

# Report - Meshing di PointCloud

Laboratorio di Meccatronica a.a. 2021/2022

Michele Lisi

Dicembre 2022

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Analisi preliminare</b>	<b>4</b>
2.1	Misura della distanza . . . . .	5
2.1.1	Lettura statica della distanza . . . . .	6
2.1.2	Lettura dinamica della distanza . . . . .	6
2.1.3	Realizzazione del tracking . . . . .	6
2.1.4	Problematiche dell'algoritmo . . . . .	7
2.2	Criticità sulle RealSense™L515 . . . . .	8
<b>3</b>	<b>Passaggio ad ambiente ROS</b>	<b>10</b>
3.1	Realsense™D435i . . . . .	10
3.2	Idea preliminare . . . . .	11
3.3	Prima implementazione . . . . .	11
3.4	Seconda implementazione . . . . .	13
3.4.1	Implementazione delle texture . . . . .	13
3.4.2	Nodo principale . . . . .	14
3.4.3	Interfaccia Grafica . . . . .	15
3.4.4	Analisi delle prestazioni . . . . .	15
3.4.5	Funzioni ancora da implementare . . . . .	16
<b>4</b>	<b>Applicazione di algoritmi ICP</b>	<b>18</b>
4.1	Ricerca preliminare . . . . .	18
4.2	Implementazione con algoritmo ICP . . . . .	18
4.3	Implementazione con RANSAC + ICP . . . . .	19
4.3.1	Integrazione in ROS . . . . .	20
4.3.2	Performance . . . . .	21
4.4	Implementazione con Pipeline ICP e PoseGraph . . . . .	21
4.5	Da Pointcloud a mesh 3D . . . . .	23
<b>5</b>	<b>Confronto con Kinect v2</b>	<b>25</b>
5.1	Nodi aggiuntivi . . . . .	26
5.1.1	Migliorie ai nodi esistenti . . . . .	27
5.2	Confronto delle Pointcloud . . . . .	28
5.2.1	Primo confronto . . . . .	28
5.2.2	Confronto di ricostruzione . . . . .	32
5.2.3	Considerazioni sul confronto . . . . .	33
<b>6</b>	<b>Conclusioni</b>	<b>35</b>

## 1 Introduzione

L'obiettivo di questo report è quello di descrivere al meglio lo sviluppo di un sistema in grado di ricostruire il modello 3D di un veicolo a partire dalle acquisizioni di camere in grado di rilevare la profondità.

Nello specifico si andranno ad utilizzare diverse tipologie di camere, basate su tecnologia Lidar o stereoscopica, in modo da ricostruire, a partire dalle loro acquisizioni, delle nuvole di punti.

Una volta ottenute si procederà nell'allinearle tramite la stima di una trasformazione che le riporti tutte nella stessa terna, pronte poi per essere ricostruite in Mesh 3D.

Utilizzando il linguaggio Python si andrà prima a creare degli script in grado di interfacciarsi con le camere, per poi passare ad un'implementazione ROS-based dello stesso, ampliata aggiungendo le funzioni mancanti.

Nonostante i risultati ottenuti siano più orientati verso una proof of concept, verranno presi degli accorgimenti e fatte delle considerazioni in vista di una possibile realizzazione industriale del progetto.

## 2 Analisi preliminare

Il primo dispositivo a disposizione per l'acquisizione di PointCloud(PC) sono state due camere Intel®RealSense™L515.

Queste fanno parte della famiglia di camere Lidar, ormai discontinue dall'azienda, ed hanno le seguenti caratteristiche:



Figura 1: Camera Intel®RealSense™L515

- **Caratteristiche:**

- Utilizzo indoor
  - Tecnologia di laser scanning
  - Range da 0.25m fino a 9m
  - IMU a 6 DOF

- **Depth:**

- Tecnologia LIDAR
  - Risoluzione fino a  $1024 \times 760$
  - Framerate fino a 30 fps
  - FOV  $70^\circ \times 55^\circ$

- **RGB:**

- Rolling Shutter
  - Immagine fino a  $1920 \times 1080$
  - Framerate fino a 30 fps
  - FOV  $70^\circ \times 43^\circ$
  - Risoluzione di 2 MP

Intel, inoltre, fornisce il Software Development Kit (SDK) completo di una vastissima scelta di wrapper, dando quindi libertà sull'ambiente di sviluppo, ed un viewer ufficiale in grado di comunicare con tutte le camere RealSense ed applicare effetti di Post Processing e correzioni.

Riguardo l'ambiente di sviluppo, è stato scelto di lavorare su Ubuntu 20.04 LTS (in grado di supportare ROS Noetic) e di utilizzare Python per realizzare la parte software.

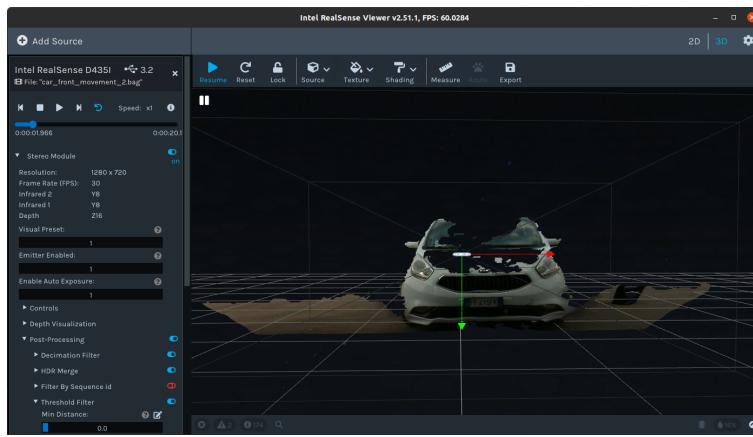


Figura 2: RealSense Viewer

## 2.1 Misura della distanza

Per prendere confidenza con le librerie del SDK (`pyrealsense2`), si è iniziato a lavorare a degli script in grado di leggere, in maniera dinamica se possibile, la distanza di un oggetto posto di fronte alla camera.

L'inizializzazione dello stream da codice, oltre ad essere ben documentata, è abbastanza semplice. Viene inizializzata una *pipeline* in cui si andrà a specificare quale dispositivo sta inviando lo stream e con quali configurazioni.

In questo passaggio è possibile abilitare lo stream sia da un'effettivo dispositivo, che emularlo attraverso un file `.bag`.

Questa possibilità ha semplificato notevolmente lo sviluppo, in quanto, previe acquisizioni soddisfacenti, non vi è il bisogno di essere costantemente connessi alla camera.

La pipeline viene a questo punto fatta partire, restituendo un profilo, che conterrà tutte le informazioni relative al dispositivo e al tipo di streaming.

### 2.1.1 Lettura statica della distanza

Una delle prime bag registrate consiste nell'aver piazzato la camera in modo da essere frontale al piatto di una stampante 3D con una scatola posta su di esso. Il piatto viene quindi portato da un estremo all'altro, facendo variare la distanza.

Acquisito lo stream, la distanza viene calcolata selezionando i valori all'interno di una regione di interesse (ROI) scelta in modo tale che contenga una sezione della scatola a qualsiasi distanza.

Viene quindi effettuata la media dei valori all'interno della ROI e moltiplicata per il fattore di scala, in modo da ottenere la distanza in metri.

I risultati ottenuti sono stati soddisfacenti, ottenendo un errore massimo registrato attorno al centimetro. La problematica principale di questo approccio statico è proprio il fatto che la selezione dei valori da utilizzare sia manuale, e quindi hardcoded.

### 2.1.2 Lettura dinamica della distanza

Dato che l'approccio precedente non è abbastanza generale, si è scelto di trovare un modo di selezionare la ROI in maniera dinamica.

Si è scelto di utilizzare la libreria `OpenCV`, che contiene diverse funzioni molto specifiche dell'ambiente della Computer Vision.

Dopo diversi tentativi, l'idea è la seguente: invece di elaborare la ROI dallo stream del LIDAR, si prende lo stream infrarosso della camera, più dettagliato, si effettua del post-processing in modo da ricavare una mask che isoli l'oggetto, frame per frame, che viene poi applicata al depth stream.

In questo modo viene realizzato un semplice, seppur efficace, tracking dell'oggetto.

### 2.1.3 Realizzazione del tracking

Dopo aver ricavato l'immagine infrarossa, si utilizza la funzione `cv2.Canny()` che processa l'immagine, in base a dei threshold, e ne disegna ciò che viene rilevato come contorno.

Successivamente `cv2.findContours()` restituisce in un vettore tutto ciò che viene rilevato come **contorno chiuso**. Viene quindi ordinato il vettore in base all'area racchiusa dai contorni e viene disegnato il contorno che rispetta i threshold imposti sull'area minima, in modo da isolare l'oggetto.

In Figura 3 è possibile vedere le varie fasi dell'elaborazione dei frame.

