

Façade segmentation from 3D Point Clouds for vehicle localization using Openstreetmap

Relatore: Prof. Domenico Giorgio Sorrenti

Co-relatore: Dr. Augusto Luis Ballardini

Relazione della prova finale di:

Federico Lodovici

Matricola 816040

Anno Accademico 2018-2019

*Basta seguire la strada
e prima o poi si fa il giro del mondo.
Non può finire in nessun altro posto, no?
Jack Kerouac, On the Road*

Indice

1	Introduzione	3
2	Segmentazione Delle Facciate	8
2.1	PCL (Point Cloud Library)	8
2.2	Filtraggio in altezza	9
2.3	Differenza di Normali	10
2.4	Clustering	14
3	Inserimento in OpenStreetMaps	16
3.1	OpenStreetMap	16
3.2	JOSM	18
3.3	Docker	18
3.4	Inserimento nelle mappe	19
3.4.1	Overpass API	19
4	Recupero e Utilizzo delle Point Cloud	24
4.1	ROS (Robotic Operating System)	24
4.2	RLE (Road Layout Estimation)	25
4.2.1	Architettura	26
4.3	IRA OpenStreetMap	27
4.3.1	Recupero delle way	29
4.3.2	Recupero delle point cloud	30
5	Conclusioni	32
Lista delle Immagini		33
Lista del Codice		35
Riferimenti		36
Bibliografia	36	
Siti	37	

Capitolo 1

Introduzione

Nella guida automatizzata, le mappe digitali completano i sensori delle auto. Utilizzando i dati della mappa, il sistema di guida automatizzata può anticipare la strada da percorrere, vedendo ben oltre la gamma dei sensori del veicolo. Il risultato è un veicolo in grado di vedere dietro gli angoli e di adattare il suo comportamento alla guida in condizioni di scarsa visibilità come pioggia o nebbia. La fusione di dati provenienti da diversi sensori con le informazioni aggiunte dalle mappe digitali rende la guida automatizzata più sicura e precisa, poiché il sistema può gestire circostanze più estreme e aggiunge ridondanza.

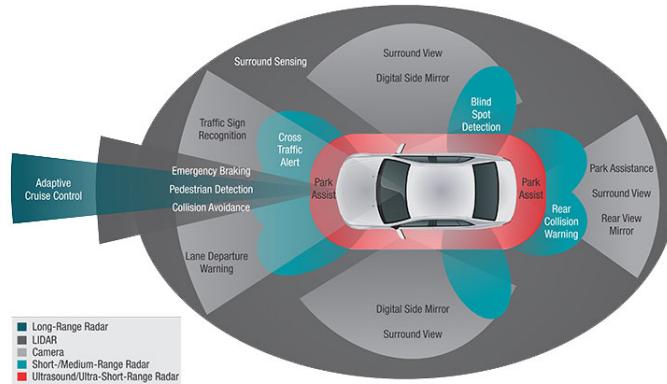


Figura 1.1: ADAS

Dal controllo adattivo della velocità di crociera (ACC) all’assistenza autostradale, fino alla guida autonoma (AD), beneficiano tutti di vari livelli di granularità forniti dalle mappe. Maggiore è il livello di autonomia del veicolo (1.2), più requisiti devono soddisfare le mappe digitali.

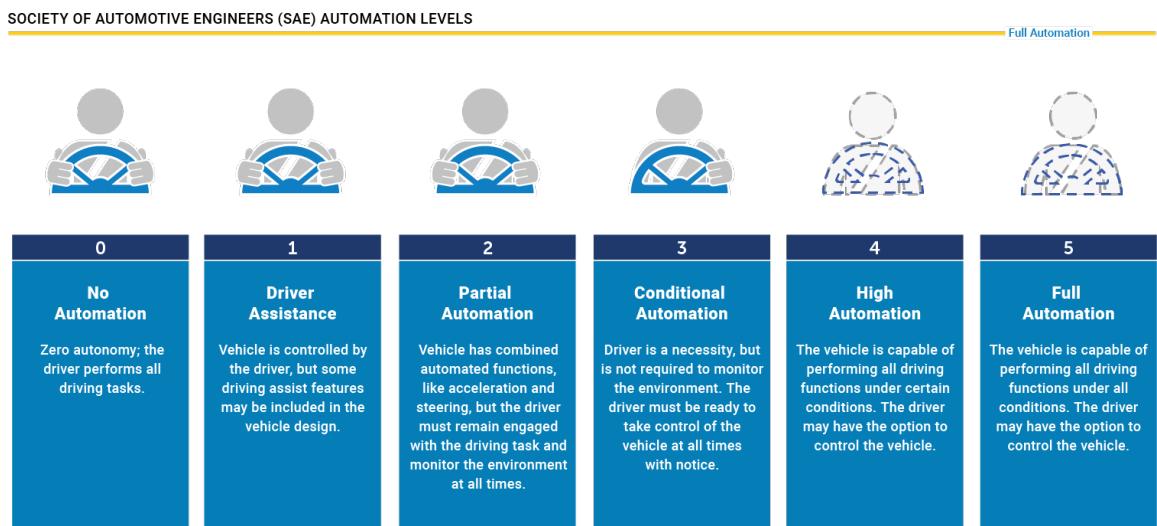


Figura 1.2: livelli di automatizzazione veicoli definiti dalla SAE.

Per soddisfare questi casi d'uso, esistono due tipi di mappe specializzate per la guida assistita e automatizzata: la mappa ADAS e le mappe HD a copertura ridotta, ma a più alta precisione.

Mappe ADAS

I veicoli che rientrano nei livelli 1 e 2 di automazione utilizzano le mappe ADAS per l'assistenza al guidatore e gli avvisi di sicurezza ad esempio per il line assist o per il controllo di velocità adattivo, esse sono quindi utilizzate per avere una guida più sicura e confortevole.

Mappe HD

Quando si passa dalla guida assistita alla piena autonomia, ovvero ai Livelli 2, 3 e oltre, la mappa HD diventa la soluzione migliore per una guida autonoma sicura. Sebbene estremamente accurate, le mappe ADAS raggiungono il loro limite quando sono richiesti maggiore precisione e livello di dettaglio. Le mappe HD invece aumentano la precisione a risoluzioni di pochi centimetri. Utilizzando informazioni dettagliate come limiti di velocità a livello di corsia, oggetti statici ad esempio segnali stradali e arredi stradali, edifici e geometria della corsia, le mappe HD consentono ai veicoli automatizzati e autonomi di diventare context-aware, ovvero di aver informazioni dettagliate rispetto alla loro posizione, all'ambiente e al percorso da seguire.

Obiettivi

Lo scopo di questo lavoro è di creare una pipeline in grado di inserire all'interno delle mappe di OpenStreetMap point cloud che descrivono le facciate degli edifici e in seguito poterle recuperare in modo da aumentare il grado di informazione disponibile, affinché le informazioni aggiunte possano essere utilizzate per migliorare la localizzazione del veicolo sulla corsia, il risultato verrà infatti processato dal sistema *Road Layout Estimation* (RLE) sviluppato all'interno del laboratorio IRA dell'Università di Milano-Bicocca, in particolare dal componente *Building Detector*[1], per cercare di fornire un'ipotesi di localizzazione più accurata rispetto alla scena percepita da un veicolo. Il diagramma di flusso seguente descrive il lavoro che è stato sviluppato:

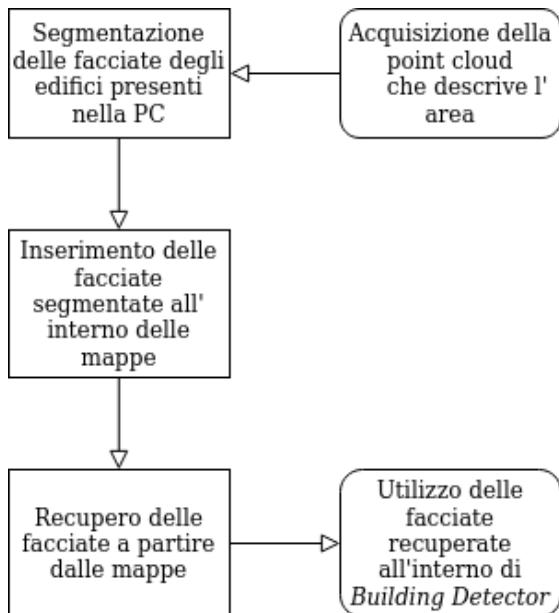


Figura 1.3: Pipeline implementata

Il progetto consiste quindi in tre parti ben distinte ma ciascuna necessaria per ottenere il risultato finale:

1. La definizione di una algoritmo di computer vision che permetta di segmentare le singole facciate degli edifici che possano essere utilizzate dalla seconda parte.
2. La modifica delle mappe per integrare i dati aggiunti e la messa in opera di un servizio web che metta a disposizione le mappe modificate.

