POLITECNICO
MILANO 1863

POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione

Dipartimento di Elettronica, Informazione e Bioingegneria

DEFINIZIONE DI UN FORMATO JSON PER LA RAPPRESENTAZIONE DI MAPPE ROBOTICHE TRIDIMENSIONALI

Relatore: prof. Francesco AMIGONI

Tesi di Laurea Magistrale di:

Francesco LAMONACA matr. 878764

Anno Accademico 2019-2020

Sommario

Lo scopo di questa tesi è la definizione di un formato per la rappresentazione di ambienti tridimensionali da utilizzare in ambito robotico. Tale lavoro si presenta come l'estensione di quello che ha definito lo standard IEEE Std 1873™-2015, standard analogo che rappresenta ambienti bidimensionali. Abbiamo individuato le principali modalità di rappresentazione del mondo tridimensionale e, per ciascuna di esse, definito il formato dei dati appropriato. Infine abbiamo effettuato alcuni test iniziali del formato da noi definito, convertendo in esso una serie di mappe già esistenti e disponibili pubblicamente.

Abstract

The aim of this thesis is the definition of a representation of the 3D space to be used in robotics. This work extends an analogous work that defined the IEEE Std 1873™-2015 for 2D environments. We identify the main representations for a 3D environment and, for each of them, we define an appropriate data format. Finally, we present some preliminary tests, that convert already existing maps, available publicly, to our format.

Indice

1 Introduzione

Per lo svolgimento efficiente delle operazioni per cui è progettato, un robot autonomo operante in un ambiente reale necessita di una rappresentazione dello spazio circostante. La conoscenza e l'esplorazione di tale ambiente diventa quindi un punto critico del suo corretto funzionamento, senza il quale esso potrebbe non essere in grado di svolgere le azioni per cui è progettato. La corretta definizione di un formato di rappresentazione dei dati spaziali diventa quindi importante tanto quanto la corretta progettazione degli algoritmi comportamentali. Spesso lo sviluppo di tale formato è compito dello sviluppatore stesso, in quanto scelte a livello di algoritmi comportamentali portano a preferire alcune strutture dati piuttosto che altre: questo comporta una stretta correlazione tra formato di rappresentazione dell'ambiente circostante e algoritmi che operano su tale rappresentazione.

Un'esigenza sempre più diffusa è quella di comunicazione tra robot differenti [1] [2]. Con un aumento sempre più importante dell'automazione di ambienti e operazioni, robot progettati da persone e per scopi differenti e realizzati da diversi costruttori si trovano ad interagire tra di loro. Per un corretto coordinamento delle azioni da svolgere, tali robot hanno la necessità di scambiarsi informazioni tra di loro e, tra queste, anche informazioni spaziali sull'ambiente circostante. È facile notare come, se ogni robot trasmettesse i dati nel proprio formato interno di rappresentazione del mondo circostante, tale comunicazione diventerebbe particolarmente difficile. In questo caso sarebbe necessaria la presenza di interpreti capaci di convertire tali informazioni da un formato all'altro. In questo modo, però, l'introduzione di una nuova tipologia di robot, con un modo differente di rappresentare il mondo circostante, renderebbe necessario aggiornare gli interpreti già presenti. L'esistenza di un formato comune di rappresentazione, stabilito a priori e conosciuto da tutti, renderebbe, invece, tale operazione di interscambio molto più semplice. Questo non significa negare i benefici di avere un formato proprietario, costruito secondo le esigenze architetturali degli specifici robot. Per ottenere entrambi i benefici (ottime prestazioni operazionali di un formato proprietario e facilità di comunicazione di uno standard unico) una possibile soluzione è quella di avere due rappresentazioni spaziali: una operazionale e una comunicativa. Tale soluzione non è esente da alcuni problemi: ad esempio, come effetto secondario, si incrementa la quantità di memoria necessaria per salvare i dati (visto che un'informazione può essere presente in entrambi i formati) oppure si aggiunge un ritardo nella comunicazione delle informazioni (in quanto tali informazioni devono essere prima convertite dal formato proprietario del mittente al formato standard comunicativo e, successivamente, dal formato standard comunicativo al formato proprietario del ricevente).

Un altro possibile rischio dell'esporre le strutture dati proprietarie è quello di rivelare troppo riguardo il comportamento del robot o la sua struttura interna. Come già detto, le strutture dati vengono definite in relazione alle esigenze operative degli algoritmi che definiscono il comportamento dei robot in questione. Spesso tali soluzioni sono soluzioni proprietarie, sviluppate da una società per vendere un determinato prodotto. La comunicazione di informazioni così legate al prodotto sviluppato comporta il rischio di rivelare anche alcuni aspetti di particolari soluzioni comportamentali, favorendo la concorrenza nello sviluppo di tecnologie competitive. Diventa, quindi, un'esigenza dell'azienda il comunicare meno informazioni possibili, col rischio di rendere la comunicazione tra robot progettati da aziende differenti difficile se non addirittura impossibile,

generando un ecosistema chiuso in cui solamente i prodotti provenienti dallo stesso produttore possono operare in coordinamento tra di loro.

Oltre ad una maggiore interoperabilità tra agenti differenti, la definizione di uno standard ha anche altri vantaggi a livello di sviluppo. La presenza di uno standard comune evita ai singoli produttori di sprecare tempo e risorse nella creazione di formati proprietari quando non necessario. In questo modo è possibile definire una soluzione architetturale di base, su cui poi, eventualmente, definire soluzioni proprietarie nel caso in cui si volessero migliorare alcune prestazioni. Inoltre, uno standard rende più semplice il confronto di prodotti differenti, sia a livello comportamentale che a livello prestazionale.

Lo scopo di questo lavoro di tesi è quello di definire un formato di rappresentazione dell'ambiente tridimensionale in cui un robot autonomo opera che possa essere proposto come standard comunicativo. Tale lavoro si presenta come l'estensione del lavoro che ha portato alla definizione dello standard IEEE Std 1873™-2015 [3] [4], standard testuale per la rappresentazione di ambienti bidimensionali. Il nostro lavoro è iniziato esaminando lo standard definito precedentemente, ricavandone la struttura base da cui partire ed eventuali problemi e limitazioni. Abbiamo, inoltre, analizzato altre soluzioni rappresentative già esistenti, considerando la loro efficacia o meno nel rappresentare ambienti tridimensionali vari e complessi in cui un robot può trovarsi. La nostra attenzione è stata rivolta principalmente su GML [5] [6], sui formati da esso derivati (principalmente CityGML [7] e IndoorGML [8]) e su X3D [9] [10]. Tali standard sono stati studiati principalmente perché rispettano molte delle specifiche da noi ritenute necessarie per le operazioni di un robot autonomo, tuttavia nessuno standard si è rivelato capace di soddisfarle tutte. In particolar modo nessuno di essi prevede la possibilità di riportare l'incertezza delle misurazioni presenti nelle mappe, una informazione spesso poco utile per la rappresentazione di ambienti tridimensionali ma di grande importanza in ambito robotico. In seguito abbiamo scelto JSON come linguaggio utilizzato per la definizione delle strutture dati, in quanto presenta la flessibilità e le performance migliori per i nostri scopi. Abbiamo poi definito quali tipologie di rappresentazione dello spazio tridimensionale supportare e come strutturare i dati riportati: le tipologie considerate includono Densegrid, Octree, Pointcloud, Polygonmesh e Sparsegrid. Per ognuna di esse abbiamo definito uno schema JSON, in grado di descrivere la struttura voluta e in grado di separare le mappe ben costruite da quelle che non seguono il formato da noi proposto. Infine, abbiamo individuato una serie di istanze di mappe già utilizzate in altri lavori e disponibili pubblicamente e abbiamo provato a rappresentarle col formato di nostra creazione, in modo tale da dimostrare la flessibilità e la capacità rappresentativa del formato definito. Inoltre, abbiamo misurato la dimensione dei file generati e il tempo impiegato per la conversione, in modo da trarre alcune prime conclusioni sulla fattibilità nell'adottare un formato del genere in ambito robotico.

Per quanto riguarda la struttura di questo elaborato:

- Nel Capitolo 2 descriveremo più approfonditamente il problema in esame, introducendo una prima classificazione di mappe e descrivendo gli standard già presenti, con una maggiore attenzione a GML, CityGML, IndoorGML e X3D.
- Nel Capitolo 3 presenteremo alcune scelte di design affrontate durante la definizione del formato, spiegando le motivazioni di tali scelte ed eventuali conseguenze.

- Nel Capitolo 4 presenteremo il formato da noi definito, descrivendo nel dettaglio le strutture dati generate e il loro significato.
- Nel Capitolo 5 analizzeremo alcuni risultati sperimentali ottenuti testando il formato da noi definito. Mostreremo alcuni esempi di file generati e trarremo alcune conclusioni in base alla loro dimensione e tempo di generazione.
- Infine, nel Capitolo 6 trarremo le conclusioni finali sul lavoro svolto ed evidenzieremo alcuni possibili sviluppi futuri.
- Nell'Appendice A riporteremo gli schemi completi che definiscono il nostro formato.
- Nell'Appendice B riporteremo una serie di tabelle dove presenteremo i vari campi che definiscono i file in questione e una breve descrizione di cosa essi rappresentino.

2 Stato dell'arte

Come già anticipato precedentemente questo lavoro di tesi si presenta come la naturale continuazione del lavoro che ha portato alla definizione dello standard IEEE Std 1873™-2015. Tale standard codifica la struttura dati necessaria per la rappresentazione e lo scambio di mappe metriche e topologiche *bidimensionali*. L'aumento dell'utilizzo di robot autonomi che operano in ambienti non controllati, aumento dovuto ad un affinarsi delle tecnologie in gioco e ad una diminuzione dei costi di adozione di tali tecnologie, ha reso evidente la necessità di uno standard del genere [4] [3]. Uno degli elementi basici affinché un robot agisca efficacemente in un ambiente di qualunque tipo è la presenza di una mappa, generata a priori o durante le operazioni (online), che gli permetta di localizzarsi e pianificare il proprio percorso. Tuttavia, sempre più spesso, un robot integra componenti sviluppati da produttori differenti o interagisce con altri device. Per rendere possibile un corretto svolgimento dei compiti previsti è necessario che questi agenti possano comunicare efficacemente tra di loro, scambiandosi le informazioni in loro possesso. Ed è in questo ambito che si inserisce il lavoro svolto: l'adozione di uno standard porta ad una maggiore interoperabilità senza dover prevedere in fase di sviluppo quali siano gli agenti in gioco e quali siano i modelli di rappresentazione da loro utilizzati. L'interoperabilità diventa un fattore critico in ambienti in cui produttori differenti contribuiscono a sviluppare solamente alcune funzionalità di quelle previste come può accadere, ad esempio, in ambito industriale o militare.

Come già detto, quindi, una *mappa* è indispensabile per l'orientamento e lo spostamento di un robot. Una prima, sostanziale, classificazione è quella che divide le mappe in mappe metriche e mappe topologiche. Una mappa metrica è una collezione di elementi con la seguente proprietà: dati due elementi a e b e una definizione di distanza, la distanza tra a e b può essere sempre calcolata sulla base delle informazioni in essa contenuta. Tale tipologia di mappa codifica esplicitamente l'aspetto fisico di un ambiente, mantenendo una forte corrispondenza tra l'ambiente reale e quello rappresentato. Una mappa metrica può, a sua volta, essere continua o discreta a seconda della granularità degli elementi rappresentati e del sistema di riferimento utilizzato. Come può essere facilmente intuibile, in una mappa metrica continua il sistema di coordinate presenta un insieme continuo di valori. Le mappe che utilizziamo quotidianamente, ad esempio quelle accessibili via Google Maps, sono esempi di mappe continue. Di contro le mappe metriche discrete decompongono la rappresentazione dell'ambiente in celle che costituiscono un pezzo atomico di informazione. È una tipologia di mappa ampiamente utilizzata in ambito digitale, proprio per la semplicità con cui programmi riescono a gestire le informazioni in essa riportate. Uno degli esempi più comuni sono le mappe presenti in ambito videoludico, specie negli strategici, in cui la rappresentazione è basata su una struttura composta da caselle esagonali. Alle mappe metriche si affiancano le mappe topologiche, mappe che basano la rappresentazione dell'ambiente su una struttura a grafo, dove i nodi rappresentano punti caratteristici dell'ambiente rappresentato e gli archi la diretta connessione tra due nodi. Un esempio di mappa topologica utilizzata anche in un contesto quotidiano è la mappa della metropolitana, dove i nodi sono le diverse stazioni e gli archi rappresentano i collegamenti tra una stazione e la successiva. Una mappa topologica rappresenta meno informazioni rispetto a una mappa metrica: non è possibile calcolare la distanza tra due nodi qualsiasi o, in alcuni casi, non è possibile calcolarla proprio (come comunemente avviene nel caso della metropolitana). Inoltre, dato che non rappresenta fedelmente tutte le proprietà dell'ambiente reale, è necessario prevedere un meccanismo di localizzazione che, ad esempio tramite il

riconoscimento di alcuni punti di riferimento, rende possibile al robot capire in quale nodo della mappa si trova. L'enorme vantaggio di una mappa topologica è che un problema di pianificazione di percorso diventa un problema di esplorazione di un grafo, ampiamente studiato e conosciuto in letteratura. In Figura 1 presentiamo un esempio per ciascuna delle tre mappe definite, in due dimensioni.

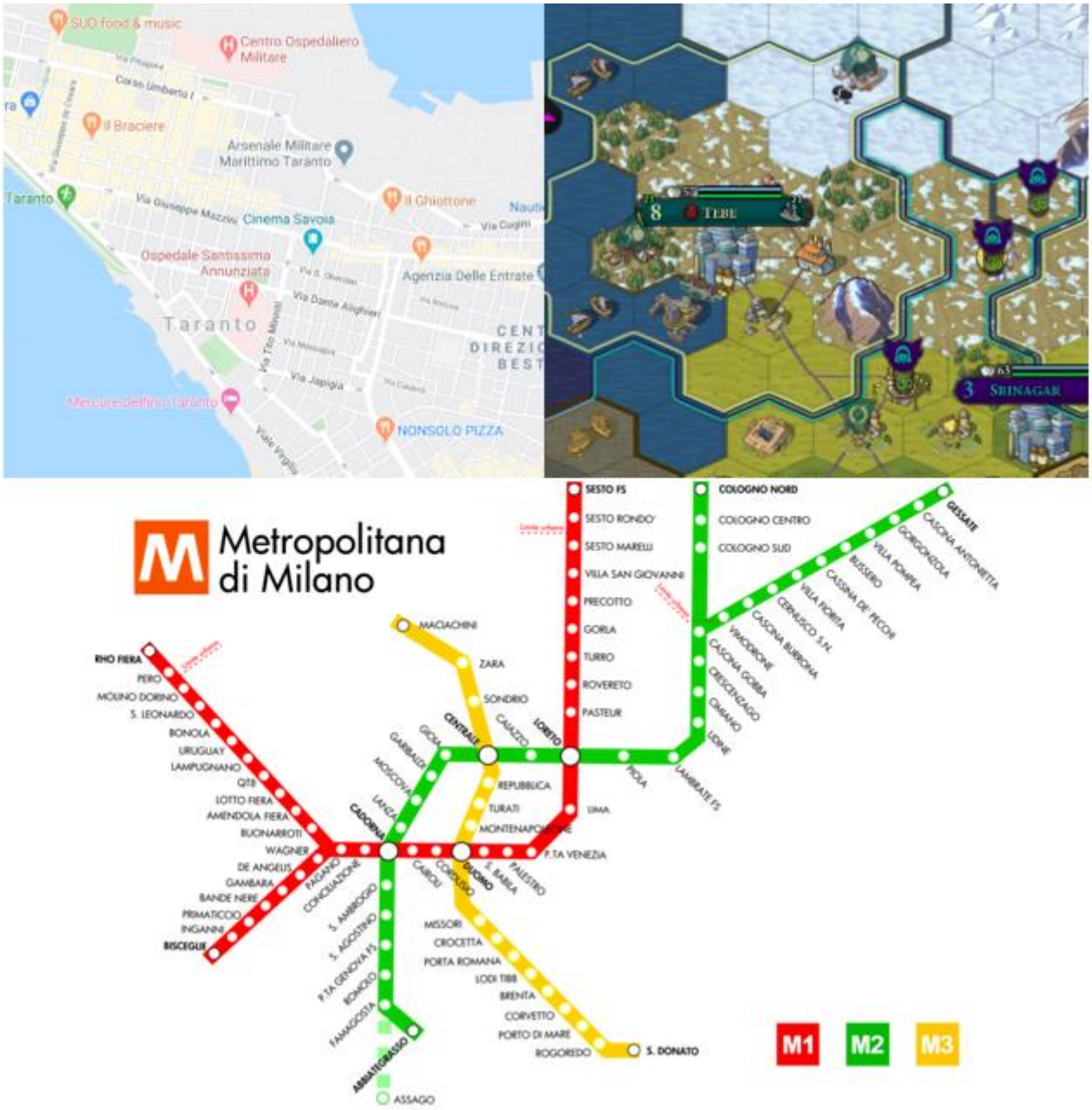

Figura 1 Tre esempi di mappe bidimensionali. In senso orario una mappa metrica continua da Google Maps [11], una mappa metrica discreta dal videogioco Civilization VI [12] e una mappa topologica (la metro di MIlano) [13]

Tutte le tipologie di mappe prima definite possono essere necessarie per il corretto comportamento di un robot e, quindi, l'idea dello standard IEEE Std 1873™-2015 è quello di poterle rappresentare tutte. La rappresentazione di tali mappe, tuttavia, è da considerare solo nell'ambito di comunicazione tra diversi agenti e non in ambito operativo. Non è compito dello standard IEEE Std 1873™-2015 (e di riflesso neanche di questo lavoro di tesi) rappresentare le mappe utilizzate per localizzare un robot o per pianificare un percorso ottimale. Si è lasciata, quindi, piena libertà ad ogni

produttore di avere la propria struttura dati operativa, purché si rendano disponibili delle API per la conversione tra il formato definito dallo standard e quello proprietario. Un altro punto fermo del lavoro precedentemente svolto è stato quello di focalizzarsi solamente su mappe statiche bidimensionali, mentre mappe tridimensionali e dinamiche sono state volutamente escluse dallo standard IEEE Std 1873™-2015 ed eventualmente possibile oggetto di lavori futuri. A questo punto si inserisce il nostro lavoro di tesi, con il compito di proporre un formato per mappe tridimensionali senza comunque andare a sostituire il lavoro svolto nel mondo 2D. L'idea, infatti, non è quella di stabilire uno standard unico che gestisca sia il mondo bidimensionale che quello tridimensionale, bensì definirne uno per le mappe in tre dimensioni che lavori in parallelo con quello prodotto precedentemente.