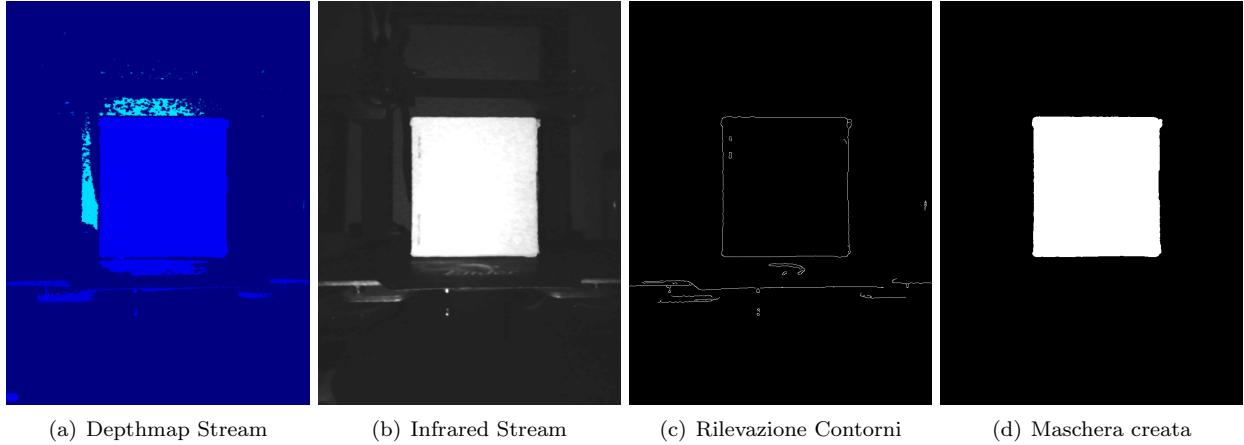


Figura 3: Esempio di processing dei frame

Questa procedura, così com'è, presenta un problema principale. Come specificato, la funzione rileva solo i contorni chiusi. Di conseguenza, non sempre lo stesso oggetto viene rilevato in frame successivi, anzi, molti dei contorni con area considerevole non vengono considerati come contorni chiusi dall'algoritmo e quindi non vengono rilevati.

Per ovviare a questo problema, si è scelto di utilizzare una variabile, `last_mask`, che ha al suo interno l'ultimo contorno disegnato.

In questo modo, durante il frame attuale, dopo aver computato i contorni presenti, se nessuno soddisfa i requisiti di area minima, viene utilizzato il contorno disegnato al frame precedente.

Se invece viene rilevato un contorno che rispetta le condizioni, viene disegnato e salvato all'interno di `last_mask`.

#### 2.1.4 Problematiche dell'algoritmo

Nonostante questo metodo sia efficacemente in grado di creare dinamicamente una ROI di un oggetto selezionato, è lontano dall'essere perfetto.

Il tracking risulta corretto solo per movimenti contenuti dell'oggetto. Se la velocità supera una certa soglia, si ha un fenomeno per il quale non è ancora stato rilevato un contorno valido per l'algoritmo e quindi viene disegnato l'ultimo valido, risultando in un offset tra l'effettiva posizione dell'oggetto e la posizione della maschera.

Inoltre, se nella scena sono presenti diversi oggetti con dimensioni paragonabili, diventa ovviamente difficile per l'algoritmo selezionarne solo uno, o anche solo discernere quale si stia selezionando.

Il sistema verrà sicuramente migliorato, tuttavia è importante ricordarsi che, nell'use case finale del progetto, la velocità sarà effettivamente contenuta e non sarà presente nessun altro oggetto oltre il veicolo.

## 2.2 Criticità sulle RealSense™L515

Durante il primo approccio con le camere, sono stati riscontrati alcuni problemi, primo tra questi il surriscaldamento delle stesse.

È stato infatti constatato come il case in alluminio delle camere raggiunga una temperatura tale da scottare al tatto dopo pochi minuti di utilizzo.

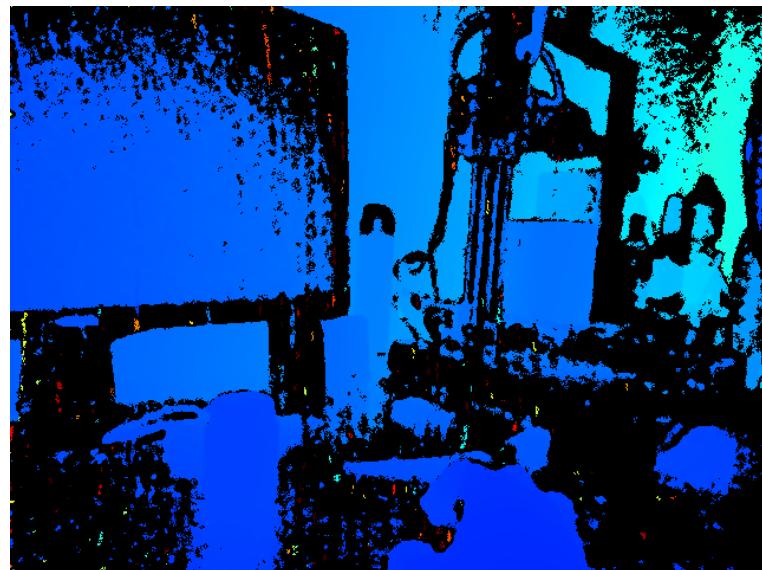


Figura 4: Interferenza creata dalle due camere

Va fatto notare che durante il periodo in cui sono state utilizzate le camere, anche solo la temperatura ambiente risultava al limite rispetto alle temperature di utilizzo da datasheet, tuttavia, visto l'use case del progetto, questo potrebbe essere un problema da non sottovalutare.

Un altro problema è l'utilizzo di più camere contemporaneamente: essendo le L515 basate su tecnologia Lidar, posizionare più di una camera risulterà in una forte interferenza sulle acquisizioni, come si può vedere in Figura 4, poichè entrambe staranno emettendo costantemente.

Una soluzione a questo problema, come documentato sul sito Intel, è quella di mettere le camere in stato di *slave*, possibile semplicemente cambiando una delle impostazioni.

Questo le manterrà costantemente disattive a meno che non ricevano un segnale attivo (ovvero una tensione di 3.3V) dalla porta appositamente creata per accettare questo segnale di *clock*.

### 3 Passaggio ad ambiente ROS

Preso confidenza con i dispositivi RealSense, si è deciso di effettuare il passaggio ad ambiente di sviluppo ROS.

Questa scelta è motivata dalla estrema flessibilità che l'ambiente offre, oltre ad una serie di vantaggi, tra cui, la possibilità di visualizzare la PointCloud direttamente su rViz, un tool di visualizzazione in grado di elaborare direttamente messaggi ROS.

Il passaggio, tuttavia, non è stato di semplice implementazione data sia la scarsa presenza di informazioni sull'ambiente RealSense, sia dal fatto che i wrapper ufficiali avessero alcune implementazioni non funzionanti.

#### 3.1 Realsense<sup>TM</sup>D435i

Il cambiamento di ambiente di sviluppo è avvenuto in concomitanza con delle prove effettuate con un'altra serie di camere, ovvero le Intel®RealSense<sup>TM</sup>D435i.



Figura 5: Intel®RealSense<sup>TM</sup>D435i

Queste, a differenza della serie L, non sfruttano la tecnologia Lidar, ma quella stereoscopica assieme agli infrarossi.

In pratica, la doppia camera è in grado di misurare la profondità, che viene ulteriormente elaborata attraverso una griglia infrarossa che viene proiettata. La RealSense D435i ha le seguenti specifiche:

- **Caratteristiche:**

- Uso esterno o interno
  - Range da 0.3m a 3m
  - Global Shutter

- **Depth:**

- Tecnologia stereoscopica

- FOV  $87^\circ \times 58^\circ$
- Risoluzione fino a  $1280 \times 720$
- Framerate fino a 90fps
- Precisione  $\pm 2\%$  a 2m

- **RGB:**

- Immagini fino a  $1920 \times 1080$
- Framerate fino a 30 fps
- FOV  $69^\circ \times 42^\circ$
- Risoluzione di 2 MP

Utilizzando queste camere, sono state registrate delle rosbag di un veicolo sia fermo che in movimento e da diverse prospettive.

Di particolare interesse saranno le acquisizioni con la camera posta di lato rispetto al veicolo in movimento, ed inclinata di  $45^\circ$  (oltre che rialzata in modo da avere una buona prospettiva) verso il basso.

### 3.2 Idea preliminare

La prima idea per il passaggio su ROS è stata quella di sfruttare il grandissimo supporto delle librerie RealSense e quindi utilizzare come base il wrapper ROS, su cui poi costruire dei nodi che avrebbero filtrato la pointcloud e poi applicato gli algoritmi necessari a ricostruire un modello 3D.

Come già anticipato però, i wrapper ufficiali presentano diversi problemi: nel caso in cui si stia leggendo una rosbag (registrata dal visualizzatore ufficiale RealSense), il nodo non pubblicherà nessun messaggio in nessun topic. Andando a ricercare, ad oggi questo è un bug noto ed ancora irrisolto.

Inoltre, andando ad avviare lo streaming da camera invece che da bag, nonostante i messaggi vengano correttamente pubblicati, la pointcloud non viene elaborata.

### 3.3 Prima implementazione

Data l'impossibilità di utilizzare i wrapper ufficiali, si è scelto di scrivere utilizzando `rospy` un nodo in grado di garantire, almeno in parte, le loro funzionalità.

Dopo aver acquisito i concetti base di ROS e `rospy`, la prima implementazione era in grado di mandare tramite topic tutti i canali di stream della camera,

grazie anche a funzioni di conversione diretta da immagine a messaggio ROS (`sensor_msgs.Image`).

Si è quindi scelto di utilizzare lo stesso approccio con la pointcloud.  
Dato che era già stata utilizzata la libreria di `open3d`, e data la disponibilità di funzioni di conversione da pointcloud `open3d` a messaggio ROS (`sensor_msgs.PointCloud2`), la prima pointcloud visualizzata è stata ottenuta con questo metodo.