3. La creazione di un servizio in grado di scaricare e mettere a disposizione le facciate integrate all'interno delle mappe.

Nel primo capitolo è stato utilizzato PCL per scrivere l'algoritmo di segmentazione degli edifici attraverso opportuni filtri effettuati sulla point cloud in ingresso.

Nella secondo capitolo è stato utilizzato Docker e le api di OSM per poter mettere a disposizione le mappe modificate con l'inclusione delle point cloud delle facciate.

Nella terzo capitolo invece si è interagito con RLE e si è quindi utilizzato ROS per poter scrivere un servizio di retrieval delle point cloud.

Ogni parte sarà trattata in modo più approfondito nei rispettivi capitoli.

Dataset

I dati utilizzati per testare questo progetto di tesi sono messi a disposizione dal KAIST Urban Dataset[8]. Questo dataset fornisce dati LiDAR (Light Detection and Ranging) e immagini stereo uniti a vari sensori di posizione che descrivono un ambiente urbano molto complesso, vengono forniti i dati di LiDAR 2D e 3D, e i dati che descrivono la navigazione del veicolo. Per comodità, gli strumenti di sviluppo sono forniti nell'ambiente Robot Operating System (ROS).

Sistema di acquisizione

La piattaforma utilizzata per acquisire i dati dei sensori è una Toyota Prius (1.4). Il veicolo è dotato di due LiDAR 2D e due 3D per raccogliere dati sull'ambiente circostante. Il 3D LiDAR è stato montato con un angolo di 45 gradi per la massima copertura. Nel caso del LiDAR 2D, il sensore posteriore è stato collocato verso il basso per informazioni sulla strada e il sensore anteriore è invece direzionato verso l'alto per informazioni sugli edifici. La telecamera stereo è stata installata di fronte alla parte anteriore del veicolo. (1.5) Inoltre, sono stati collocati vari sensori per stimare la posizione del veicolo. GPS e VRS GPS sono stati installati per stimare la posizione globale. Il GPS consente di immagazzinare informazioni generiche sulla posizione, al contrario il VRS GPS stima accuratamente la posizione del veicolo. Inertial Measurement Unit (IMU) e Fibre Optic Gyro (FOG) sono stati montati per misurare l'odometria del veicolo. L'IMU è un sensore che restituisce informazioni approssimative, il FOG fornisce invece valori estremamente più precisi rispetto all'IMU. Tutte le informazioni sui sensori sono fornite in formato raw con un timestamp. Prima del loro utilizzo i dati acquisiti sono stati fusi con uno script

sviluppato all'interno del laboratorio IRA in un'unica mappa contenente tutti i valori provenienti dai sensori.



Figura 1.4: Auto utilizzata per acquisire i dati

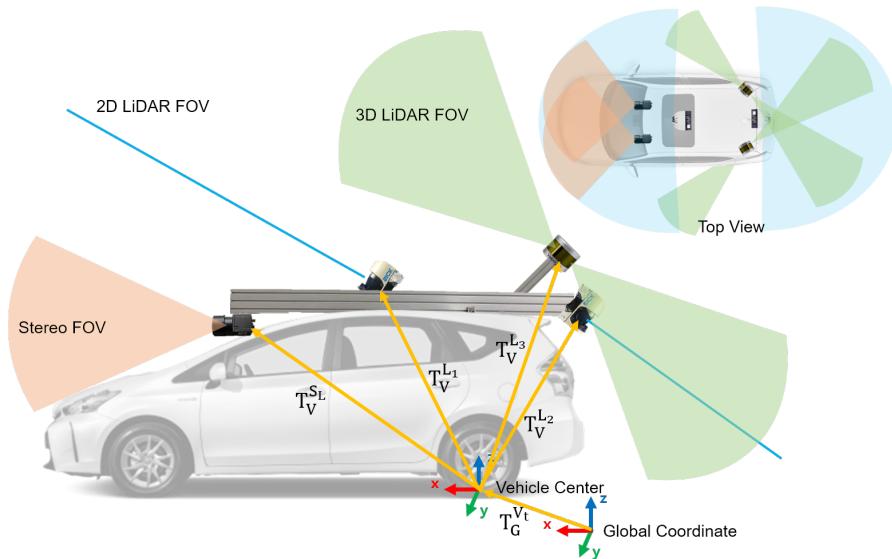


Figura 1.5: Vista dettagliata dei sensori utilizzati per acquisire i dati

Capitolo 2

Segmentazione Delle Facciate

In questo capitolo si descriverà come è stato affrontato il problema dell'estrazione delle facciate degli edifici a partire da una grande point cloud che rappresentava una vasta area. Per questo scopo è stata utilizzata Point Cloud Library.

2.1 PCL (Point Cloud Library)

Point Cloud Library [12] è un progetto open source per processare point cloud. Esso include e implementa un grande numero di algoritmi che consentono di filtrare, manipolare, segmentare ed estrarre features da point cloud. PCL è un progetto modulare, diviso in tante librerie che possono essere usate e compilate separatamente. L'albero delle librerie è il seguente:

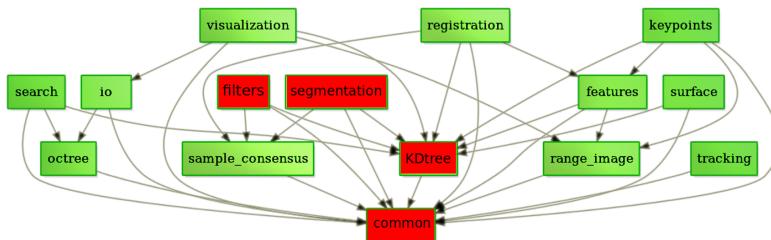


Figura 2.1: Grafo delle librerie di PCL, in rosso le librerie utilizzate

In particolare sono state utilizzate le librerie inerenti al filtraggio, alla segmentazione e all'utilizzo di alberi KDTree.

PCL è stata utilizzata per la segmentazione delle facciate degli edifici ovvero per andare ad estrarre le singole facciate a partire dalla point cloud della mappa. Si andrà ora a descrivere nel dettaglio il procedimento di segmentazione attuato.

2.2 Filtraggio in altezza

Per riuscire a trovare più facilmente le facciate si è deciso di effettuare un filtraggio della point cloud in modo da eliminare punti superflui come quelli del piano stradale o dei marciapiedi e ridurre così i tempi di calcolo.

Si è deciso di optare per un filtro sulle altezze strutturato nel seguente modo:

$$P_{cf}(x, y, z) = \{P(x, y, z) \in PC : h_{min} < z < h_{max}\} \quad (2.1)$$

Nel nostro caso si è deciso per filtrare tutti i punti con un'altezza inferiore ai 15 cm, considerandoli come manto stradale, e tutti i punti con un'altezza superiore ai 50m.

La condizione si è tradotta nel seguente frammento di codice:

```
pcl::PassThrough<pcl::PointXYZ> pass;
pass.setInputCloud (cloud);
pass.setFilterFieldName ("z");
pass.setFilterLimits (0.15 ,50);
pass.filter (*cloud_filtered);
```

Il risultato dell'operazione di filtraggio è il seguente:

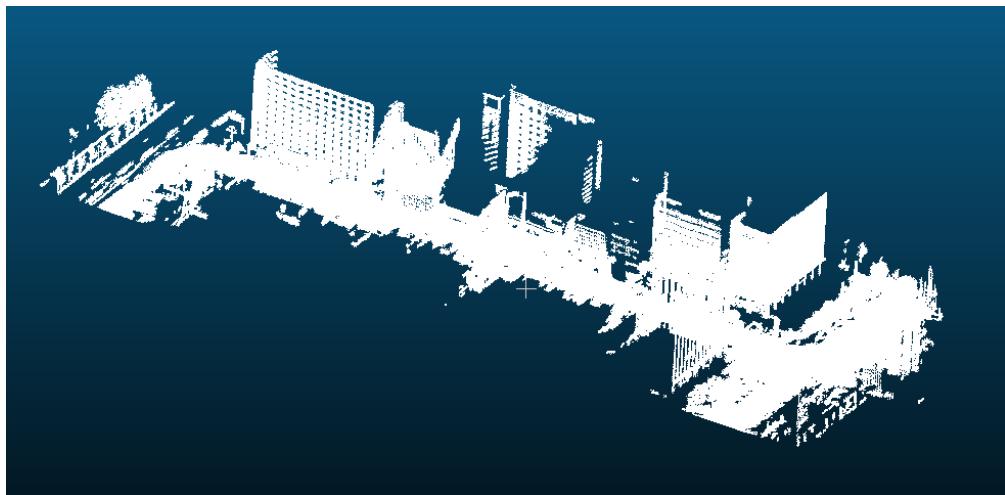


Figura 2.2: Point cloud prima del filtraggio



Figura 2.3: Point cloud dopo il filtraggio

Come si può notare sono stati rimossi tutti i punti relativi al manto stradale e ai marciapiedi.