Un'ulteriore distinzione tra tipologie di mappe è stata introdotta nel lavoro precedente e verrà ereditata in questo lavoro di tesi. Si è voluta creare una gerarchia, definendo una mappa globale e diverse mappe locali in essa contenuta. Senza entrare troppo nel dettaglio di questa implementazione (che verrà discussa in seguito) ecco una breve introduzione:

- Una mappa globale è rappresentata tramite un grafo. I nodi di questo grafo sono le singole mappe locali, mentre gli archi sono le rototraslazioni necessarie per passare dal sistema di riferimento di una mappa locale a quello di un'altra mappa locale.
- Una mappa locale è una mappa nel senso stretto del termine, metrica o topologica, e rappresenta una porzione di ambiente più o meno circoscritta. Presenteremo in seguito diverse tipologie di mappe locali, ognuna con i propri vantaggi e svantaggi in termini di informazioni presentate.

Un modello del genere presenta diversi punti di forza. Sicuramente una flessibilità maggiore nel rappresentare realtà differenti. Dato che è possibile scegliere una diversa tipologia di mappa locale per ogni porzione dell'ambiente senza perdere di vista l'ambiente nel suo insieme, è possibile ogni volta ottimizzare tale scelta in base alle peculiarità della porzione in esame. Inoltre si garantisce una certa modularità alla mappa globale, con la possibilità di aggiungere nuove porzioni senza dover riscrivere la mappa nel suo insieme. Terzo, non è necessario avere l'intera mappa in memoria ma è possibile caricarne solamente la porzione utile al problema in esecuzione.

A livello implementativo lo standard IEEE Std 1873™-2015 ha scelto XML come il linguaggio più adatto per rappresentare i dati. Il formato XML (ovvero eXtensible Markup Language) [14] è un metalinguaggio per la definizione di linguaggi marcatori che consente di definire il significato dei singoli elementi del file analizzato, ampiamente utilizzato anche in ambito di dati geospaziali, come dimostrato successivamente. Grazie alla possibilità data da XML di generare tag personalizzati è stato possibile ricreare tutte quelle meta-informazioni necessarie per la rappresentazione della mappa. Il modello ad albero su cui si basa XML è facilmente adattabile al sistema di mappe globali e locali descritto precedentemente. Nonostante XML sia un'ottima scelta dal punto di vista di design, si è deciso di abbandonare tale standard per il lavoro nell'ambito tridimensionale in favore del formato JSON (JavaScript Object Notation) [15]. L'adozione di tale formato eredita gran parte dei punti di forza di XML, andando ad aggiungerne di nuovi, quali una minor dimensione dei file, una maggior velocità di lettura e una struttura più in linea con i principali linguaggi di programmazione. Tali novità verranno approfondite successivamente.

Un legittimo dubbio potrebbe essere l'effettiva necessità di sviluppare un nuovo standard quando potrebbero esistere soluzioni già affermate e consolidate. In effetti in letteratura esistono già altri standard dedicati alla rappresentazione di ambienti (sia 2D che 3D). L'Open Geospatial Consortium (OGC) [16] è un'organizzazione no-profit col compito di raccogliere e definire standard open source per dati geospaziali e di localizzazione. Una lista non esaustiva di standard da essa definiti comprende: Geography Markup Language (GML) [17] [6], le sue varie derivazioni come CityGML [7], InfraGML [18], IndoorGML [8], GeoSPARQL [19]. Procederemo in seguito a descrivere i più importanti, ora ci dedichiamo ad una presentazione generale. Ognuno di questi standard presenta una propria specializzazione, che sia questa relativa alla tipologia di ambiente da rappresentare (CityGML è specializzato, ad esempio, sulla rappresentazione di ambienti cittadini) o relativa all'ambito di utilizzo (GeoSPARQL è specializzato in mappe compatibili con query SPARQL).

Tali standard, tuttavia, presentano due criticità importanti per l'ambito di applicazione robotico considerato in questo lavoro di tesi:

- Risoluzione della mappa. Ogni standard si dedica ad una particolare realtà geografica, che può essere un appartamento, un terreno o una città. Nessuno standard sembra avere la possibilità di descrivere in maniera ottimale un ambiente a prescindere dalla sua estensione o articolazione.
- Incertezza della misurazione. Gli standard sopra citati si basano sull'assunzione che i dati rappresentati siano certi. Tale assunzione è perfettamente ragionevole, considerando che per la costruzione del modello di un ambiente si parte spesso dai dati di progetto (per esempio modelli CAD) e ci si avvale di strumentazione avanzata per la misurazione e modellizzazione dell'ambiente in esame. Tale assunzione, tuttavia, viene meno in ambito robotico: la costruzione della mappa dell'ambiente in cui opera un robot viene effettuata dal robot stesso sulla base delle proprie misurazioni, intrinsecamente rumorose e soggette ad errori. È necessaria la possibilità di rappresentare l'incertezza associata all'informazione rappresentata. In un certo senso, gli standard esistenti sono più orientati alla rappresentazione di ambienti *progettati* o disegnati che alla rappresentazione di ambienti *percepiti* tramite sensori.

Procederemo, ora, ad una descrizione più approfondita di GML e delle sue espansioni più importanti: CityGML e IndoorGML. Inoltre presenteremo anche X3D come alternativa alla famiglia di standard definiti a partire da GML.

2.1 GML

Geographic Markup Language (GML) è uno standard di codifica di dati geografici definito in XML e sviluppato dall'OpenGIS Consortium (GIS) [17] [6] [5]. GML rappresenta lo spazio tramite collezione di *feature*. Una feature altro non è che la rappresentazione di un oggetto nel mondo reale, come una strada, un edificio, un fiume. Ad una feature vengono assegnate due categorie di qualificazioni:

- Le **property**: una property è una caratteristica dell'oggetto rappresentato. Un esempio potrebbe essere il nome di un edificio, la fonte di un fiume o il materiale di una strada.
- Le **geometry**: una geometry è una caratteristica geometrica di una feature, che la posiziona nello spazio e la caratterizza. Le geometry possono essere definite a vari livelli di risoluzione: da un singolo punto che rappresenta il centro dell'oggetto in questione, ad una

rappresentazione lineare che definisce il suo perimetro in due dimensioni o, nel caso di una strada o di un fiume, il suo asse centrale di sviluppo.

Una mappa in GML, quindi, è definita tramite una collezione di feature posizionate nello spazio attraverso le loro geometry. Come abbiamo detto sono quest'ultime le responsabili del posizionamento dell'oggetto nello spazio e, quindi, è in questo campo che vengono riportate le sue coordinate. Coordinate che in GML (e negli standard da esso derivati) possono essere espresse sia in relazione ad un sistema di riferimento proprio della mappa, sia tramite delle coordinate geografiche, riportando esplicitamente latitudine e longitudine. Negli esempi che riporteremo in seguito, per semplicità utilizzeremo la prima possibilità, ipotizzando l'esistenza di un sistema di riferimento relativo alla mappa in esame. In XML vi sono due possibilità per riportare il campo coordinate: o specificando i singoli campi tramite l'annotazione *<coord>* oppure come una singola stringa annotata con *<coordinates>*. Di seguito riportiamo l'esempio di un punto in uno spazio bidimensionale espresso in entrambi i modi:

```
<Point srsName="http://www.opengis.net/gml/srs/epsg.xml#4326">
    <coord><X>5.0</X><Y>40.0</Y></coord>
</Point>
```

```
<Point srsName="http://www.opengis.net/gml/srs/epsg.xml#4326">
    <coordinates>5.0,40.0</coordinates>
</Point>
```

Il vantaggio del primo approccio è una maggior granularità del file che permette ad un parser di effettuare controlli sulle singole coordinate, controlli complessi o impossibili da fare su una stringa. Il vantaggio del secondo approccio, invece, è un minor impiego di memoria. Come vedremo più approfonditamente nel Capitolo 3 sul confronto tra i formati JSON e XML, una maggior verbosità del documento porta ad un consumo maggiore di risorse, specie ipotizzando mappe complesse in cui il numero di elementi rappresentati è di diverse migliaia.

Il punto è l'unità geometrica più semplice ipotizzabile, tuttavia strutture più complesse vengono definite, come *LineString*, ovvero un elenco di punti collegati tra loro da segmenti lineari, *Box*, una sezione di spazio compresa tra due punti che rappresentano le coordinate minime e massime o *Polygon*, una superficie connessa piana, dove ogni coppia di punti in essa contenuta è collegabile tramite un percorso interno al poligono stesso.

```
<Polygon gid="_98217" srsName="http://www.opengis.net/gml/srs/epsg.xml#4326">
    <outerBoundaryIs>
        <LinearRing>
            <coordinates>
                0.0,0.0 100.0,0.0 100.0,100.0 0.0,100.0 0.0,0.0
            </coordinates>
        </LinearRing>
    </outerBoundaryIs>
    <innerBoundaryIs>
        <LinearRing>
            <coordinates>
                10.0,10.0 10.0,40.0 40.0,40.0 40.0,10.0 10.0,10.0
```

```
            </coordinates>
        </LinearRing>
    </innerBoundaryIs>
</Polygon>
```

Nella sezione sopra riportata è rappresentato un poligono definito su uno spazio piano. Tale geometry è a sua volta definita da due geometry di tipo LinearRing (un tipo di LineString i cui estremi coincidono rendendo la linea ciclica): una che definisce il perimetro esterno e una che definisce il perimetro interno. In base al contesto in cui tale geometry si trova, si possono dare diversi significati ad essa: può rappresentare una piazza (rappresentata dal perimetro esterno) al cui interno vi è una fontana (rappresentata dal perimetro interno) o uno stadio rappresentato sempre dal perimetro esterno, mentre il perimetro interno rappresenta il campo da gioco.

Definite le geometry, è possibile definire le feature. Come abbiamo detto una feature è una rappresentazione in GML di un oggetto del mondo reale. A differenza delle geometry, già definite dagli schemi base presenti nello standard, per quanto riguarda le feature è compito dello sviluppatore definire gli schemi associati alle feature più complesse. Come vedremo successivamente, tale compito è relegato alle espansioni di GML che, partendo dalla stessa struttura base, introducono delle feature specifiche di determinati domini di applicazione.

```
<Building>
    <buildingName>Politecnico di Milano</buildingName>
    <foundationYear>1863</foundationYear>
    <abbreviation>PoliMI</abbreviation>
    <gml:location>
        <gml:Point>
            <gml:coord><gml:X>1.0</gml:X><gml:Y>1.0</gml:Y></gml:coord>
        </gml:Point>
    <gml:location>
</Building>
```

Nel frammento XML sopra riportato definiamo una prima descrizione del Politecnico di Milano, definendo alcune property (il nome, l'anno di fondazione e un suo nome abbreviato) e una geometry (puntiforme) che descrive le sue coordinate. Questa è solo una delle possibili descrizioni del Politecnico, un'altra potrebbe usare il concetto di LinearRing per definire il perimetro del campus o, addirittura, dei singoli edifici.

Definite geometry e feature possiamo ora definire una collezione di feature, che rappresenterà la mappa:

```
<CityModel fid="Cm1456">
    <dateCreated>Feb 2000</dateCreated>
    <gml:featureMember>
        <River fid="Rv567">....</River>
    </gml:featureMember>
    <gml:featureMember>
        <River fid="Rv568">....</River>
    </gml:featureMember>
    <gml:featureMember>
        <Road fid="Rd812">....</Road>
    </gml:featureMember>
```

```
</CityModel>
```

Nell'esempio sopra riportato definiamo una città che è costituita da due fiumi e una strada. Ovviamente il modello può essere arricchito a piacimento, aggiungendo ulteriori tipologie di feature e associando ad ognuna di essa più property o geometry.

L'approccio di GML al problema di rappresentazione geografica è senza dubbio valida e interessante. Tramite la rappresentazione separata di oggetto fisico (feature) e sua rappresentazione geografica (geometry) si semplifica enormemente la definizione dei campi da riportare. Le feature con le property ad esse associate permettono di avere una struttura comune a tutti gli oggetti di una certa categoria (ad esempio tutti gli edifici devono avere un nome, un numero di piani, un indirizzo…). Tuttavia non si è vincolati nella rappresentazione grafica, rendendo più semplice adattarla in relazione alle esigenze del singolo edificio (ad esempio un grattacielo, di sviluppo verticale, può avere esigenze rappresentative differenti rispetto ad un campus universitario, di estensione prevalentemente orizzontale, nonostante siano entrambi edifici o ancora mappe con scopi differenti possono aver bisogno di rappresentazioni differenti dello stesso edificio). Come vedremo questa possibilità sarà sfruttata da CityGML, descritta nella Sezione successiva. Per quanto interessante, GML non sembra essere la soluzione ottimale per la descrizione di ambienti per robot. Per prima cosa, perché GML pone il suo focus principale sugli elementi dello spazio, le feature appunto, descrivendo l'ambiente in funzione di esse. Così facendo, però, rende impossibile la descrizione dello spazio vuoto e l'assegnazione di alcune caratteristiche a punti specifici dello spazio, indipendentemente dalla loro appartenenza o meno ad un particolare oggetto. Ad esempio vorremmo poter associare ad ogni punto se questo è occupato o meno, in modo da poter elaborare dove sia possibile muoversi e dove no. Inoltre GML suppone una conoscenza a priori dello spazio da rappresentare, degli oggetti in esso contenuto e delle loro posizioni. Tale supposizione è perfettamente razionale nel problema di costruzione di una mappa, meno nel caso di navigazione di un robot. Infatti solitamente è quest'ultimo che acquisisce informazioni dall'ambiente circostante tramite una serie di sensori e poi, solo in seguito, inizia il processo di classificazione. Infine, l'ultima importante limitazione di GML è che non è uno standard completo. Gli schemi che definisce sono, in realtà, dei meta-schemi che definiscono solamente gli elementi base con cui costruire le definizioni personalizzate di feature. Come vedremo sarà poi compito delle estensioni di GML, come CityGML o IndoorGML, a portare a termine questo compito, sviluppando le feature relative ad un particolare dominio di sviluppo.

2.2 CityGML

CityGML è uno standard di rappresentazione di dati geospaziali costruito a partire da GML. È un'estensione dello standard prima definito da cui eredita i concetti di feature e geometry descritti nella Sezione 2.1. A GML aggiunge una serie di definizioni sia per le geometry sia, soprattutto, per le feature necessari per descrivere in maniera ottimale ambienti cittadini 3D [20].

Per quanto riguarda le geometry, le modifiche riguardano soprattutto l'estensione dei concetti dal mondo bidimensionale al mondo tridimensionale. Questo perché, seppur GML permetta di rappresentare oggetti in tre dimensioni, tale possibilità non viene supportata appieno, mantenendo il focus sul mondo bidimensionale. CityGML, al contrario, pone il suo focus esplicitamente sul mondo 3D. Le geometry Point in CityGML devono obbligatoriamente avere tre valori per le coordinate, le varie geometrie lineari possono svilupparsi lungo i tre assi, così come i poligoni. L'unica limitazione

ancora in vigore è il vincolo nel rappresentare solamente figure piane. Tale vincolo non pone grandissime limitazioni su cosa si possa rappresentare e cosa no, purché si scompongano figure complesse in figure semplici piane, con un approccio analogo a quanto vedremo successivamente nella Sezione 4.2.4.

Per quanto riguarda le feature, invece, CityGML introduce una serie di nuovi schemi in grado di definire gli oggetti più comuni che caratterizzano il paesaggio cittadino. Per semplicità di consultazione, queste definizioni vengono divise in moduli, ognuno con un tema specifico. Sono presenti, ad esempio, un modulo dedicato agli edifici, uno dedicato alle strade, uno ai vari tipi di terreno… In questo modo lo sviluppatore avrà la possibilità di scelta su quali elementi incorporare nel proprio modello e quali no. Un'importante aggiunta di CityGML è il concetto di *Level of Details* (**LoD**), il livello di dettaglio con cui viene rappresentata una feature. Ogni feature, infatti, può essere descritta con LoD differenti, che via via aggiungono sempre più dettagli alla descrizione. In questo modo, a differenza di GML, non avremo più una definizione per ogni feature, ma una collezione di definizioni, una per ogni LoD supportato. Per ogni modulo vengono descritte le linee guida che caratterizzano ogni LoD. Ad esempio, per quanto riguarda il modulo degli edifici, CityGML individua cinque LoD, indicati con numeri da 0 a 4. Un edificio di cui si conosce solamente il perimetro esterno è rappresentato con LoD 0. Se, invece, viene riportato il volume che esso occupa allora è LoD 1. Aumentano la risoluzione è possibile descrivere l'aspetto esterno (LoD 2), le aperture sulla facciata come porte e finestre (LoD 3) e, infine, l'aspetto interno (LoD 4). In Figura 2 un esempio dei cinque LoD previsti da CityGML per un edificio.