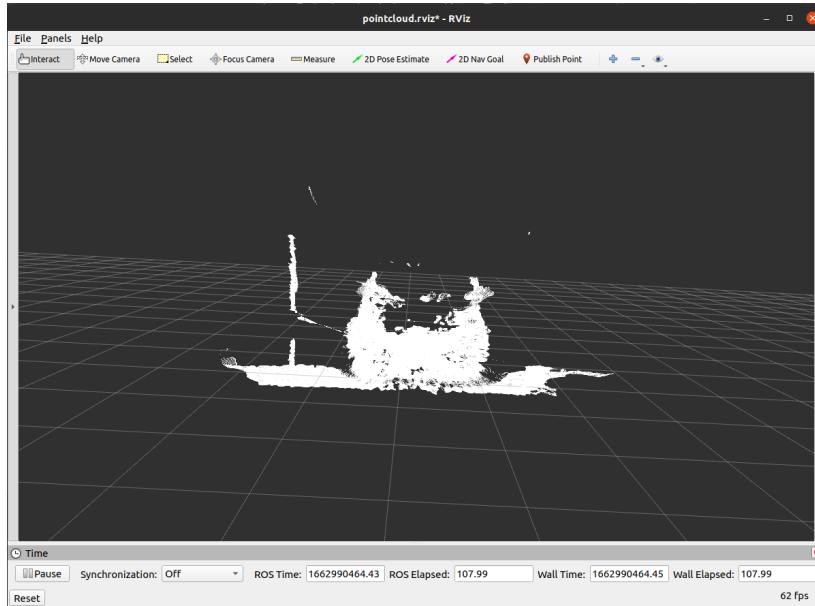


Figura 6: PointCloud elaborata con `open3d`

Il problema riscontrato in questo approccio, è che, data l'impossibilità di convertire la classe pointcloud di `pyrealsense2` in quella `open3d` direttamente, quest'ultima viene calcolata a partire dall'immagine depth e da quella a colori.

La pointcloud ottenuta in questo modo, nonostante fosse correttamente visualizzata, presentava discrepanze rispetto alla pointcloud originale e quindi è risultata insoddisfacente se paragonata al visualizzatore ufficiale RealSense.

### 3.4 Seconda implementazione

Non essendo la prima implementazione soddisfacente, si è deciso di andare a creare una funzione che si occupi della conversione da classe pointcloud di `pyrealsense2` a messaggio PointCloud2.

Il primo passo è stato effettivamente capire il funzionamento del messaggio. Un aiuto sostanziale è stato ricevuto da uno dei branch del wrapper ROS: tra i vari vi era `py-rs2`, un'implementazione in Python (al contrario del main branch, scritto principalmente in C++) di alcune funzioni del wrapper.

Alcuni problemi riscontrati con il wrapper ufficiale, in particolare riguardo la pubblicazione di PointCloud, sono risolti in questa versione, ma non tutti: nonostante la PC fosse convertita correttamente, non era stato preso nessun accorgimento che prevedesse la presenza di texture nel messaggio PointCloud2. A questo punto è stata utilizzata questa versione per apprendere appieno il funzionamento del messaggio PointCloud2 e realizzare una funzione in grado di convertire una PC RealSense in messaggio ROS.

Senza soffermarsi sui dettagli, i campi principali che identificano ciò che si sta passando sul topic sono le sezioni *data* e *PointField*.

Il primo sarà un array di bytes raw, mentre il secondo campo indica come è stato strutturato un singolo elemento dell'array.

```
msg.fields = [
    PointField('x', 0, PointField.FLOAT32, 1),
    PointField('y', step, PointField.FLOAT32, 1),
    PointField('z', step*2, PointField.FLOAT32, 1),
    PointField('r', step*3, PointField.FLOAT32, 1),
    PointField('g', step*4, PointField.FLOAT32, 1),
    PointField('b', step*5, PointField.FLOAT32, 1),
]
```

Figura 7: Definizione dei PointFields

Come si può vedere in figura 7, ogni singolo punto inviato prevede 3 `float32` per le coordinate nello spazio, e 3 `float32` per i canali RGB, normalizzati quindi tra 0 e 1 invece che 0 e 255.

#### 3.4.1 Implementazione delle texture

Uno dei problemi non risolti dal wrapper scritto in Python è quello delle texture: nonostante fosse in parte presente del codice che le prevedesse, la funzione non era correttamente implementata.

Oltre a questo, tramite `pyrealsense2` non è possibile ottenere direttamente i valori RGB di un singolo punto, bensì una texture map, anche detta UV map, un espediente molto comune nella grafica 3D.

Si è quindi realizzata una funzione che passasse da coordinate UV ad RGB: le prime, infatti, rappresentano le coordinate nello spazio della texture relative ad ogni punto della PointCloud.

Avendo il vettore della texture map la stessa lunghezza di quello che contiene i punti, si crea una corrispondenza tra il singolo punto e il pixel, quindi i valori RGB, dell'acquisizione a colori.

### 3.4.2 Nodo principale

Trovato il modo per inviare correttamente la pointcloud, si è passato alla rifinitura dei dettagli per un corretto funzionamento del nodo e una visualizzazione soddisfacente su rViz.

Il nodo principale è `streamer.py`, che si occupa di tradurre le classi `pyrealsense2` nei relativi messaggi ROS. I topic creati dal nodo in questa implementazione sono i seguenti:

- `/camera/color`
- `/camera/depth`
- `/camera/depth/color`
- `/camera/infrared`
- `/camera/points`
- `/camera/camera.info`

Attualmente è possibile lanciare il nodo attraverso il comando `roslaunch lidar lidar.launch`, che provvederà a lanciare il nodo principale ed avviare rViz con la configurazione necessaria per visualizzare correttamente la pointcloud.

Sono inoltre presenti due argomenti per il launch file: `stream`, che specifica al nodo se si sta effettuando stream da camera quando messo a `True`, o se si sta leggendo una rosbag. In questo secondo caso il parametro `bag_filename` specificherà il percorso della bag.

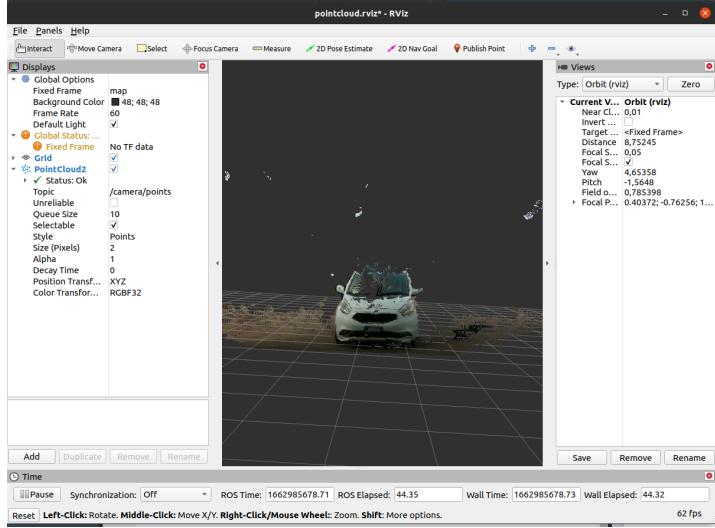


Figura 8: Pointcloud visualizzata in rViz

### 3.4.3 Interfaccia Grafica

Assieme al nodo principale e ad rViz, una terza finestra verrà aperta. Quest'ultima ha l'utilità di gestire tutta la parte di filtraggio prima che la pointcloud venga convertita.

In questa implementazione sono presenti alcuni dei filtri di post processing standard di RealSense: un filtro threshold, uno spatial filter ed un filtro di decimazione.

Come si vede dalla Figura 9, sono presenti diversi slider che permettono di agire dinamicamente sui valori dei vari filtri. Oltre a questo è possibile attivarli e disattivarli singolarmente, oltre che a mettere in pausa lo stream.

Il risultato in figura è stato ottenuto usando la libreria `tkinter` come base per realizzare una classe creata appositamente per gestire in maniera semplice e versatile l'interfaccia per i valori utilizzati dai filtri: per come questa è strutturata, aggiungere o rimuovere altri filtri, con relativi slider dei valori e checkbox, può essere fatto in pochissime righe di codice.

### 3.4.4 Analisi delle prestazioni

Andando a confrontare ciò che viene visualizzato in rViz contro ciò che si vede con il RealSense Viewer (Figura 10), i risultati ottenuti sono più che soddisfa-

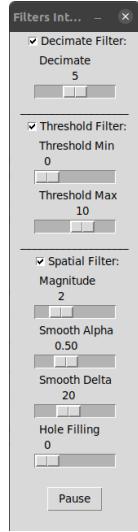


Figura 9: Interfaccia Grafica

centi.

Il nodo riesce ad elaborare e visualizzare la pointcloud mantendo una media di 18 fps con filtro di decimazione al minimo, e 22 fps col filtro al massimo, su una rosbag registrata a 30 fps.

Questi valori sono stati ottenuti attraverso il comando `rostopic hz`.

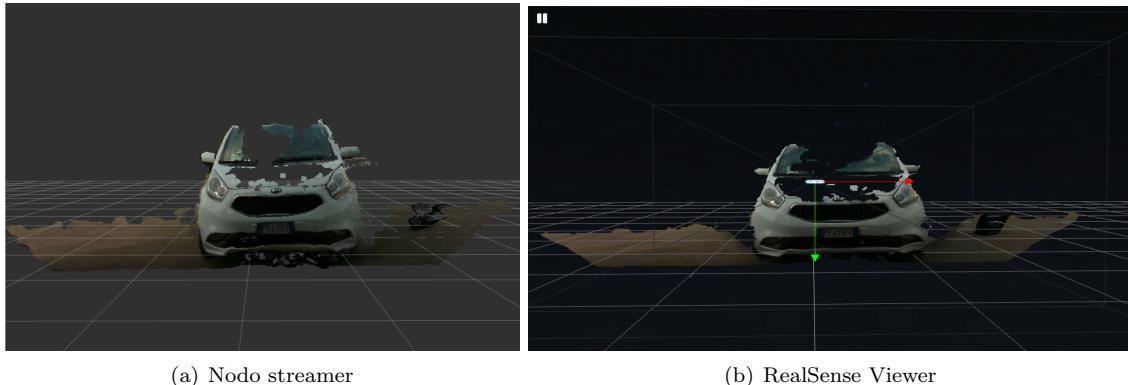


Figura 10: Confronto tra i visualizzatori

### 3.4.5 Funzioni ancora da implementare

A questo punto, data la modularità di ROS, implementare nuovi nodi in grado di manipolare e filtrare le pointcloud risulta molto meno complicato rispetto al

precedente approccio.