2.3 Differenza di Normali

E' stata successivamente applicata una segmentazione attraverso l'utilizzo della differenza di normali [7].

L'operatore *DoN* si basa sull'uso delle normali stimate dalle superfici appartenenti ad una point cloud non organizzata ovvero in cui i punti non hanno un particolare ordine. L'operatore è definito come:

$$\Delta\hat{n}(p, r_s, r_l) = \frac{\hat{n}(p, r_s) - \hat{n}(p, r_l)}{2} \quad (2.2)$$

Dove

$$r_s \text{ e } r_l$$

sono rispettivamente i raggi di supporto della normale.

La motivazione principale che sta alla base dell'uso di questo operatore è data dall'osservazione del fatto che, dato un raggio di supporto, la normale viene stimata trovando il piano tangente alla componente principale dell'intorno dei punti appartenenti al raggio stesso.

In questo modo la normale riflette la geometria della superficie sottostante al punto in base alle dimensioni del raggio di supporto. Cambiando il raggio di supporto la direzione della normale cambia in base alla geometria della superficie sottostante (2.4).

Inoltre le normali sono caratteristiche che, all'interno di una point cloud, risultano

poco affette da rumore per la loro natura. Esse infatti vengono sempre stimate tramite un raggio di supporto, o un numero prefissato di vicini, caratteristica che premette di ridurre il peso di eventuali punti rumorosi.

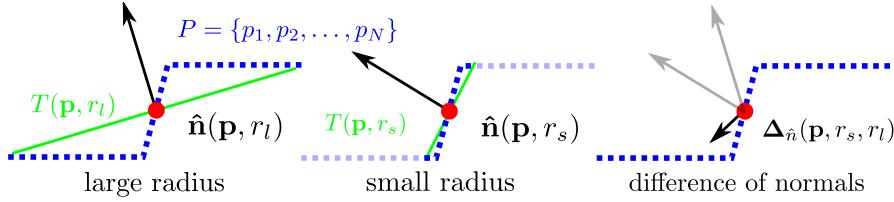


Figura 2.4: Effetti dei differenti raggi per la stima di una superficie

L'operazione può facilmente essere svolta con **PCL**:

```

pcl::NormalEstimationOMP<PointXYZ, PointNormal> ne;
ne.setInputCloud (cloud);
ne.setSearchMethod (tree);
ne.setViewPoint (std::numeric_limits<float>::max (), std::numeric_limits<
    float>::max (), std::numeric_limits<float>::max ());
pcl::PointCloud<PointNormal>::Ptr normals_small_scale (new pcl::PointCloud<
    PointNormal>);
ne.setRadiusSearch (scale1);
ne.compute (*normals_small_scale);
pcl::PointCloud<PointNormal>::Ptr normals_large_scale (new pcl::PointCloud<
    PointNormal>);
ne.setRadiusSearch (scale2);
ne.compute (*normals_large_scale);
pcl::DifferenceOfNormalsEstimation<PointXYZ, PointNormal,
PointNormal> don;
don.setInputCloud (cloud);
don.setNormalScaleLarge (normals_large_scale);
don.setNormalScaleSmall (normals_small_scale);

```

Come è stato evidenziato da dati sperimentali [7] sono stati scelti come raggi 0.8 e 8.0.

Il risultato dell'applicazione della DoN sulla pointcloud è il seguente:

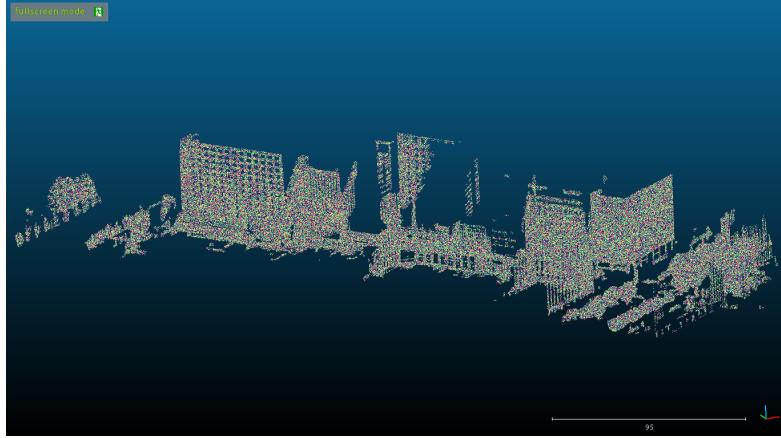


Figura 2.5: E' possibile vedere le diverse normali calcolate sulla point cloud

Si è passato poi al filtraggio della nuvola di punti generata dopo l'applicazione dell'operatore DoN (2.2) in modo da discriminare i punti appartenenti ad una facciata.

Dal momento che il risultato dell'applicazione dell'operatore DoN (2.2) è un vettore, si è scomposto nelle sue 3 componenti:

$$\begin{cases} \Delta\hat{n}(p, r_s, r_l)_x \in (-1, 1) \\ \Delta\hat{n}(p, r_s, r_l)_y \in (-1, 1) \\ \Delta\hat{n}(p, r_s, r_l)_z \in (-1, 1) \end{cases} \quad (2.3)$$

Si è poi evidenziato empiricamente che il miglior modo per filtrare i punti appartenenti alla stessa facciata è quello di prendere punti con $\Delta\hat{n}(p, r_s, r_l)_z$ e $\Delta\hat{n}(p, r_s, r_l)_y$ molto grandi e con $\Delta\hat{n}(p, r_s, r_l)_x$ piccola. Per tanto si è impostato un filtro che filtrasse la nuvola di punti rispettando le condizioni sopra imposte. Si è ottenuta quindi la seguente condizione:

```

pcl::ConditionOr<PointNormal>::Ptr range_cond (
    new pcl::ConditionOr<PointNormal> ()
);
range_cond->addComparison (pcl::FieldComparison<PointNormal>::ConstPtr (
    new pcl::FieldComparison<PointNormal> ("normal_z",
        pcl::ComparisonOps::GT, z_normthreshold))
);
range_cond->addComparison (pcl::FieldComparison<PointNormal>::ConstPtr (
    new pcl::FieldComparison<PointNormal> ("normal_y",
        pcl::ComparisonOps::GT, y_normthreshold))
);

pcl::ConditionAnd<PointNormal>::Ptr range1_cond (
    new pcl::ConditionAnd<PointNormal> ()
);
range1_cond -> addComparison (pcl::FieldComparison<PointNormal>::ConstPtr (

```

```
        new pcl::FieldComparison<PointNormal> ("normal_x",
                                                pcl::ComparisonOps::LT, x_normthreshold))
    );
range1_cond -> addCondition(range_cond);

pcl::ConditionalRemoval<PointNormal> condrem;
condrem.setCondition (range1_cond);
condrem.setInputCloud (doncloud);
pcl::PointCloud<PointNormal>::Ptr doncloud_filtered (new pcl::PointCloud<
    PointNormal>);

condrem.filter (*doncloud_filtered);
doncloud = doncloud_filtered;
```

Il risultato ottenuto è il seguente:

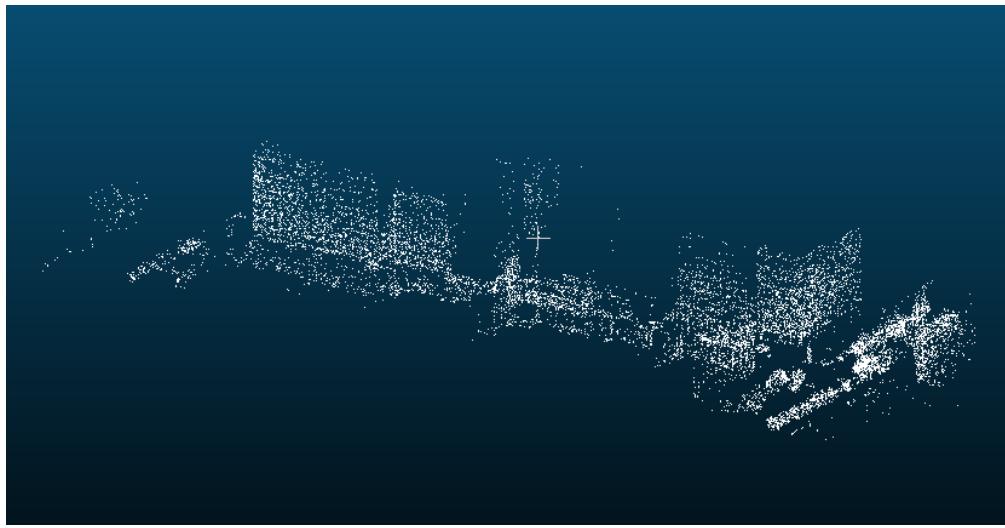


Figura 2.6: E' possibile notare come gli alberi e altre strutture non appartenenti alle facciate degli edifici siano state eliminate

2.4 Clustering

L'operazione finale che è stata applicata per poter ottenere le facciate dei singoli edifici è stata quella del clustering.