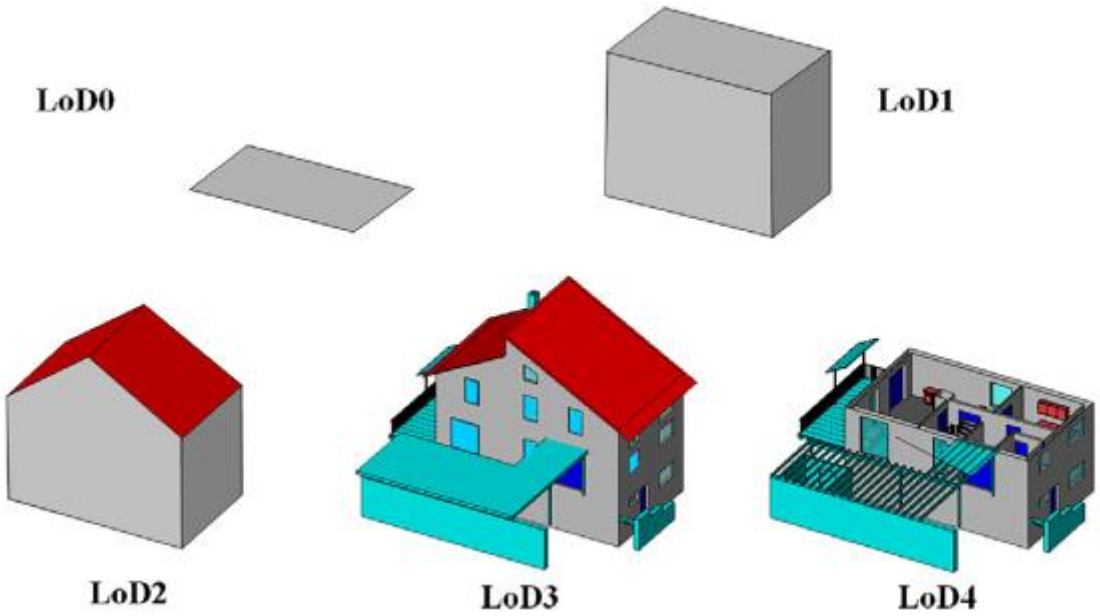

Figura 2 Cinque livelli di dettaglio previsti da CityGML [21]

CityGML si è dimostrato, nel corso degli anni, uno strumento valido per rappresentare ambienti cittadini anche vasti e complessi, grazie alla sua natura modulare e alla sua collezione completa di feature. Un esempio importante è il lavoro svolto nella rappresentazione della città di Berlino, progetto sviluppato inizialmente grazie al supporto dell'Institute for Cartography and Geoinformation, presso L'università di Bonn, con il benestare del Senate Department for Economics, Technology and Research di Berlino. Il modello di Berlino vanta all'incirca 550.000 edifici descritti

tramite le specifiche CityGML di LoD 2 e compresi in un'area di circa 890 km^2 [22] [23]. Tale modello, nel corso degli anni, è stato utilizzato negli ambiti più disparati, dalla localizzazione per servizi di business, al sostegno di analisi e formulazione di policy e decisioni in ambito politico e civile [24].

Anche nel caso di CityGML, come nel caso di GML, il focus sulle feature e l'impossibilità di descrivere lo spazio al di fuori di esse rende tale standard inadatto allo scopo del nostro lavoro. Definire e posizionare delle feature nello spazio presuppone una conoscenza a priori dell'ambiente circostante che non sempre è possibile. In secondo luogo, la specializzazione di CityGML per ambienti cittadini porta ad una perdita di flessibilità dello standard. Ad esempio, se non ci sono problemi nel modellare le città, diventa più complesso farlo per spazi interni, come un magazzino o un appartamento, e per spazi esterni non strutturati, come foreste o ambienti sottomarini, in quanto mancano una serie di feature specifiche per questi ambienti. Questo è in conflitto con il nostro proposito di proporre uno standard il quanto più generico e agnostico possibile, in modo tale da applicarlo ai differenti contesti di utilizzo in ambito robotico.

2.3 IndoorGML

IndoorGML si presenta come l'estensione GML speculare a CityGML: se quest'ultimo pone il suo focus principale sugli ambienti esterni, specialmente cittadini, IndoorGML si specializza sugli ambienti interni e, in particolar modo, su come questi possono essere navigati. Infatti un ambiente interno, a differenza di uno esterno, presenta una geometria più articolata che rende più complesso definire il percorso che collega due punti. La presenza di ostacoli (come, ad esempio, i muri) e di punti di passaggio (porte, scale…) rendono necessaria la definizione di un concetto di raggiungibilità che sia proprio di questa tipologia di ambienti [8] [25].

Essendo anch'esso un'estensione di GML, IndoorGML eredita da questo i concetti di feature e geometry visti nella Sezione 2.1. A questi affianca una serie di concetti nuovi, quali la definizione di cella, la definizione di uno spazio duale e un modello di spazio multilivello. Già nell'esperienza quotidiana, dividiamo lo spazio interno in sezioni ben definite: un appartamento è suddiviso in stanze, un edificio è suddiviso in piani… Tale suddivisione viene formalizzata in IndoorGML tramite il concetto di cella (*cell*). Una cella è l'elemento atomico che costituisce lo spazio rappresentato e ad essa vengono associate le proprietà (ad esempio il nome, la funzionalità…) e una geometria. Ad esempio una stanza può essere vista come l'elemento atomico della mappa rappresentata e la sua geometria è definita dal solido che la delimita. In IndoorGML vi sono tre possibilità di rappresentazione della geometria assegnata ad una cella:

1. Non viene assegnata alcuna geometria. La prima possibilità è la più semplice di tutte: la geometria assegnata alla cella non viene rappresentata, di fatto facendo degenerare la mappa in una mappa topologica, la cui definizione è stata affrontata precedentemente.
2. La definizione di una geometria viene demandata ad un altro standard. L'esempio più diffuso è definire una geometria tramite una rappresentazione in CityGML di LoD4 e nella rappresentazione in IndoorGML assegnare un link a tale definizione.
3. La definizione della geometria viene gestita internamente. In questo caso la geometria viene gestita utilizzando gli schemi XML propri di IndoorGML. Nel caso in cui la cella così definita debba essere integrata in uno spazio esterno (ad esempio se vogliamo rappresentare diversi appartamenti ognuno con la sua rappresentazione in IndoorGML) bisognerà gestire i collegamenti tra sezioni differenti, secondo un meccanismo spiegato successivamente.

Una volta definito l'insieme di celle che costituiscono l'ambiente rappresentato, è necessario definire la raggiungibilità tra celle differenti. IndoorGML risolve tale problema tramite la costruzione di uno spazio duale: un elemento K-dimensionale in uno spazio N-dimensionale nello spazio topografico viene convertito in un oggetto (N-K)-dimensionale nello spazio duale. In questo modo, definito uno spazio tridimensionale, una stanza (elemento a tre dimensioni) viene ridotto ad un elemento a zero dimensioni (3-3=0), ovvero un punto, mentre una porta (intesa come un oggetto a due dimensioni) viene ridotta ad un elemento a una dimensione (3-2=1), ovvero una linea. Quello che si viene a creare, alla fine, è un grafo che rappresenta la connettività tra celle differenti.

In Figura 3 troviamo un esempio di spazio topografico e duale. Lo spazio rappresentato è composto da due celle (R1 e R2) e da uno spazio esterno (Ext), convertiti in nodi del grafo nello spazio duale. I muri e le porte sono, invece, convertiti in archi del suddetto grafo. In questo modo è possibile capire l'adiacenza tra le celle (la cella R1 è adiacente allo spazio esterno attraverso il muro B1, mentre è adiacente alla cella R2 attraverso i muri B2 e B3) e la raggiungibilità (dal grafo si può capire che per raggiungere lo spazio esterno dalla cella R1 è necessario prima raggiungere la cella R2 attraverso le porte D1 o D2 e, da lì, raggiungere lo spazio esterno attraverso la porta D3). In particolare il nodo Ext è un nodo speciale, detto *Anchor Node*, in quanto non rappresenta di per sé una cella ben definita, ma è più un nodo di collegamento tra spazi interni differenti.

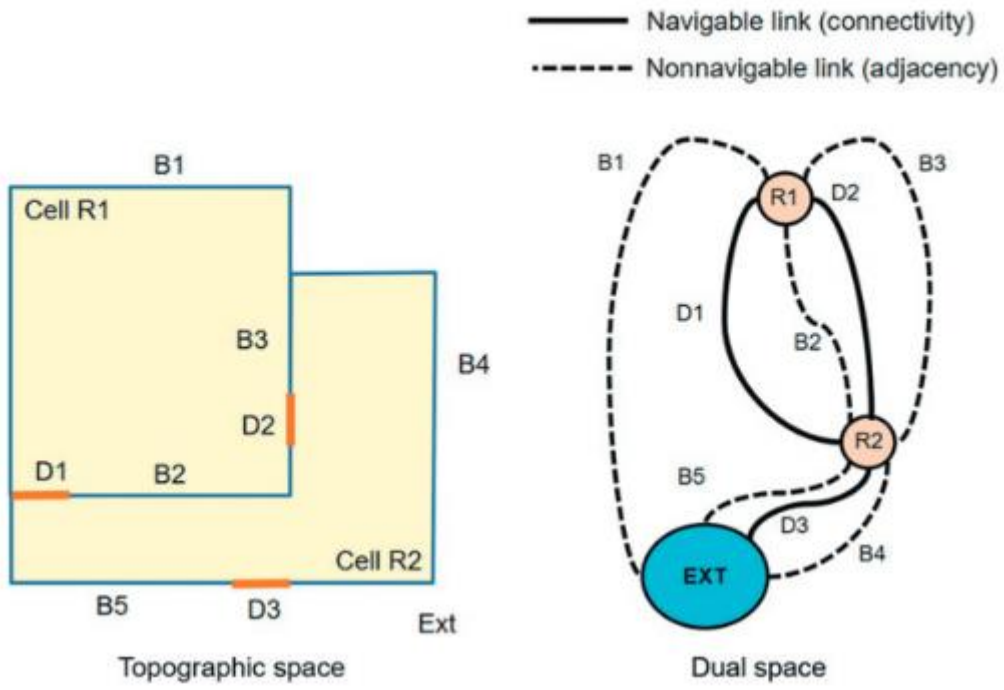

Figura 3 Esempio di spazio topografico e duale [8]

Infine, l'ultimo concetto introdotto da IndoorGML è la struttura multilivello dello spazio rappresentato. Uno stesso spazio può essere interpretato in maniera differente a seconda dell'attore coinvolto o del contesto in cui opera. Ipotizziamo, ad esempio, uno spazio costituito da una serie di stanze in cui due sono collegate tra di loro attraverso una scala. A seconda che l'attore coinvolto sia su una sedia a rotelle o meno, il grafo di raggiungibilità cambia. In Figura 4 possiamo

vedere un esempio in tal senso. La Room 3 presenta al suo interno una rampa di scale. Per una persona abile nel percorrerla tale stanza è una cella unica, mentre per una persona su sedia a rotelle tale stanza viene suddivisa in due celle separate e non collegate tra di loro: per andare dalla Room 3a alla Room 3b o viceversa è necessario passare dall'esterno oppure passare dalle Room 1 e 2. IndoorGML definisce una particolare tipologia di arco, chiamato *interlayer connection* e rappresentato in figura con una linea tratteggiata, che mette in relazione celle di livelli differenti.

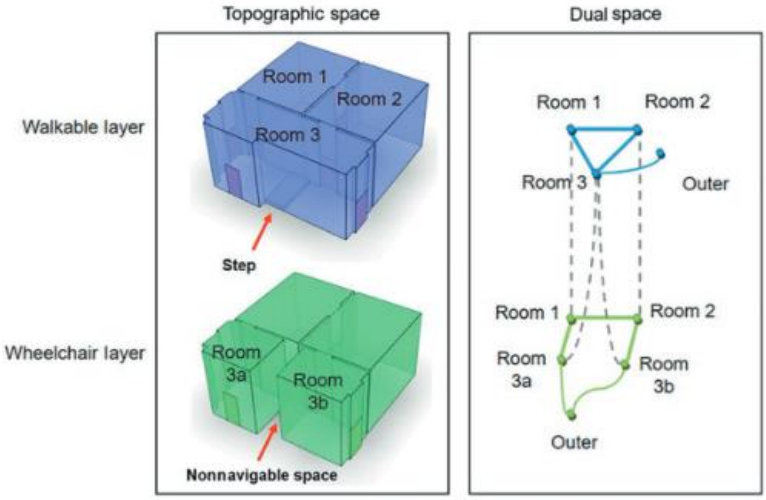

Figura 4 Un esempio di uno spazio multilivello [26]

Ancora una volta, IndoorGML presenta gli stessi problemi già discussi per gli standard precedenti. Essendo una derivazione di GML, esso eredita il focus sulle feature, rendendo, anche in questo caso, impossibile rappresentare lo spazio vuoto. Inoltre, analogamente a CityGML, il suo utilizzo è limitato solamente ad una particolare applicazione: la rappresentazione degli spazi esterni è, infatti, al di fuori dello standard. Anche per gli spazi interni, inoltre, vi è una criticità: infatti IndoorGML definisce come feature solamente oggetti strutturali degli ambienti interni (porte, muri, scale…) ignorando completamente oggetti di arredamento quali tavoli, sedie, scaffali… Oggetti, invece, essenziali per la corretta rappresentazione dell'ambiente circostante in ambito robotico.

2.4 X3D

X3D (abbreviazione di eXtensible 3D) è un formato di rappresentazione di ambienti tridimensionali generato a partire da VRML [9]. A differenza di GML e dei suoi formati derivati, X3D descrive la struttura della rappresentazione, lasciando libera scelta sul formato del file associato. Infatti, nel corso degli anni X3D ha supportato una serie di possibilità, quali XML, file binari e, ultimamente, anche JSON. X3D si pone come obiettivo quello di rappresentare uno spazio tridimensionale navigabile attraverso un browser e con cui l'utente può interagire. Sono previsti, infatti, una serie di sensori che si attivano in seguito a particolari comportamenti da parte dell'utente, allo scorrere del tempo o a seguito di altri stimoli. Oltre alla rappresentazione di uno spazio tridimensionale e alla sua navigazione, X3D permette anche la rappresentazione di altre caratteristiche relative ad un ambiente come suoni e fonti di luce. Come CityGML, anche in questo caso lo standard è diviso in diversi moduli, chiamati *profile*, che si focalizzano su una specifica funzionalità. Qualunque sia l'ambito di applicazione, il *core profile* deve comunque essere sempre presente, in quanto è quello che implementa le funzionalità base dello standard.

X3D basa la sua rappresentazione su uno *scene graph*, un grafo diretto aciclico il cui compito è quello di rappresentare lo spazio [10]. Un nodo di tale grafo rappresenta la struttura atomica per descrivere l'ambiente tridimensionale. È un concetto molto simile alla feature presente in GML. Attraverso gli archi, invece, si definiscono le relazioni che intercorrono tra essi. Tali relazioni sono descritte lungo diverse dimensioni:

- **Scene graph hierarchy**: ogni nodo può avere uno o più *field* che, a loro volta, possono contenere altri nodi. Si viene a creare un grafo di gerarchie, in cui un arco che va da A a B significa che il nodo A presenta un campo che contiene il nodo B.
- **Transformation hierarchy**: questa gerarchia definisce le relazioni spaziali che intercorrono tra diversi nodi. X3D introduce il concetto di sistema locale di coordinate, definito a partire da un altro sistema di coordinate. Viene, inoltre, definito un sottoinsieme di nodi detti *root nodes*. Un nodo radice è un nodo il cui sistema di coordinate coincide con quello del mondo rappresentato e da cui sono derivati tutti gli altri sistemi di riferimento.
- **Behaviour graph**: definisce delle connessioni (*routes*) tra i campi di un nodo e quelli di un altro. Il cambio di valore del campo del nodo padre si riflette sul nodo figlio, propagando un evento. Un evento può scaturire da un'interazione con l'utente, dallo scorrere del tempo o, ancora, X3D permette l'introduzione di script esterni che possono a loro volta generare eventi.

Come vedremo più approfonditamente nella Sezione 4.1, la transformation hierarchy è molto simile alla struttura da noi proposta per la mappa globale. Inoltre X3D supporta una serie di rappresentazioni geometriche per noi importanti come 3D voxels grid, points cloud e polygonmesh.

Tuttavia, anche X3D non è la soluzione ideale per la rappresentazione di mappe per robot autonomi. Infatti, X3D, come già GML, non prevede né la rappresentazione dell'incertezza di una misurazione né la rappresentazione dello spazio vuoto. Inoltre la definizione dello scene graph come grafo aciclico può risultare limitante per la nostra applicazione: desideriamo, invece, definire un formato che possa supportare un grafo ciclico, se necessario. Ad esempio, nel caso in cui ogni nodo del grafo rappresentasse una porzione di un anello circolare, vorremmo poter descrivere la circolarità attraverso un arco che collega l'ultimo nodo al primo, chiudendo l'anello. Infine adottare X3D come standard di rappresentazione significherebbe adottare anche una serie di opzioni per noi poco utili: il core profile, infatti, introduce anche il supporto ai sensori che generano eventi, mentre l'*interchange profile* introduce sia il supporto alle geometrie necessarie, ma anche il supporto alla gestione e il rendering di fonti di luci e ombre. Per questo, quindi, nonostante non sia impossibile definire un nuovo profile che risponda alle nostre esigenze (principalmente incertezza e graficiclici), si è preferito sviluppare un nuovo formato, in modo da essere liberi da quelle funzionalità extra per noi poco utili.

3 Impostazione del problema di ricerca

Nel capitolo precedente abbiamo effettuato una prima panoramica sul problema in esame, introducendo una prima distinzione tra mappe metriche e topologiche, lo standard definito precedentemente per il mondo bidimensionale con la sua distinzione tra mappa globale e mappa locale e mostrando come gli standard ad oggi esistenti siano poco efficaci nel risolvere il problema in esame. In questo capitolo, invece, approfondiremo ulteriormente alcune scelte implementative compiute, analizzando più nel dettaglio il loro motivo e come queste hanno influenzato il prodotto finale.

Una prima, importante, decisione è stata quella di rendere lo standard proposto per il mondo tridimensionale completamente indipendente da quello del mondo bidimensionale. Pur mantenendo la stessa filosofia di fondo (ad esempio la suddivisione in mappe globali e locali o le tipologie di mappe locali rappresentate) il lavoro da noi svolto non è dipeso da alcuna scelta fatta precedentemente. In questo modo abbiamo avuto piena libertà di ponderare le scelte migliori per questo nuovo possibile standard senza essere soggetti ad alcun vincolo di retrocompatibilità. Il prezzo da pagare per tale scelta, ovviamente, è che i due lavori producono effettivamente due standard differenti: in questo modo un eventuale robot che lavori sia con mappe bidimensionali che con mappe tridimensionali è costretto a conoscere entrambi gli standard ed effettuare, nel caso, le relative conversioni. Ovviamente nulla vieta di utilizzare lo standard da noi definito per rappresentare anche mappe bidimensionali, avendo comunque l'accortezza di aggiungere le informazioni extra che richiede, come, ad esempio, la terza coordinata di tutti i punti che la mappa contiene.