Mancano, infatti, un nodo che sia in grado di applicare algoritmi di ICP in modo da allineare le varie pointcloud ed un altro che sia in grado di ricostruire in 3D la stessa.

Va notato, inoltre, che i 22 fps massimi raggiunti, nonostante rappresentino un ottimo risultato, potrebbero diventare un collo di bottiglia, quindi ci si impegna ad ottimizzare lo stream.

## 4 Applicazione di algoritmi ICP

Avendo sviluppato in maniera soddisfacente un nodo che permetta di visualizzare una pointcloud, il passo successivo è quello di prenderle, campionandole possibilmente, ed unirle in modo tale da formare un'unica pointcloud.

Quindi, data una serie di pointcloud contenenti frammenti diversi dello stesso oggetto, trovare quelle trasformazioni tali per cui se applicate ai frammenti li portino da terna locale a terna globale.

La prima è definita come l'orientamento della camera in quel frammento, mentre l'ultima per convenzione sarà l'orientamento della camera del primo.

Si è subito scartata l'opzione di leggere i dati dell'IMU ed integrarli in modo da ottenere la posizione istantanea della camera.

Questo sia per l'onerosità d'implementazione di un filtro di Kalman che abbia la precisione necessaria a ricostruire correttamente i frammenti, sia perchè nell'use case finale la posizione ed i movimenti della camera saranno controllati, e quindi noti.

### 4.1 Ricerca preliminare

L'approccio alternativo a quello descritto nella sezione precedente è utilizzare degli algoritmi che riescano a stimare con un buon grado di precisione la trasformazione necessaria ad unire in maniera corretta i vari frammenti.

Andando a ricercare lo stato dell'arte, gli algoritmi ICP (Iterative Closest Point) sono largamente utilizzati per questo scopo.

La libreria utilizzata finora per la gestione delle Pointcloud, `open3d`, ha specificatamente delle sottolibrerie (nello specifico `open3d.pipelines.registration`) dedicate a questo.

### 4.2 Implementazione con algoritmo ICP

La prima prova di registrazione di Pointcloud è stata fatta utilizzando la funzione `registration_icp()`.

Questa riceve in input due pointcloud (source e target), una matrice 4 x 4 (la matrice di rototraslazione da cui inizializzare l'algoritmo), ed i parametri relativi all'algoritmo ICP, e restituisce la matrice di trasformazione stimata.

Dopo alcuni test, è stato trovato che l'applicazione diretta dell'algoritmo ICP con queste modalità non è efficace, in quanto non produce accoppiamenti soddisfacenti.

### 4.3 Implementazione con RANSAC + ICP

A questo punto, è stata effettuata una ricerca più approfondita sia sulla documentazione di `open3d` che sulle tecniche utilizzate attualmente per effettuare ricostruzione.

Le prestazioni poco soddisfacenti della precedente implementazione sono dovute al fatto che questi algoritmi performano male quando devono ricercare una trasformazione da zero.

Vengono infatti utilizzati più per ottimizzare la stima, ma a partire da una trasformazione approssimativa invece che una nulla. Da qui la seconda implementazione che sfrutta diversi algoritmi.

Come primo step vengono computate le feature FPFH (Fast Point Feature Histogram) delle due pointcloud, che vengono poi date in input ad una funzione che applica un algoritmo RANSAC che cerca delle corrispondenze con le suddette feature e calcola una trasformazione che, con un buon grado di approssimazione, porta la il secondo frammento nella terna del primo.

Già in questo primo step i risultati sono molto più consistenti rispetto alla prima implementazione, seppur non precisissimi.

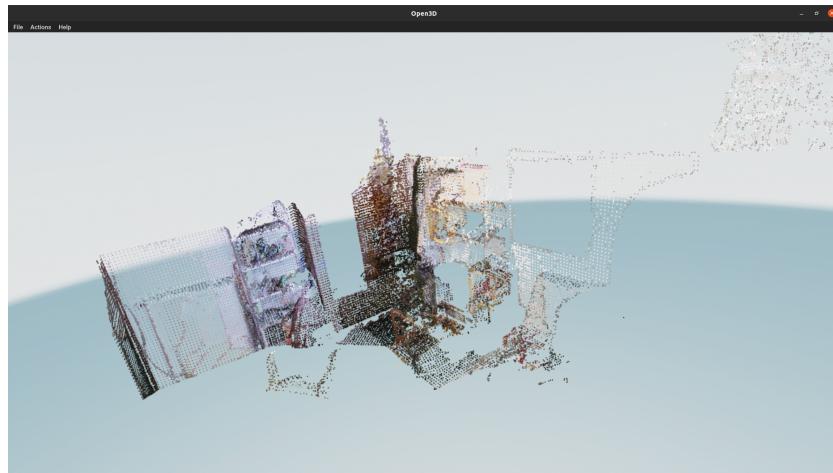


Figura 11: Ricostruzione con RANSAC ed ICP

Lo step successivo è quello di applicare l'algoritmo ICP a partire dalla trasformazione calcolata dal RANSAC ed ottenerne una nuova, più precisa rispetto alla precedente.

In Figura 11 è possibile vedere una ricostruzione della mia stanza fatta a partire da 3 acquisizioni in angolazioni diverse. I tre frammenti sono perfettamente allineati.

#### 4.3.1 Integrazione in ROS

Trovato un metodo funzionante per stimare le trasformate, finora applicate in modo statico con acquisizioni separate, va adesso implementato un nodo che legga ciò che `streamer.py` trasmette su topic e lo elabori.

È stato quindi creato il nodo `registration.py`, in cui viene definita la classe `PointCloud` che gestisce ogni aspetto, dalla subscription al topic `/camera/points`, all'elaborazione della pointcloud ricostruita.

Data l'impossibilità di trovare un metodo efficace per effettuare sampling dello stream a valle, si è deciso di farlo direttamente a monte, quindi nel nodo `streamer.py`:

nel momento in cui `registration.py` viene eseguito, il nodo passa da uno streaming a 60 fps ad uno pari ad 1 frame al secondo, che è risultato un sampling più che efficace per ottenere delle Pointcloud che abbiano diverse angolazioni del veicolo.

Avviato il nodo, quindi, dopo l'interazione con l'utente, verranno registrati i frame dello stream fino al riempimento del buffer, anch'esso definito dall'utente. Nel momento in cui viene avviata la computazione, oltre al calcolo delle trasformate per ogni coppia di frame, queste vengono anche applicate agli stessi nel corretto ordine in modo da ottenere alla fine un'unica pointcloud che viene poi pubblicata in `/computed/points`.

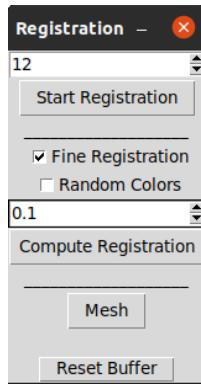


Figura 12: Graphic interface of `registration.py`

In Figura 12 è possibile vedere l'interfaccia grafica del nodo, creata con la

stessa libreria ideata per serializzare una GUI semplice e versatile. Quest'ultima è stata ampliata in modo da poter aggiungere spinbox, in modo da avere un controllo di valori più preciso rispetto agli slider, e da poter avere dei pulsanti generici le cui funzioni possono essere assegnate in momenti diversi dalla creazione.

Un altro elemento da menzionare è la seconda casella: questa serve a selezionare la grandezza dei voxel. Questa influenza la precisione e, molto spesso, se l'accoppiamento risulti soddisfacente o meno.

È stato trovato che una scelta di valori tra 0.7 e 0.1 produce molto spesso degli accoppiamenti soddisfacenti.

Vi è anche la scelta di utilizzare solo l'algoritmo di RANSAC, per una minor precisione a vantaggio della velocità, oppure entrambi gli algoritmi tramite la checkbox 'Fine Registration'.

Infine, è possibile assegnare ai diversi frammenti dei colori casuali per meglio discernere le diverse pointcloud e valutare l'accoppiamento.

#### 4.3.2 Performance

Nonostante l'implementazione sia funzionante, e la ricostruzione della pointcloud sia stata ottenuta, questo approccio non è stato ritenuto soddisfacente. Questo perchè i tempi computazionali risultano non indifferenti ( 5 secondi per coppia di frammenti), e gli accoppiamenti finali molte volte sono risultati totalmente errati.

Risolvere quest'ultima problematica avrebbe richiesto un fine tuning sui ( numerosi) parametri dei vari algoritmi che ha portato alla ricerca di soluzioni diverse.

### 4.4 Implementazione con Pipeline ICP e PoseGraph

L'ultima implementazione del nodo è basata sì sull'utilizzo di algoritmi ICP, ma sfruttando i PoseGraph.

Questi ultimi, ampiamente implementati in `open3d`, sono degli elementi contenenti nodi e spigoli. I nodi sono le  $P_i$  pointcloud a cui è associata l'information matrix  $\Lambda_i$  e una matrice di posa  $T_i$ , che rappresentano le variabili da ottimizzare.

Gli spigoli invece sono composti dalla trasformazione  $T_{i,j}$  che allinea  $T_i$  a  $T_j$ , ovvero frammenti di breve distanza temporale.

