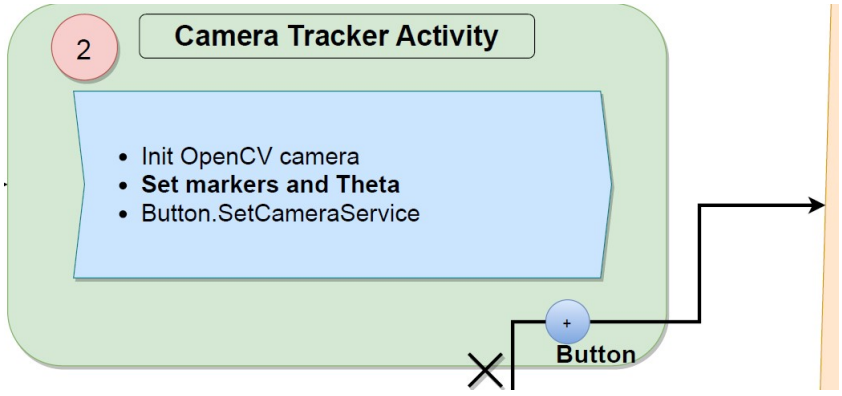


Figura 3.17: Seconda parte dell'analisi del flusso dati.

Una volta avviata la fotocamera vengono avviate le componenti **OpenCv** per l'identificazione dei **markers**. Se una *board* (4.3) viene identificata, viene reso visibile un *button* che avvia 2 processi: inizia l'esecuzione del **Service**, che vedremo nella sezione successiva, e invia il valore iniziale e timestamp dell'angolo θ all'activity principale. Eseguiti i processi, la fotocamera viene chiusa tornando alla sezione principale.

3.7.3 Parte 3

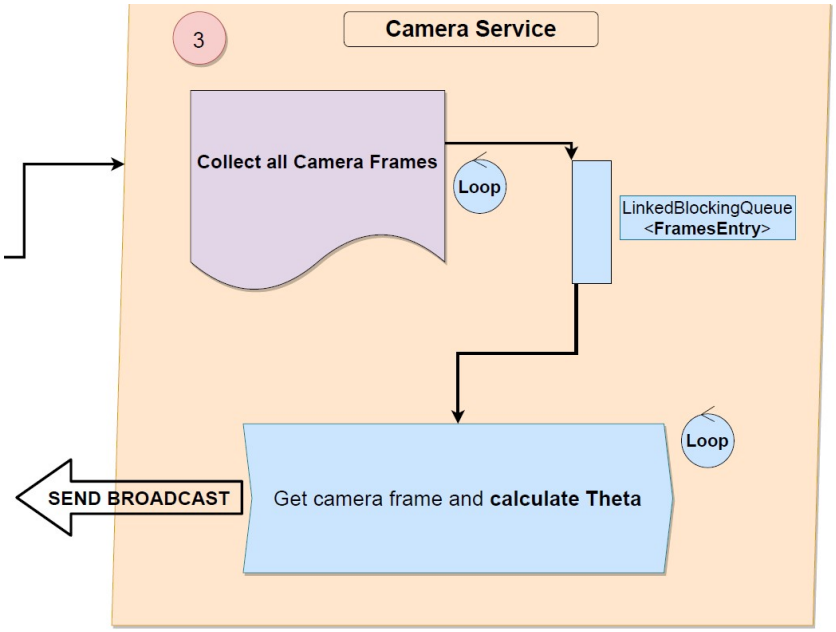


Figura 3.18: Terza parte dell'analisi del flusso dati.

Il Service acquisisce in background i frame uno alla volta tramite un loop e li inserisce in una lista. Successivamente crea un *thread* che, sfruttando il pattern produttore-consumatore (3.1), preleva continuamente i frame dalla lista e ne calcola l'angolo θ inviandolo in **broadcast**.

3.7.4 Parte 4

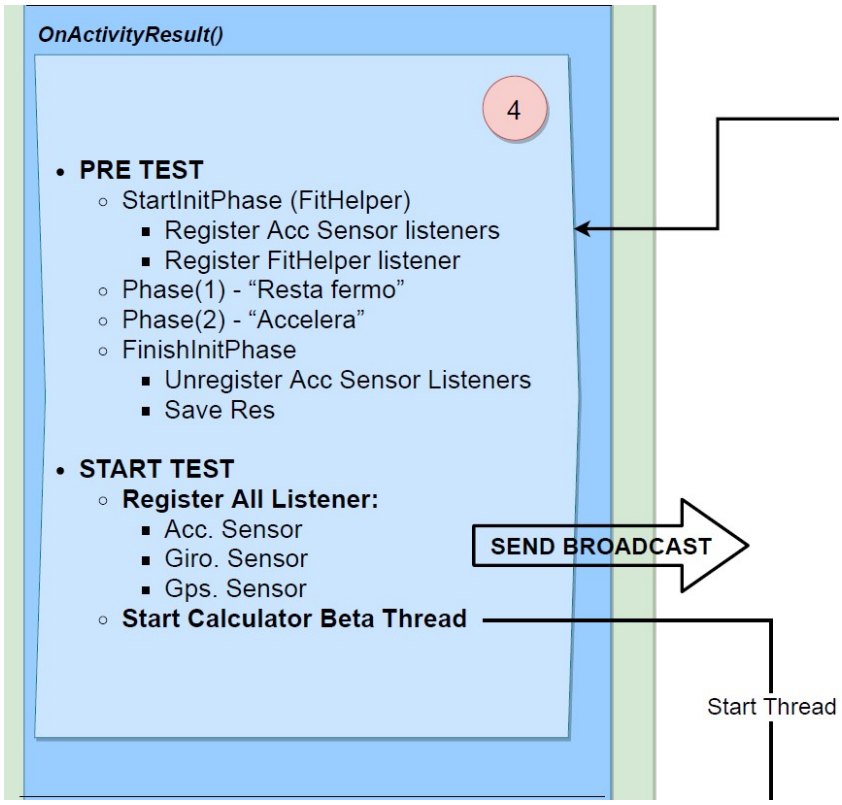


Figura 3.19: Quarta parte dell'analisi del flusso dati.

A questo punto viene avviata una fase pre-test per acquisire i sistemi di riferimento iniziali e salvarli per essere usati durante l'elaborazione dati della parte 5. I listener dei sensori (accelerazione, giroscopio, gps) vengono registrati e parte l'acquisizione dati trasmessi in **broadcast**. Infine si effettua lo start del thread per il calcolo del β.

3.7.5 Parte 5

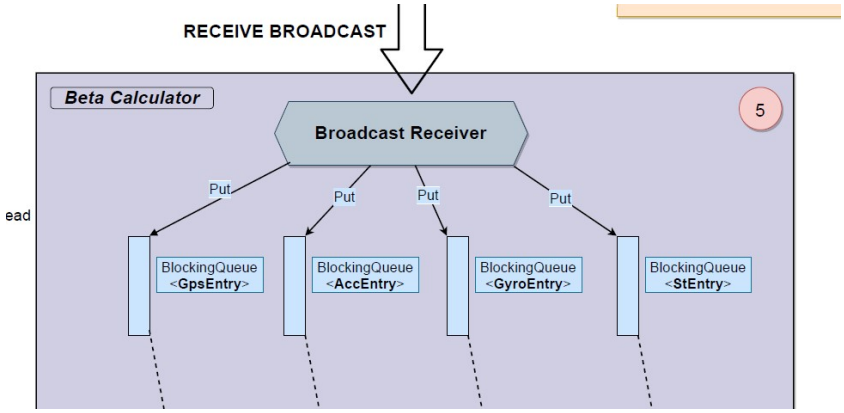


Figura 3.20: Quinta parte dell'analisi del flusso dati: Acquisizione.

Il thread generato è fondamentale per l'elaborazione finale dei dati. Tramite un Broadcast Receiver[17] raccoglie, per tutta la durata del suo processo, i dati provenienti da tutti i sensori e li inserisce nelle rispettive liste.

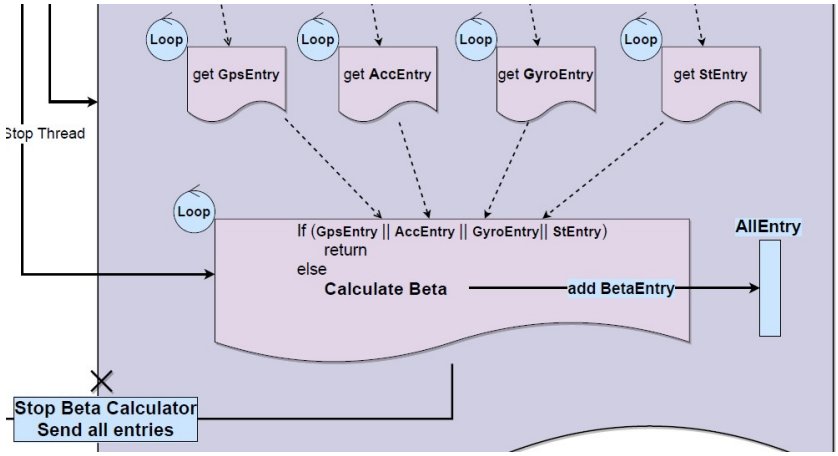


Figura 3.21: Quinta parte dell'analisi del flusso dati: Elaborazione.

Successivamente, quattro thread annidati prelevano in loop dalle rispettive liste. Se uno dei dati non è presente si aspetta; nel caso contrario si procede con il calcolo del β. Terminata l'elaborazione, i dati in input e output risultate vengono salvati in una lista.