Questo distacco tra i due standard è subito comprensibile osservando il linguaggio utilizzato: come già detto precedentemente, mentre lo standard IEEE Std 1873™-2015 utilizza XML per rappresentare le mappe in questione, per questo lavoro di tesi si è scelto di utilizzare JSON. Tale scelta è stata fatta per motivi prestazionali: infatti, pur presentando caratteristiche comuni, il diverso modo di rappresentare i documenti porta JSON ad essere più efficiente di XML in alcuni aspetti prestazionali. Un primo vantaggio è la velocità di trasmissione dei file, con differenze che possono arrivare anche ad uno o due ordini di grandezza [27]. Tale aspetto è particolarmente importante proprio per la natura stessa dello standard: come abbiamo già accennato precedentemente, infatti, tale standard non rappresenterà mappe su cui i robot effettueranno le loro operazioni, bensì mappe utilizzate per lo scambio di informazioni tra robot differenti. Un file più leggero e più semplice da trasferire rende tali comunicazioni molto più agevoli. Oltre ad una maggior velocità di trasmissione, JSON consuma anche meno risorse e, soprattutto, meno memoria. Tale risultato è spiegabile principalmente da due fattori. Il primo è la differente gestione dei tag di apertura e chiusura dei campi. In XML ogni istanza di dato deve essere racchiusa tra due tag, uno di apertura ed uno di chiusura, che definiscono cosa esso rappresenti e rendono possibile la sua individuazione all'interno del documento da parte di un parser. Ipotizziamo, ad esempio, di voler modellare una persona. Questa avrà un campo nome, rappresentato da una stringa. Una eventuale istanza dell'oggetto persona avrà, quindi, un campo del genere:

```
<person>
    <name>Francesco</name>
</person>
```

In questo modo il parser capisce che la stringa *Francesco* rappresenta il nome di questa particolare persona. Anche in JSON c'è la necessità di marcare i campi presentati nel documento, dando un significato a quel dato e definendo un sistema di delimitatori che definiscono l'inizio e la fine dell'informazione fornita. Tale funzionalità viene effettuata in maniera differente da XML, ereditando una rappresentazione simile a quella dei linguaggi di programmazione. JSON, infatti, definisce una coppia chiave/valore ed utilizza come delimitatori caratteri speciali quali i doppi apici, le parentesi quadre e graffe. Lo stesso campo nome presentato precedentemente in JSON diventa quindi:

```
{
    "name": "Francesco"
}
```

Da notare come tale meccanismo permette di evitare la ripetizione del nome del campo in questione, portando ad una diminuzione dei caratteri utilizzati. Già nel nostro semplice esempio, per riportare la sola informazione del nome, XML utilizza 22 caratteri contro i 18 di JSON. Questo risparmio può sembrare misero, ma bisogna tener presente che *name* è una parola relativamente breve, di soli quattro caratteri. Nomi di campi molto più lunghi renderebbero la differenza molto più marcata, in quanto ogni carattere aggiunto al nome di un campo viene ripetuto due volte in XML. Il secondo motivo per cui JSON produce documenti più brevi rispetto ad XML è l'introduzione del concetto di lista di oggetti (o array). XML, infatti, non supporta in maniera nativa tale costrutto, ma lo definisce solo come ripetizione di singoli elementi. Riprendendo l'esempio della persona fatto precedentemente, ipotizziamo che questa possa avere più di un indirizzo. È necessario, quindi, definire una lista di indirizzi (arbitrariamente lunga) dove ogni elemento è il singolo indirizzo. Un'istanza del genere in XML viene rappresentata come:

```
<person>
    <list_of_addresses>
        <address>address_1</address>
        <address>address_2</address>
        <address>address_3</address>
    </list_of_addresses>
</person>
```

Possiamo notare come, oltre al tag di apertura e chiusura della lista di indirizzi, ogni singolo elemento è esso stesso racchiuso tra una nuova coppia di tag. La stessa struttura in JSON invece diventa:

```
{
    "list_of_addresses": ["address_1", "address_2", "address_3"]
}
```

Come anticipato, JSON introduce il supporto nativo alle liste di oggetti: una lista è delimitata da una coppia di parentesi quadre ed ogni oggetto è separato dal successivo da una virgola. Se il risparmio di caratteri nella diversa definizione di delimitatori poteva sembrare trascurabile, nel caso delle liste invece è lampante: 119 caratteri utilizzati da XML contro i 55 caratteri di JSON, meno della metà. E, ancora una volta, tale risparmio diventa sempre più notevole con l'aumento della complessità del dato rappresentato: come vedremo, per alcune tipologie di mappe avremo una lista di punti geometrici, la cui dimensione può raggiungere anche le diverse migliaia di elementi.

Individuato in JSON il metalinguaggio più adeguato alle nostre esigenze, è sorto il problema di come assicurarci che i documenti prodotti (ovvero le mappe) rispettino una serie di limitazioni da noi desiderate. Tali limitazioni spaziano da esigenze strutturali (ad esempio potremmo volere che ogni mappa abbia un nome, una data di creazione …) ad esigenze di consistenza (ad esempio se volessimo rappresentare un triangolo vorremmo avere esattamente tre vertici). Questa necessità è soddisfatta dallo schema, ovvero un documento, scritto anch'esso in JSON, che descrive la struttura voluta per l'istanza di documento. Ognuna di queste deve poi essere validata rispetto allo schema scritto. Nel nostro caso specifico, quindi, andremo a produrre uno schema per ogni tipologia di mappa, rispetto a cui la rispettiva istanza deve essere validata. Sono vari i vincoli che JSON supporta. Alcuni esempi sono: la possibilità o meno che alcuni dei campi elencati nello schema possano essere tralasciati, la possibilità o meno di avere campi extra non definiti nello schema, minimo e massimo numero di elementi di una lista… JSON supporta anche i tipi di dato elementari quali numeri interi o decimali, stringhe, caratteri, booleani e anche su questi vi è la possibilità di validarli (definendo, ad esempio, minimo e massimo per un numero o una espressione regolare per una stringa). Sebbene JSON presenti una vasta gamma di possibilità a cui attingere, è in questo aspetto che vi sono state le prime difficoltà. Infatti, dopo aver definito le limitazioni basilari (campi richiesti e limitazioni sui dati elementari) abbiamo iniziato a chiederci quali altri vincoli potessimo imporre. In particolare i vincoli di consistenza si sono rivelati particolarmente ostici, se non impossibili da definire: ad esempio imporre che due liste siano della stessa lunghezza o che il dato presente in uno specifico campo sia consistente con quello presente in un altro campo sono due esigenze che JSON non supporta. Esso, infatti, presenta un sistema di puntatori interni al documento particolarmente elementare: è possibile definire nello schema delle strutture dati da riutilizzare all'interno del documento e, per estensione, definire delle strutture ricorsive, ma non è possibile avere dei riferimenti all'istanza del documento, invece che al suo schema. Ad esempio, una necessità nata durante lo sviluppo era la definizione di due liste, entrambe di lunghezza variabile ed ignota a priori ma che doveva essere uguale tra di loro. Tale vincolo, infatti, non poteva essere rappresentato in JSON a causa della mancanza di puntatori verso l'istanza di documento finale. Approfondendo lo studio del formato, siamo venuti a conoscenza di *Ajv* (acronimo di Another JSON Validator) [28]. Ajv è un validatore di documenti JSON scritto in JavaScript e che, oltre alle funzionalità base di JSON, ne aggiunge di nuove. In particolare, introduce la funzionalità *$data*, capace di utilizzare un puntatore non per navigare lo schema, ma la stessa istanza [29]. Un esempio, tratto dalla guida fornita dagli sviluppatori, può essere un oggetto composto da due campi numerici: *smaller* e *larger*. Entrambi non sono vincolati singolarmente ma, intuitivamente, il campo smaller deve essere più piccolo del campo larger. Tale vincolo in JSON base non è definibile ma diventa possibile con Ajv:

```
var ajv = new Ajv({$data: true});

var schema = {
    "properties": {
        "smaller": {
            "type": "number",
            "maximum": { "$data": "1/larger" }
        },
        "larger": { "type": "number" }
    }
};
```

La prima riga segnala ad Ajv che utilizzeremo la notazione $data. In seguito vediamo come il massimo del valore smaller è pari a *1/larger*, ovvero, secondo la notazione dei puntatori definita da JSON, al campo larger del documento. Il campo larger non è vincolato, dato che basta che uno dei due campi rispetti il vincolo affinché entrambi lo rispettino. Questa nuova notazione rende possibile definire vincoli molto più complessi e raffinati del JSON base. Tuttavia, dopo un'analisi approfondita, abbiamo deciso di non sfruttarla. L'utilizzo di una notazione del genere, infatti, legherebbe la validazione delle mappe prodotte ad un particolare linguaggio di programmazione (JavaScript) e ad un particolare validatore (Ajv, appunto). Tale decisione è in contrasto con la definizione stessa di standard, che deve essere il più generica possibile e priva di dipendenze tecnologiche. Abbiamo scelto, quindi, di non rappresentare esplicitamente negli schemi tutti quei vincoli impossibili da definire senza la notazione $data, limitandoci ad esprimerli a parole nella documentazione. Sarà poi compito dello sviluppatore popolare adeguatamente le proprie istanze JSON senza violare questi limiti "impliciti". Ci rendiamo conto che è un modello più debole, ma ne guadagna in genericità, rendendolo compatibile con tutte quelle applicazioni che supportano JSON nativo.

Come abbiamo anticipato nel Capitolo 2, a livello strutturale abbiamo deciso di suddividere la rappresentazione di un ambiente complesso in più file: una mappa globale che presenta una visione d'insieme dello spazio rappresentato e una o più mappe locali che descrivono più nel dettaglio una particolare sezione. Come già anticipato precedentemente questo approccio rende la soluzione modulare, rendendo più semplice sia accedere ad una singola porzione che corrisponde ad una mappa locale sia crearne di nuove. In entrambi i casi, infatti, è necessario manipolare solo il relativo file e, nel caso di una creazione, il file di mappa globale. Non si costringe a caricare in memoria tutte le informazioni, che possono pesare anche diversi gigabyte di dati. L'unico svantaggio di tale soluzione è un impoverimento delle informazioni presenti all'interno della mappa globale. Come approfondiremo maggiormente nel capitolo dedicato, una mappa globale è una raccolta di mappe locali in relazione tra di loro attraverso rototraslazioni. Anche in questo caso si ripropone l'esigenza di avere un sistema di puntatori più complesso, questa volta addirittura tra file differenti. Infatti una mappa globale dovrà avere al suo interno i riferimenti alle mappe locali che la compongono. Una soluzione perfetta sarebbe costituita dalla possibilità di avere un sistema tale per cui la validazione della singola mappa globale porti automaticamente alla validazione delle mappe locali. Tuttavia, ciò non è possibile e la soluzione scelta, più semplice ma più efficace, è quella di riportare all'interno della mappa globale un identificativo delle mappe locali che la compongono, ovvero il nome del relativo file. Sarà poi, ancora una volta, compito dello sviluppatore controllare in primo luogo l'esistenza dei singoli file e, successivamente, validarli con il corretto schema. È possibile, ad esempio, scrivere uno script che in maniera automatica estragga la lista dei nomi dei file dal relativo campo della mappa globale e poi vada ad esaminare e validare file per file, ma questo è al di fuori del lavoro di definizione di standard da noi svolto.

4 Architettura del sistema

Abbiamo riportato quali sono state le scelte di design più significative, in questo capitolo procederemo con la descrizione del formato dei dati prodotto, analizzando gli schemi JSON definiti per tutti i tipi di mappa. Per ognuno di questi elencheremo nel dettaglio quali sono i campi riportati, che tipologia di dato è presente e cosa significhi. L'intenzione è avere un capitolo più tecnico, quasi una guida all'uso dello standard da noi proposto. Per facilità di lettura divideremo lo schema JSON in frammenti che commenteremo di volta in volta. Lo schema completo è, invece, riportato nell'Appendice A.

4.1 Mappa globale

Come anticipato precedentemente una mappa globale è una sorta di "contenitore" di mappe. Questa è rappresentabile come un grafo i cui nodi sono le singole mappe locali e i cui archi sono le rototraslazioni necessarie per passare dal sistema di riferimento di una mappa locale a quello di un'altra. La mappa globale rappresenta l'intero ambiente rappresentato. Ad esempio, possiamo supporre di avere la mappa globale di un condominio. Per semplicità di rappresentazione ipotizziamo che ogni mappa locale rappresenti, invece, un appartamento. L'arco che collega un appartamento a quello subito sopra rappresenterà una traslazione del sistema di riferimento lungo l'asse verticale, mantenendo invariati sia i restanti due assi sia l'orientamento. La struttura che abbiamo sviluppato è quindi vuota dei dati effettivi, presenti nei file delle singole mappe locali, ma presenta solamente dei riferimenti alle mappe locali che la compongono. Inoltre una mappa globale può essere facilmente utilizzata per la costruzione di una mappa topologica, condividendo con essa la struttura a grafo.

```
{
    "$id": "https://github.com/effelam/JSON_3dMaps/blob/master/globalmap.json",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "globalMap",
    "description": "Implementation of the global map. A global map is a graph
                    whose vertices are local maps and whose (directed) arcs
                    represent the transformation between the coordinate systems of
                    the two connected local maps.",
    "type": "object",
```

La prima parte dello schema di ogni mappa, globale o locale che sia, ha una struttura identica. Abbiamo il riferimento a dove lo schema è salvato e un riferimento al concetto di JSON schema, in modo tale da evidenziare che questo non è un documento JSON ma, appunto, uno schema. Segue, poi, il nome della tipologia di mappa (in questo caso *globalMap*) e una breve descrizione di cosa rappresenti. Infine vi è il tipo di dato che stiamo rappresentando. Dato che una mappa non è un dato elementare ma una raccolta di dati, in JSON viene rappresentato tramite un oggetto.

```
    "properties":{
        "globalmap_id": {"type": "string"},
        "list_of_vertices": {
            "description": "List of all the vertices. Each vertex is a local map,
                            represented by a triple: its id, the type of local
                            map, and the path where the map is stored.",
```

```
            "type": "array",
            "items": {"$ref": "#/definitions/vertex"}
        },
        "list_of_arcs": {
            "description": "List of all the arcs. Each arc represents a
                            transformation between two local maps.",
            "type": "array",
            "items": {"$ref": "#/definitions/transformation"}
        }
    },
```

Nel JSON schema per un oggetto viene definita una lista di proprietà che rappresentano i campi che ci aspettiamo essere presenti nel documento JSON. Nel caso di una mappa globale i campi da noi definiti sono tre:

- **Globalmap_id**: un identificativo univoco della mappa globale. È definito da una stringa e, per adesso, non abbiamo imposto alcuna limitazione sulla sua struttura.
- **List_of_vertices**: la lista dei vertici del grafo che rappresenta la mappa globale. È un array i cui elementi sono del tipo vertex, da noi definito. In questa sezione possiamo vedere in uso i puntatori JSON: *$ref* indica che la definizione di tale campo non è presente qui ma in un'altra sezione dello stesso file, il simbolo # rappresenta la radice del documento, seguito, poi dal path per raggiungere l'informazione voluta.
- **List_of_arcs**: la lista degli archi del grafo che rappresenta la mappa globale. Anche questo è un array i cui elementi sono di un tipo da noi definito, in questo caso il tipo trasformazione.

```
    "definitions": {
        "vertex" :{
            "type": "object",
            "properties": {
                "vertex_id": {"type": "string"},
                "vertex_type": {
                    "type": "string",
                    "enum": ["densegrid", "octree", "pointcloud", "polygonmesh",
                             "sparsegrid", "other"]
                },
                "vertex_path": {"type": "string"}
            },
            "required": ["vertex_id", "vertex_type"]
        },
```

La sezione sopra riportata rappresenta l'inizio della porzione JSON relativa alle definizioni di oggetti complessi, utilizzabili in altri campi del file. Come anticipato poco prima, per la mappa globale noi abbiamo due definizioni: vertex e transformation. Un vertex è il vertice del grafo che costituisce la mappa globale e rappresenta una specifica mappa locale. Questo è definito da tre caratteristiche:

- **Vertex_id**: id della mappa locale. Si suggerisce di riportare in questo campo l'id presente nel campo localmap_id, descritto successivamente nella Sezione 4.2.
- **Vertex_type**: la tipologia di mappa locale del vertice in esame. È una stringa che può avere un valore tra quelli supportati dallo standard. A questi si affianca anche il valore *other*, utile

sia per garantire maggiore flessibilità allo sviluppatore, sia per rendere possibile rappresentare le mappe topologiche. Infatti una mappa globale può rappresentare, tramite alcune accortezze, una mappa topologica. Un vertice di tale mappa può non essere associato a nessuna mappa locale e, per questo motivo, avrà come tipologia other. Sebbene il valore di questo campo possa essere ottenuto facilmente andando ad analizzare il contenuto del file associato alla mappa locale in questione, si è preferito riportarlo anche in questa sezione. In questo modo è possibile scrivere un parser che per ogni vertice che costituisce la mappa globale, ricavi il file della mappa locale e lo validi con il JSON schema appropriato.