E' stato scelto di applicare un algoritmo di clustering basato sulla distanza euclidea tra i punti nello spazio [14]. L'algoritmo procede nel modo seguente:

Algorithm 1 Funzionamento dell'algoritmo di clustering

```

 $C = \emptyset, P, Q$ 
for  $p_i \in P$  do
     $Q \leftarrow p_i$ 
    for all  $p_i \in Q$  do
        for  $r < d_{th}$  do
             $P_k^i \leftarrow p_i$ 
        end for
        for  $p_k^i \in P_k^i$  do
            if  $p_k^i \notin Q$  then
                 $Q \leftarrow p_k^i$ 
            end if
        end for
    end for
     $C \leftarrow Q$ 
     $Q = \emptyset$ 
end for
return  $C$ 

```

Dove P è la point cloud da clusterizzare, C è una lista contenente i cluster e Q è una coda che contiene i vari punti da processare, d_{th} indica la distanza massima a cui cercare da ciascun punto ed r è il raggio della sfera costruita a partire da un punto p_i .

L'algoritmo va a cercare tutti i punti vicini ad un altro all'interno di un raggio prestabilito e li inserisce dentro lo stesso cluster, il funzionamento è similare a quello di k-Nearest Neighbour [4]. **PCL** consente di implementare facilmente questo tipo di clustering attraverso la classe *pcl:EuclideanClusterExtraction class* contenuta all'interno di **PCL**, questa classe consente inoltre di specificare la dimensione minima e massima di un cluster. Attraverso varie prove si è ottenuto che le dimensioni ideali siano rispettivamente di 300 e di 1,000,000 di punti. In codice ciò viene tradotto come:

```

pcl::search::KdTree<PointNormal>::Ptr segtree (new pcl::search::KdTree<
    PointNormal>);
segtree->setInputCloud (doncloud);
std::vector<pcl::PointIndices> cluster_indices;
pcl::EuclideanClusterExtraction<PointNormal> ec;
ec.setClusterTolerance (segradius);
ec.setMinClusterSize (300);
ec.setMaxClusterSize (1000000);
ec.setSearchMethod (segtree);
ec.setInputCloud (doncloud);
ec.extract (cluster_indices);

int j = 0;
for (std::vector<pcl::PointIndices>::const_iterator it = cluster_indices.begin
    (); it != cluster_indices.end (); ++it, j++)
{
    pcl::PointCloud<PointNormal>::Ptr cloud_cluster_don (new pcl::PointCloud<
        PointNormal>);
    for (std::vector<int>::const_iterator pit = it->indices.begin (); pit != it->
        indices.end (); ++pit)
    {
        cloud_cluster_don->points.push_back (doncloud->points[*pit]);
    }
    cloud_cluster_don->width = int (cloud_cluster_don->points.size ());
    cloud_cluster_don->height = 1;
    cloud_cluster_don->is_dense = true;

    stringstream ss;
    ss << "don_cluster_" << j << ".pcd";
    writer.write<pcl::PointNormal> (ss.str (), *cloud_cluster_don, false);
}

```

Abbiamo quindi diversi cluster ciascuno corrispondente ad una diversa facciata di ogni edificio.

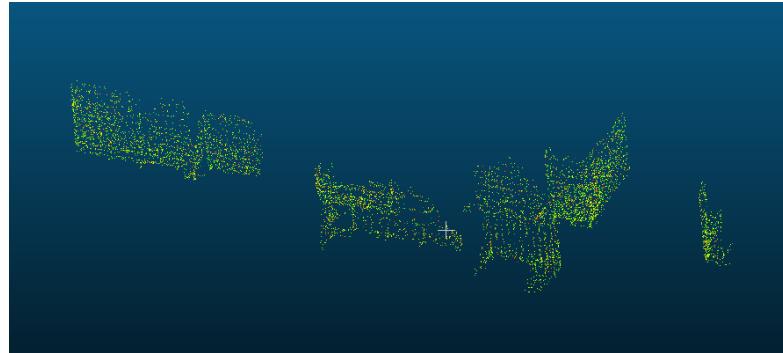


Figura 2.7: E' possibile vedere i cluster ottenuti rappresentanti le varie facciate degli edifici

I Cluster ottenuti verranno poi utilizzati per l'inserimento all'interno delle mappe di **OpenStreetMaps**.

Capitolo 3

Inserimento in OpenStreetMaps

In questo capitolo si descriverà come sono state inserite le point cloud all'interno delle mappe e di come queste sono state messe a disposizione per l'utilizzo.

3.1 OpenStreetMap

OpenStreet Map[10] è un'iniziativa per creare e fornire dati geografici a chiunque gestita dalla OpenStreetMap Foundation. Essa permette di contribuire, accedere ai dati e ospitare una propria infrastruttura privata senza la necessità di appoggiarsi a infrastrutture chiuse e proprietarie. OpenStreetMap è un database che utilizza un proprio formato di scambio dati basato su XML, tutti gli elementi che possono essere inseriti e che modellano la realtà sono di quattro tipologie:

1. punto (node): singolo punto. E' utilizzato per indicare oggetti puntuali.



Figura 3.1: OSM Logo



Figura 3.2: Node

2. linea (way): un insieme di punti non chiuso, il percorso è un segmento tra punti che può descrivere oggetti che seguono una linea come vie, muri e simili.

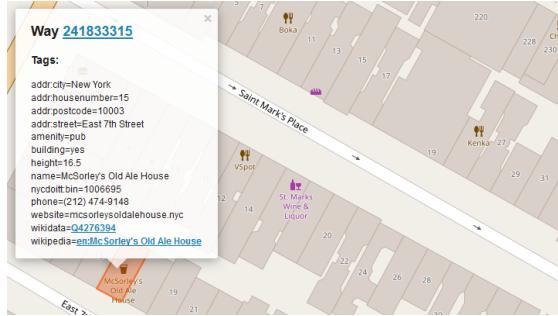


Figura 3.3: Way

3. area (polygon): una linea chiusa ovvero un insieme di segmenti che si chiudono, è utilizzato per rappresentare oggetti che hanno un qualche tipo di area.
4. relazione (relation): è un insieme degli elementi scritti sopra, esso consente di rappresentare oggetti complessi composti da un insieme di elementi.

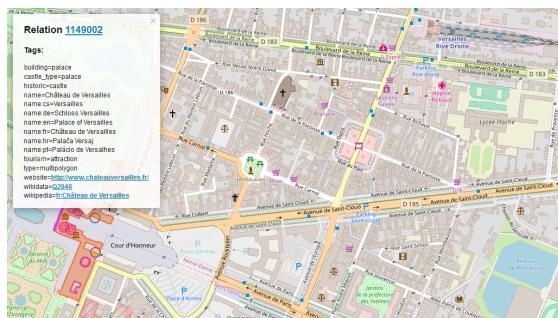


Figura 3.4: Relation

Nel progetto si sono manipolati elementi di tipo way e relation. Per descrivere gli oggetti vengono utilizzate delle etichette (tag) che sono codificate secondo uno standard internazionale e sono composte da una tupla di elementi consistenti in una chiave (key) e un valore (value). Ogni oggetto dev'essere descritto da almeno un tag principale per identificarlo ed è inoltre possibile aggiungerne di secondari che consentono di descrivere particolari proprietà. Il concetto di etichetta è stato utilizzato all'interno del progetto per consentire di aggiungere una chiave che possa descrivere la facciata di un edificio.

3.2 JOSM

JOSM[9] è un editor per OpenStreetMap(OSM) Open Source licenziato sotto GPL estensibile scritto in Java. Supporta l'editing avanzato delle mappe senza le limitazioni date dall'interfaccia web di OSM. Consente inoltre il caricamento di tracciati GPX, l'utilizzo di immagini di background, la modifica dei dati di OSM da sorgenti locali oltre che da sorgenti online. Consente inoltre di modificare tutte le tipologie di dato associate ad OSM e i loro tag.

JOSM è stato scelto per la capacità di modificare in modo avanzato le mappe in particolare per poter modificare un'istanza locale di OSM e non dover modificare le mappe globali.