[17] Classe base che riceve e gestisce strutture dati inviate in broadcast

3.7.6 Parte 6

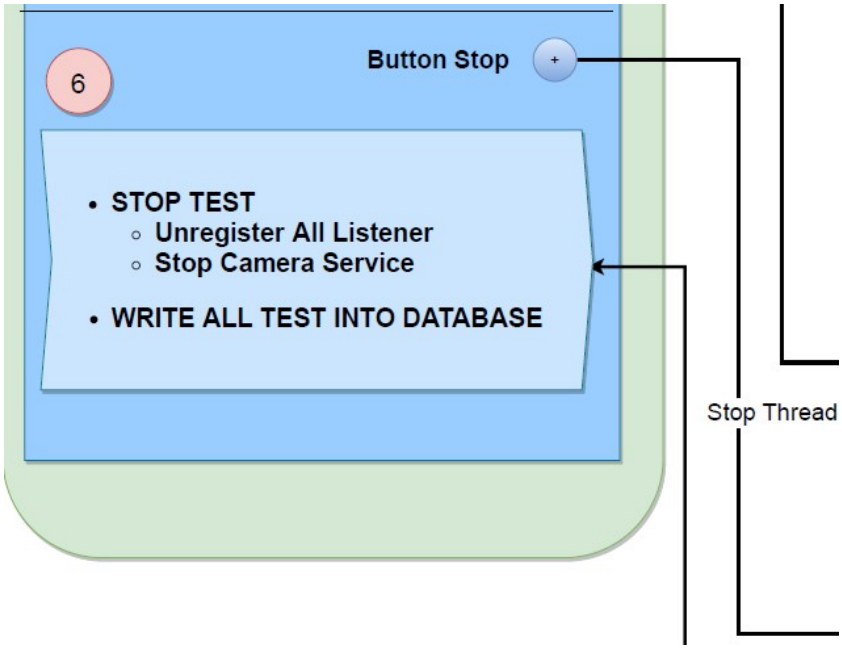


Figura 3.22: Sesta parte dell'analisi del flusso dati.

Non appena l'utente decide di terminare la prova il thread per il calcolo del β ed il Camera Service vengono terminati e tutti i sensori disattivati. Vengono recuperati tutti i dati e salvati in un database. Su richiesta dell'utente potranno essere inviati tramite mail, per poterli elaborare ulteriormente su altri dispositivi.

Capitolo 4

Libreria OpevCV

Figura 4.1: Logo OpenCv

Per l'acquisizione dell'angolo volante, l'utilizzo della libreria OpenCv [8] si è rivelato fondamentale. In questa sezione vedremo quali sono stati gli upgrade per semplificare e migliorare l'acquisizione dei markers e quindi il passaggio dalla libreria **ArUco** [9] a **ChArUco** [10].

4.1 Compilazione

Le nuove versioni di OpenCV non contengono le librerie per il riconoscimento dei markers, pertanto, è stata necessaria una compilazione ad-hoc. Per raggiungere questo obiettivo è stato utilizzato CMake. Una volta scaricati tutti i file sorgente della libreria, in particolare **OpevCV-4.5.1**, è stato possibile eseguire la compilazione [11] integrando **ArUco** e **ChArUco**.

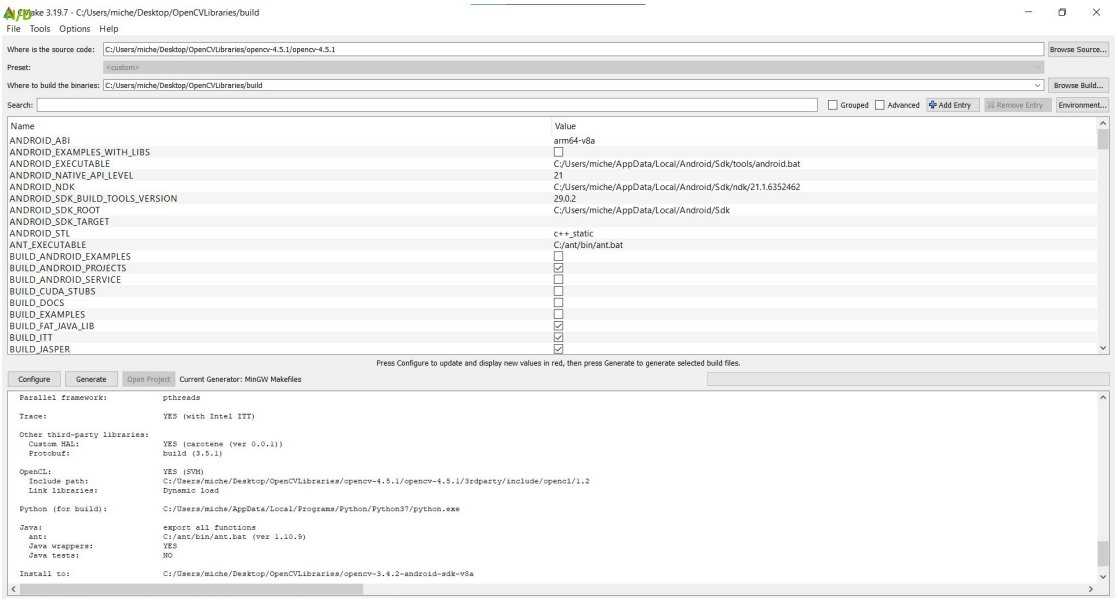


Figura 4.2: Utilizzo di Cmake per compilare file OpenCv

4.2 ArUco

La stima della posizione dei markers è stata possibile grazie ad ArUco, una libreria, che permette di trovare una corrispondenza tra i punti in un ambiente reale e la loro proiezione in un immagine 2d. Nelle seguenti sottosezioni parleremo delle funzioni e del codice utilizzato nel progetto.

4.2.1 Creazione ed utilizzo dei markers

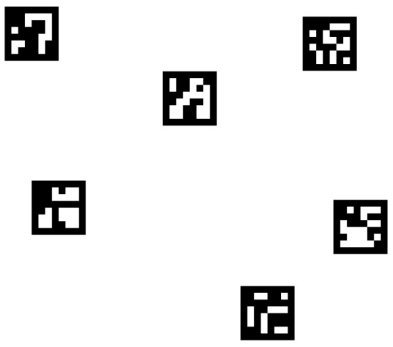


Figura 4.3: Insieme di markers.

Un marker ArUco è composto da un bordo nero, che ne definisce il perimetro e facilita la detection, e da una matrice binaria interna che costituisce l'id del marker. L'id è un numero

ottenuto convertendo la codifica binaria in un numero su base decimale. Un marker presenta ogni angolo univoco poiché dev'essere distinguibile da altri marker che possono essere disposti diversamente nell'ambiente circostante creando così errori di detection. I markers vengono raggruppati in set, chiamati **dizionari**, in funzione della dimensione che determina anche la grandezza della matrice interna. Per il progetto, ad esempio, è stato utilizzato un dizionario rappresentante l'insieme dei markers 5x5.

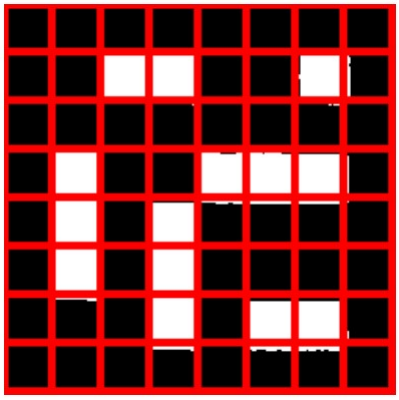


Figura 4.4: Marker diviso in celle.

Per poter effettuare una detection il marker dev'essere prima stampato. La stampa può essere effettuata da codice, tramite una funzione specifica, o da un sito web https://chev.me/arucogen/ che ne fornisce l'immagine jpeg.

Una volta acquisita l'immagine contenente uno o più markers, il processo di detection viene effettuato tramite 2 step principali.
Nel **primo step** si effettua una detection cercando di trovare, all'interno dell'immagine, delle forme quadrate candidate ad essere possibili markers. All'interno dell'immagine, tutto il resto viene scartato utilizzando dei filtri che permettono il thresholding dell'immagine.
Nel **secondo step** si effettua la detection dei possibili candidati applicando delle trasformazioni ed ottenendo cosi un'immagine del markers divisa in celle secondo la grandezza del marker e dei suoi bordi.

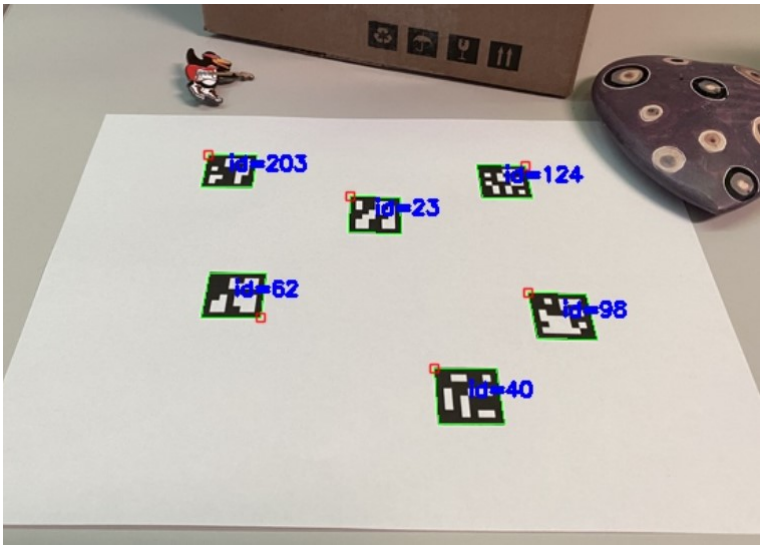


Figura 4.5: Detection dei markers.

Questo processo viene eseguito tramite la funzione **detectMarkers()**:

```
private java.util.List<org.opencv.core.Mat> corners = new ArrayList<>();
private MatOfInt ids = new MatOfInt();

    ...

Aruco.detectMarkers(mRgb, Aruco.getPredefinedDictionary(Aruco.DICT_ARUCO_ORIGINAL)
                                                        , corners, ids);
    ...
```

- Il primo parametro **mRgb** è l'immagine che contiene i marker da rilevare.

- Il secondo rappresenta l'oggetto dizionario: in questo caso il dizionario scelto è il DICT_ARUCO_ORIGINAL.