- **Vertex_path**: percorso completo dove è salvato il file associato al vertice in esame. È l'istanza di mappa locale da validare tramite il JSON schema appropriato. Tale campo è l'unico non necessario proprio per rendere possibile la rappresentazione di mappe topologiche: i nodi di tali mappe, infatti, possono non avere nessuna mappa locale associata, rendendo inutile questo campo.

```
        "transformation": {
            "type": "object",

            "properties": {
                "transformation_id": {"type": "string"},
                "arc_head": {"type": "string"},
                "arc_tail": {"type": "string"},
                "rotation": {
                    "type": "array",
                    "minItems": 4,
                    "maxItems": 4,
                    "items": {"type": "number"}
                },
                "translation": {
                    "type": "array",
                    "minItems": 3,
                    "maxItems": 3,
                    "items": {"type": "number"}
                },
                "uncertainty": {
                    "description": "7x7 uncertainty matrix",
                    "type": "array",
                    "minItems": 28,
                a     "maxItems": 28,
                    "items": {"type": "number"}
                }
            },
            "required": ["transformation_id", "arc_head", "arc_tail"]
        }
    },
```

La sezione sopra riportata definisce una trasformazione, Ovvero una rototraslazione tra il sistema di riferimento di una mappa locale e quello di un'altra a cui viene associata anche l'incertezza di tale trasformazione. Come proprietà questa presenta:

- **Transformation_id**: un identificativo univoco della trasformazione.
- **Arc_head**: la testa dell'arco. Indica il nome della mappa di arrivo risultato dell'applicazione della trasformazione.
- **Arch_tail**: la coda dell'arco. Indica il nome della mappa di partenza a cui viene applicata la trasformazione.
- **Rotation**: la rotazione degli assi di riferimento generata dalla trasformazione. È un array di quattro numeri decimali che rappresentano il quaternione associato alla rotazione. La rappresentazione tramite quaternione è particolarmente efficace in quanto non presenta informazioni ridondanti, a differenza di quanto succede per una matrice di rotazione. Un quaternione è un vettore in uno spazio quadridimensionale del tipo $q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$. Identifichiamo con q_0 la parte reale (a) e con **q** la parte immaginaria (b**i** + c**j** +d**k**). Dato un quaternione unitario, abbiamo che: $\|q\|^2 = \|q_0\|^2 + \|\mathbf{q}\|^2 = 1$. Possiamo, quindi, associare un angolo θ tale per cui: $\|q_0\|^2 = cos^2(\theta)$ e $\|\mathbf{q}\|^2 = sin^2(\theta)$. Dato, quindi, un vettore e un quaternione unitario è possibile dimostrare che il vettore **v'** così definito: $\boldsymbol{v}' = q\boldsymbol{v}q*$ (con q* il complesso coniugato di q) è il vettore risultante da una rotazione di angolo θ applicata al vettore **v** [30]**.**
- **Traslation**: la traslazione degli assi di riferimento generata dalla trasformazione. È un array di tre numeri decimali che definiscono le coordinate dell'origine del nuovo sistema di riferimento rispetto al precedente.
- **Uncertainty**: il campo di incertezza della trasformazione. È una matrice 7x7 che rappresenta la covarianza tra tutte le possibili coppie delle sette variabili che definiscono la traslazione (quattro variabili per la rotazione e tre per la traslazione). Essendo una matrice simmetrica sono sufficienti 28 valori (la metà superiore della matrice) riportati per riga.

Da notare, infine, che per definire un arco sono necessari solamente i primi tre campi. In questo modo possiamo rappresentare una trasformazione nulla, dove non è definita né la traslazione né la rotazione. Questo è utile perché, così facendo, la mappa globale degenera in una mappa topologica, supportando anche tale tipologia.

```
    "additionalProperties": false,
    "required": ["globalmap_id", "list_of_vertices", "list_of_arcs"]
}
```

Infine, affermiamo che per la creazione di una mappa globale tutti e tre i campi prima definiti, l'id, la lista dei vertici e quella degli archi, sono necessari e non accettiamo ulteriori proprietà.

4.2 Mappa locale

Una mappa locale rappresenta una porzione di una mappa globale. In realtà noi non avremo mai un'istanza di mappa locale, che è intesa come una classe astratta, bensì una mappa di una delle tipologie riportate in seguito. Benché ogni mappa locale presenta delle caratteristiche e una struttura diverse, vi sono alcuni campi in comune che non avrebbe senso commentare più volte in ogni sezione. Quindi in questa sezione presenteremo la struttura in comune tra tutte le mappe locali, mentre nelle sezioni dedicate mostreremo solamente la parte specifica di quella tipologia di mappa.

```
{
    "$id": "LOCAL MAP ID",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "LOCAL MAP TITLE",
    "description": "LOCAL MAP DESCRIPTION",
    "type": "object",

    "properties":{
        "localmap_id": {"type": "string"},
        "time": {"description": "Time of creation of the map.", "type": "string",
                 "format": "date-time"},
        "map_description": {"type": "string"},
        "coordinate_system": {"type": "string",
                              "enum": ["absolute", "relative"]},
        "absolute_pose": {"description": "Geo-referenced location of the local
                                          map. Required if coordinate_system is
                                          absolute, not required if
                                          coordinate_system is relative."},
```

Il JSON schema di una mappa locale inizia in maniera molto simile alla mappa globale. Abbiamo i due campi iniziali di id (con un link a dove è salvato lo schema che stiamo consultando) e il campo $schema che definisce il file in questione come uno schema JSON. Abbiamo, poi il titolo (che rappresenta la tipologia di mappa locale in questione), una descrizione che spiega cosa sia questa particolare tipologia di mappa e, dato che anche questo è un dato strutturato, la tipologia che è oggetto. Abbiamo, in seguito, delle proprietà generiche:

- **Localmap_id**: un identificativo univoco dell'istanza della mappa in questione.
- **Time**: la data di creazione dell'istanza.
- **Map_description**: una breve descrizione testuale della mappa.
- **Coordinate_system**: questo campo può avere solamente due valori: absolute e relative. Absolute significa che il sistema di riferimento di questa mappa locale è geolocalizzato in relazione a latitudine, longitudine e altitudine. Tali informazioni saranno presenti nel campo absolute_pose. Relative significa che il sistema di riferimento di questa mappa, invece, è definito solamente in relazione a quello di un'altra mappa locale, tramite il meccanismo di trasformazione a livello di mappa globale. Il campo absolute_pose, in questo caso, non è significativo. Da notare come, nonostante il sistema di riferimento sia geolocalizzato nel caso in cui coordinate_system sia absolute, rimane sempre possibile definire delle trasformazioni a livello di mappa globale. Si lascia allo sviluppatore il compito di controllare che tali trasformazioni siano consistenti tra di loro.
- **Absolute_pose**: come già detto questo campo è significativo solamente nel caso in cui il valore del campo precedente sia absolute. In seguito mostreremo come ci assicuriamo di ciò e come popoliamo questo campo.

```
        "list_of_characteristics": {
            "description": "List of the voxels' characteristics. Each voxel
                            should present a value for each one of these.",
```

```
            "type":"array",
            "items": {
                "type": "object",
                "properties": {
                    "C_name": {"type": "string"},
                    "C_description": {"type": "string"},
                    "C_values": {"type": "string"}
                },
                "required": ["C_name"]
            },
            "minItems": 1
        }
            [PROPERTIES SPECIFIC OF THE LOCAL MAP]
        },
```

Ogni mappa locale è costruita su un particolare elemento atomico, che varia a seconda della tipologia di mappa. Esempi possono essere il punto, il voxel, il poligono… Come vedremo nell'istanza di una mappa locale prevederemo l'esistenza di una lista di elementi del genere, ad ognuno dei quali saranno associate delle caratteristiche come, ad esempio, il suo colore, il fatto che sia occupato o meno, un'etichetta… Nel campo *list_of_characteristics* elenchiamo quali caratteristiche ci aspettiamo che ogni elemento atomico della mappa corrente abbia e imponiamo che ne sia riportata almeno una. Questa imposizione verrà meno per la mappa Polygonmesh. Una caratteristica la definiamo riportando il suo nome, una sua descrizione e i valori che questa può assumere (elencandoli esplicitamente o descrivendoli in maniera informale). Di questi tre campi solamente il nome è obbligatorio. In seguito a questo campo sono riportate le caratteristiche specifiche di ciascuna mappa locale, che descriveremo nelle apposite sezioni.

```
    "if": {
        "properties": {
            "coordinate_system": {"const": "absolute"}
        }
    },
    "then": {
        "properties": {
            "absolute_pose": {
                "type": "object",
                "properties": {
                    "translation": {
                        "type": "array",
                        "items": [
                            {
                                "type": "number",
                                "minimum": -90,
                                "maximum": 90
                            },
                            {
                                "type": "number",
                                "minimum": -180,
```

```
                                "maximum": 180
                            },
                            {
                                "type": "number"
                            }
                        ]
                    },
                    "orientation": {
                        "type": "array",
                        "items": [
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            },
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            },
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            }
                        ]

                    }
                },
                "required": ["translation", "orientation"]
            }
        },
        "required": ["absolute_pose"]
    },
    "else": {
        "properties": {
            "absolute_pose": false
        }
    },
```

Nella sezione sopra riportata vi è il meccanismo, anticipato precedentemente, che collega il campo *absolute_pose* al valore presente nel campo *coordinate_system*. Infatti, se tale valore è pari ad *absolute*, allora il campo *absolute_pose* deve essere costituito da due array di tre elementi ciascuno, entrambi obbligatori. Il primo rappresenta la traslazione (ovvero le coordinate geografiche del sistema di riferimento, quindi latitudine, longitudine e altitudine) mentre il secondo rappresenta l'orientamento del sistema di riferimento, tramite tre angoli che esprimono la rotazione attorno all'asse verticale. Il ramo *else*, invece, impone il campo *absolute_pose* falso, ovvero tale campo deve mancare nell'istanza. In questo modo quando il campo *coordinate_system* non è pari ad *absolute* (e

dato che tale campo può avere solo due valori sarà pari a *relative*) il campo *absolute_pose* non deve essere definito.

```
    "additionalProperties": false,
    "required": ["localmap_id", "coordinate_system", "resolution", "size",
                 "list_of_voxels", "list_of_characteristics"]
}
```

Infine, anche per una mappa locale non accettiamo altre proprietà oltre a quelle da noi elencate e definiamo quali siano quelle obbligatorie. Da sottolineare che questa è una lista parziale, che verrà arricchita con le proprietà obbligatorie caratteristiche della specifica mappa locale.

4.2.1 Densegrid

Una *Densegrid* è una discretizzazione regolare dello spazio tridimensionale. L'unità atomica di tale mappa è il voxel, che altro non è che l'estensione tridimensionale del pixel. Esso rappresenta un parallelepipedo di spazio a cui assoceremo le varie caratteristiche. Gli assi di riferimento, quindi, presenteranno solamente valori interi e ogni tripletta [x,y,z] rappresenterà in maniera univoca un singolo voxel. In Figura 5 una rappresentazione di una Densegrid: lo spazio è suddiviso in cubi, indicizzati tramite una tripletta di numeri interi.

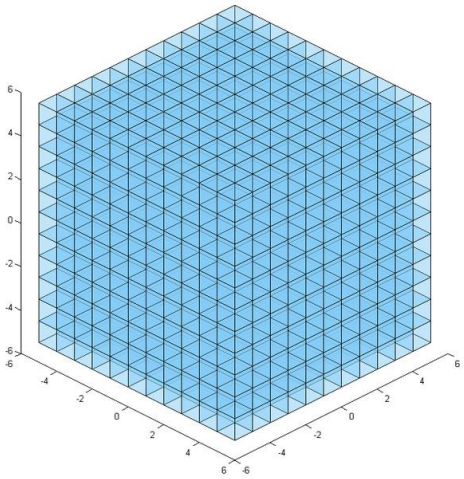

Figura 5 Rappresentazione di una Densegrid [31]

Tramite una rappresentazione del genere diventa molto più semplice gestire le informazioni spaziali: un voxel, ad esempio, può essere occupato o meno, e un oggetto reale (ad esempio un'automobile o un tavolo) è rappresentato (approssimativamente) da un gruppo di voxel. Di contro, come tutte le discretizzazioni, viene introdotto un certo grado di *aliasing*.

```
{
    "$id": "https://github.com/effelam/JSON_3dMaps/blob/master/local_maps_JSON/de
nsegrid.json",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "densegrid",
    "description": "Implementation of the local map type densegrid. A densegrid
```

```
                is a parallelepiped divided in equal voxels, which are
                described by the proper field.",
    "type": "object",

    "properties":{
        "localmap_id": {"type": "string"},
        "time": {"description": "Time of creation of the map.", "type": "string",
                 "format": "date-time"},
        "map_description": {"type": "string"},
        "coordinate_system": {"type": "string",
                              "enum": ["absolute", "relative"]},
        "absolute_pose": {"description": "Geo-referenced location of the local
                                          map. Required if coordinate_system is
                                          absolute, not required if
                                          coordinate_system is relative."},
```

Quelli riportati qui sopra sono i campi comuni per ogni tipologia di mappa locale, commentati precedentemente.

```
        "resolution": {
            "description": "Resolution (length of a single voxel's edges) along
                            the x, y, and z axes, expressed in meters.",
            "type": "array",
            "minItems": 3,
            "maxItems": 3,
            "items": {"type": "number", "exclusiveMinimum": 0}
        },

        "size": {
            "description": "Size of the dense grid, expressed as the number of
                            voxels along the x, y, and z axes.",
            "type": "array",
            "minItems": 3,
            "maxItems": 3,
            "items": {"type": "integer", "exclusiveMinimum": 0}
        },
```

I due campi sopra riportati sono i primi due esclusivi delle Densegrid, entrambi triplette di numeri. Il primo rappresenta la risoluzione della mappa, rappresentata come la lunghezza (in metri) di ogni lato del voxel. Per evitare degenerazioni, tali valori devono essere strettamente maggiori di zero. La seconda tripletta, invece, rappresenta il numero di voxel lungo i tre assi x, y e z. In questo caso, ovviamente, tali numeri devono essere interi e, anche in questo caso, strettamente maggiori di zero.

```
        "list_of_characteristics": {
            "description": "List of the voxels' characteristics. Each voxel
                            should present a value for each one of these.",
            "type":"array",
            "items": {
                "type": "object",
                "properties": {
                    "C_name": {"type": "string"},
```

```
                    "C_description": {"type": "string"},
                    "C_values": {"type": "string"}
                },
                "required": ["C_name"]
            },
            "minItems": 1
        },

        "list_of_voxels": {
           "description": "List of all the voxels. The first one is the voxel
                           with local coordinate (0,0,0), the second (1,0,0), ...
                           (size[0]-1,0,0), (0,1,0), ..., (size[0]-1,size[1]-1,0),
                           (0,0,1), ..., (size[0]-1,size[1]-1,size[2]-1).
                           Each voxel is a list of characteristics, described in
                           the field characteristics.",
            "type": "array",
            "items": {"$ref": "#/definitions/densegrid_voxel"}
        }
    },
```

Seguono la lista delle caratteristiche (già discussa precedentemente) e, infine, la lista dei voxel. Tale campo è definito come un array di elementi complessi di tipo voxel (definiti successivamente). In una Densegrid elenchiamo tutti i voxel che la compongono (come vedremo, diversamente da quanto faremo con la Sparsegrid). Per questo motivo non le coordinate di ciascun voxel sono informazioni ridondanti, purché si rispetti uno specifico ordinamento. Per convenzione il primo voxel in elenco avrà coordinate [0,0,0] e in seguito verranno elencati tutti i voxel aumentando prima la coordinata x, fino al voxel in coordinata [x_{max}, 0, 0], dopo la coordinata y, raggiungendo il voxel [x_{max}, y_{max},0] e infine la coordinata z fino al voxel [x_{max}, y_{max}, z_{max},].

```
    "definitions": {
        "densegrid_voxel": {
            "description": "A densegrid_voxel is an array of values, one for each
                            characteristic reported in the list_of_characteristics
                            field.",
            "type": "array",
            "minItems": 1
        }
    },
```

Nella sezione sopra riportata è presente la definizione di voxel, utilizzata nel campo list_of_voxel, descritto precedentemente. Questo è il nostro primo esempio di elemento atomico, concetto introdotto nella Sezione 4.2. Come abbiamo già detto, è ad esso che assoceremo le caratteristiche elencate nel campo *list_of_characteristics*. Ad esempio possiamo riportare per ogni voxel se questo è occupato o meno (rappresentabile con un booleano), la sua temperatura (rappresentato da un numero decimale) o il suo colore (rappresentato, per esempio, da un dato strutturato composto da tre interi). Vista la grande varietà di caratteristiche possibili, un voxel è definito solamente da un array (i cui elementi sono i valori delle sue caratteristiche) su cui non imponiamo alcuna limitazione di sorta, tranne per il fatto che deve essere presente almeno un valore. L'unica altra limitazione (peraltro implicita perché non la esercitiamo con nessun controllo da parte dello schema) è che ogni

caratteristica qui presente deve essere definita nel campo list_of_characteristic precedente e viceversa.

Alla sezione delle definizioni segue il ramo if-then-else descritto precedentemente e, in conclusione:

```
    "additionalProperties": false,
    "required": ["localmap_id", "coordinate_system", "resolution", "size",
                 "list_of_voxels", "list_of_characteristics"]
}
```

Ancora una volta definiamo quali siano i campi necessari e che non accettiamo campi non presenti in questo schema.

4.2.2 Octree

Un Octree è una mappa che rappresenta lo spazio circostante tramite octree, una struttura dati ad albero estremamente efficiente per rappresentare lo spazio tridimensionale [32]. La sua costruzione avviene tramite procedimento ricorsivo. La radice di tale albero è l'intero ambiente rappresentato. Dato un nodo *n*, questo può essere un nodo foglia oppure può avere esattamente otto figli. I suoi otto figli altro non sono che la suddivisione dello spazio rappresentato da *n* in otto sotto-spazi uguali fra loro. In Figura 6 una rappresentazione grafica del processo di esplorazione dello spazio: il cubo che rappresenta il nodo padre (a sinistra) viene suddiviso in 8 cubi che rappresentano i nodi figli (a destra).