Figura 3.5: JOSM Logo

3.3 Docker

Docker[5] è un progetto open-source nato per automatizzare il deployment di applicazioni all'interno di contenitore che in Docker si chiama Container, fornendo un livello ulteriore di astrazione grazie alla virtualizzazione a livello di sistema operativo concesso da Linux.

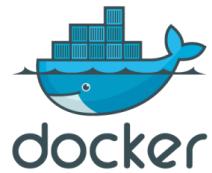


Figura 3.6: Docker Logo

Docker utilizza le funzionalità di isolamento delle risorse del kernel Linux, come cgroup e namespace per consentire a container indipendenti di coesistere sulla stessa istanza di Linux, evitando l'installazione e la manutenzione di una macchina virtuale (VM): i namespace del kernel Linux isolano ciò che l'applicazione può vedere dell'ambiente operativo, incluso l'albero dei processi, la rete, gli ID utente e i file system montati, mentre i cgroup forniscono l'isolamento delle risorse, inclusa la CPU, la memoria, i dispositivi di I/O a blocchi e la rete.

Docker accede alle funzionalità di virtualizzazione del kernel Linux o direttamente utilizzando la libreria libcontainer, che è disponibile da Docker 0.9, o indirettamente attraverso libvirt, LXC o systemd-nspawn. I container condividono lo stesso kernel, ma ciascuno di essi può dover utilizzare una certa quantità di risorse, come la CPU, la memoria e l'I/O.

3.4 Inserimento nelle mappe

Per associare i singoli cluster agli edifici si è deciso di intervenire andando ad aggiungere una nuova etichetta agli edifici che presentavano il tag *building*. Il tag aggiunto è denominato *building:facade:pcl*, ad esso viene associato come valore l'url presso cui è salvato l'oggetto pcd che contiene la facciata dell'edificio. L'aggiunta del tag è stata fatta manualmente attraverso l'utilizzo di JOSM, come si può vedere nella figura successiva.

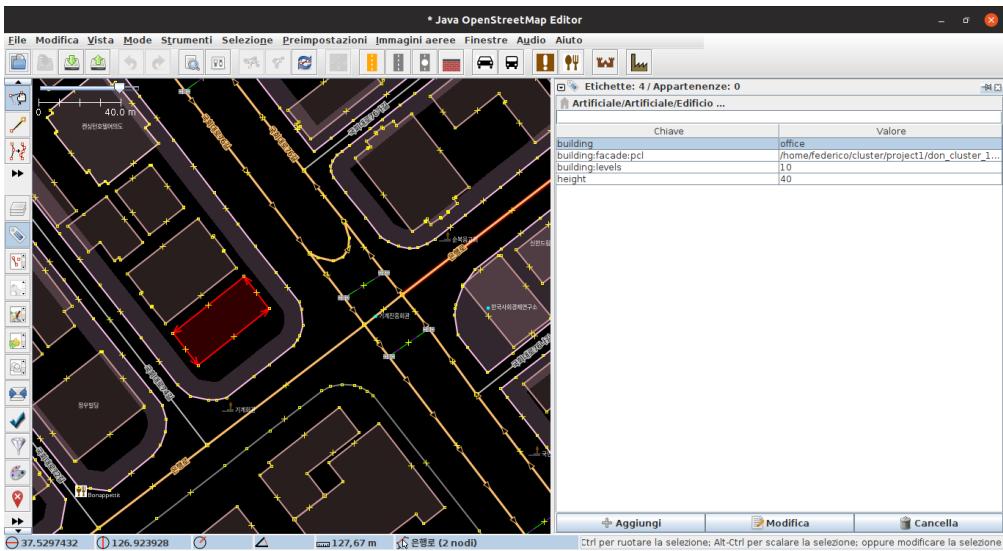


Figura 3.7: Esempio di modifica di un edificio con JOSM

Nel pannello di sinistra si può vedere il tracciato selezionato di tipo *polygon* in quanto linea chiusa. Nel pannello di destra si possono invece evidenziare i tag associati allo specifico poligono, in particolare si può vedere che esso è un edificio e che vi è associato uno specifico cluster oltre che a dati di importanza secondaria come l'altezza dell'edificio o il numero di piani. Il procedimento di associazione è stato ripetuto per ogni cluster precedentemente estratto, la mappa modificata è stata infine esportata per essere servita.

3.4.1 Overpass API

Per consentire l'estrazione delle informazioni si è deciso di utilizzare Overpass[11].

Overpass API è un'API read-only che permette di servire i dati di regioni specifiche della mappa di OSM, funziona come un database, il client lancia una query

all'API e riceve i dati che corrispondono alla query.

A differenza dell'API principale di OSM che è ottimizzata per l'editing, Overpass è ottimizzata per fornire pochi elementi in pochi secondi o fino a 10 milioni di elementi in pochi minuti entrambi filtrati da specifici criteri di ricerca. Funziona come backend per vari servizi.

Dal momento che non si voleva utilizzare la mappa globale ma una nostra versione modificata è stato necessario ospitare in locale un server Overpass, per poter far fronte alle difficoltà dovute all'utilizzo di versioni di librerie diverse e incompatibili tra loro si è deciso di utilizzare Docker che permette di fare il deploy del server in un container portatile ed isolato.

Il container è stato costruito a partire da un container già esistente[6] al quale sono state apportate modifiche per ottenere il seguente listato:

```

FROM ubuntu:16.04
RUN apt-get update
RUN apt-get install -y apache2 vim
RUN apt-get install -y \
    autoconf \
    automake1.11 \
    expat \
    git \
    g++ \
    libtool \
    libexpat1-dev \
    make \
    zlib1g-dev \
    bzip2 \
    wget \
    liblz4-1 liblzf4-dev
RUN apt-get clean && rm -rf /var/lib/apt/lists/*
RUN git clone https://github.com/drolbr/Overpass-API.git
WORKDIR /Overpass-API
#Checkout latest version
RUN git checkout $(git describe --abbrev=0 --tags)
#Configure
WORKDIR /Overpass-API/src
RUN \
    autoscans && \
    aclocal-1.11 && \
    autoheader && \
    libtoolize && \
    automake-1.11 --add-missing && \
    autoconf
#Compile
RUN \
    ./configure --enable-lz4 CXXFLAGS="-O2" --prefix="`pwd`" && \
    make -j $(nproc --all)
COPY vhost_apache.conf /etc/apache2/sites-available
RUN a2enmod ext_filter cgi

```

```

RUN a2dissite 000-default.conf
RUN a2ensite vhost_apache.conf
WORKDIR /
COPY *.sh /
ADD www /www
RUN useradd overpass_api
CMD ["/run.sh"]
USER root
RUN chmod 777 /Overpass-API/src/bin/*
VOLUME "/overpass_DB"
EXPOSE 80

```

Il Dockerfile può essere diviso in 4 parti:

1. Il pull dell'immagine dal dockerhub e l'installazione delle dipendenze necessarie.
2. Il recupero del sorgente di Overpass API e la sua compilazione.
3. La configurazione di apache per poter servire l'API.
4. La definizione degli script da eseguire nel container e il binding con volumi che risiedono sul filesystem dell'ospite.

Il binding è molto importante in quanto si è dovuto generare prima dell'esecuzione un database a cui l'API potesse accedere partendo dalla mappa personalizzata esportata in precedenza.

Dopo aver generato il DB di overpas attraverso degli strumenti dedicati messi a disposizione dal progetto OpenStreetMap, il container è stato lanciato nel seguente modo:

```
docker run -d --restart=always -v ~/overpass_DB:/overpass_DB
-p 5001:80 overpass-api
```



Figura 3.8: Overpass API in funzione

E' ora possibile sottomettere query all'API utilizzando Overpass QL. Grazie a Overpass QL possiamo andare ad effettuare query mirate, circoscritte all'interno di bounding box specifiche e andando a chiedere che il risultato contenga delle proprietà specifiche Una query può essere composta da:

- Un'istruzione di query che inizia con "*node*", "*way*", "*relation*", "*rel*", "*nwr*" (*node+way+relation*).
- Un' istruzione di output che inizia con "*out*".