- Il terzo parametro **corners** è la lista degli angoli dei markers rilevati. Gli angoli di ogni marker vengono rilevati in senso orario partendo dall'angolo in alto a sinistra.

- Il quarto parametro **ids** è la lista degli Id di ogni marker rilevato in *corners*. Da notare che il vettore ritornato in questo quarto parametro ha le stesse dimensioni del terzo.

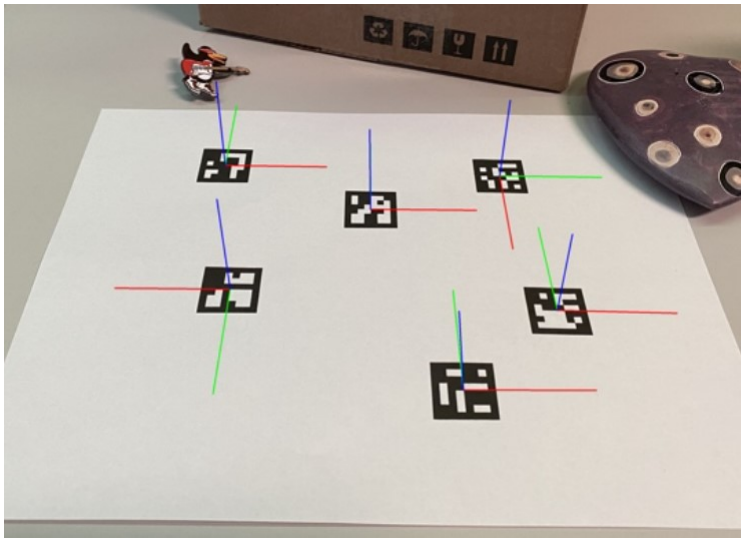


Figura 4.6: Stima della posizione dei markers.

Ottenuti i markers è stata ottenuta la loro posizione e rotazione rispetto alla fotocamera, utilizzando la funzione **estimatePoseSingleMarkers()**:

```
private Mat rvecs = new Mat();
private Mat tvecs = new Mat();

    ...

for (int i = 0; i < corners.size(); i++) {
        Aruco.estimatePoseSingleMarkers(corners.subList(i, i + 1), (float) 0.05, cameraMatrix
                                                                                , coeffs
                                                                                , rvecs, tvecs);

    ...
}
```

- Il primo parametro rappresenta il vettore di angoli dei markers ritornati dalla funzione *detectMarkers()*. Viene considerata una subList in un ciclo **for** poiché nel nostro caso è necessario effettuare la dection di più markers.
- Il secondo parametro è la lunghezza di un lato del marker in metri.
- Il quinto parametro (di output) **rvecs** rappresenta il vettore di rotazione per ogni marker rilevato in **corners**.
- Il sesto parametro (di output) **tvecs** rappresenta il vettore di traslazione per ogni marker rilevato in **corners**.

Per ottenere una stima della posizione è necessario calibrare i parametri della fotocamera utilizzata. Tali parametri sono **cameraMatrix** e **distCoeffs** (Coefficienti di distorsione). per la calibrazione viene utilizzata la funzione **loadCalib()**:

```
loadCalib(){
    cameraMatrix = new Mat(3, 3, CvType.CV_32F);
     coeffs  = new Mat(5, 1, CvType.CV_32F);

    cameraMatrix.put(0, 0, 1024.3446);
    cameraMatrix.put(0, 1, 0);
    cameraMatrix.put(0, 2, 640.0);
    cameraMatrix.put(1, 0, 0);
    cameraMatrix.put(1, 1, 1024.3446);
    cameraMatrix.put(1, 2, 360);
    cameraMatrix.put(2, 0, 0);
    cameraMatrix.put(2, 1, 0);
    cameraMatrix.put(2, 2, 1);

     coeffs.put(0,  0,  0.08816299);
     coeffs.put(1,  0,  −0.21675685);
     coeffs.put(2,  0,  0);
     coeffs.put(3,  0,  0);
     coeffs.put(4,  0,  0);
}
```

Nel progetto in questione è stata effettuata una calibrazione manuale. Tramite il metodo *put* viene inserito ogni valore della matrice rispettivamente, nella riga (primo parametro) e colonna (secondo parametro) designata, il valore per la calibrazione passato nel terzo parametro.

4.3 ChArUco

Come vedremo nel capitolo successivo, la sostituzione di ArUco con la libreria ChArUco porta a dei miglioramenti per quanto riguarda l'accuratezza del calcolo della posizione di un marker. Questo è dovuto al fatto che una board di markers è molto efficiente nella detection ma produce risultati con un'accuratezza elevata. Con una scacchiera, invece, è possibile avere un'accuratezza migliore ma una scarsa versatilità in termini di visibilità poiché la parziale occlusione non è permessa. Per questo motivo è stato definito un terzo approccio che risulta essere la combinazione dei due descritti precedentemente e consiste nell'utilizzo della **ChArUco board**.

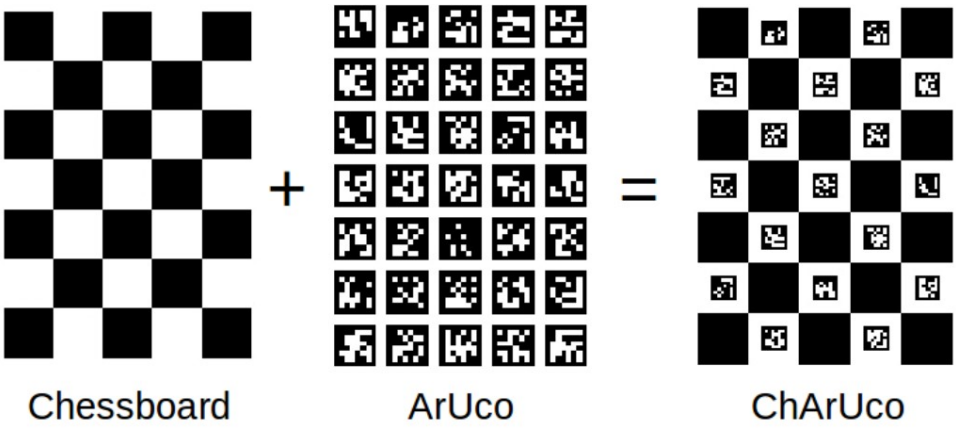


Figura 4.7: Definizione della ChArUco board.

4.3.1 Creazione ed utilizzo della ChArUco Board

La creazione di una ChArUco board è simile a quella, descritta precedentemente, per la creazione di un Marker se non per il concetto di univocità, che ora è esteso a tutta la board.

```
try {

    Imgproc.cvtColor(inputFrame, mRet, Imgproc.COLOR_BGRA2BGR);
    Aruco.detectMarkers(mRet, Aruco.getPredefinedDictionary(Aruco.DICT_6X6_250), corners
                                                                                , ids
                                                                                , params);

    // if at least one marker detected
    if (ids.total() > 0) {
        //Aruco.drawDetectedMarkers(mRet, corners, ids);
        charucoCorners = new Mat();
        charucoIds = new MatOfInt();
        Aruco.interpolateCornersCharuco(corners, ids, mRet, board, charucoCorners, charucoIds
                                                                        , cameraMatrix
                                                                        , coeffs );

    ...

}


if (charucoCorners.total() > 0) {
    Aruco.drawDetectedCornersCharuco(mRet, charucoCorners, charucoIds, new Scalar(255, 0, 0));

    ...
```

```
    boolean valid = Aruco.estimatePoseCharucoBoard(charucoCorners, charucoIds, board
                                                   , cameraMatrix
                                                   , coeffs
                                                   , rvecs
                                                   , tvecs);

    ...


    if (valid)
        Aruco.drawAxis(mRet, cameraMatrix, coeffs, rvecs, tvecs, (float) 0.1);

    ...

}
```

Infatti, come per Aruco (4.2), si effettua la detection dei markers tramite **detectMarkers()** e successivamente si identifica la board tramite **interpolateCornersCharuco()**. Se la board è stata rilevata, si calcola la posizione e la rotazione tramite **estimatePoseCharucoBoard()**. I risulati vengono inseriti nei vettori di rotazione e traslazione **rvecs** e **tvecs**. Successivamente vengono disegnati sul frame elaborato tramite la funzione **drawAxis()**.

4.3.2 Utilizzo della board su supporto volante

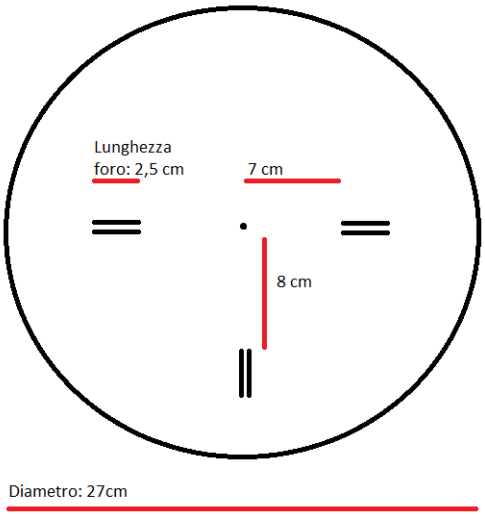


Figura 4.8: Misure del supporto per la board da applicare al volante.

Il calcolo dell'angolo volante θ prevede l'applicazione dei markers sul volante per poterne rilavare la posizione. Per facilitare l'applicazione è stato costruito un supporto circolare:

la stima delle grandezze ideali è stato ottenuta costruendo un supporto in cartone da applicare al volante. Il supporto, dopo una serie di prove, è stato modellato ottenendo delle grandezze che garantiscano una guida priva di interferenze: è stato scelto un diametro di 27 cm con l'aggiunta di piccole fenditure lunghe 2.5 cm e distanti 7/8 cm dal centro. Attraverso le fenditure vengono inserite delle fascette in velcro per poter attaccare il supporto alle razze del volante.
In futuro, il supporto potrebbe essere stampato tramite stampante 3d con dei ganci mobili per adattarsi a qualsiasi tipo di volante.