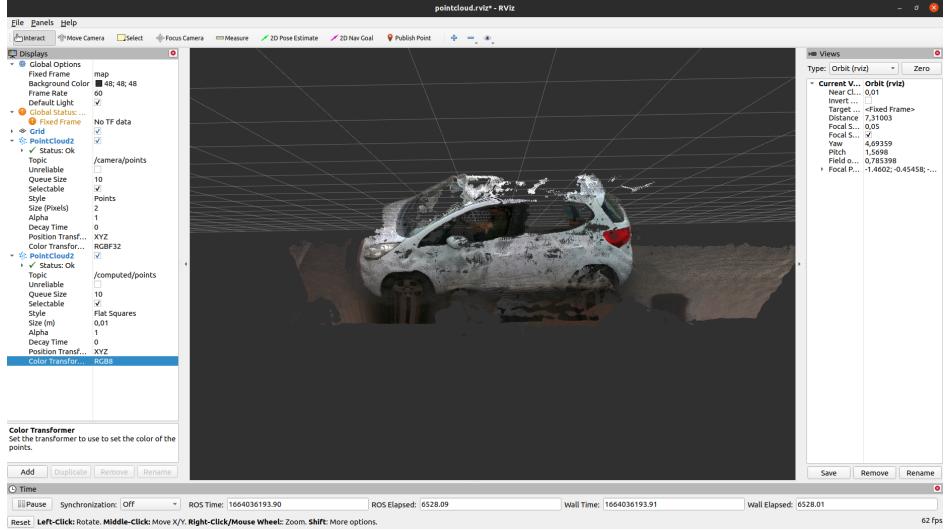


Figura 13: Registrazione ottenuta con l’ottimizzazione del PoseGraph

L’inizializzazione delle  $T_i$  viene effettuata sfruttando i diversi algoritmi ICP in una pipeline.

Ad ogni coppia di pointcloud vengono applicati diversi criteri di stima di ICP; nell’implementazione attuale la pipeline è la seguente:

1. ICP con alto distance treshold e criterio Point to Plane
2. ICP con basso distance treshold (più fine) con criterio Point to Plane
3. ICP con alto distance threshold con criterio Point to Point
4. ICP con alto distance threshold con criterio basato sul colore dei punti

Aggiornati gli elementi del PoseGraph, si avranno delle prime trasformazioni con un buon grado di approssimazione, pronte per essere ottimizzate attraverso la funzione `global_optimization`, che sfrutta l’Algoritmo di Levenberg-Marquardt.

Si fa notare come ogni funzione ICP richieda come argomento una *threshold distance*, ovvero la distanza con cui si vanno a cercare gli accoppiamenti.

È pratica comune quando si utilizzano le Pipeline ICP utilizzare un’unica voxel size con cui effettuare un downsample iniziale e poi utilizzare dei moltiplicatori per definire questo parametro.

In questo modo è possibile definire la granularità della ricerca.

In Figura 13 è possibile vedere il risultato ottenuto con l'algoritmo descritto che risulta essere più che soddisfacente.

Nonostante bisogni comunque effettuare un fine tuning dei parametri, più rilassato rispetto alla precedente implementazione, la differenza di costo computazionale è considerevole: è possibile calcolare mediamente tra le 4 e le 6 trasformate al secondo.

## 4.5 Da Pointcloud a mesh 3D

Avendo ottenuto una ricostruzione più che soddisfacente, il passo successivo è quello di ricostruire una Pointcloud in Mesh triangolare.

La libreria `open3d` mette a disposizione diverse funzioni per il remeshing. Di particolare interesse per questo caso d'uso sono la ricostruzione attraverso il Ball Pivoting Algorithm (BPA), ed il Poisson Screened Reconstruction.

Il primo dei due algoritmi presentati, il BPA, come suggerisce il nome, consiste nel far ‘rotolare’ una sfera di raggio arbitrario sulla nuvola di punti. Se questa tocca 3 punti, questi diventano un triangolo, altrimenti si continua. È possibile scegliere raggi di diverse dimensioni, riproponendo l'algoritmo.



Figura 14: Remeshing di Pointcloud con Ball Pivoting Algorithm

In Figura 14 è possibile vedere una ricostruzione effettuata attraverso il BPA. Come raggi per l'algoritmo è stata calcolata la distanza media tra i vari punti della Pointcloud e ne sono stati usati multipli e sottomultipli.

Il secondo algoritmo invece, consiste nel prendere un piano e cercare di avvolgerlo attorno ad i punti, risultando quindi intrinsecamente indicato per superfici chiuse.

In Figura 15 è possibile vedere la ricostruzione effettuata dall'algoritmo.

Entrambe le ricostruzioni purtroppo risultano insoddisfacenti. Ciò è dovuto al forte rumore presente nelle acquisizioni.



Figura 15: Remeshing di PointCloud con Screened Poisson Reconstruction

## 5 Confronto con Kinect v2

Dato che il problema principale della ricostruzione del modello risulta essere la rumorosità delle acquisizioni, si è scelto di confrontare le prestazioni con dei Kinect v2.

Nonostante questi ultimi abbiano una risoluzione minore per quanto riguarda i punti, ci si aspetta riescano a produrre una pointcloud molto meno rumorosa rispetto a quelle delle due RealSense.



Figura 16: Kinect v2

Uscito nel 2013, il Kinect v2 presenta le seguenti caratteristiche:

- **Depth:**

- Risoluzione di  $512 \times 424$
- Framerate di 30 fps
- FOV  $70^\circ \times 60^\circ$

- **Color:**

- Risoluzione di  $1920 \times 1080$
- Framerate di 30 fps
- FOV  $70^\circ \times 60^\circ$

Nonostante gli anni passati dalla sua uscita, grazie alle librerie `libfreenect2` (wrapper C++) e `freenect2` (wrapper Python), è possibile tuttora utilizzare il dispositivo, senza tuttavia poter accedere alle opzioni di processing interne.

## 5.1 Nodi aggiuntivi

L'idea, a questo punto, è quella di creare altri nodi in modo tale che lo streaming da Kinect venga gestito in maniera analoga a quanto ottenuto per le camere RealSense.

Il wrapper Python risulta molto semplice da utilizzare, nonostante non sia perfetto. Uno dei primi problemi riscontrati è stata la Queue.

Durante lo streaming, la queue assegnata di default alla funzione che gestisce l'acquisizione dei frame veniva riempita e questo risultava in un errore da parte della libreria.

Per risolvere il problema è bastato creare un'altra queue che avesse più spazio, in modo da non riempirla durante il normale runtime del nodo.

Come per lo streamer RealSense, dopo l'inizializzazione dei topic e delle classi relative al wrapper, inizia il loop principale.

Nella prima sezione di quest'ultimo vengono acquisite le immagini di profondità e colore che, date come argomento ad una delle funzioni di `freenect2`, trasformano l'immagine di profondità, nativamente a  $512 \times 424$ , nella stessa grandezza di quella a colori, quindi  $1920 \times 1080$ .

Questa immagine di profondità viene poi elaborata in modo da ottenere il vettore dei punti.

A questo punto, la conversione in Pointcloud `open3d` sarebbe immediata. Tuttavia, è stato riscontrato che il vettore dei punti restituito ha dimensioni pari a  $1920 \times 1082$ .

Questa discrepanza tra le dimensioni di immagine di profondità e colore le rende incompatibili per la conversione.

Piuttosto che ridimensionare l'immagine a colori, operazione che porta lo streaming dei dati ad 1 fps, è stato scelto di ignorare le ultime due righe del vettore dei punti, in modo da diminuire il tempo di computazione.

Un'altra operazione che inizialmente è risultata un bottleneck per lo streaming è stata la conversione a `open3d`: per assegnare i punti e i relativi colori, gli array contenenti gli stessi devono essere convertiti in `open3d.utility.Vector3dVector`, classe che gestisce i vettori in 3 dimensioni.

È stato trovato che nel caso in cui il vettore di partenza fosse in `float32`, formato standard per quanto riguarda `open3d`, si avranno dei rallentamenti non indifferenti.

Questo perché nella conversione in `Vector3dVector`, avviene inizialmente un cast del vettore in formato `float64`.

È quindi bastato cambiare il formato del vettore in ingresso per eliminare il collo di bottiglia ed ottenere uno stream attorno ai 24 fps.

### 5.1.1 Migliorie ai nodi esistenti

Durante questa fase dello sviluppo, sono state effettuate delle migliorie ai nodi già esistenti, con particolare enfasi su `streamer.py`, a cui è stato cambiato il nome in `realsense_streamer.py`.

Infatti, oltre agli aggiornamenti sulle librerie usate dai vari nodi, ci si è concentrati sull'implementazione dei filtri di `pyrealsense2` nel nodo principale.

In particolare, allo stato attuale è stato implementato un filtro temporale ed un filtro in grado di chiudere, per quanto non in maniera precisa, eventuali 'buchi' della Pointcloud.