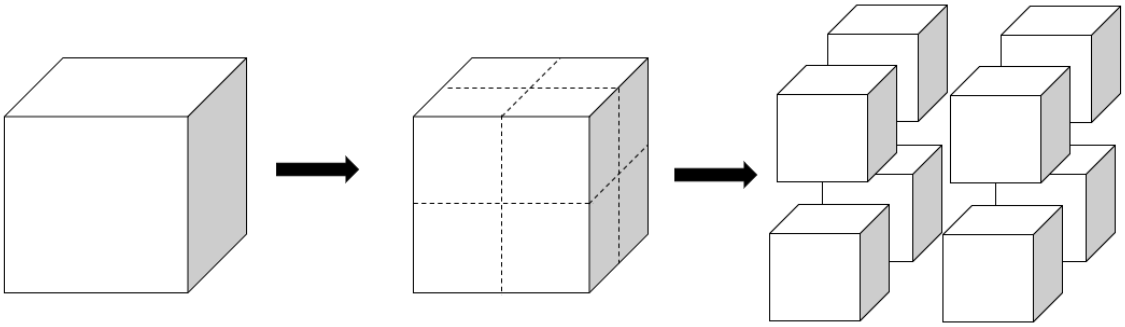

Figura 6 A sinistra il nodo padre e a destra gli 8 nodi figli

Tale procedimento di suddivisione viene applicato in maniera ricorsiva finché un set di caratteristiche definito precedentemente ha valore costante per ogni punto presente nel nodo. Il vantaggio di tale struttura rispetto alla Densegrid è un notevole risparmio di memoria: immaginiamo, per semplicità, che l'unica caratteristica coinvolta è l'essere occupato o meno. Normalmente il mondo circostante ha tanto spazio vuoto: basti pensare alla mappa di una stanza, lo spazio occupato è costituito solamente dal confine della stanza (muri, soffitto, pavimento) e da eventuali oggetti presenti. Tutto il resto è, appunto, spazio vuoto, senza informazioni di sorta. In una Densegrid dovremmo elencare uno ad uno tutti i voxel vuoti, mentre in un octree tali voxel collassano in un uno o più nodi foglia, riducendo il carico di informazioni presenti nel file.

```
{
    "$id": "https://github.com/effelam/JSON_3dMaps/blob/master/local_maps_JSON/octree.json",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "octree",
    "description": "Implementation of the local map type octree. An octree
                    represents the space with an octree, that is a tree. Each
                    node of this tree that is not a leaf has exactly 8 children.
                    Each node (including leaves) has the characteristics listed
                    in the corresponding field.",
    "type": "object",

    "properties":{
        "localmap_id": {"type": "string"},
        "time": {"description": "Time of creation of the map.", "type": "string",
                 "format": "date-time"},
        "map_description": {"type": "string"},
        "coordinate_system": {"type": "string",
                              "enum": ["absolute", "relative"]},
        "absolute_pose": {"description": "Geo-referenced location of the local
                                          map. Required if coordinate_system is
                                          absolute, not required if
                                          coordinate_system is relative."},
```

Quanto sopra riportato è stato già descritto nella Sezione 4.2.

```
        "size": {
            "description": "Size of the space, expressed in meters along the x,
                            y, and z axes.",
            "type": "array",
            "minItems": 3,
            "maxItems": 3,
            "items": {"type": "number", "exclusiveMinimum": 0}
        },
```

La prima caratteristica (sopra riportata) rappresenta la dimensione dello spazio descritto, espresso in metri, lungo i tre assi. Per evitare casi degeneri, tutti e tre i valori devono essere strettamente maggiori di zero.

```
        "max_depth": {
            "description": "Maximum number of levels of the tree. If not
                            specified the tree is fully explored.",
            "type": "integer",
            "exclusiveMinimum": 0
        },
```

Segue una seconda caratteristica, la massima profondità dell'albero, definita da un intero (la radice ha profondità 0, i figli della radice hanno profondità 1 e così via). Tale informazione acquisisce significato (ed è presente) solamente se l'albero non è esplorato completamente, bensì viene introdotta una condizione di terminazione basata sulla profondità dell'albero. In questo modo si lascia la possibilità allo sviluppatore di non avere alberi illimitatamente grandi, ma di limitarne le

dimensioni in cambio di una perdita di informazioni. Nel caso in cui lo sviluppatore sfrutti tale funzionalità, infatti, è tenuto a definire un meccanismo di risoluzione del valore delle caratteristiche da attribuire al nodo "sporco" non espanso. Riprendendo l'esempio dello spazio con l'unica caratteristica occupato/non occupato, nel caso in cui si volesse terminare prima l'esplorazione nel ramo si presenterebbe il problema di stabilire, appunto, se il nodo non espanso sia occupato o meno. Due possibilità entrambe percorribili sarebbero stabilire un valore di default (ad esempio definire che i nodi non espansi essendo parzialmente occupati vengono trattati come se lo fossero completamente), oppure lasciare la decisione alla maggioranza (se il 70% dei punti al suo interno sono non occupati, allora lo sarà anche il nodo non espanso).

```
        "tree": {
            "description": "The actual tree. The first node is the root.",
            "$ref": "#/definitions/node"
        }
    },
```

Il campo sopra riportato rappresenta dove viene effettivamente salvato l'albero che costituisce la mappa. Come vedremo è una struttura ricorsiva il cui nodo qui riportato è la radice.

```
    "definitions": {
        "node": {
            "type": "object",

            "properties": {
                "node_characteristics": {
                    "type": "array",
                    "minItems": 1
                },
                "node_children": {
                    "anyOf": [
                        {
                            "type": "array",
                            "minItems": 0,
                            "maxItems": 0
                        },
                        {
                            "type": "array",
                            "minItems": 8,
                            "maxItems": 8,
                            "items": {"$ref": "#/definitions/node"}
                        }
                    ]
                }
            },
            "required": ["node_children", "node_characteristics"]
        }
    },
```

Nella porzione di schema sopra riportato definiamo un nodo Questo ha due sole proprietà: un array di caratteristiche (che riflettono le caratteristiche definite nel campo list_of_characteristics precedentemente incontrato) e uno di figli. L'array di figli o è vuoto (questo vuol dire che il nodo è un nodo foglia) oppure ha esattamente otto figli di tipo nodo, rendendo la struttura ricorsiva. Due importanti considerazioni al riguardo:

- Qualunque nodo può avere delle caratteristiche, non solo i nodi foglia. In questo caso, quindi, non supponiamo una relazione biunivoca tra le caratteristiche assegnate ad un singolo nodo e quelle elencate nel campo list_of_characteristics, anzi ci aspettiamo che un nodo abbia un sottoinsieme delle caratteristiche definite.
- Nell'istanza di mappa di tipo octree non avremo un elenco di nodi, bensì una struttura innestata, con i dati dei nodi figli presenti all'interno del campo children del nodo padre. Tale struttura è particolarmente efficace per la navigazione dell'albero.

```
    "additionalProperties": false,
    "required": ["localmap_id", "coordinate_system", "size", "octree",
                 "list_of_characteristics"]
}
```

Concludiamo, come sempre, con l'imposizione di non avere altri campi al di fuori di quelli da noi definiti e l'elenco dei campi necessari.

4.2.3 Pointcloud

Una Pointcloud è la struttura dati più semplice definibile: altro non è che un elenco di punti di cui riportiamo le coordinate e le caratteristiche associate. Tale struttura è estremamente efficace nel descrivere ampi spazi vuoti, in cui la maggior parte dei punti non presenta caratteristiche degne di nota, a causa del fatto che riportiamo solamente i punti significativi, tipicamente rilevati sulle superfici degli ostacoli.

```
{
    "$id": "https://github.com/effelam/JSON_3dMaps/blob/master/local_maps_JSON/po
intcloud.json",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "pointcloud",
    "description": "Implementation of the local map type pointcloud. A pointcloud
                    is a list of points, each one that has a coordinate and a list
                    of characteristics.",
    "type": "object",

    "properties":{
        "localmap_id": {"type": "string"},
        "time": {"description": "Time of creation of the map.", "type": "string",
                 "format": "date-time"},
        "map_description": {"type": "string", "maxLength": 250},
        "coordinate_system": {"type": "string",
                              "enum": ["absolute", "relative"]},
        "absolute_pose": {"description": "Geo-referenced location of the Local
                                          map. Required if coordinate_system is
```

```
                                    absolute, not required if
                                    coordinate_system is relative."},

        "list_of_characteristics": {
            "description": "List of the points' characteristics. Each node should
                            present a value for each one of these.",
            "type":"array",
            "items": {
                "type": "object",
                "properties": {
                    "C_name": {"type": "string"},
                    "C_description": {"type": "string"},
                    "C_values": {"type": "string"}
                },
                "required": ["C_name"]
            },
            "minItems": 1
        },
```

La porzione sopra riportata è stata già descritta nella Sezione 4.2.

```
        "list_of_points": {
            "description": "List of all the points. Each point is a pair: the
                            first element is its coordinates and the second one is
                            a list of its characteristics as listed in the
                            characteristic field.",
            "type": "array",
            "items": {"$ref": "#/definitions/point"}
        }
    },
```

La prima caratteristica esclusiva (sopra riportata) è la lista dei punti che compongono la Sparsegrid. È un array di oggetti di tipo point, definito successivamente.

```
    "definitions": {
        "point": {
            "description": "A point is a pair. The first element represents the
                            coordinates (x,y,z) in the space and the second
                            element is an array of values, one for each
                            characteristic reported in the wanted_characteristics
                            field.",
            "type": "array",
            "items": [
                {
                    "type": "array",
                    "items": {"type": "number"},
                    "minItems": 3,
                    "maxItems": 3
                },
                {
                    "type": "array"
                }
```

```
            ]
        }
    },
```

Qui sopra riportiamo la definizione di un punto. Ogni punto è definito da una coppia di array. Il primo array rappresenta le coordinate del punto in questione, mentre il secondo sono le caratteristiche di quel punto, che, analogamente a quanto detto precedentemente per gli altri tipi di mappe locali, devono essere presenti nel campo list_of_characteristics.

```
    "additionalProperties": false,
    "required": ["localmap_id", "coordinate_system", "list_of_points",
                 "list_of_characteristics"]
}
```

Come sempre chiudono il nostro schema l'imposizione di non avere altri campi oltre a quelli descritti e la lista di campi obbligatori.

4.2.4 Polygonmesh

Una Polygonmesh (o mesh poligonale) è una rappresentazione tridimensionale basata sull'utilizzo di poligoni in cui una superficie tridimensionale complessa viene suddivisa in una collezione di vertici e figure piane. Tale rappresentazione ha guadagnato grande popolarità di recente grazie all'aumento dell'utilizzo di tecnologie di acquisizione tridimensionale, ad esempio nel campo della medicina, dell'ingegneria ma anche dell'intrattenimento (film o videogiochi). Analisi geometriche, ricostruzione di superfici a partire da punti nello spazio e operazioni di filtraggio per ovviare a misurazioni imprecise sono alcuni dei problemi che un campo relativamente nuovo deve affrontare. Il vantaggio di utilizzare Polygonmesh in questi ambiti è dato dalla grande flessibilità del formato e della sua efficienza [33].

```
{
    "$id": "https://github.com/effelam/JSON_3dMaps/blob/master/local_maps_JSON/po
lygonmesh.json",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "polygonmesh",
    "description": "Implementation of the local map type polygonmesh. In this
                    type of map the 3D world is described by a collection of 2D
                    polygons. Each polygon has a value for each characteristic
                    listed in the wanted_characteristics field.",
    "type": "object",

    "properties":{
        "localmap_id": {"type": "string"},
        "time": {"description": "Time of creation of the map.", "type": "string",
                 "format": "date-time"},
        "map_description": {"type": "string"},
        "coordinate_system": {"type": "string",
                              "enum": ["absolute", "relative"]},
        "absolute_pose": {"description": "Geo-referenced location of the local
                                          map. Required if coordinate_system is
                                          absolute, not required if
```

```
                                        coordinate_system is relative."},

        "list_of_characteristics_point": {
            "description": "List of the wanted characteristics for the points.
                            Each point should present a value for each one of
                            these.",
            "type":"array",
            "items": {
                "type": "object",
                "properties": {
                    "C_name": {"type": "string"},
                    "C_description": {"type": "string"},
                    "C_values": {"type": "string"}
                },
                "required": ["C_name"]
            }
        },
        "list_of_characteristics_polygon": {
            "description": "List of the wanted characteristics for the polygons.
                            Each polygon should present a value for each one of
                            these.",
            "type":"array",
            "items": {
                "type": "object",
                "properties": {
                    "C_name": {"type": "string"},
                    "C_description": {"type": "string"},
                    "C_values": {"type": "string"}
                },
                "required": ["C_name"]
        }

        },
```

L'inizio di un JSON schema (sopra riportato) inizia sempre con quanto descritto nella Sezione 4.2. Da notare come, a differenza delle altre tipologie di mappe locali, nel caso di una Polygonmesh vi sono due liste di caratteristiche, una esclusiva per i punti ed una esclusiva per i poligoni. In questo modo diamo la libertà allo sviluppatore di definire delle caratteristiche a livello di poligono e altre a livello di punti. Questo perché è lecito supporre che ve ne siano alcune che devono essere applicate all'intera faccia del poligono (ad esempio il colore della faccia o il grado di trasparenza), mentre altre vengono applicate solamente al vertice in questione. Notiamo, inoltre, che nel caso della Polygonmesh viene meno il vincolo che debba esistere almeno una caratteristica. Questo perché, questa particolare mappa locale già di per sé riporta delle informazioni sullo spazio rappresentato (le facce che delimitano gli oggetti).

```
    "list_of_points": {
        "description": "List of all the points.",
        "type": "array",
        "items": {"$ref": "#/definitions/point"}
```

```
    },

        "list_of_polygons": {
            "description": "List of all the polygons.",
            "type": "array",
            "items": {"$ref": "#/definitions/polygon"}
        }
    },
```

Nella sezione sopra riportata definiamo due liste, una per i punti e una per i poligoni che definiscono la Polygonmesh. Entrambe le liste traggono vantaggio dal campo definizioni riportato in seguito.

```
    "definitions": {
        "point": {
            "description": "It represents a point. Each point is represented as a
                            pair of arrays. The first one represents its
                            coordinates, as 3 real numbers. The second one is a
                            collection of the values of the wanted characteristics
                            for that point.",
            "type": "array",
            "items": [
                {
                    "type": "array",
                    "items": {"type": "number"},
                    "minItems": 3,
                    "maxItems": 3
                },
                {
                    "type": "array"
                }
            ]
        },
```

La definizione di punto in una Polygonmesh ricalca fedelmente quanto definito per la Pointcloud. Ogni punto è definito come una coppia di array: il primo, composto da tre numeri, rappresenta le sue coordinate, mentre il secondo è una lista su cui non facciamo alcuna assunzione che rappresenta i valori delle caratteristiche associate al singolo vertice in esame (prese da list_of_characteristics_point).

```
        "polygon": {
            "type": "object",
            "properties": {

                "polygon_characteristics": {
                    "description": "List of the characteristics of this polygon.
                                    These are taken from
                                    list_of_characteristcs_polygon field",
                    "type": "array"
                },

                "list_of_vertices": {
```

```
                    "description": "The list of points that defines the current
                                    polygon.",
                    "type": "array",
                    "items": {"type": "integer"},
                    "minItems": 3
                }
            },
            "required": ["list_of_points"]
        }
```

La seconda definizione in una Polygonmesh è quella di poligono, sopra riportata. Questo ha due campi: uno che rappresenta le eventuali caratteristiche definite a livello di poligono (prese dal campo list_of_characteristics_polygon) e un altro che è la lista dei suoi vertici, ovviamente quest'ultima con almeno tre elementi. Per evitare di ripetere ulteriormente le coordinate dei punti, per individuare i vertici di un poligono utilizziamo la sua posizione nel campo list_of_points precedente, ordinamento che, come solitamente avviene per gli array nei linguaggi di programmazione, parte da 0.

```
        "point": {
            "description": "It represents a point. Each point is represented as a
                            pair of arrays. The first one represents its
                            coordinates, as 3 real numbers. The second one is a
                            collection of the values of the wanted characteristics
                            for that point.",
            "type": "array",
            "items": [
                {
                    "type": "array",
                    "items": {"type": "number"},
                    "minItems": 3,
                    "maxItems": 3
                },
                {
                    "type": "array"
                }
            ]
        }
    },
```

La seconda definizione presente in una Polygonmesh è quella dell'oggetto punto, che ricalca fedelmente quanto definito per la Pointcloud. Ogni punto è definito come una coppia di array, il primo, composto da tre numeri, rappresenta le sue coordinate, mentre il secondo è una lista su cui non facciamo alcuna assunzione che rappresenta i valori delle caratteristiche associate al singolo vertice in esame (prese da list_of_characteristics_point).

```
    "additionalProperties": false,
    "required": ["localmap_id", "coordinate_system", "list_of_points",
                 "list_of_polygons", "list_of_characteristics_point",
                 "list_of_characteristics_polygon"]
}
```

Infine abbiamo, come sempre, le caratteristiche necessarie e l'imposizione di non avere campi al di fuori di quelli descritti.

4.2.5 Sparsegrid

Una Sparsegrid ha una struttura analoga a quella della Densegrid: lo spazio viene suddiviso in porzioni discrete chiamate voxel a cui associamo le varie caratteristiche. La differenza è che, mentre per le Densegrid rappresentiamo tutti i voxel esistenti, per cui non è necessario riportare le coordinate di ogni voxel ma basta definire un ordinamento, nel caso della Sparsegrid riportiamo solamente i voxel considerati significativi. Analogamente alle Pointcloud, ciò avviene per ottimizzare la memoria nel caso in cui volessimo rappresentare spazi per la maggior parte vuoti.

```
{
    "$id": "https://github.com/effelam/JSON_3dMaps/blob/master/local_maps_JSON/Sp
arsegrid.json",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "sparsegrid",
    "description": "Implementation of the local map type sparsegrid. A sparsegrid
                    is a parallelepiped divided in smaller voxels, like
                    densegrids. Unlike densegrids, in sparsegrids only voxels with
                    at least one non-null characteristic are explicitly reported
                    in the voxels field.",
    "type": "object",

    "properties": {
        "localmap_id": {"type": "string"},
        "time": {"description": "Time of creation of the map.", "type": "string",
                 "format": "date-time"},
        "map_description": {"type": "string"},
        "coordinate_system": {"type": "string",
                              "enum": ["absolute", "relative"]},
        "absolute_pose": {"description": "Geo-referenced location of the local
                                          map. Required if coordinate_system is
                                          absolute, not required if
                                          coordinate_system is relative."},
```

Già descritta nella Sezione 4.2.

```
        "resolution": {
            "description": "Resolution (length of a single voxel's edges) along
                            the x, y, and z axes, expressed in meters.",
            "type": "array",
            "minItems": 3,
            "maxItems": 3,
            "items": {
                "type": "number",
                "exclusiveMinimum": 0
            }
        },
```

```
        "size": {
            "description": "Size of the grid, expressed as the maximum index of
                            sparse voxels nx, ny, and nz along the x, y, and z
                            axes.",
            "type": "array",
            "minItems": 3,
            "maxItems": 3,
            "items": {
                "type": "integer",
                "exclusiveMinimum": 0
            }
        },
```

Essendo la Sparsegrid molto simile ad una Densegrid, hanno anche i campi in comune. Resolution e size, già descritti nella Sezione 4.2.1, rappresentano rispettivamente le dimensioni di un voxel (in metri) e la dimensione dello spazio (in voxel).

```
        "list_of_voxels": {
            "description": "List of all the voxels that have at least one
                            non-null characteristic. Each voxel is a pair: the
                            first element is its coordinates in the grid and the
                            second one is a list of its characteristics as listed
                            in the characteristic field.",
            "type": "array",
            "items": {"$ref": "#/definitions/sparsegrid_voxel"}
        }
    },
```

Infine abbiamo la lista dei voxel, banalmente un array di oggetti di tipo sparsegrid_voxel.

```
    "definitions": {
        "sparsegrid_voxel": {
            "description": "A sparsegrid_voxel is a pair. The first element
                            represents the coordinates (x,y,z) in the 3D grid and
                            the second element is an array of values, one for each
                            characteristic reported in the wanted_characteristics
                            field.",
            "type": "array",
            "items": [
                {
                    "type": "array",
                    "minItems": 3,
                    "maxItems": 3,
                    "items": {"type": "integer", "minimum": 0}
                },
                {
                    "type": "array",
                    "minItems": 1
                }
            ]
        }
```

```
    },
```

Qui definiamo l'oggetto *sparsegrid_voxel*. Questo si differenzia dall'omologo nella Densegrid perché presenta, oltre all'array delle caratteristiche, anche le coordinate del voxel in questione all'interno della griglia. In questo modo possiamo posizionare ogni voxel nello spazio dato che manca l'informazione implicita dell'ordinamento presente nella Densegrid.

```
    "additionalProperties": false,
    "required": ["localmap_id", "coordinate_system", "list_of_voxels",
                 "resolution", "size", "list_of_characteristics"]
}
```

Come tutte le mappe, anche qui l'ultima informazione definisce quali campi sono necessari e vieta la presenza di campi differenti da quelli presenti nello schema.