4.4 Camera2 API

Durante il lavoro svolto è stata utilizzata la nuova classe di API disponibile in Android. Nella versione precedente dell'app era stata utilizzata, anche se deprecata, la prima versione (3.6.1) di API per necessità legate all'implementazione di OpenCv. E' stato possibile integrare le **Camera2 API** [12] con le nuove versioni di OpenCv. La nuova classe di API porta un aumento sostanziale delle capacità delle applicazioni di controllare il sottosistema della fotocamera garantendo maggiore affidabilità e massimizzando la qualità e le prestazioni. L'API modella il sottosistema della camera [13] come una pipeline ricevendo in input delle richieste per ogni singolo frame ed effettuandone la cattura restituendone in output i metadata del frame ed un set di immagini bufferizzate.

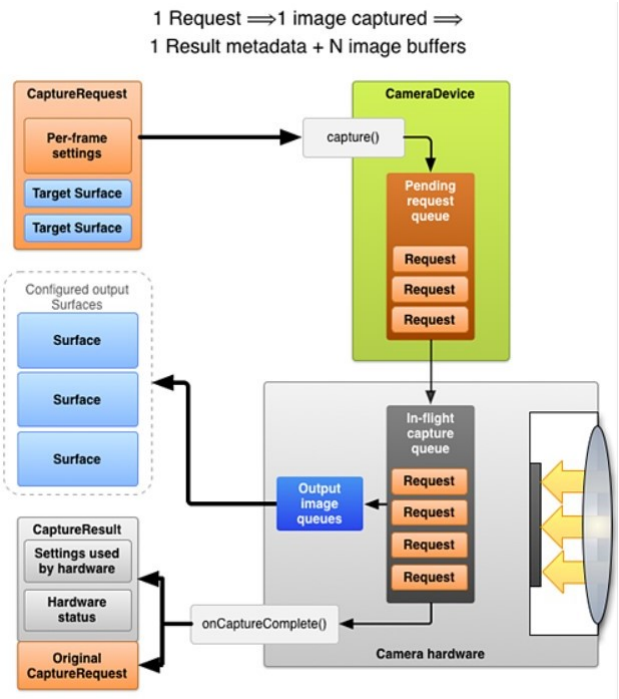


Figura 4.9: Modello sottosistema Camera2 API.

Vengono così eliminati dei limiti su alcuni aspetti legati allo sviluppo dell'algoritmo per il calcolo dell'angolo volante. Ad esempio è stato possibile modificare il valore del framerate della registrazione video permettendo di fruttare a pieno l'hardware del dispositivo.

Parte III

Terza Parte

Capitolo 5

Soluzioni al problema dell'angolo volante

5.1 Algoritmo di Tracking e stima dell'angolo volante

L'algoritmo di Tracking utilizzato è composto da una fase in **foreground**, realizzata nella Camera2Tracker Activity (3.4.2), che ha il compito di dare un feedback iniziale all'utilizzatore e costituisce il punto di partenza della prova.
Finita la prima fase, vengono abilitate la seconda e la terza fase che agiscono in **background** effettuando il calcolo dell'angolo volante senza essere percepite dall'utente. Come descritto nella sezione 3.7.3, le due fasi lavorano in maniera asincrona sfruttando il pattern *Produttore-Consumatore* contenuto realizzato nel *Service* (3.2.3).

5.1.1 Descrizione delle fasi

Vedremo in dettaglio come operano tutte le fasi per ottenere il risultato finale:

- **Prima fase in Foreground**

  Come vediamo dalla figura (5.1), in questa fase viene acquisito il valore di partenza dell'angolo volante. Infatti, non appena l'utente inquadra i markers, si attiva un pulsante che da un feedback sulla corretta detection dei markers ed, una volta premuto, determina il salvataggio dell'angolo di partenza e l'inizio della prova. Nei primi 30 secondi, l'utente viene guidato con un'interfaccia così da ottenere i dati necessari per l'assetto e la correzione del sistema di riferimento.

- **Seconda fase in Background**

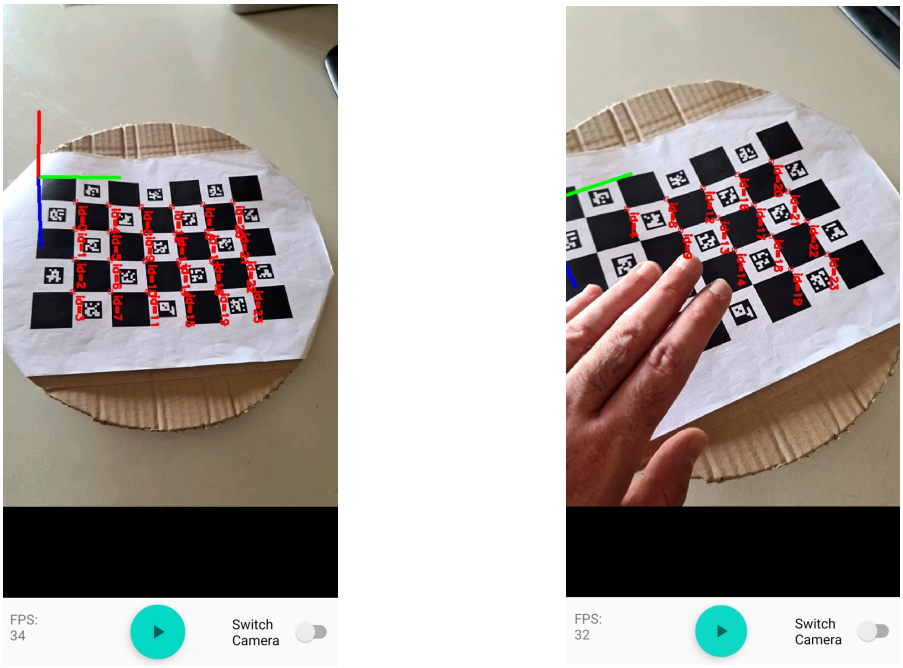
  In questa fase viene avviato il *Camera2Service* che si occupa di acquisire un frame dopo l'altro ed inserirlo in una *LinkedBlockingQueue* (3.1). Il service svolge quindi il ruolo di produttore del dato in input. Contemporaneamente crea ed esegue un *Thread* che svolge tutti i processi della fase successiva.

- **Terza fase in Background**

  Nell'ultima fase, il thread in questione acquisisce il ruolo di consumatore prelevando il un frame alla volta dalla *LinkedBlockingQueue*. Successivamente trasforma il frame in un oggetto **matrice** per poter essere elaborato dalla **libreria OpenCv**: Si sfruttano le librerie **Aruco/Charuco** per stimare il valore dell'**angolo volante**.

5.2 Copertura accidentale dei Markers

L'implementazione della Charuco board, come visto nella sezione 4.3, ha contribuito ad attenuare la problematica della detection in caso di occlusione parziale dei markers.



(a) Board non occultata. (b) Board parzialmente occultata.

Figura 5.1: Detection dei markers e calcolo dell'angolo con occlusione parziale.

Con questo approccio è possibile aumentare le probabilità di riuscita del calcolo dell'angolo volante durante le manovre del conducente. Per garantirne il calcolo è necessario che almeno 5 markers della board siano visibili. Maggiore è il numero di markers rilevati, migliore e preciso sarà l'angolo θ calcolato. La fase dei test (6.1) è stata importante per capire fino a che punto è possibile spingersi per ottenere un compromesso tra un buon risultato ed una detection durante le manovre rapide.

5.3 Basso framerate

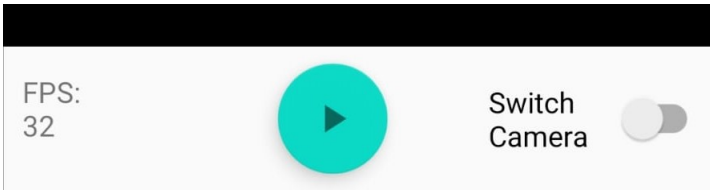


Figura 5.2: Media della velocità di cattura dei frame.

Utilizzando un Samsung Galaxy S20 che implementa una fotocamera capace di registrare a 60 fps, è stato possibile superare le limitazioni della vecchia API passando da una registrazione che si aggirava intorno ai 20 fps ad una media di 32/34 fps con picchi di 45 fps. Naturalmente queste medie sono state ottenute tenendo conto del peso computazionale dovuto all'elaborazione real-time dei singoli frame per la detection dei markers. La velocità, oltre ad essere influenzata dall'hardware, dipende dal variare delle condizioni di luce e da eccessive vibrazioni. Il problema della variazione delle condizioni di luce e la possibile presenza di riflessi è stato attenuato permettendo l'attivazione del flash durante l'esecuzione della prova: la luce diretta sulla board migliora la detection dei markers aumentando il framerate medio di circa 5 fps.
Tutte le seguenti migliorie hanno indirettamente contribuito al problema del de-allineamento dei dati riducendo possibili ritardi dovuti alla cattura.

5.4 De-allineamento dei dati

5.4.1 Inserimento dei Timestamp

Trovare una soluzione al problema del de-allineamento dei dati è stato fondamentale, poiché, nella versione precedente i dati venivano acquisiti senza un riferimento all'istante di cattura. Questo portava uno sfasamento tra i valori acquisiti dall'applicazione e quelli ideali ottenuti tramite un simulatore o una IMU[1]. L'errore andava quindi a ripercuotersi sul calcolo dei valori finali e soprattutto sull'angolo di assetto β.
Il problema è stato risolto inserendo in fase di acquisizione un **timestamp** associato ad ogni dato catturato da un sensore: In questo modo interpolando i dati, acquisiti dai sensori con frequenze diverse, è possibile ottenere una frequenza comune costante.