Le query consistono nel tipo di oggetto che dev'essere trovato:

node
way
rel

seguito da almeno una condizione che deve descrivere l' oggetto da trovare ad esempio:

["name"="Milano"]

In questo modo è possibile filtrare i singoli tag, cosa necessaria per il nostro progetto. Infatti è stata costruita la seguente query:

```
[out:json];
(way["building:facade:pcl"]
(37.52871919454917,
 126.91968441009521,
 37.531458875784935,
```

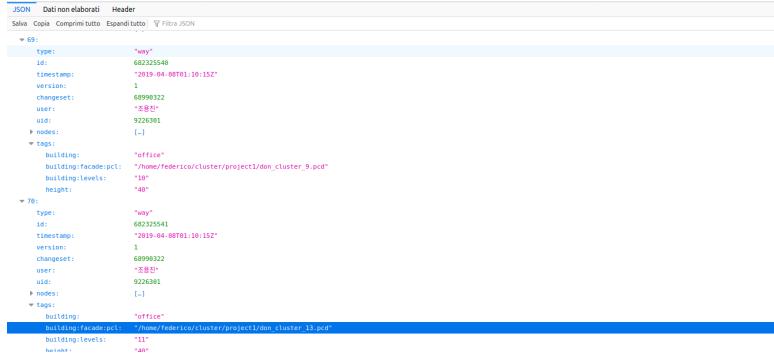
```

    126.9231390953064)););
(. -; >%;););
out meta;

```

Listing 3.2: Query di Overpass QL

Ciò consente di restituire tutti le way (e quindi anche i polygon) presenti all'interno del distretto di Yeongdeungpo a Seoul contenenti il tag "building:facade:pcl".



The screenshot shows a JSON editor interface with tabs for 'JSON', 'Dati non elaborati', and 'Header'. The 'Header' tab is selected. The JSON data is displayed in a tree view:

```

{
  "type": "way",
  "id": "68235548",
  "timestamp": "2019-04-08T01:10:15Z",
  "version": 1,
  "changeset": "6899322",
  "user": "federico",
  "uid": 9225391,
  "nodes": [...],
  "tags": {
    "building": "office",
    "building:facade:pcl": "/home/federico/cluster/project1/don_cluster_9.pcd",
    "building:levels": "18",
    "height": "48"
  }
}

{
  "type": "way",
  "id": "68235541",
  "timestamp": "2019-04-08T01:10:15Z",
  "version": 1,
  "changeset": "6899322",
  "user": "federico",
  "uid": 9225391,
  "nodes": [...],
  "tags": {
    "building": "office",
    "building:facade:pcl": "/home/federico/cluster/project1/don_cluster_11.pcd",
    "building:levels": "11",
    "height": "48"
  }
}

```

The second way object is highlighted with a blue background, indicating it contains the tag 'building:facade:pcl'.

Figura 3.9: Come si può vedere si trovano way con il tag building:facade:pcl presente

Il risultato della query verrà poi utilizzato per poter recuperare le point cloud quando sarà necessario il loro utilizzo durante il recupero da parte del componente di RLE dedicato.

Capitolo 4

Recupero e Utilizzo delle Point Cloud

In questo capitolo si descriverà come è stato scritto un servizio di retrieval delle pointcloud a partire dalle mappe e di come queste vengano poi utilizzate.

4.1 ROS (Robotic Operating System)

ROS è un framework flessibile open source che nasce per lo sviluppo e la programmazione dei robot ed è infatti molto utilizzato nell'ambito della robotica di servizio. Il software è liberamente disponibile dal sito ufficiale[13]. Le peculiarità del framework sono simili a quelle di un qualsiasi sistema operativo:

1. Capacità di fornire astrazione dell'hardware.
2. Modularità (cioè la possibilità di avere diverse parti del sistema che lavorano autonomamente al fine di raggiungere un obiettivo comune).
3. Possibilità di controllare i dispositivi tramite driver.
4. Possibilità di gestione delle applicazioni (package).

ROS è costituito fondamentalmente da una rete di processi che vengono eseguiti in parallelo e che comunicano tra di loro sfruttando un architettura Peer-to-Peer. Essi vengono comunemente chiamati *Nodi* e costituiscono uno degli elementi principali del sistema. Per essere eseguiti è necessario, innanzitutto, avviare un server principale tramite comando *roscore*. Denominato ROS Master (o Core) consente di effettuare operazioni di routing permettendo ai processi di comunicare tra loro. Nella comunicazione, i *Nodi* si scambiano *messaggi*. ROS fornisce una grande

quantità di messaggi predefiniti ma è ovviamente possibile crearne di nuovi a seconda delle proprie esigenze. Sostanzialmente un nodo è un programma che si interfaccia con il ROS Master per comunicare con altri nodi tramite messaggi. Il sistema di scambio di messaggi è molto simile all'architettura pub/sub: esistono nodi *talker* (Publisher) e nodi *listener* (Subscriber).

Per circolare i messaggi possono sfruttare *topic* o *servizi*:

- **Topic:** i nodi Publisher creano il Topic (canale di comunicazione) e su di esso pubblicano i messaggi, mentre i nodi Subscriber vi si sottoscrivono per poterne ricevere il contenuto. Più nodi possono sottoscriversi allo stesso topic e ogni nodo può pubblicare su uno o più topic contemporaneamente.

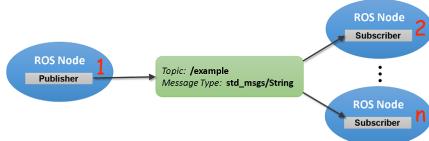


Figura 4.1: schema di comunicazione *Publisher-Subscriber* tramite topic

- **Servizi:** forniscono la possibilità di creare una comunicazione sincrona tra un nodo client, che può richiedere un servizio mandando una *request* ed un nodo server, che riceverà la richiesta e risponderà con una *response*.

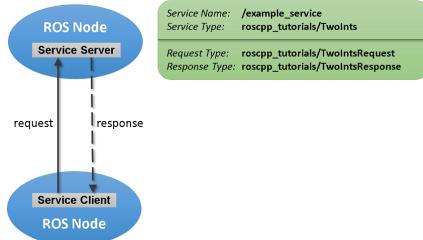


Figura 4.2: schema di comunicazione *Client-Server* tramite servizio

ROS in particolare nella versione Kinetic basata sulla versione 16.04 di ubuntu è alla base di RLE su cui si è andato a lavorare.

4.2 RLE (Road Layout Estimation)

Road Layout Estimation[2] è un framework probabilistico per l'interpretazione di scene in contesto urbano. Gli elementi da interpretare in una scena urbana possono

essere sia di natura *statica*, come edifici, cartelli stradali o incroci, sia di natura *dinamica*, come altri veicoli o persone.

Il framework è modulare e consente la localizzazione attraverso diverse fonti di date. I dati possono provenire da sensori fisici come camere, LiDAR e GPS, o da sensori virtuali ovvero moduli software che analizzano la scena e ne estraggono diverse caratteristiche.

A differenza dei framework già esistenti che prevedono un insieme finito di sensori definito a priori durante la fase di design, *RLE* si basa sull’assunzione che diverse informazioni da diverse fonti possono essere disponibili a frequenze diverse, assenti per un certo periodo, o addirittura utili solo in specifiche situazioni.

4.2.1 Architettura

L’approccio con cui *RLE* localizza il veicolo nella mappa si basa su un filtro definito particle filtering[15], la scena viene descritta attraverso delle *layout hypothesis*(LH). Le LH sono un insieme di elementi che descrivono lo stato del veicolo. In particolare danno informazioni riguardo:

- Lo *stato* vero e proprio del veicolo, espresso tramite pose e velocità in *6DoF*.
- Il vettore di *Layout Component*, o LC, ovvero i modelli geometrici o semantici associati agli elementi della scena.
- lo *score* del layout, ovvero una stima della verosimiglianza dell’ipotesi
- Il *modello di movimento*, che descrive come l’ipotesi evolve nel tempo.

I *layout component*(LC) sono alla base del processo di interpretazione della scena, ogni istanza di LC vive indipendentemente dalle altre, sia che appartengano alla stessa ipotesi o meno, per questo nell’implementazione di ognuna sono definite le funzioni che calcolano sia lo score che la propagazione del moto.

Ogni LC viene creato nel momento in cui è riconosciuto da uno specifico detector, che comunica poi al *Layout Manager* i dati di interesse. Il *Layout Manager* è il nodo principale del sistema, e il suo compito è quello di gestire tutte le diverse ipotesi di struttura della scena nel tempo. Per fare questo assegna a ciascuna ipotesi un punteggio su quanto essa risulti essere verosimile. Di seguito è possibile vedere lo schema ad alto livello di funzionamento del framework.

Nel lavoro ci si è concentrati sull’estensione di IRA OpenStreetMaps e di IRA Building Detector[3] per permettere il recupero e l’utilizzo delle point cloud salvate all’interno delle mappe.