Un altro filtro degno di nota è il Disparity to Depth (D2D). Questo è uti-

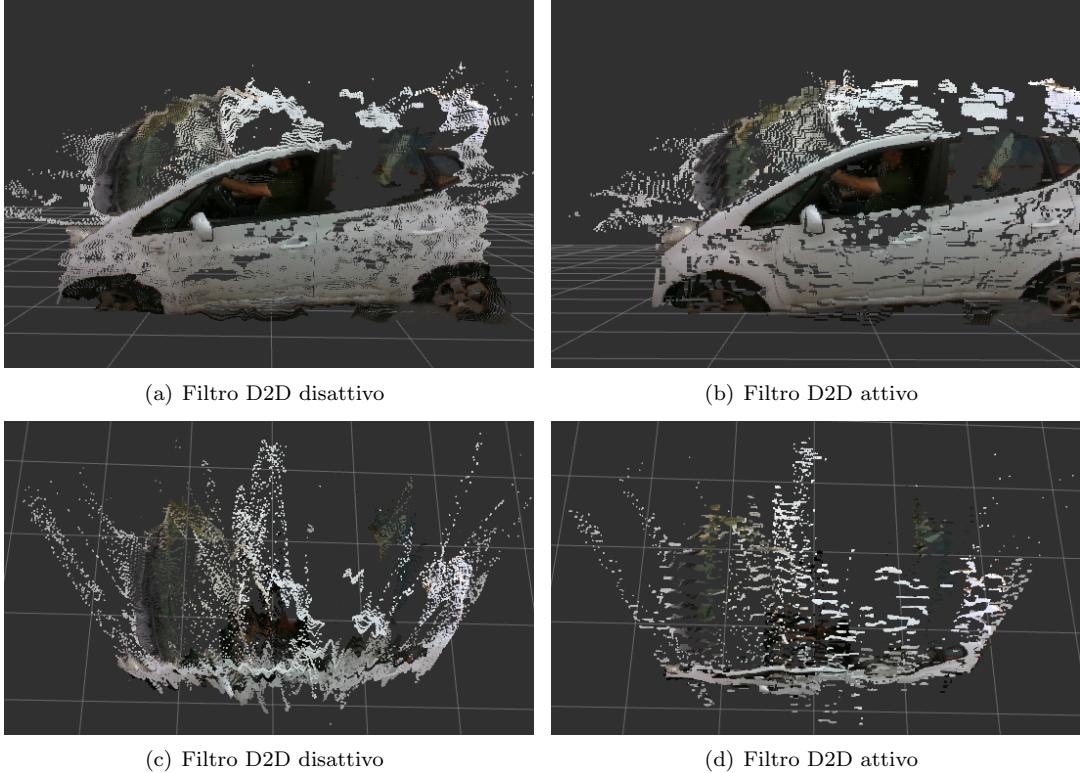


Figura 17: Confronto dell'acquisizione con e senza il filtro D2D

lizzabile solo con camere stereoscopiche (quindi con la serie RealSense DXXX) e, se combinato con il filtro spaziale, diminuisce molto il rumore presente, come si può vedere in Figura 17.

## 5.2 Confronto delle Pointcloud

Con il nodo `kinect_streamer.py` funzionante, è possibile confrontare le acquisizioni delle camere RealSense con quelle del Kinect v2.

È stata riscontrata l'impossibilità di effettuare in maniera semplice acquisizioni di un veicolo tramite Kinect v2.

Questo principalmente perché viene richiesta un'alimentazione esterna per operare la camera, al contrario delle RealSense, che vengono alimentate direttamente dal cavo USB, rendendo quindi impossibile operare la camera all'esterno senza avere una presa da cui alimentarla.

Si è scelto quindi di effettuare il confronto sia visualizzando le acquisizioni della stessa sezione di una stanza, sia confrontando la ricostruzione di un oggetto di medie dimensioni.

### 5.2.1 Primo confronto

Il primo confronto consiste nell'effettuare delle acquisizioni della stessa sezione di una stanza e valutarne la rumorosità.

Sono state quindi registrate due bag di pochi secondi, una per la RealSense™L515 e una per il Kinect v2, in cui le camere vengono poste ad 80 centimetri dal muro.



Figura 18: Confronto acquisizioni

In Figura 18 si possono vedere le acquisizioni. Le prime due immagini sono relative alla RealSense L515, la prima ha il filtro di decimazione disattivato, per un totale di 2.073.600 punti, mentre la seconda ha il filtro di decimazione a 5, su un massimo di 8, ed è composta da 82.944 punti.

La terza immagine invece, è un'acquisizione da Kinect v2, per un totale di 691.200 punti. Va fatto notare, tuttavia, come non tutti questi punti siano validi.

Passare la Pointcloud acquisita da Kinect così com'è creerebbe diversi problemi nella visualizzazione infatti. Questo è dovuto al fatto che vengono prodotti dei punti con coordinate non finite, ovvero i cui valori sono infiniti o NaN (Not a Number).

Eliminando questi punti, il numero di quelli rimasti è variabile, ma in media ne vengono rilevati 470.804, un numero confrontabile con quelli rilevati dalla RealSense con il filtro di decimazione al valore minimo (518.400).

A questo punto, è stato realizzato un nodo che analizzasse i dati delle due acquisizioni.

Nelle Figure 19 e 20 nelle pagine successive, è possibile vedere una sezione delle Pointcloud.

In particolare, è stata isolata una sezione in un determinato istante di tempo in cui è presente solo il muro grande 20 cm × 10 cm, in entrambe le acquisizioni, e ne è stato fatto un plot 3D.

La differenza nei valori delle X e delle Y è dovuto al fatto che l'orientamento degli assi per le due camere differiscono.

Entrambe le figure evidenziano quanto rumorosa sia la misura della distanza con due viste arbitrariamente orientate (per poter apprezzare il tipo di rumore nell'insieme) e due viste relative all'asse X e all'asse Y.

Per quanto riguarda la RealSense L515, questa ha rilevato punti che vanno da 81.8 cm a 82.7 cm, mentre il Kinect ne ha prodotti da 82.1 cm a 83.1 cm. Entrambe le camere quindi producono un range di valori di approssimativamente un centimetro, ed hanno come valor medio, rispettivamente 82,47 cm e 82,58 cm.