[1]Inertial Measurement Unit: dispositivo elettronico utilizzato per misurare accelerazione, orientazione, velocità angolari ed altre forze gravitazionali. E' costituito da 3 accelerometri, 3 giroscopi e, a seconda delle esigenze di acquisizione, da 3 magnetometri. Possiede un componente per ognuno dei 3 assi del veicolo.

5.4.2 Duplicazione dei dati

L'inserimento dei timestamp ha portato alla scoperta di un'altra problematica, rappresentata in Figura 5.3, che era passata del tutto inosservata: Lo stesso dato rilevato, veniva acquisito e salvato più volte a causa della struttura di acquisizione implementata. Il valore di un dato catturato in un determinato istante di tempo, veniva elaborato e salvato un numero di volte del tutto casuale. Infatti, senza timestamp, si aveva la sensazione che il valore si mantenesse semplicemente costante nel tempo. Questo fenomeno contribuiva al de-allineamento dei dati e determinava l'utilizzo di una quantità di memoria necessaria maggiore per il salvataggio della prova effettuata.

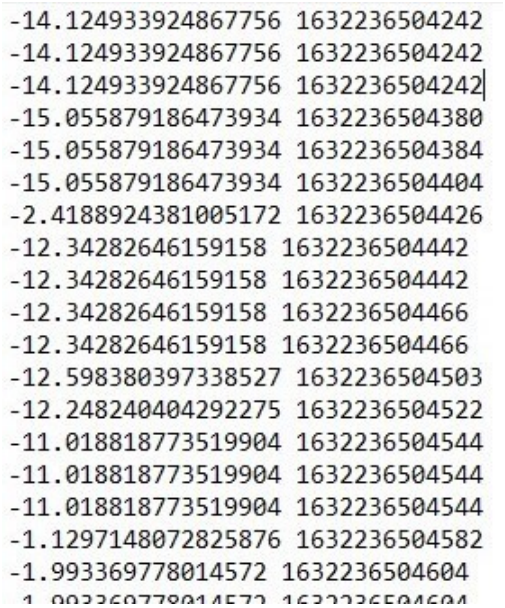


Figura 5.3: Duplicati dell'angolo volante: a sinistra i *valori*, a destra i rispettivi *timestamp*.

Il problema è stato risolto modificando la struttura di acquisizione implementata e soprattutto l'algoritmo per il salvataggio dei dati su file di testo. Questa soluzione è importante per una corretta interpolazione dei dati in RAW[2], effettuata in post produzione. Vediamo nella seguente immagine un file con i valori dell'angolo volante θ senza la presenza di duplicati.

[2]Si intende un dato che contiene informazioni non ancora elaborate.

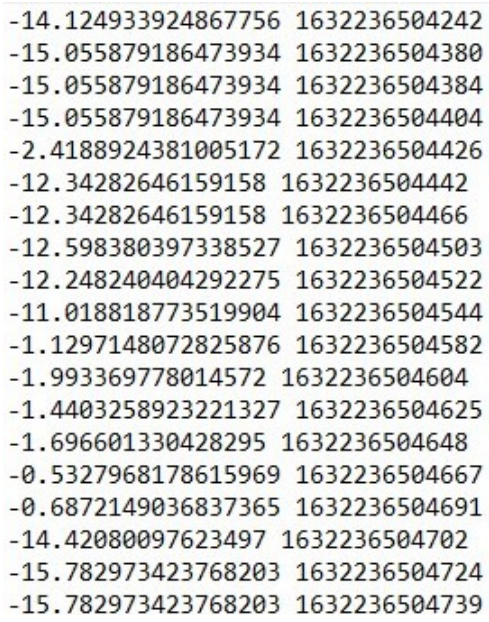


Figura 5.4: Assenza di duplicati dell'angolo volante: a sinistra i *valori*, a destra i rispettivi *timestamp*.

Capitolo 6

Risultati

In quest'ultima parte verranno inizialmente analizzati i risultati ottenuti dalle prove simulate e, nella seconda parte, le prove reali svolte su strada.

6.1 Prove Simulate

La prima sezione del capitolo descrive le prove svolte tramite simulatore composto dal software *CarMaker* e da una postazione di guida. Successivamente una board con i marker e stata applicata sul volante. Lo smartphone, invece, è stato montato su un supporto laterale in direzione del volante.
Lo scopo principale di queste simulazioni è quello di confrontare il valore dell'angolo volante restituito dal simulatore, con quello calcolato dall'applicazione.
Per le prime 2 prove è stato utilizzato come modello un *BMW Serie 5*.

Figura 6.1: Primo veicolo utilizzato: *BMW Serie 5*.

6.1.1 Prova 1: Rotatoria

La prima simulazione è stata effettuata su un segmento circolare di diametro 100 m per riprodurre l'andamento del veicolo su una rotatoria. Dopo una fase di accelerazione, il veicolo mantiene una velocità costante per tutta la prova.

(a) Simulazione con visuale conducente. (b) Mappatura del percorso: Rotatoria.

Figura 6.2: Raffigurazioni del percorso della prima prova simulata.

L'obiettivo principale è di ottenere un primo riscontro sull'affidabilità delle misure percorrendo un tragitto uniforme.

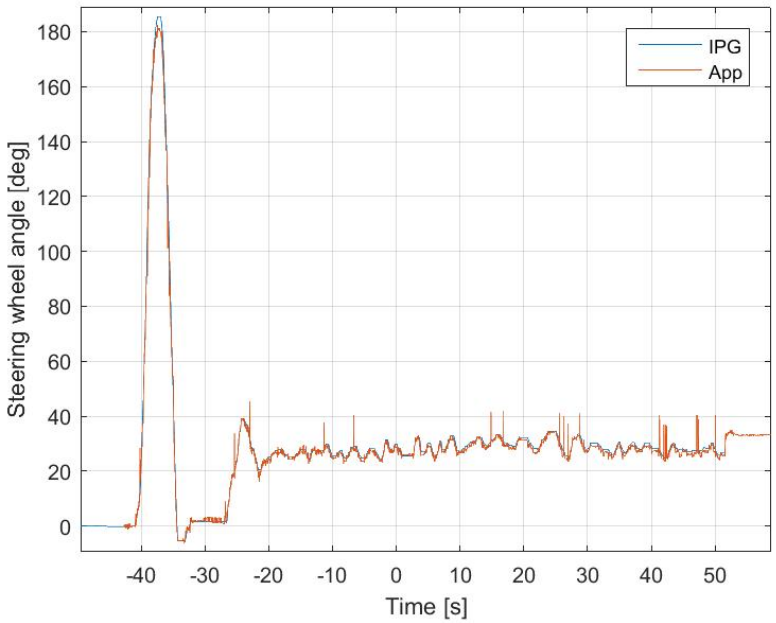


Figura 6.3: Grafico della prima prova che mostra il confronto tra l'angolo volante del simulatore e dell'app su un segmento circolare con diametro 42m.

Confrontando i risultati ottenuti dall'applicazione con quelli del simulatore vediamo come l'allineamento sull'asse dei tempi sia praticamente identico. Anche il valore dell'angolo volante, in un percorso privo di cambiamenti di direzione, risulta fedele. Solo in piccoli istanti il valore risulta diverso ma tale differenza è dovuta a piccole interferenze date dalla luce e dalla calibrazione della camera. La prima prova ha dato comunque dei risultati soddisfacenti.
Da notare come in tutte le prove, nella parte iniziale dei grafici, vedremo un picco generato volontariamente: Poiché non è possibile generare un input che determini lo start della prova in contemporanea, inizialmente, viene ruotato il volante di 180 gradi generando una **singolarità** per permettere l'allineamento dei grafici sull'asse del tempo.

Successivamente la prova è stata replicata su un segmento circolare con un diametro di 42 metri per generare valori di θ più elevati.

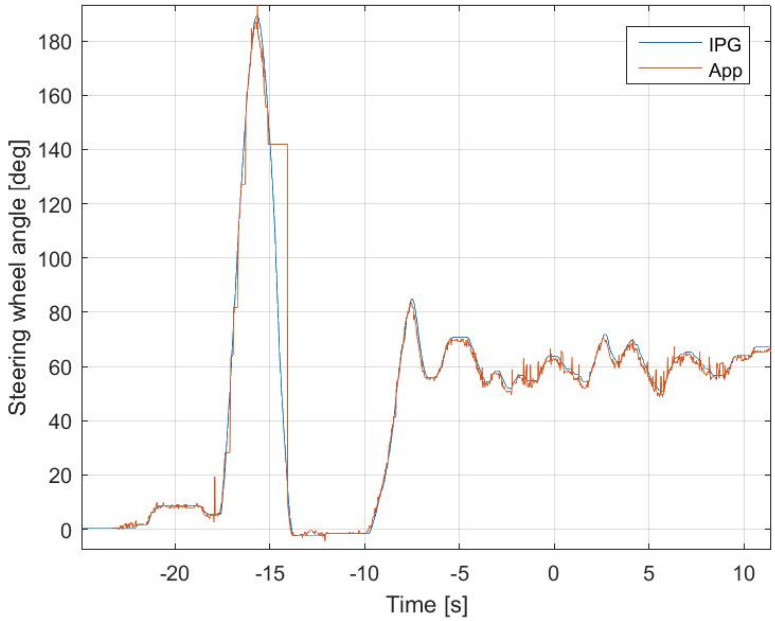


Figura 6.4: Grafico della prima prova che mostra il confronto tra l'angolo volante del simulatore e dell'app su un segmento circolare con diametro 42m.

Anche in questo caso, nonostante le maggiori oscillazioni, i due andamenti risultano ben allineati e i valori coincidono a parte una singolarità presente a 140°. L'angolo misurato ha subito un incremento di 30° avendo percorso una curva più stretta.

6.1.2 Prova 2: Percorso semplice

La secondo prova simulata risulta molto simile alla prima ma è resa più complicata con un percorso e un andamento meno lineari.

(a) Simulazione con visuale conducente.

(b) Mappatura del percorso semplice.

Figura 6.5: Raffigurazioni del percorso della seconda prova simulata.