Con la Sparsegrid terminano le tipologie di mappe locali da noi definite. Nell'Appendice A riportiamo gli schemi nella loro interezza, mentre nell'Appendice B sono presenti delle tabelle riepilogative che presentano i nomi dei campi, una loro piccola descrizione e se sono obbligatori o meno.

5 Realizzazioni sperimentali e valutazioni

Nel capitolo precedente abbiamo analizzato ogni JSON schema da noi definito, descrivendo nel dettaglio tutti i campi che li compongono. In questo, invece, ci soffermeremo sulla produzione di istanze di mappe che rispettano gli schemi prima definiti. Così facendo ci prefissiamo un duplice obiettivo: da un lato vogliamo dimostrare la possibilità di rappresentare con il nostro standard realtà complesse e dall'altro analizzare quanto grandi sono i file che produciamo.

Per rendere il processo il più significativo possibile abbiamo deciso, dove possibile, di non generare mappe artificiali, bensì di provare a convertirne di già esistenti utilizzate in altri lavori o disponibili pubblicamente. Il vantaggio di tale approccio è che, in questo modo, si evita di generare dei dati in funzione della struttura definita, rischiando di produrre solamente istanze rappresentabili, ma si testa lo standard, provando a cercare l'esistenza di esempi che esso non possa rappresentare. In generale l'approccio è stato lo stesso per tutte le tipologie di mappe: come primo passo abbiamo cercato una mappa consultabile in formato testuale (quindi niente formati binari, più complessi da convertire e, soprattutto, da confrontare), in seguito abbiamo scritto uno script Python che convertisse il contenuto del file sorgente nel JSON desiderato. Abbiamo scelto Python come linguaggio di programmazione perché ben si presta ad operazioni così leggere e veloci: in questo modo siamo stati in grado di generare codice rapidamente, senza spendere troppo tempo nel creare la giusta struttura di classi. Inoltre il supporto nativo della struttura dati *dictionary* [34], struttura dati che ricalca fedelmente quella dei documenti JSON, rende estremamente semplice creare e modificare documenti JSON, mentre la libreria Pandas [35] velocizza e semplifica la gestione di una grande quantità di dati.

Oltre ad un confronto di dimensioni tra il file originario e il file generato da noi, presenteremo il tempo di conversione impiegato per ogni mappa. Tale dato è puramente indicativo, in quanto viene influenzato da altri fattori quali, ad esempio, ottimizzazione del codice di conversione, struttura dati del file sorgente e potenza della macchina. Tuttavia può essere interessante come ordine di grandezza, per capire se sia possibile ipotizzare una conversione online quando necessario da parte del robot o se, invece, è necessario dedicare alla conversione un tempo significativo.

Infine, data la grande dimensione della maggior parte delle mappe prodotte, non riporteremo in questo documento le istanze generate. Tuttavia, per chi fosse interessato, i file sorgenti, gli script di conversione e i file generati sono consultabili dal progetto GitHub associato a questo lavoro [36]. A titolo di esempio, invece, riporteremo esempi semplificati, in modo tale da dare un'idea di come sia un'istanza di ogni mappa.

5.1 Mappa globale

Iniziamo questa analisi con l'unica tipologia di mappa a cui non siamo stati in grado di associare esempi reali. Il concetto di mappa globale, infatti, è un concetto poco comune che noi abbiamo deciso di porre maggiormente in primo piano. Per questo motivo non siamo riusciti a trovare esempi utilizzati da altri da convertire. Non potendo fare analisi sulla bontà o velocità della conversione, l'unica cosa da riportare rimane un esempio semplificato di mappa globale:

```
{
    "globalmap_id": "Globalmap di prova",
```

```
    "list_of_vertex": [
        {
            "vertex_id": "densegrid_1",
            "vertex_type": "densegrid",
            "vertex_path": "C:\\json\localmap\densegrid_1.json"
        },
        {
            "vertex_id": "octree_1",
            "vertex_type": "octree",
            "vertex_path": "C:\\json\localmap\octree_1.json"
        },
        {
            "vertex_id": "octree_2",
            "vertex_type": "octree",
            "vertex_path": "C:\\json\localmap\octree_2.json"
        },
        {
            "vertex_id": "sparsegrid_1",
            "vertex_type": "sparsegrid",
            "vertex_path": "C:\\json\localmap\sparsegrid_1.json"
        }
    ],
    "list_of_transformations": [
        {
            "head": "densegrid_1",
            "tail": "octree_1",
            "rotation": [1,2,3,4],
            "traslation": [1,2,3],
            "uncertainty": [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                            21,22,23,24,25,26,27,28]
        },
        {
            "head": " octree_2",
            "tail": " densegrid_1",
            "rotation": [1,2,3,4],
            "traslation": [1,2,3],
            "uncertainty": [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                            21,22,23,24,25,26,27,28]
        },
        {
            "head": "sparsegrid_1",
            "tail": "octree_1",
            "rotation": [1,2,3,4],
            "traslation": [1,2,3],
            "uncertainty": [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                            21,22,23,24,25,26,27,28]
        }    ]
}
```

Possiamo notare nella sezione sopra riportata come tale mappa globale sia costituita da quattro mappe locali: una Densegrid, due Octree e una Sparsegrid. Le trasformazioni ci permettono di raggiungere a partire dalla mappa octree_1 le mappe sparsegrid_1 e densegrid_1 e da quest'ultima raggiungere octree_2. Le eventuali altre trasformazioni dei sistemi di riferimento vanno fatte combinando tali trasformazioni riportate. Infine possiamo notare come il campo uncertainty presenti 28 valori che, come spiegato nella Sezione 4.1, equivalgono a definire la matrice simmetrica 7x7:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 8 & 9 & 10 & 11 & 12 & 13 \\ 3 & 9 & 14 & 15 & 16 & 17 & 18 \\ 4 & 10 & 15 & 19 & 20 & 21 & 22 \\ 5 & 11 & 16 & 20 & 23 & 24 & 25 \\ 6 & 12 & 17 & 21 & 24 & 26 & 27 \\ 7 & 13 & 18 & 22 & 25 & 27 & 28 \end{bmatrix}$$

Un altro esempio, più realistico, è riportato nella sezione qui sotto:


```
{
    "globalmap_id": "Globalmap di prova",
    "list_of_vertex": [
        {
            "vertex_id": "bildstein_station1_1",
            "vertex_type": "pointcloud",
            "vertex_path": "C://json/localmap/bildstein_station1_1.json"
        },
        {
            "vertex_id": "bildstein_station1_2",
            "vertex_type": "pointcloud",
            "vertex_path": "C://json/localmap/bildstein_station1_2.json"
        },
        {
            "vertex_id": "bildstein_station1_3",
            "vertex_type": "pointcloud",
            "vertex_path": "C://json/localmap/bildstein_station1_3.json"
        },
        {
            "vertex_id": "bildstein_station1_4",
            "vertex_type": "pointcloud",
            "vertex_path": "C://json/localmap/bildstein_station1_4.json"
        },
        {
            "vertex_id": "bildstein_station1_5",
            "vertex_type": "pointcloud",
            "vertex_path": "C://json/localmap/bildstein_station1_5.json"
        },
        {
            "vertex_id": "bildstein_station1_6",
```

```
            "vertex_type": "pointcloud",
            "vertex_path": "C://json/localmap/bildstein_station1_6.json"
        },
        {
            "vertex_id": "bildstein_station1_7",
            "vertex_type": "pointcloud",
            "vertex_path": "C://json/localmap/bildstein_station1_7.json"
        },
        {
            "vertex_id": "bildstein_station1_8",
            "vertex_type": "pointcloud",
            "vertex_path": "C://json/localmap/bildstein_station1_8.json"
        }
    ],
    "list_of_transformations": [
        {
            "head": "bildstein_station1_2",
            "tail": "bildstein_station1_1",
            "traslation": [7.939,0,0]
        },
        {
            "head": "bildstein_station1_3",
            "tail": "bildstein_station1_1",
            "traslation": [0,-10.335,0]
        },
        {
            "head": "bildstein_station1_4",
            "tail": "bildstein_station1_1",
            "traslation": [7.9393,-10.335,0]
        },
        {
            "head": "bildstein_station1_5",
            "tail": "bildstein_station1_1",
            "traslation": [0,0,15.131]
        },
        {
            "head": "bildstein_station1_6",
            "tail": "bildstein_station1_1",
            "traslation": [7.939,0,15.131]
        },
        {
            "head": "bildstein_station1_7",
            "tail": "bildstein_station1_1",
            "traslation": [0,-10.335,15.131]
        },
        {
            "head": "bildstein_station1_8",
            "tail": "bildstein_station1_1",
            "traslation": [7.939,-10.335,15.131]
```

```
        }
    ]
}
```

In questo esempio abbiamo usato come base di partenza la Pointcloud bildestein_station1. Tale mappa è una di quelle del dataset Semantic3D che utilizzeremo come test per la mappa locale Pointcloud e verrà descritta nella Sezione 5.2.2. Per ora è sufficiente sapere che è una Pointcloud a cui ad ogni punto riportato vengono assegnate alcune caratteristiche. I valori minimi e massimi di ogni coordinata sono riportati nella seguente tabella:

Coordinata	Minimo	Massimo
X	-104,7	120,58
Y	-133,53	112,87
Z	-17,11	47,37

Considerato, quindi, il minimo volume che include tutti i punti presenti, lo dividiamo in otto sottospazi tra loro identici, secondo un meccanismo analogo a quanto visto per la creazione di un Octree nel Capitolo 4.2.2. Per ognuno di tali sottospazi definiamo un file del formato Pointcloud che rappresenta solamente quella particolare porzione di spazio. Definendo come piani di taglio i piani di coordinate x = 7,94, y = -10,33 e z = 15,13, gli otto file da noi definiti sono:

Nome file	X	Y	Z
bildstein_station1_1	< 7,94	< 10,33	< 15,13
bildstein_station1_2	>= 7,94	< 10,33	< 15,13
bildstein_station1_3	< 7,94	>= 10,33	< 15,13
bildstein_station1_4	>= 7,94	>= 10,33	< 15,13
bildstein_station1_5	< 7,94	< 10,33	>= 15,13
bildstein_station1_6	>= 7,94	< 10,33	>= 15,13
bildstein_station1_7	< 7,94	>= 10,33	>= 15,13
bildstein_station1_8	>= 7,94	>= 10,33	>= 15,13

Definiamo, quindi, 7 trasformazioni, ognuna delle quali definisce il sistema di riferimento di uno degli ultimi sette frammenti in relazione con quello del primo. Vista la natura di queste trasformazioni, vi è presente solamente il campo traslation, in quanto non applichiamo alcun tipo di rotazione e non vi sarà incertezza.

5.2 Mappa locale

5.2.1 Densegrid, Octree, Sparsegrid

Densegrid, Octree e Sparsegrid condividono la stessa tipologia di file sorgenti e per questo motivo verranno analizzati insieme. Le mappe convertite provengono da una collezione di scansioni laser di alcuni ambienti dell'Università di Freiburg: il campus di informatica, un corridoio e due aule [37]. Tali mappe sono fornite come file binario, quindi il primo passaggio è stato quello di generare un file testuale equivalente, in modo tale da avere un approccio analogo a quanto fatto per le altre tipologie di mappe.

Il file binario fornito rappresenta l'Octree corrispondente, in formato Octomap [32], gestibile tramite Matlab. Quest'ultimo, infatti, associa alla Octomap una Densegrid equivalente, con un grado di risoluzione stabilito dal file stesso. In questo modo è possibile indagare in maniera sistematica le caratteristiche dello spazio rappresentato: chiamando una funzione passando come input le

coordinate di un voxel, Matlab esplora l'albero e ritorna come output il valore delle caratteristiche associate al voxel. Così facendo è possibile ricostruire la Densegrid associata all'Octomap, salvare tutto in un file testuale e, in un secondo momento, ricostruire l'Octree a partire dalla Densegrid generata. Per quanto riguarda i file da noi considerati, Matlab generava delle Densegrid i cui voxel avevano tutte le dimensioni pari ad 0,1 m e l'unica caratteristica associata era se il voxel fosse occupato o meno. Nella seguente tabella riportiamo le dimensioni del file binario, del file testuale generato e del file JSON finale per le varie tipologie di mappe locali.

Nome file	File binario	File testuale	Densegrid	Octree	Sparsegrid
fr_078_tidyup	408 KB	5.022 KB	1.901 KB	185.611 KB	525 KB
fr_079	190 KB	16.057 KB	6.980 KB	20.935 KB	985 KB
fr_campus	17.625 KB	1.545.089 KB	596.407 KB	359.280 KB	6.060 KB

Possiamo notare come Densegrid e Sparsegrid occupino meno memoria del file testuale originario, mentre Octree ha un comportamento ambivalente. Lasciando un attimo da parte quest'ultima tipologia di mappa, inizieremo prima a commentare le due tipologie di griglie. Sebbene la diminuzione nell'utilizzo della memoria sia presente in entrambi i casi, l'entità e la motivazione sono differenti. Per quanto riguarda la Densegrid, infatti, il file finale è grande circa un terzo di quello originario: ciò è dovuto al fatto che nel file testuale di partenza per ogni voxel, oltre al valore occupato/non occupato, sono presenti le coordinate del voxel stesso. Come abbiamo già detto nella Sezione 4.2.1, tale dato è ridondante, in quanto ricavabile dall'ordinamento con cui presentiamo i voxel. Per ognuno di questi, quindi, delle quattro informazioni ricevute (le tre coordinate e il valore della caratteristica associata) ne riportiamo solo uno, riducendo di un quarto i dati presenti. Tuttavia il risparmio in termini di memoria è di due terzi invece che di tre quarti a causa della presenza delle parentesi quadre che racchiudono ogni singolo voxel, separandolo dal successivo. Per le Sparsegrid tale spiegazione non vale, in quando in questo modello di mappa locale le coordinate sono presenti: il risparmio, in questo caso, è dovuto alla riduzione di voxel riportati. Come abbiamo spiegato nella Sezione 4.2.5 in una Sparsegrid vengono riportati solamente i voxel ritenuti significativi. In questo caso riteniamo significativo un voxel occupato, implicando che quelli non riportati siano liberi. In questo modo riduciamo enormemente il numero di voxel presenti, andando ad impattare nella sulla dimensione del file finale.

Un discorso a parte va fatto per l'Octree. Come si può vedere, questa presenta un aumento delle dimensioni del file per le prime due istanze, mentre una diminuzione per la terza, presentando performance perfino migliori di Densegrid. Tale comportamento può essere giustificato andando ad analizzare cosa rappresentano le mappe in questione. Tutte e tre sono scansioni dell'Università di Freiburg, la prima (fr_078_tidyup) che rappresenta due aule di informatica, la seconda (fr_079) un corridoio e la terza (fr_campus) lo spazio all'aperto del campus di informatica. Possiamo notare come le dimensioni dei tre ambienti siano molto differenti: se i primi due hanno delle dimensioni più o meno simili (14m x 12,8m x 4,7m del primo contro i 43,8m x 18,2m x 3,3m del secondo) il terzo è molto più ampio (292m x 167m x 28m). Tale discorso è importante perché Octree introduce un importante overhead nelle informazioni, creando una struttura dati complessa per ogni nodo. Di contro la struttura dei voxel nelle altre due mappe locali è più semplice, con due array per la Sparsegrid e solo uno per la Densegrid. Tale overhead vanifica nelle mappe più piccole l'effetto positivo di risparmio di memoria dato dalle caratteristiche della struttura dati descritte nella Sezione 4.2.2. Da notare, inoltre, come nella prima istanza l'aumento di memoria è anche più marcato che non nella seconda. È ipotizzabile pensare che in un'aula (l'ambiente rappresentato nella prima

mappa) vi siano molti oggetti sparsi per la mappa (ad esempio banchi, sedie, strumentazione varia…). Tali oggetti impongono un'espansione maggiore dei nodi dell'albero, impattando maggiormente sulla memoria. La seconda istanza, invece, rappresenta un corridoio, uno spazio relativamente vuoto. È possibile, quindi, che la struttura dati dell'Octree ci permette di risparmiare memoria, ma non in maniera da controbilanciare l'overhead. È solo nella terza istanza, un ambiente sempre abbastanza vuoto e, questa volta, sufficientemente grande, che l'Octree ha le sue performance migliori, non solo occupando meno memoria della versione testuale completa ma riuscendo perfino ad essere più efficiente della Densegrid. È ipotizzabile che con mappe ancora più grandi tale guadagno sia ancora più apprezzabile.

Un discorso finale va fatto nei confronti dei file binari. Tutte e tre le tipologie di mappe presentano un peggioramento di performance. Ciò è atteso, in quanto un file testuale sacrifica tale aspetto a favore di una maggior facilità di lettura. Questo, tuttavia, non inficia la bontà dello standard, in quanto l'obiettivo principale è proprio definirne uno che sia il più semplice da leggere per i robot ma anche per l'essere umano.