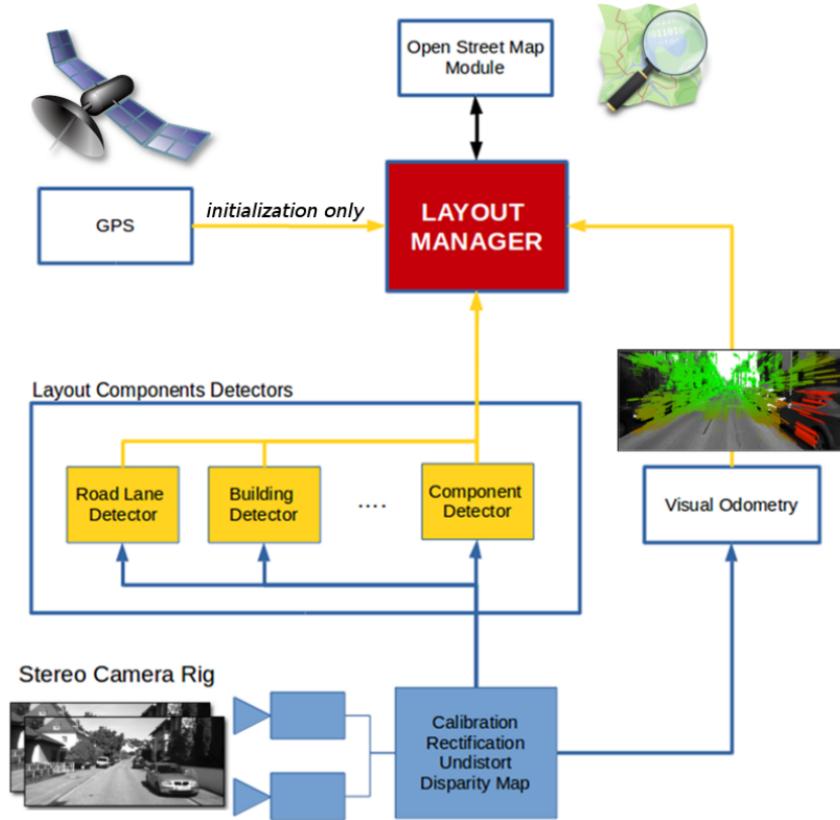


Figura 4.3: Schema ad alto livello del funzionamento di RLE

4.3 IRA OpenStreetMap

IRA OpenStreetMap è un componente di RLE che consente di caricare le mappe per poi utilizzarle successivamente durante la navigazione, esso è un fork di ROS Open_street_map con alcune sostanziali modifiche per il suo utilizzo all'interno di RLE.

Esso è stato modificato in un suo nodo che si occupa dell'acquisizione della mappa. Ad esso sono state fatte delle modifiche riguardanti:

1. La selezione della sorgente da cui scaricare la mappa.
2. La possibilità di scaricare solo gli edifici con una point cloud associata.

Per quanto riguarda la prima parte il nodo preesistente consentiva il download soltanto dall'API Overpass globale è stato per tanto necessario inserire la possibilità di utilizzare l'API locale che è stata descritta nel capitolo precedente.

Si è quindi aggiunto un argomento booleano *local* al nodo che consente di selezionare su quale server andare ad effettuare la query:

```
#Aggiunta di un nuovo parametro booleano
TCLAP::SwitchArg localArg("l","local","use local server", cmd, false);
bool local = localArg.getValue();
```

Si è poi andato ad introdurre un selettore per selezionare l'url da utilizzare.

```
if (local){
    osm_api_url = "http://gas2008-server.disco.unimib.it:5001/api/
        interpreter?data=( " + highwaysQuery + buildingQuery + paramQuery +
        ");(.;>);out%20meta;" ;
} else {
    osm_api_url = "http://www.overpass-api.de/api/interpreter?data=( " +
        highwaysQuery + buildingQuery + paramQuery + ");(.;>);out%20meta
        ;" ;
}
```

Il secondo punto invece implicava l'inserimento di un nuovo parametro per la scelta di scaricare tutti gli edifici o solamente gli edifici che avevano una point cloud associata. Esso è risultato nell'implementazione all'interno del nodo della query di Overpass API descritta precedentemente 3.2, aggiunta in modo molto semplice, dopo aver definito il nuovo parametro booleano *buildingPCL* in modo analogo a sopra, con il seguente frammento di codice:

```
if (buildingPCL)
{
    buildingQuery = "node[building:facade:pcl]" + area + ";" +
    "way[building:facade:pcl]" + area + ";" +
    "relation[building:facade:pcl]" + area + ";" ;
}
```

E' stato poi necessario implementare un *service* ROS che andasse a recuperare le point cloud ottenute dagli edifici, esso si comporta in modo similare al *service* *getNearBuildings* già esistente dentro IRA OpenStreetMap che invece di restituire un messaggio di tipo *geometry_msgs/Point* ovvero una lista di punti restituisce un messaggio di tipo *sensor_msgs/PointCloud2* ovvero la point cloud che descrive la facciata richiesta in quel momento.

Il servizio, denominato *getNearestBuildingFacade*, è definito come segue:

```
#request fields (float64 = C++ double)
float64 latitude
float64 longitude
float64 theta
float64 radius
---
```

```
#response fields (float64 = C++ double)
sensor_msgs/PointCloud2 facadecloud
```

Il servizio prende in output latitudine, longitudine correnti e i raggi a cui ricercare gli edifici e ritorna la point cloud dell'edificio più vicino, il funzionamento è il seguente:

4.3.1 Recupero delle way

```
Xy center = latlon2xy_helper(req.latitude, req.longitude);
vector<shared_ptr<Osmium::OSM::Way const>> way_vector;
for(std::set<shared_ptr<Osmium::OSM::Way const>>::iterator way_itr = oh.
    m_ways.begin(); way_itr != oh.m_ways.end(); way_itr++)
{
    const char* building_tag = (*way_itr)->tags().get_value_by_key("building:
        facade:pcl");
    if (!building_tag)
        continue;
    Osmium::OSM::WayNodeList waylist = (*way_itr)->nodes();
    for(Osmium::OSM::WayNodeList::iterator node_list_itr = waylist.begin();
        node_list_itr != waylist.end(); node_list_itr++)
    {
        double lat = node_list_itr->position().lat();
        double lon = node_list_itr->position().lon();
        Xy way_node_xy = latlon2xy_helper(lat, lon);
        double distance = get_distance_helper(center.x, center.y, way_node_xy.
            x, way_node_xy.y);
        if(distance <= req.radius)
        {
            way_vector.push_back(*way_itr);
            break;
        }
    }
}
if(way_vector.size() == 0){
    ROS_ERROR_STREAM("      No map nodes found next to particle. Distance
        radius: " << req.radius << " m");
    return false;
}
```

Questa prima parte è molto simile al servizio già esistente, essa consente di andare a trovare e salvare tutte le way in un raggio ben definito rispetto alla posizione corrente, la differenza rispetto al servizio per il recupero dei generici edifici è che si va a filtrare non solo sul tag *building* che indica se la way è un edificio o meno ma si utilizza il più specifico tag *building:facades:pcl* che indica se un edificio ha una point cloud associata o meno.

4.3.2 Recupero delle point cloud

```

1  for(vector<shared_ptr<Osmium::OSM::Way const> >::iterator way_itr = way_vector.
2      begin(); way_itr != way_vector.end(); way_itr++)
3  {
4      Osmium::OSM::WayNodeList way_n_list = (*way_itr)->nodes();
5      for(Osmium::OSM::WayNodeList::iterator node_list_itr = way_n_list.begin();
6          node_list_itr != way_n_list.end() - 1; node_list_itr++)
7      {
8          const char* facadepclurl = obj.tags().get_value_by_key("building:facade:
9              pcl");
10         if (facadepclurl) {
11             resource_retriever::Retriever r;
12             resource_retriever::MemoryResource resource;
13             try
14             {
15                 resource = r.get(facadepclurl);
16             }
17             catch (resource_retriever::Exception& e)
18             {
19                 ROS_ERROR("Failed to retrieve file: %s", e.what());
20             }
21             FILE* f = fopen("cloud.pcd", "w");
22             fwrite(resource.data.get(), resource.size, 1, f);
23             fclose(f);
24
25             sensor_msgs::PointCloud2 output;
26             pcl::PointCloud<pcl::PointXYZ> cloud;
27             pcl::io::loadPCDFile ("cloud.pcd", cloud);
28             pcl::toROSMsg(cloud, output);
29             resp.facadecloud = output;
30         }
31     }
32 }
```

La seconda parte, che si occupa del recupero delle singole facciate rispetto alle way trovate nella parte precedente, può essere suddivisa ulteriormente in 2 parti:

1. La parte dalla riga 7 alla riga 20 si occupa del download della point cloud dall'url a cui punta il valore del tag *buildings:facades:pcl*
2. La parte dalla riga 20 fino alla fine si occupa invece della pubblicazione della PC come messaggio ROS

Il servizio viene poi messo a disposizione per poter essere chiamato dal modulo di building detection[1] che effettuerà un confronto tra la point cloud che riceve come messaggi dai vari sensori disponibili (LiDAR o Camere) e la point cloud di riferimento all'interno delle mappe.