Durante questa prova sono stati percorsi 2 tragitti diversi che chiameremo tragitto **A** e **B**.

- **Tragitto A:**

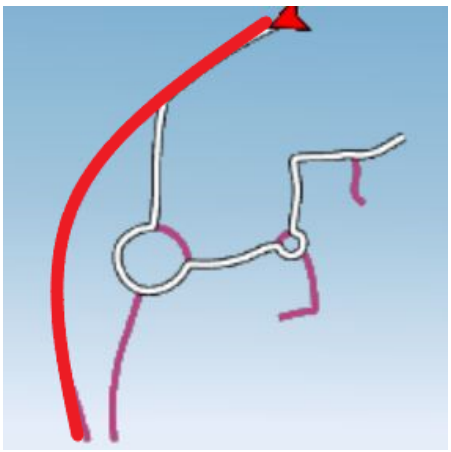


Figura 6.6: Percorso semplice: Tragitto A.

In questo primo tragitto è stato percorso il primo tratto ad una velocità crescente per poi affrontare una curva a velocità costante.

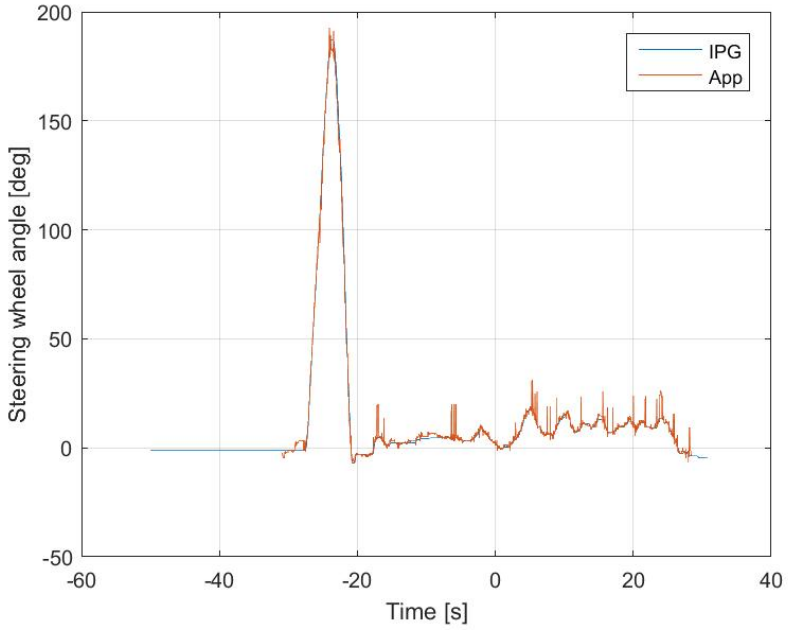


Figura 6.7: Grafico della seconda prova che mostra i valori rilevati per il tragitto A.

Nonostante la velocità elevata, la tipologia della curva ha portato ad un valore massimo θ di circa 25 gradi ma con oscillazioni più fitte: nonostante siano più difficili da seguire, anche qui i due andamenti sono risultati per allineati.

- **Tragitto B:**

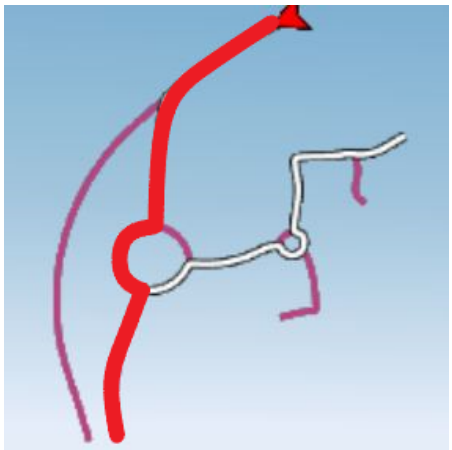


Figura 6.8: Percorso semplice: Tragitto B.

Nel secondo tragitto, dopo una fase di accelerazione viene compiuta una curva a sinistra seguita da metà rotatoria e una leggera curva.

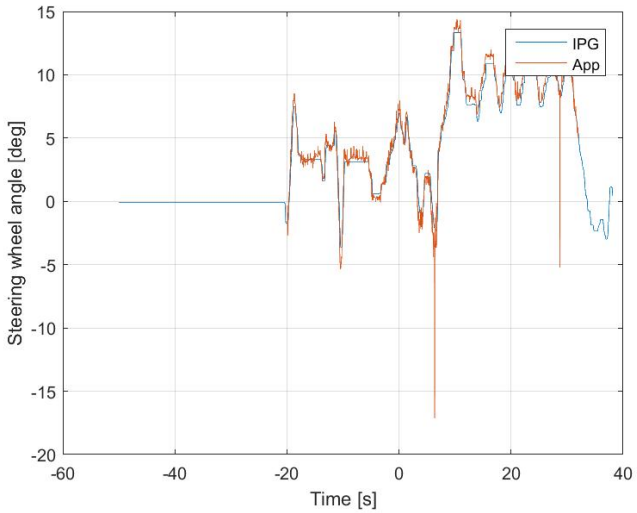


Figura 6.9: Grafico della seconda prova che mostra i valori rilevati per il tragitto B.

In questo caso le oscillazioni del valore θ sono nettamente maggiori e, nella parte finale, l'applicazione ha generato valori con un leggero ma costante delay: Questo fattore è molto importante poiché la versione precedente dell'app generava un delay variabile generando notevoli difficoltà nell'allineamento e nell'interpolazione dei dati.

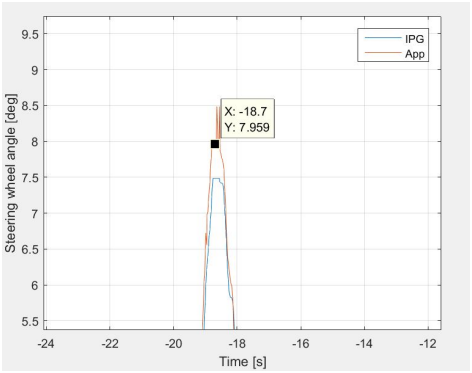
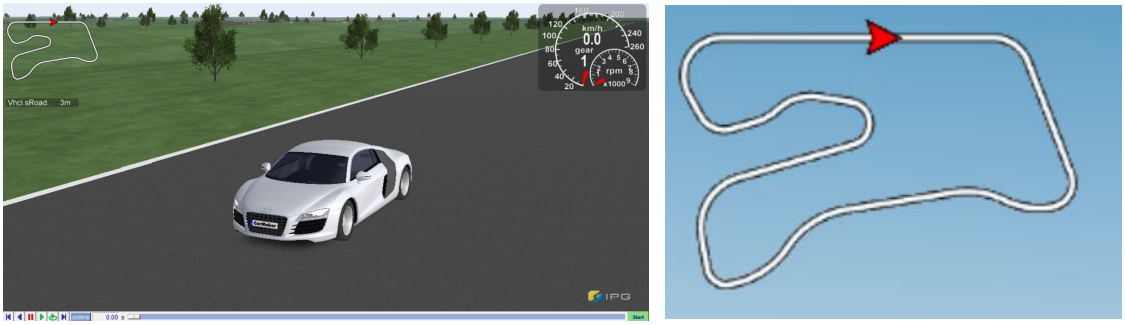


Figura 6.10: Grafico della seconda prova che mostra la differenza massima di θ rilevati per il tragitto B sui due dispositivi.

Il grafico in Figura 6.10 è importante poiché rileva uno scostamento medio di circa 0.5 gradi dal valore ideale IPG. Il massimo scostamento rilevato in pochissimi istanti è di circa 1 grado.

6.1.3 Prova 3: Percorso su pista



(a) Simulazione su pista con visuale autovettura. (b) Mappatura del percorso: Circuito pista.

Figura 6.11: Raffigurazioni del percorso della prova simulata su pista.

In questa prova su pista è stata utilizzata una supercar (Audi r8) per testare la capacità dell'applicazione nel rilevare l'angolo θ a velocità elevate. Le curve sono state percorse seguendo una traiettoria ideale con uno stile di guida aggressivo. E' interessante analizzare il confronto tra i valori calcolati dai due sistemi.

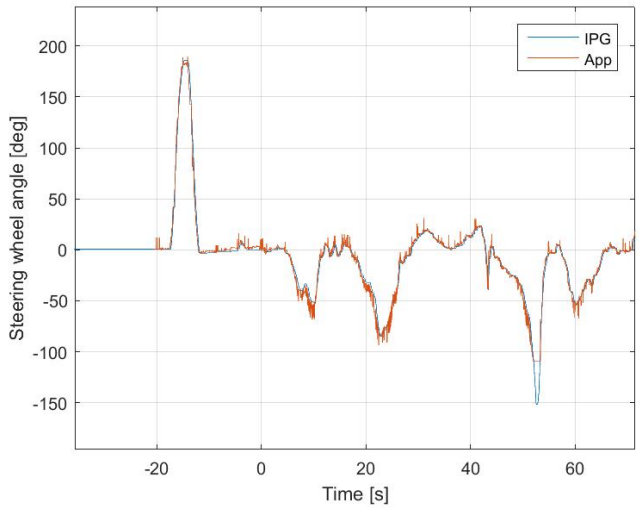


Figura 6.12: Grafico della terza prova simulata su pista.

Dal grafico si notano picchi non uniformi dovuti a piccoli cambi di direzioni o, in alcuni punti, fasi di sottosterzo per riportare il veicolo in traiettoria. Nonostante la guida sportiva i valori di θ rimangono sempre molto fedeli soprattutto lungo l'asse tempi. Nella fase finale si può notare una piccola perdita dell'angolo volante da parte dell'applicazione dovuta ad una brusca sterzata seguita da una copertura della board con entrambe le braccia. Nonostante ciò, il valore è stato subito recuperato restando fedele nel tempo.