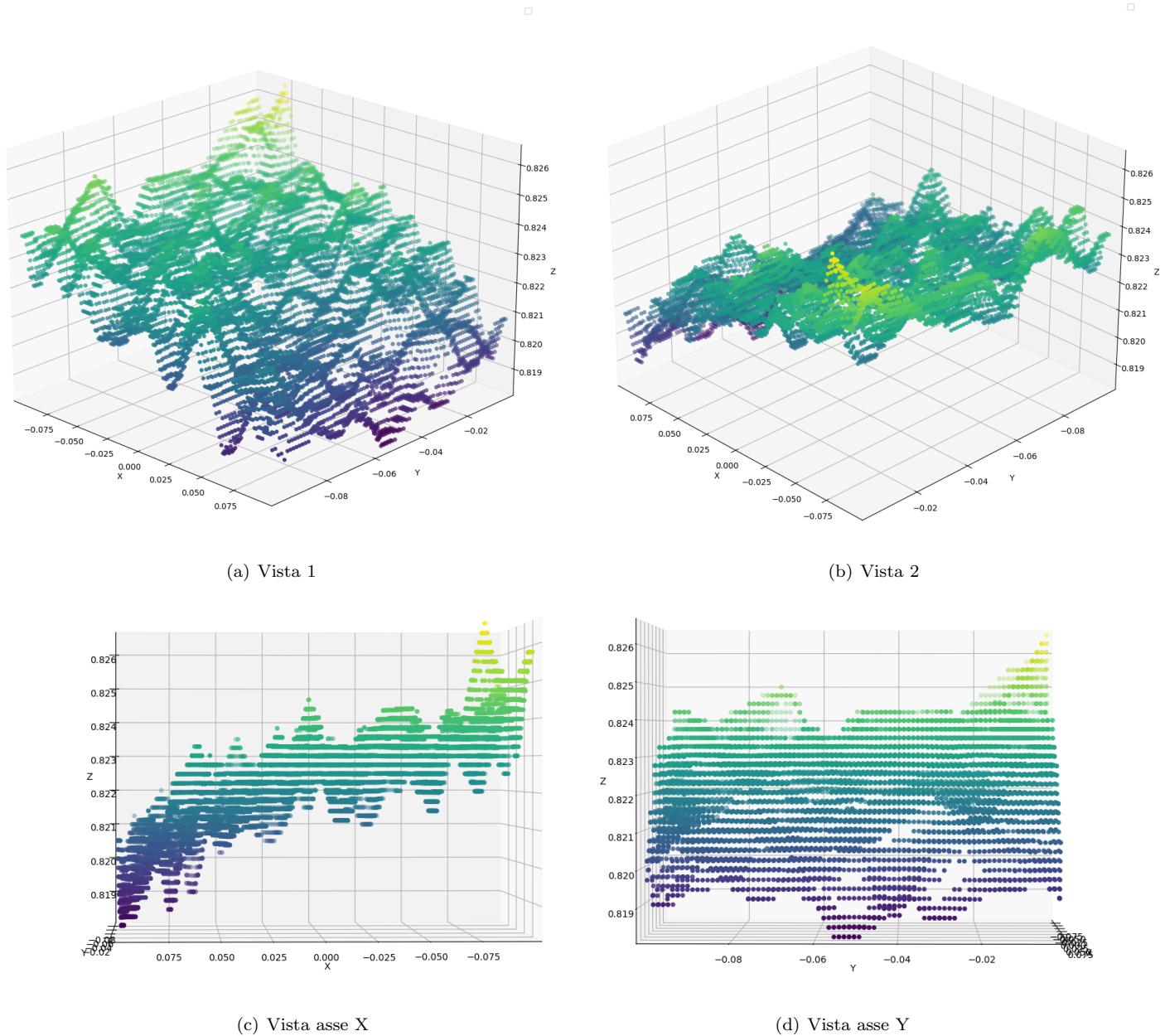


Figura 19: Variazione di profondità con RealSense L515

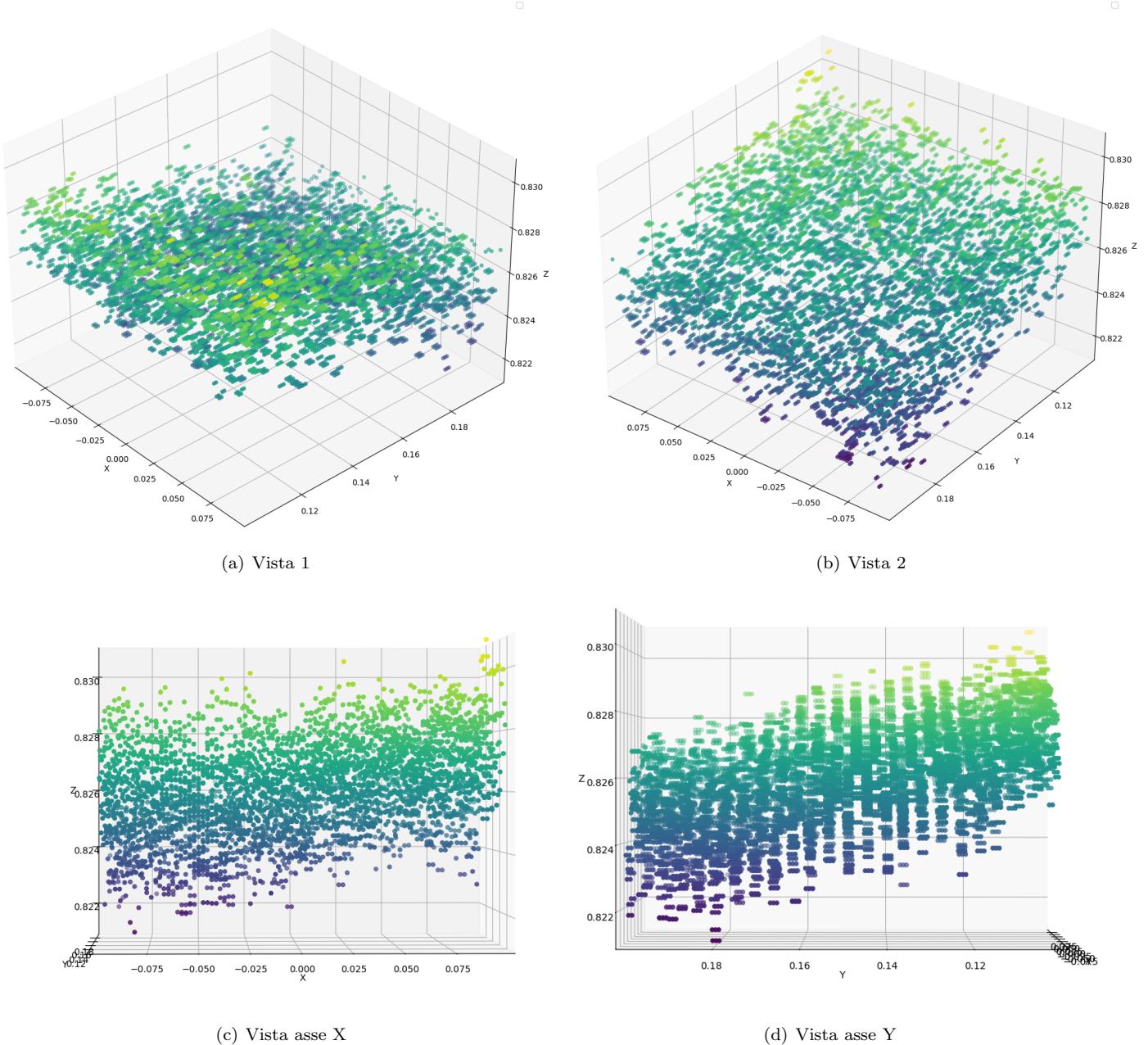


Figura 20: Variazione di profondità con Kinect v2

### 5.2.2 Confronto di ricostruzione

Come ulteriore metodo di paragone si è deciso di cercare di effettuare una ricostruzione da Pointcloud a Mesh 3D.

Data l'impossibilità di operare entrambe le camere in ambiente outdoor, è stato inizialmente deciso di effettuare il confronto di ricostruzione utilizzando un oggetto con forme di media complessità.

Tuttavia, dopo aver ricostruito correttamente la Pointcloud, il passaggio a Mesh 3D è risultato insoddisfacente.

Si è quindi scelto di utilizzare una scatola come oggetto di ricostruzione, in modo da mantenere dimensioni apprezzabili dalle camere, ma semplificando la geometria.

Essendo comunque la scatola di dimensioni sensibilmente più piccole rispetto a quelle di un veicolo, durante il processo di registrazione è stato deciso di mantenere lo sfondo (quindi il resto della stanza) invece che di eliminarlo con l'uso di un Threshold Filter.

Questo perché, date le dimensioni, sarebbe difficile ricostruire l'oggetto utilizzando come riferimento solo se stesso, cosa invece possibile nel caso di un veicolo.

Le acquisizioni sono state effettuate posizionando l'oggetto su un piano rialzato e girandoci attorno con la camera fino a prenderne ogni angolazione.

La bag registrata è stata poi elaborata dal nodo `registration.py` in modo da ottenere una pointcloud unica con accoppiamenti soddisfacenti e salvata nella sua interezza in formato `.ply` (comunemente utilizzato per salvare pointcloud).

Durante questo passaggio le acquisizioni Kinect sono risultate molto più semplici da allineare rispetto a quelle RealSense.

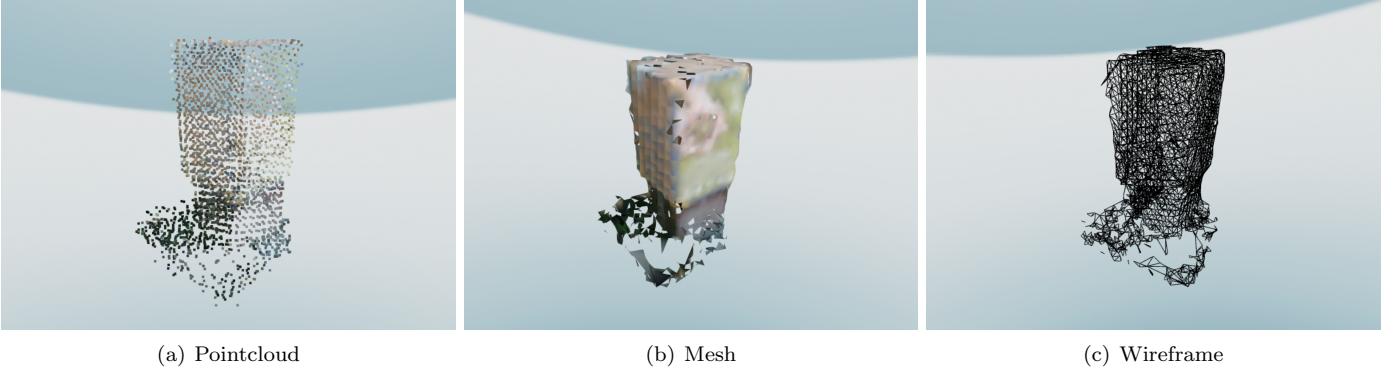
Per le prime infatti, la pipeline di ricostruzione utilizzata consiste semplicemente nell'applicazione di Colored ICP con moltiplicatore di 2 e successivamente una Point To Plane sempre con moltiplicatore di 2, che hanno prodotto sin da subito accoppiamenti soddisfacenti.

Per la camera RealSense invece si è utilizzata una pipeline leggermente più complessa: vengono utilizzate 3 coppie, composte da Colored ICP e Point To Plane, ma con moltiplicatore discendente, ovvero 100 per la prima, 10 per la seconda e 2 per la terza.

A questo punto, si è creato uno script per caricare le pointcloud, isolare l'oggetto ed effettuare la ricostruzione della Mesh.

In Figura 21 e 22 è possibile vedere i risultati in forma di Pointcloud, Mesh e wireframe della mesh.

Confrontandoli è possibile vedere, da una parte, come il maggior numero di punti delle acquisizioni RealSense producano una texture leggermente più niti-

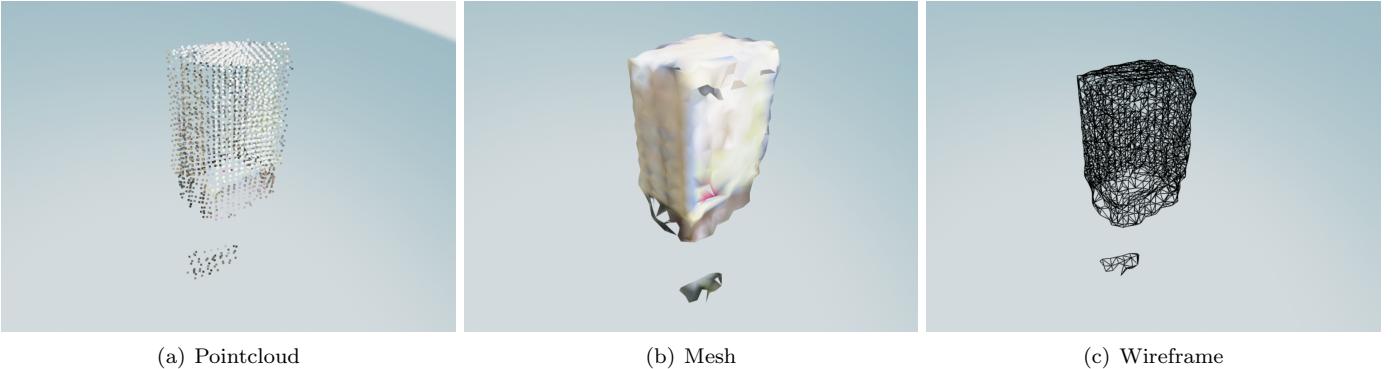


(a) Pointcloud

(b) Mesh

(c) Wireframe

Figura 21: Ricostruzione dell'acquisizione RealSense L515



(a) Pointcloud

(b) Mesh

(c) Wireframe

Figura 22: Ricostruzione dell'acquisizione Kinect v2

da e degli spigoli più evidenti, mentre dall'altra, l'acquisizione Kinect produce punti più ordinati che risultano in una mesh leggermente migliore.