All'interno di ROS è possibile visualizzare la struttura del servizio che è la seguente:

```
lodovic@gas-2008-server:~/catkin_ws/devel$ rossrv show pcd_service/getNearBuildingsPCL
float64 latitude
float64 longitude
float64 theta
float64 radius
...
sensor_msgs/PointCloud2 facadecloud
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
    uint32 height
    uint32 width
    sensor_msgs/PointField[] fields
      uint8 INT8=1
      uint8 UINT8=2
      uint8 INT16=3
      uint8 UINT16=4
      uint8 INT32=5
      uint8 UINT32=6
      uint8 FLOAT32=7
      uint8 FLOAT64=8
      string name
      uint32 offset
      uint8 datatype
      uint32 count
      bool is_bigendian
      uint32 point_step
      uint32 row_step
      uint8[] data
      bool is_dense
```

Figura 4.4: Struttura del servizio di richiesta delle point cloud

Il servizio è stato poi testato facendo delle richieste grazie a *rosservice call* che permette di interrogare i servizi ros

```
facadecloud:
  header:
    seq: 0
    stamp:
      secs: 0
      nsecs: 0
    frame_id: ''
  height: 1
  width: 1507
  fields:
    -
      name: "x"
      offset: 0
      datatype: 7
      count: 1
    -
      name: "y"
      offset: 4
      datatype: 7
      count: 1
    -
      name: "z"
      offset: 8
      datatype: 7
      count: 1
    is_bigendian: False
    point_step: 16
    row_step: 2412
  data: [159, 159, 124, 198, 196, 232, 108, 70, 113, 136, 95, 65, 0, 0, 128, 63, 102, 161, 124, 198, 185, 249, 108, 70, 177, 100, 36, 63, 0, 0, 128, 63, 161, 151, 124, 198, 213, 19, 109, 70, 97, 127, 4, 64, 0, 0, 128, 63, 57, 163, 124, 198, 46, 239, 108, 70, 26, 247, 228, 65, 0, 0, 128, 63, 119, 191, 124, 198, 222, 19, 109, 70, 56, 51, 8, 65, 0, 0, 128, 63, 109, 239, 124, 198, 211, 84, 109, 70, 134, 248, 39, 65, 0, 0, 128, 63, 228, 162, 124, 198, 140, 237, 108, 70, 76, 51, 95, 65, 0, 0, 128, 63, 09, 162, 124, 198, 228, 238, 108, 70, 138, 6, 5, 65, 0, 0, 128, 63, 185, 239, 124, 198, 57, 82, 109,
```

Figura 4.5: Si può notare che il servizio risponde con una pointcloud contenente la facciata dell'edificio corrente

Capitolo 5

Conclusioni

L’obiettivo del lavoro è stato quello di ottenere una pipeline funzionante che potesse mettere a disposizione, incorporandole nelle mappe, informazioni aggiuntive riguardanti le facciate degli edifici attraverso le point cloud che le descrivono.

Il sistema è risultato funzionante e facilmente riproducibile, il lavoro potrà quindi essere utilizzato per testare se esiste un miglioramento sostanziale dell’accuratezza di localizzazione attraverso l’utilizzo di mappe aumentate. Durante la prototipazione della pipeline sono stati fatti alcuni passaggi manuali che possono essere facilmente automatizzati con l’ausilio di script dedicati che potranno essere sviluppati in seguito, in particolare l’aggiunta del tag che rappresenta le facciate all’interno di OSM è facilmente automatizzabile attraverso l’utilizzo di librerie dedicate.

La parte di segmentazione ha dato risultati ottimali solo su specifici dataset con poche fonti di disturbo come alberi e segnali stradali, per tanto è necessario trovare e utilizzare un modello per la segmentazione delle facciate più robusto, in grado di funzionare in una maggior quantità di scene. Inoltre si potrebbero sottomettere le modifiche effettuate alle mappe ”globali” in modo da eliminare la necessità di utilizzarne una loro istanza locale, ciò ha tuttavia problemi legislativi ed è, al momento, impossibile.

Elenco delle figure

1.1	ADAS	3
1.2	livelli di automatizzazione veicoli definiti dalla SEA.	4
1.3	Pipeline implementata	5
1.4	Auto utilizzata per acquisire i dati	7
1.5	Vista dettagliata dei sensori utilizzati per acquisire i dati	7
2.1	Grafo delle librerie di PCL, in rosso le librerie utilizzate	8
2.2	Point cloud prima del filtraggio	9
2.3	Point cloud dopo il filtraggio	10
2.4	Effetti dei differenti raggi per la stima di una superficie	11
2.5	E' possibile vedere le diverse normali calcolate sulla point cloud . .	12
2.6	E' possibile notare come gli alberi e altre strutture non appartenenti alle facciate degli edifici siano state eliminate	13
2.7	E' possibile vedere i cluster ottenuti rappresentanti le varie facciate degli edifici	15
3.1	OSM Logo	16
3.2	Node	16
3.3	Way	17
3.4	Relation	17
3.5	JOSM Logo	18
3.6	Docker Logo	18
3.7	Esempio di modifica di un edificio con JOSM	19
3.8	Overpass API in funzione	22
3.9	Come si può vedere se trovano way con il tag building:facade:pcl presente	23
4.1	schema di comunicazione <i>Publisher-Subscriber</i> tramite topic	25
4.2	schema di comunicazione <i>Client-Server</i> tramite servizio	25
4.3	Schema ad alto livello del funzionamento di RLE	27
4.4	Struttura del servizio di richiesta delle point cloud	31

- 4.5 Si può notare che il servizio risponde con una pointcloud contenente la facciata dell’edificio corrente 31

Listings

2.1	Filtraggio della pointcloud	9
2.2	Applicazione della DoN	11
2.3	Filtraggio della pointcloud dopo l'applicazione dell' operatore DoN .	12
2.4	Clustering usando PCL	14
3.1	Dockerfile per OverpassAPI	20
3.2	Query di Overpass QL	22
4.1	Aggiunta del parametro per il download dal server in locale	28
4.2	Selezione del server da utilizzare per la query	28
4.3	Inserimento della query con cui scaricare gli edifici con la PCL associata.	28
4.4	Definizione del servizio per il retrival delle facciate	28
4.5	Recupero delle way con il tag inerente alla facciata.	29
4.6	Recupero e download delle pointcloud	30

Riferimenti

Bibliografia

- [1] A. L. Ballardini et al. «Leveraging the OSM building data to enhance the localization of an urban vehicle». In: *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. Nov. 2016, pp. 622–628. DOI: [10.1109/ITSC.2016.7795618](https://doi.org/10.1109/ITSC.2016.7795618).
- [2] Augusto Luis Ballardini et al. «A Framework for Outdoor Urban Environment Estimation». In: *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC 2015*-October (2015), pp. 2721–2727. DOI: [10.1109/ITSC.2015.437](https://doi.org/10.1109/ITSC.2015.437).
- [3] Sergio Cattaneo. «Leveraging the OSM building data in the Road Layout Estimation Framework». Tesi di laurea mag. Dipartimento di Informatica, Sistemistica e Comunicazione, 2015.
- [4] Padraig Cunningham e Sarah Delany. «k-Nearest neighbour classifiers». In: *Mult Classif Syst* (apr. 2007).
- [7] Y. Ioannou et al. «Difference of Normals as a Multi-scale Operator in Unorganized Point Clouds». In: (ott. 2012), pp. 501–508. ISSN: 1550-6185. DOI: [10.1109/3DIMPVT.2012.12](https://doi.org/10.1109/3DIMPVT.2012.12).
- [8] Jinyong Jeong et al. «Complex urban dataset with multi-level sensors from highly diverse urban environments». In: *The International Journal of Robotics Research* (2019), p. 0278364919843996.
- [14] Radu Bogdan Rusu. «Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments». Tesi di dott. Computer Science department, Technische Universitaet Muenchen, Germany, ott. 2009.
- [15] Sebastian Thrun, Wolfram Burgard e Dieter Fox. *Probabilistic robotics*. MIT press, 2005.

Siti

- [5] *Docker*. URL: <https://docs.docker.com/>.
- [6] *Docker Overpass API*. URL: <https://github.com/Frankkkkk/docker-overpass-api>.
- [9] *JOSM*. URL: <https://josm.openstreetmap.de/>.
- [10] *OpenStreetMap*. URL: <https://www.openstreetmap.org/about>.
- [11] *Overpass API*. URL: https://wiki.openstreetmap.org/wiki/Overpass_API.
- [12] *Point Cloud Library*. URL: <http://pointclouds.org>.
- [13] *ROS*. URL: <https://www.ros.org/about-ros/>.