6.2 Prove su strada

Un volta verificato, nel capitolo precedente, la correttezza dell'angolo θ e del suo valore nel tempo, si è deciso di fare lo stesso per le altre componenti di dinamica del veicolo. Infatti, lo scopo di questa sezione è quello di verificare la correttezza delle altre componenti, confrontando, i valori calcolati dai sensori dello smartphone con quelli calcolati da una **IMU**[1]. Quest'ultima è sta fissata specularmente allo smartphone sul finestrino del passeggero. Al termine di ogni prova i dati sono stati esportati, elaborati e confrontati su *MatLab*. Per ogni test effettuato sarà mostrato il percorso visto da Google Maps e il valore delle accelerazioni.

6.2.1 Prova 4: Percorso Urbano

La quarta prova è stata effettuata su un percorso che circumnaviga il Politecnico di Torino a bordo di una Hyundai Veloster. La prova, nonostante il percorso non sia molto lungo, dura circa quattro minuti per via dei semafori.

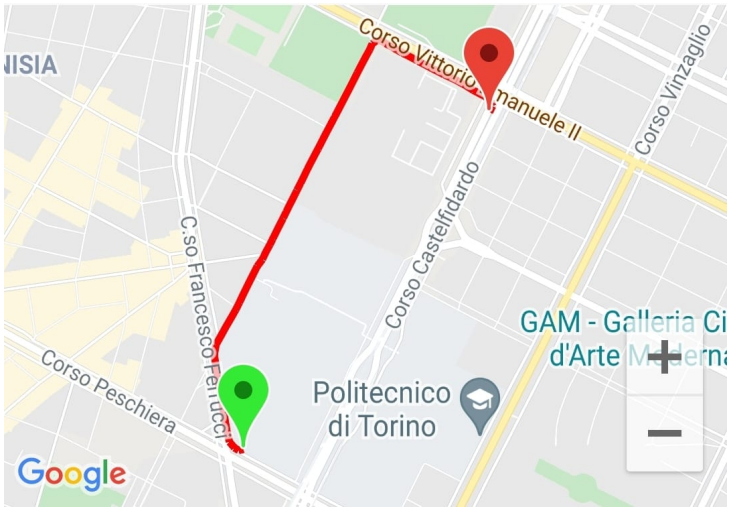


Figura 6.13: Tragitto della quarta prova fatta in città.

Tuttavia, questa prima prova su strada ha permesso di testare il funzionamento e l'affidabilità dei dispositivi e di ottenere i primi confronti tra i dati.

[1]Inertial Measurement Unit: dispositivo elettronico utilizzato per misurare accelerazione, orientazione, velocità angolari ed altre forze gravitazionali. E' costituito da 3 accelerometri, 3 giroscopi e, a seconda delle esigenze di acquisizione, da 3 magnetometri. Possiede un componente per ognuno dei 3 assi del veicolo.

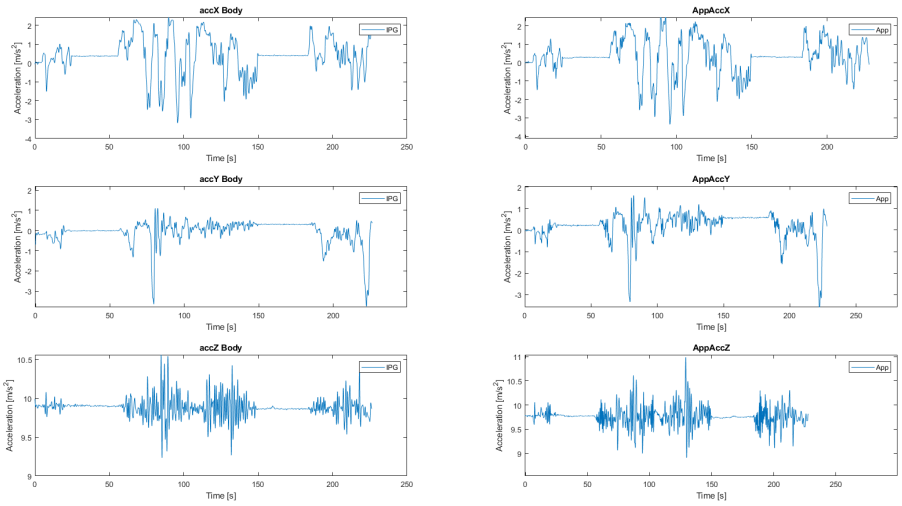


Figura 6.14: Confronto delle accelerazioni tra IMU e Applicazione.

Un primo confronto, tra le accelerazioni dei 2 sistemi, è stato necessario per allineare i rispettivi assi. In particolare l'**IMU** presenta gli assi x ed y invertiti e l'asse y negativo rispetto a quello dell'applicazione.

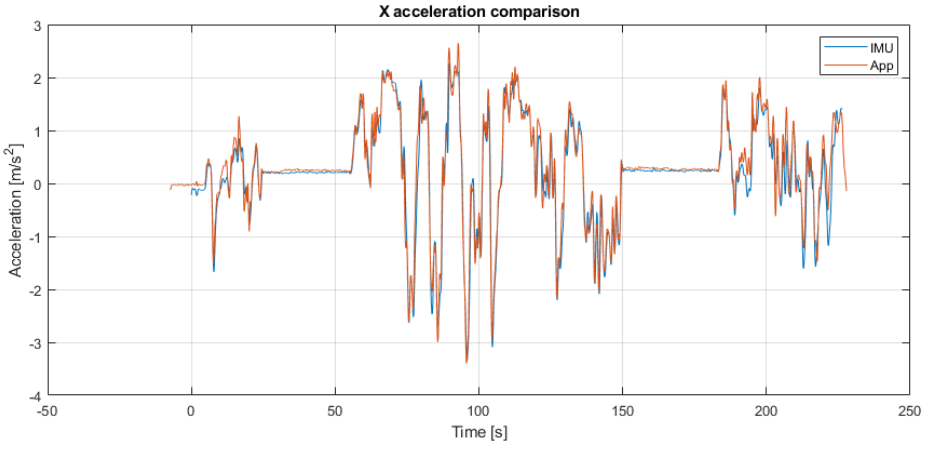


Figura 6.15: Confronto con sovrapposizione dell'accelerazione X tra IMU e Applicazione.

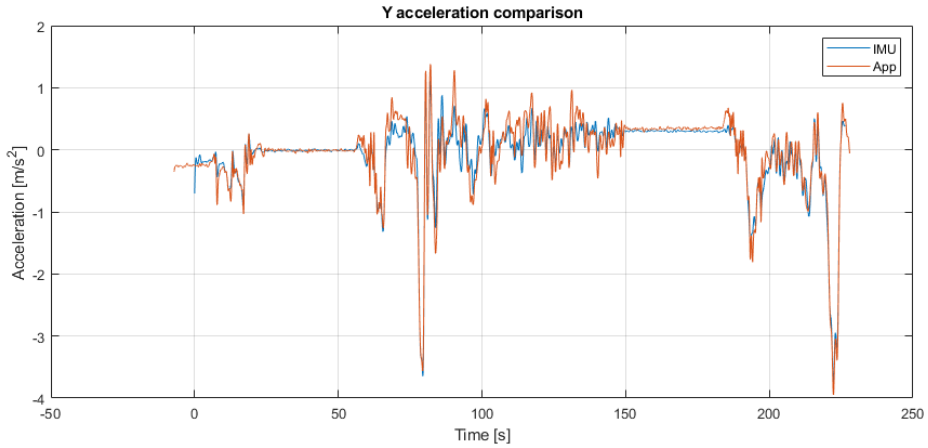


Figura 6.16: Confronto con sovrapposizione dell'accelerazione Y tra IMU e Applicazione.

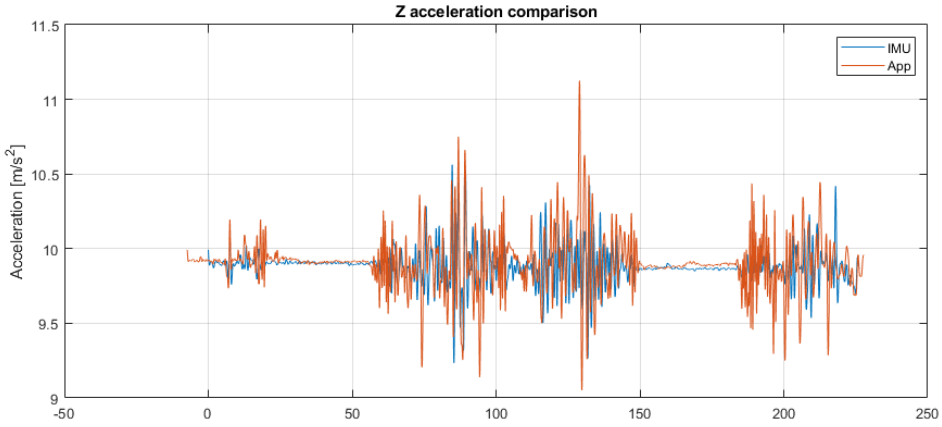


Figura 6.17: Confronto con sovrapposizione dell'accelerazione Z tra IMU e Applicazione.

Successivamente, ogni componente dell'accelerazione dei due sistemi è stata sovrapposta. I risultati ottenuti sono abbastanza fedeli soprattutto per quanto riguarda la componente longitudinale x dell'accelerazioni. Anche le altre risultano allineate ma con valori leggermente diversi a causa del rumore creato dalle oscillazioni dello smartphone attaccato al supporto.

6.2.2 Prova 5: Percorso su sentiero collinare

La quinta prova è stata fatta nella zona collinare di Torino su un percorso ricco di tornanti che costeggia il Parco di San Vito.

(a) Percorso totale. (b) Zoom sulla prima parte del percorso collinare

Figura 6.18: Raffigurazioni del percorso della prova su sentiero collinare.

Grazie a questo percorso è stato possibile ottenere dei dati durante una guida più sportiva con una velocità sostenuta e priva dei rallentamenti tipici del percorso urbano.