### 5.2.3 Considerazioni sul confronto

Alla luce dei confronti tra le due camere, non vi è una netta distinzione tra i livelli di prestazioni.

Da una parte, la RealSense L515 offre sicuramente una precisione più alta ed una maggiore versatilità rispetto al Kinect. Basti pensare a tutte le opzioni che offre la camera, assieme alla possibilità di diversi filtri di post processing.

Dall'altra parte, il Kinect v2, in quanto sistema chiuso, non può offrire ciò che la L515 permette, ma, al prezzo di una precisione leggermente minore, non presenta quello che è uno dei suoi più grandi difetti: l'interferenza da parte della

luce solare.

Oltre a questo, il Kinect v2 permette la connessione di più camere senza l'inconveniente dell'interferenza reciproca, esonerando quindi dall'esigenza di hardware dedicato a temporizzare le camere per evitarla.

Un altro elemento degno di nota alla valutazione delle camere è che empiricamente è stato trovato che le Pointcloud prodotte dalla Kinect v2 vengono ricostruite in maniera molto più immediata rispetto a quelle prodotte dalla RealSense.

Si ritiene quindi che, dato il caso d'uso, tra le due camere il Kinect v2 risulti la scelta più cost effective

## 6 Conclusioni

Al termine di tutto il lavoro svolto, nonostante i problemi di ricostruzione, si ritiene che il progetto concept sia stato dimostrato appieno.

Il sistema è in grado di effettuare lo streaming su topic a partire da qualsiasi camera RealSense, tramite il nodo `realsense_streamer.py`, o da Kinect v2 tramite `kinect_streamer.py`.

Per entrambe le camere vengono pubblicati i frame a colori, ad infrarossi, di profondità, e la PointCloud calcolata; nel caso di streaming RealSense, vi sono in aggiunta le informazioni dell'IMU (se disponibile) e sulla camera.

È possibile, inoltre, sempre per lo streaming RealSense, applicare tutti i metodi di filtraggio disponibili nell'SDK fornito da Intel, quindi:

- Decimation Filter
- Threshold Filter
- Spatial Filter
- Temporal Filter
- Hole Filling Filter
- Depth To Disparity (solo serie D)

Il secondo nodo creato, `registration.py`, è in grado di mantenere in un buffer definito dall'utente le PointCloud pubblicate.

Con il buffer pieno è poi possibile calcolare il PoseGraph, allineando quindi le PointCloud, scegliendone la voxel size e se effettuare l'ottimizzazione o meno; è inoltre possibile scegliere se rimuovere o meno gli outlier delle PointCloud prima di calcolarne gli accoppiamenti.

La PointCloud ricostruita viene poi rimandata sul topic `/computed/points` o può essere salvata in formato `.ply`.

Infine, attraverso degli script Python, è possibile ricostruire utilizzando il Ball Pivoting Algorithm o la Poisson Screened Reconstruction il file appena salvato.

## Sviluppi futuri

Nonostante l'obiettivo come proof of concept sia stato raggiunto, il margine di miglioramento per il sistema è ancora molto.

### Miglioramento delle prestazioni

Una delle prime cose a poter essere migliorata sono le prestazioni del sistema: attualmente l'interezza del codice è implementato in Python, registrando comunque delle buone performance.

Nonostante questo, le limitazioni di Python sono note: non vi è il supporto nativo per il multithreading e, nonostante molte librerie (come `numpy` e in parte `open3d`) siano state ottimizzate, le prestazioni ne risentono.

Di conseguenza, passare ad un implementazione totalmente scritta in C++ si ritiene porterebbe dei grossi vantaggi, come ad esempio una ricostruzione delle Pointcloud in real time,.

Inoltre, nel caso in cui tutti i nodi girino sullo stessa macchina, sarebbe vantaggioso passare da nodo ROS a nodelet, in modo tale da non dover passare dalle limitazioni del protocollo TCP utilizzato dai normali nodi.

### Selezione della Pipeline

Per quanto riguarda invece la parte di accoppiamento delle nuvole di punti, nonostante sia perfettamente funzionante, non è ancora automatica.

Un'idea per raggiungere questo obiettivo potrebbe essere quello di creare un tool per scegliere dinamicamente la Pipeline ICP da utilizzare e quindi trovarne una che sia generalmente valida per il caso d'uso.

Un altro punto critico del sistema è sicuramente la fase di ricostruzione della Mesh 3D. I metodi utilizzati non sono risultati soddisfacenti per avere una ricostruzione completa.

Alcune soluzioni per migliorare questa parte del sistema potrebbero essere sia effettuare un filtraggio più pesante sulle Pointcloud che ricercare nuovi algoritmi per il meshing di Pointcloud.

### Ricostruzione 3D con Deep Learning

Un'altra strada percorribile per migliorare la ricostruzione 3D è quella di utilizzare reti neurali che si occupino del meshing al posto degli algoritmi proposti. Allo stato dell'arte attuale sono già presenti diverse reti in grado di ricostruire con ottimi risultati delle nuvole di punti.

## **Considerazioni sull'implementazione reale**

Per una possibile implementazione reale del sistema, una delle prime considerazioni da fare sono il numero di camere.

Dalle registrazioni effettuate con le RealSense D435i è stato reso evidente che posizionare solo una camera per lato orientata a 45° non permette una registrazione completa. Questo è dovuto principalmente al fatto che essendo limitato il FOV, la parte inferiore del veicolo viene leggermente distorta.

Immaginando il sistema come un portale, solidale con le camere, che si sposta su dei binari, il numero di camere ritenuto minimo è pari a 4: due poste lateralmente rispetto al veicolo e due in alto, ruotate verso il basso di 45°.

Nel caso in cui questa configurazione non fosse in grado di acquisire correttamente i punti relativi alla parte anteriore o posteriore del veicolo, si potrebbero aggiungere ulteriori camere, o semplicemente fare in modo che le due laterali possano ruotare sul posto e quindi riuscire nell'acquisizione.

Con la configurazione così descritta, si farebbe meno affidamento sulla ricostruzione ICP, essendo la posizione di ogni camera nota in ogni istante, che potrebbe comunque essere utilizzata per ovviare ad eventuali errori di misura.

Inoltre, il sistema sarebbe di semplice realizzazione: utilizzando ad esempio dei profilati Bosh-Rexroth, si garantirebbe facilità di costruzione, modularità e stabilità, oltre a poter utilizzare una vastissima gamma di accessori, tra cui motori che predispongono le canaline dei profilati ad essere usate appunto come binari.

A pagina successiva è presente un'immagine che schematizza il sistema appena descritto.

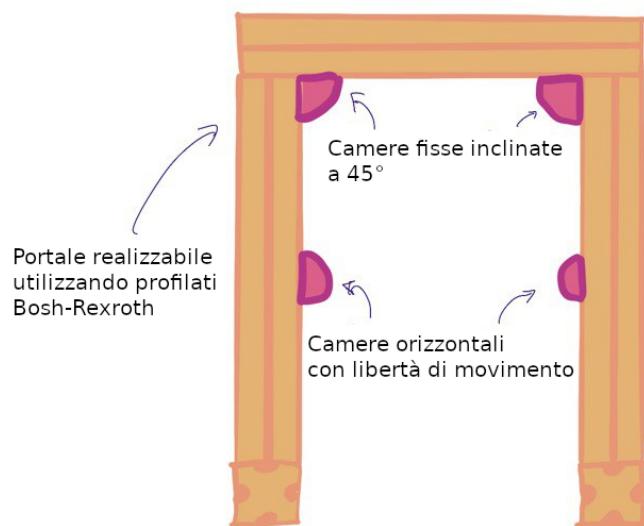
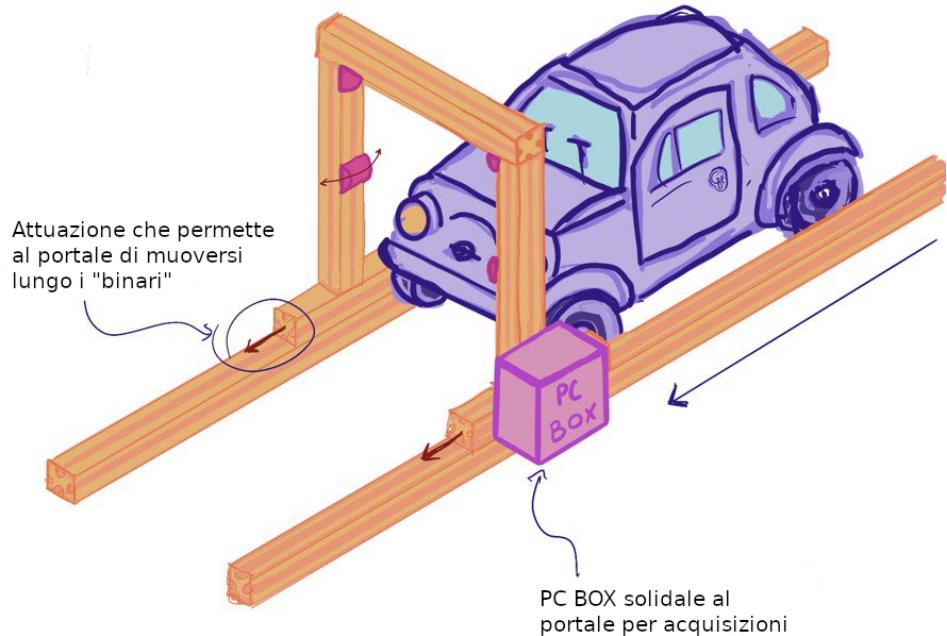


Figura 23: Idea di "portale" per acquisizioni