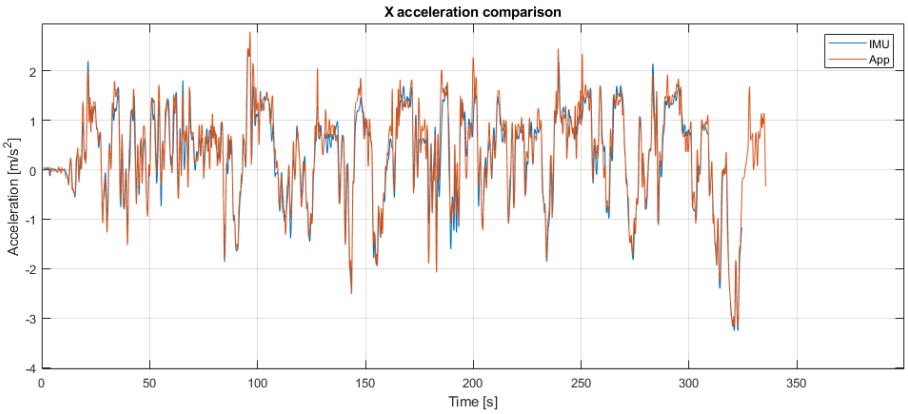


Figura 6.19: Confronto con sovrapposizione dell'accelerazione X tra IMU e Applicazione su sentiero collinare.

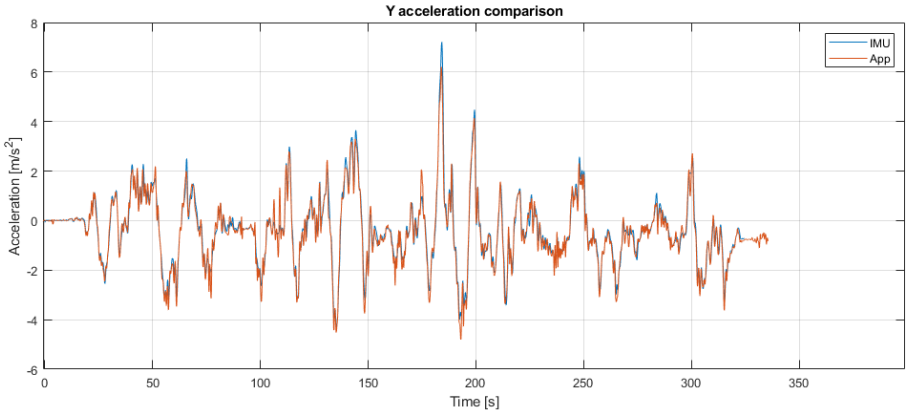


Figura 6.20: Confronto con sovrapposizione dell'accelerazione Y tra IMU e Applicazione su sentiero collinare.

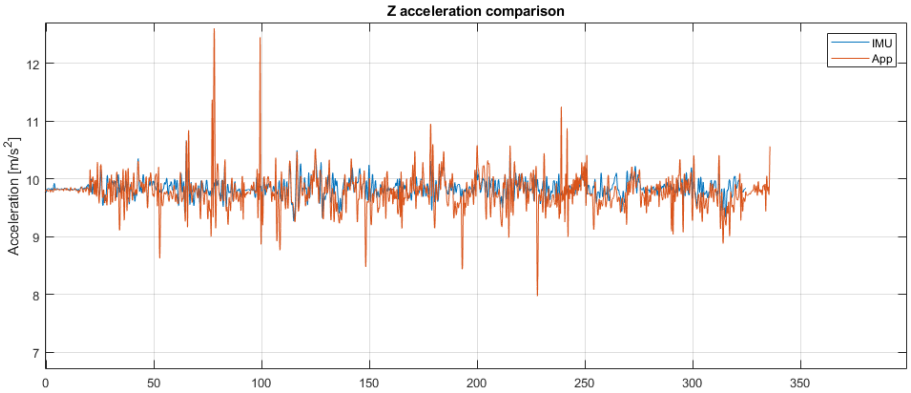


Figura 6.21: Confronto con sovrapposizione dell'accelerazione Z tra IMU e Applicazione su sentiero collinare.

Anche in questa prova, sovrapponendo tutte le componenti dell'accelerazione dei due sistemi, si ottengono ottimi risultati sia per i valori che per l'allineamento sull'asse dei tempi. Nonostante le componenti Z risultino diverse in alcuni istanti, i calcoli complessivi risultano essere molto promettenti se si pensa all'aver utilizzato uno stile di guida sportivo in un percorso caratterizzato da stretti tornanti con pendenza variabile.

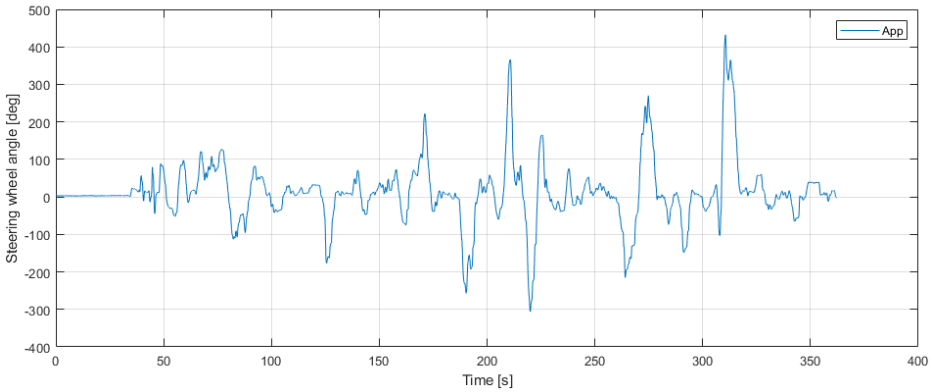


Figura 6.22: Andamento dell'Angolo volante θ durante la prova su percorso collinare.

Avendo verificato l'affidabilità del calcolo dell'angolo volante θ tramite le prove simulate (6.1), viene infine mostrato l'andamento dell'angolo volante durante la prova su percorso collinare. Dal grafico non si notano irregolarità se non una perdita dell'angolo, in pochissimi istanti, che viene recuperata senza ritardi: queste singolarità sono state registrate solo durante alcune sterzate brusche e sono comunque state corrette in post filtrando i dati con *MatLab*.

Capitolo 7

Conclusioni

Il lavoro di tesi svolto mi ha permesso di attenzionare gli aspetti principali relativi alla dinamica del veicolo per capire come raggiungere gli obiettivi prefissati relativi all'upgrade dell'applicazione. Le implementazioni apportate mi hanno permesso di studiare e comprendere bene sia concetti legati all'interazione con la sensoristica dello smartphone, sia argomenti riguardanti la computer vision ed in particolare la realtà aumentata nel mondo mobile.
Le prove simulate tramite CarMaker e su strada hanno dato risultati soddisfacenti: confrontando i risultati prodotti dall'applicazione con CarMaker e la piattaforma inerziale (IMU), si è potuto constatare come tutte le modifiche implementate abbiano portato alla soluzione del problema di sincronizzazione dell'angolo volante θ garantendo dei valori affidabili.

7.1 Miglioramenti Futuri

Attualmente i sensori dello smartphone acquisiscono dati con frequenze variabili a causa della mancanza di un clock unico. Per risolvere questo problema ed elaborare tutti i dati avendo un unico asse dei tempi, si potrebbe implementare un algoritmo che vada, in post, ad interpolare tutti i dati prodotti dallo smartphone. Per quanto riguarda la gestione dei database si potrebbe implementare la totale gestione del database tramite *ROOM* ovvero una libreria che conferisce un design pattern modulare e meno *verbose*. Si potrebbe anche sperimentare una sostituzione dei Markers con *Tiny YOLO* ovvero una *Convolutional Neural Network* utilizzata per il riconoscimento di oggetti e rotazione istante per istante. Per ottenere un ulteriore incremento della precisione di calcolo dell'angolo volante, potrebbe essere implementata una calibrazione automatica della board con i Markers. Tramite una stampante 3D si potrebbe progettare e costruire un supporto ad-hoc per garantire una veloce e semplice installazione dei markers sul volante.

Bibliografia

[1] Rakesh Singh. *Explain Java BlockingQueue with Producer-Consumer thread.* URL: https://www.interviewsansar.com/java-blockingqueue-example-with-produce-and-consumer-design-pattern/.

[2] Android Developers. *Save data in a local database using Room.* URL: https://developer.android.com/training/data-storage/room.

[3] L. Di Chio G.Malnati. *Mobile Application Development, Corso di Laurea Magistrale in Ingegneria Informatica, Politecnico di Torino.* 2018.

[4] Android Developers. *Navigation Component.* URL: https://developer.android.com/guide/navigation/navigation-getting-started.

[5] Android Developers. *Class details of Bottom navigation view.* URL: https://developer.android.com/reference/com/google/android/material/bottomnavigation/BottomNavigationView.

[6] mrmans0n. *Smart Location Library.* URL: https://github.com/mrmans0n/smart-location-lib.

[7] lessthanoptimal. *GeoRegression.* URL: https://github.com/lessthanoptimal/GeoRegression.

[8] OpenCv. *OpenCv Documentation.* URL: https://docs.opencv.org/4.5.1/index.html.

[9] OpenCv. *Detection of Aruco Markers.* URL: https://docs.opencv.org/4.5.2/d5/dae/tutorial_aruco_detection.html.

[10] OpenCv. *Detection of ChArUco Corners.* URL: http://www.bim-times.com/opencv/3.3.0/df/d4a/tutorial_charuco_detection.html.

[11] Homan Huang. *Android with OpenCv and Contrib Modules.* URL: https://homanhuang.medium.com/android-opencv-part-8-diy-sdk-contrib-modules-on-windows-7d7e618402fc.

[12] Android Developers. *Android hardware camera 2.* URL: https://developer.android.com/reference/android/hardware/camera2/package-summary.

[13] Android Developers. *Fotocamera HAL3.* URL: https://source.android.com/devices/camera/camera3.