Nome file	Densegrid	Octree	Sparsegrid
fr_078_tidyup	1,776964 s	3.695,636693 s	0,522997 s
fr_079	6,206035 s	501,003563 s	1,209577 s
fr_campus	543,603248 s	7.273,099038 s	55,706952 s

Per quanto riguarda i tempi di conversione, ancora una volta dobbiamo affrontare due discorsi separati, uno per Densegrid e Sparsegrid e uno per Octree. Per quanto riguarda le due tipologie di griglie, possiamo notare che per i primi due file (fr_078_tidyup e fr_079) il tempo di conversione è nell'ordine dei pochi secondi. Per quanto riguarda l'ultimo file (fr_campus), il più grande dei tre, il tempo di conversione è nell'ordine del minuto per Sparsegrid e di poco meno di dieci minuti per Densegrid. Come è lecito supporre, il tempo di conversione aumenta con la dimensione del file da convertire ma, mentre per Sparsegrid questo aumento rimane comunque trascurabile e rende possibile una conversione online, per quanto riguarda Densegrid questo potrebbe non essere possibile. Infatti un'attesa di dieci minuti per la conversione del file da trasferire potrebbe essere un'attesa eccessiva da affrontare in alcune applicazioni. Tuttavia, e questo è una costante per tutte le conversioni da noi effettuate, il collo di bottiglia che rallenta enormemente la conversione è la gestione di un file così grosso in memoria. Un possibile approccio potrebbe essere la gestione tramite stream di dati: così facendo la memoria non verrebbe completamente utilizzata, aumentando le prestazioni. Per quanto riguarda Octree, invece, vediamo come la durata di conversione va dai dieci minuti a alle due ore. A maggior ragione la conversione non può essere fatta online, bensì sarebbe più conveniente avere salvata direttamente la copia definita nel nostro standard. Tale soluzione è fattibile ricordando che le mappe in questione sono mappe statiche che, una volta ottenute, non verranno più cambiate. Tuttavia è bene ricordare che il file di partenza su cui noi abbiamo effettuato la conversione non rappresentava un Octree, bensì una Densegrid. Il nostro script Python, quindi, non ha solo effettuato la conversione, ma prima ha effettivamente costruito l'Octree associato alla Densegrid proposta. È lecito supporre che una conversione a partire da una struttura dati in memoria che rappresenti un octree sia più veloce ed efficiente.


```
{
    "localmap_id": "densegrid_prova",
    "time": "2020-06-28T18:48:36.497552Z",
```

```
    "map_description": "Una densegrid di prova",
    "coordinate_system": "relative",
    "resolution": [1,1,1],
    "size": [2,2,2],
    "list_of_characteristics": [
        {
            "C_name": "occupation",
            "C_description": "Indica se il voxel è occupato o meno. Se 0 è libero
                              se 1 è occupato.",
            "C_values": "{0,1}"
        }],
    "list_of_voxels": [
        [0],[1],[1],[0],[0],[1],[1],[0]
    ]
}
```


Nella sezione sopra riportata vi è un semplice esempio di Densegrid. Notiamo come l'unica caratteristica presente (occupation) sia completamente specificata, con il nome (obbligatorio) ma anche con una descrizione e un insieme di valori ammissibili. Un singolo voxel ha solamente questa come caratteristica e, quindi, viene rappresentato come un array con un solo elemento.


```
{
    "localmap_id": "octree_prova",
    "time": "2020-06-28T18:48:36.497552Z",
    "map_description": "Un octree di prova",
    "coordinate_system": "relative",
    "list_of_characteristics": [
        {
            "C_name": "occupation",
            "C_description": "Indica se il voxel è occupato o meno. Se 0 è libero
                              se 1 è occupato.",
            "C_values": "{0,1}"
        }],
    "tree": {
        "node_characteristics": [],
        "node_children": [
            {
                "node_characteristics": [0],
                "node_children": []
            },
            {
                "node_characteristics": [1],
                "node_children": []
            },
            {
                "node_characteristics": [1],
                "node_children": []
            },
            {
                "node_characteristics": [0],
```

```
                "node_children": []
            },
            {
                "node_characteristics": [0],
                "node_children": []
            },
            {
                "node_characteristics": [1],
                "node_children": []
            },
            {
                "node_characteristics": [0],
                "node_children": []
            },
            {
                "node_characteristics": [1],
                "node_children": []
            }
        ]
    }
}
```

Nella sezione sopra riportata vi è un semplice esempio di Octree. Notiamo per prima cosa l'assenza del campo max_depth, che vuol dire che l'albero è completamente esplorato. In secondo luogo qui sopra vi è un esempio della struttura innestata descritta nella Sezione 4.2.2: nel campo tree abbiamo un nodo (la radice) nel cui campo node_children sono presenti gli altri nodi dell'albero. Avendo definito un'unica caratteristica (l'occupancy) ipotizziamo che questa sia quella che governa l'espansione: i nodi intermedi (quindi la sola radice) non presentano alcun valore assegnato, mentre viene assegnato ai nodi foglia (che, analogamente, hanno un array vuoto nel campo node_children).

```
{
    "localmap_id": "sparsegrid_prova",
    "time": "2020-06-28T18:48:36.497552Z",
    "map_description": "Una sparsegrid di prova",
    "coordinate_system": "relative",
    "resolution": [1,1,1],
    "size": [50,50,50],
    "list_of_characteristics": [
        {
            "C_name": "occupation",
            "C_description": "Indica se il voxel è occupato o meno. Se 0 è libero
                              se 1 è occupato.",
            "C_values": "{0,1}"
        }],
    "list_of_voxels": [
        [[0,0,0],[1]],
        [[1,2,3],[1]],
        [[42,42,42],[1]]
    ]
}
```

Infine qui sopra riportiamo un semplice esempio di Sparsegrid. Notiamo come stiamo rappresentando uno spazio di 50m per lato ma, grazie alla semplificazione dovuta da questa tipologia di mappa locale, non siamo costretti a riportare ogni singolo voxel che lo compone ma solamente i tre qui presenti (che ipotizziamo essere gli unici tre occupati).

5.2.2 Pointcloud

Pointcloud è la tipologia di mappa locale che è stata più facile da testare, in quanto è un formato estremamente diffuso e comune nella rappresentazione di ambienti tridimensionali. Per questo motivo abbiamo un gran numero di mappe da convertire e da portare come esempi. In particolare le istanze da noi riportate provengono da due fonti differenti: la prima è costituita da scansioni laser ottenute da una piattaforma mobile in un contesto urbano nei dintorni del campus del Carnegie Mellon University della città di Oakland [38]. Ad ogni punto è associata un'etichetta numerica, che codifica un particolare significato (come veicolo, umano, facciata…). La seconda fonte è Semantic3D [39] che fornisce anch'essa una serie di scansioni laser a cui vengono associate delle etichette di classificazione, il cui obiettivo è essere da benchmark per algoritmi di classificazione. La più importante differenza tra i due dataset è la dimensione dei file: se il dataset Oakland contiene circa 100.000 punti per file, un file di Semantic3D arriva anche a diversi milioni di punti. Inoltre il dataset Semantic3D presenta due file per ogni porzione rappresentata: in un primo file sono riportate per ogni punto le coordinate, un colore ad esso associato (rappresentato tramite tre numeri che corrispondono ai valori di rosso, giallo e blu) e l'intensità del punto. Nel secondo file sono presenti i label assegnati ad ogni punto del primo file.

Nome file	Dimensione originale	Dimensione formato
Bildstein_station1	87.005 + 1.075.565 KB	1.896.572 KB
Bildstein_station3	70.300 + 876.882 KB	1.536.741 KB
Bildstein_station5	73.007 + 915.803 KB	1.595.522 KB
Domfountain_station1	131.809 + 1.625.752 KB	2.862.655 KB
oakland_part2_ac	2.530 KB	3.595 KB
oakland_part2_ad	2.604 KB	3.670 KB
oakland_part2_ae	2.728 KB	3.797 KB
oakland_part2_ag	2.766 KB	3.835 KB
oakland_part2_ah	2.784 KB	3.856 KB
oakland_part2_ai	2.874 KB	3.946 KB
oakland_part2_aj	2.861 KB	3.935 KB
oakland_part2_ak	2.798 KB	3.869 KB
oakland_part2_al	2.845 KB	3.917 KB
oakland_part2_ao	2.773 KB	3.846 KB
oakland_part3_aj	2.210 KB	3.434 KB
oakland_part3_ak	2.522 KB	3.587 KB
oakland_part3_al	2.370 KB	3.694 KB
oakland_part3_am	2.541 KB	3.606 KB
oakland_part3_an	2.601 KB	3.668 KB
oakland_part3_ao	2.487 KB	3.550 KB
oakland_part3_ap	1.003 KB	1.436 KB

Possiamo notare come, in tutti i casi, la dimensione dei file dello standard da noi definito è più grande dei file testuali di partenza. Ciò è dovuto dalla presenza di informazioni extra riportate (il nome della mappa, la data di creazione…) e dalla sintassi di un file JSON che, per quanto snella, rimane comunque più verbosa di quella di un file csv. Nonostante questo, possiamo notare che gli

incrementi non cambiano l'ordine di grandezza dei file in questione: per i file di Semantic3D (i primi quattro) l'aumento si aggira intorno al 60% delle dimensioni, mentre scende al 40% per i file del dataset Oakland (gli ultimi). In generale, quindi, utilizzare lo standard da noi definito non peggiora in maniera significativa le performance, aumentando tuttavia le informazioni trasmesse e la leggibilità del file.

Nome file	Durata conversione
Bildstein_station1	940,237572 s
Bildstein_station3	599,089276 s
Bildstein_station5	610,963479 s
Domfountain_station1	3.683,320757 s
oakland_part2_ac	1,829947 s
oakland_part2_ad	1,928402 s
oakland_part2_ae	1,879189 s
oakland_part2_ag	1,950808 s
oakland_part2_ah	1,952881 s
oakland_part2_ai	1,899942 s
oakland_part2_aj	1,897422 s
oakland_part2_ak	1,982393 s
oakland_part2_al	1,871874 s
oakland_part2_ao	1,934361 s
oakland_part3_aj	1,698915 s
oakland_part3_ak	1,893342 s
oakland_part3_al	1,879581 s
oakland_part3_am	1,89388 s
oakland_part3_an	2,001918 s
oakland_part3_ao	2,017482 s
oakland_part3_ap	0,693235 s

Possiamo notare come, per quanto riguarda i file del dataset Oakland, il tempo di conversione si aggira intorno ai due secondi, un tempo compatibile con la conversione online. Per quanto riguarda Semantic3D, invece, notiamo come il tempo di conversione aumenta aggirandosi intorno ai dieci minuti con un solo file che raggiunge l'ora. A parte la grandezza del file in sé (una cui possibile soluzione è già stata proposta nella Sezione 4.2.1 per le Densegrid) qui è bene ricordare che la conversione dei file Semantic3D è resa ancora più complicata dal fatto che i dati sono divisi su due file testuali differenti. Lo script Python è obbligato, quindi, a leggere contemporaneamente entrambi, in modo da associare a ciascun punto l'etichetta corrispondente, peggiorando ulteriormente le performance.

```
{
    "localmap_id": "pointcloud_prova",
    "time": "2020-06-28T18:48:36.497552Z",
    "map_description": "Una pointcloud di prova",
    "coordinate_system": "relative",
    "list_of_characteristics": [
        {
            "C_name": "occupation",
            "C_description": "Indica se il punto è occupato o meno. Se 0 è libero
                              se 1 è occupato.",
            "C_values": "{0,1}"
```

```
        }],
    "list_of_points": [
        [[0,0,0],[1]],
        [[1.3,2.2,3.1],[1]],
        [[10,15,20],[1]],
        [[42,42,42],[1]]
    ]
}
```

Nella sezione sopra riportata presentiamo un semplice esempio di una Pointcloud. Possiamo notare come sia molto simile ad una Sparsegrid, se non fosse che l'elemento atomico che la definisce non è più il voxel ma il punto. Non ha più senso, quindi, definire le dimensioni del punto e, in secondo luogo, le sue coordinate possono essere numeri reali e non più interi, come mostrato dal secondo punto dell'elenco.

5.2.3 Polygonmesh

Per quanto riguarda Polygonmesh, questa tipologia di mappa locale si è dimostrata particolarmente ostica da testare. Questo perché, come detto nella Sezione 4.2.4, tale rappresentazione dello spazio è usata principalmente per rappresentare oggetti in tre dimensioni, specialmente in ambito di design e progettazione. Presentiamo due tipologie di esempi: la prima non rappresenta delle mappe vere e proprie, ma dei singoli oggetti tridimensionali. Riteniamo che sia comunque un esempio valido e interessante da convertire, in quanto una mappa può essere rappresentata come una naturale estensione di un file del genere, in quanto intesa come una serie di oggetti tridimensionali che condividono lo stesso sistema di riferimento. Uno dei formati in grado di rappresentare oggetti tramite Polygonmesh è Ply [40], un formato open source per la rappresentazione di file poligonali. La peculiarità di tale formato è la possibilità di rappresentare i dati in un file binario o (per noi più utile) in un file testuale. Un file ply è diviso in sezioni, una che elenca le coordinate di tutti i vertici nello spazio e un'altra che elenca le facce, associando ad ognuna di essa i suoi vertici.

Nome file	Dimensione originale	Dimensione formato
Aeroplane	73 KB	219 KB
Ant	24 KB	78 KB
Apple	52 KB	152 KB
Balance	59 KB	193 KB
Beethoven	156 KB	453 KB
Big_atc	440 KB	1.246 KB
Big_dodge	535 KB	1.522 KB
Big_porche	332 KB	952 KB
Big_spider	270 KB	819 KB
Canstick	137 KB	383 KB
Chopper	62 KB	185 KB
Cow	182 KB	525 KB
Cube	1 KB	0 KB
Dart	1 KB	0 KB
Dodecahedron	1 KB	1 KB
Dolphins	47 KB	146 KB
Egret	50 KB	141 KB
Ellell	1 KB	2 KB
F16	146 KB	411 KB

Footbones	133 KB	382 KB
Fracttree	211 KB	518 KB
Galleon	150 KB	429 KB
Hammerhead	160 KB	464 KB
Helix	198 KB	577 KB
Hind	189 KB	570 KB
Icosahedron	1 KB	2 KB
Kerolamp	97 KB	297 KB
Ketchup	30 KB	86 KB
Mug	114 KB	318 KB
Octahedron	1 KB	1 KB
Part	37 KB	109 KB
Pickup_big	436 KB	1.231 KB
Pump	72 KB	206 KB
Pyramid	1 KB	0 KB
Sandal	162 KB	456 KB
Saratoga	229 KB	594 KB
Scissors	19 KB	55 KB
Shark	26 KB	69 KB
Sphere	17 KB	70 KB
Steeringwheel	37 KB	118 KB
Stratocaster	276 KB	790 KB
Streetlamp	281 KB	804 KB
Teapot	66 KB	199 KB
Tennis_shoes	113 KB	328 KB
Tetrahedron	1 KB	0 KB
Tommygun	267 KB	753 KB
Trashcan	56 KB	163 KB
Turbine	206 KB	598 KB
Urn2	18 KB	53 KB
Walkman	125 KB	371 KB
Weathervane	162 KB	471 KB

Data la relativa semplicità dei file in questione, l'analisi delle performance in questo caso è solamente indicativo. Notiamo un incremento nella dimensione dei file, in alcuni casi anche di un ordine di grandezza, tuttavia, ciò è perfettamente spiegabile da un overhead iniziale presente in ogni file. Campi come il nome, la descrizione o la data di creazione della mappa in questione, o il nome dei campi stessi impattano sulle performance. Nonostante la relativa semplicità dei file in esame, possiamo ritenerci soddisfatti di constatare che, anche in questo esempio, il formato da noi definito si è mostrato sufficientemente flessibile nel rappresentare anche singoli oggetti.

Nome file	Durata conversione
Aeroplane	0,345845 s
Ant	0,055985 s
Apple	0,058033 s
Balance	0,07503 s
Beethoven	0,132016 s
Big_atc	0,351997 s
Big_dodge	0,503214 s
Big_porche	0,31201 s
Big_spider	0,270026 s

Canstick	0,161212 s
Chopper	0,083961 s
Cow	0,177994 s
Cube	0,002003 s
Dart	0,001999 s
Dodecahedron	0,009008 s
Dolphins	0,059021 s
Egret	0,053024 s
Ellell	0,00895 s
F16	0,125996 s
Footbones	0,124025 s
Fracttree	0,186006 s
Galleon	0,136017 s
Hammerhead	0,152003 s
Helix	0,188999 s
Hind	0,171034 s
Icosahedron	0,015994 s
Kerolamp	0,097971 s
Ketchup	0,035997 s
Mug	0,115 s
Octahedron	0,001998 s
Part	0,043998 s
Pickup_big	0,368986 s
Pump	0,096035 s
Pyramid	0,022 s
Sandal	0,16686 s
Saratoga	0,193021 s
Scissors	0,036025 s
Shark	0,05 s
Sphere	0,024994 s
Steeringwheel	0,049036 s
Stratocaster	0,228931 s
Streetlamp	0,25604 s
Teapot	0,06199 s
Tennis_shoes	0,094037 s
Tetrahedron	0,001975 s
Tommygun	0,312978 s
Trashcan	0,061027 s
Turbine	0,169032 s
Urn2	0,026035 s
Walkman	0,117001 s
Weathervane	0,145012 s

Per completezza anche per le Polygonmesh riportiamo i tempi impiegati per la conversione dei file. Tuttavia, data la semplicità dei file trattati, sono tutti inferiori al secondo. È incoraggiante vedere dei tempi così brevi, tuttavia per stabilire se sia possibile o meno una conversione online sarebbe necessario effettuare qualche test più approfondito su mappe di ambienti e non su oggetti.

Un test più significativo è effettuato grazie al lavoro effettuato dal professor Matteucci sullo sviluppo di un algoritmo in grado di generare una mesh poligonale a partire da una serie di immagini acquisite da una camera mobile [41] [42] [43]. A partire dalla fontana P-11 del dataset EPFL [44] e

dalla sequenza 95 del dataset Kitti [45] sono stati generati rispettivamente il file Fountain-P11 e tutti i restanti altri. I file sono in formato *.off* [46], formato analogo a *.ply*, definito precedentemente: dopo un header iniziale che indica il numero di punti e di facce presenti nel file, vengono elencati prima tutti i punti con le loro coordinate e, in seguito, le facce, a cui ad ognuna di loro vengono assegnati i tre vertici, presi dal primo elenco. In Figura 7 la visualizzazione grafica del file Fountain-P11.

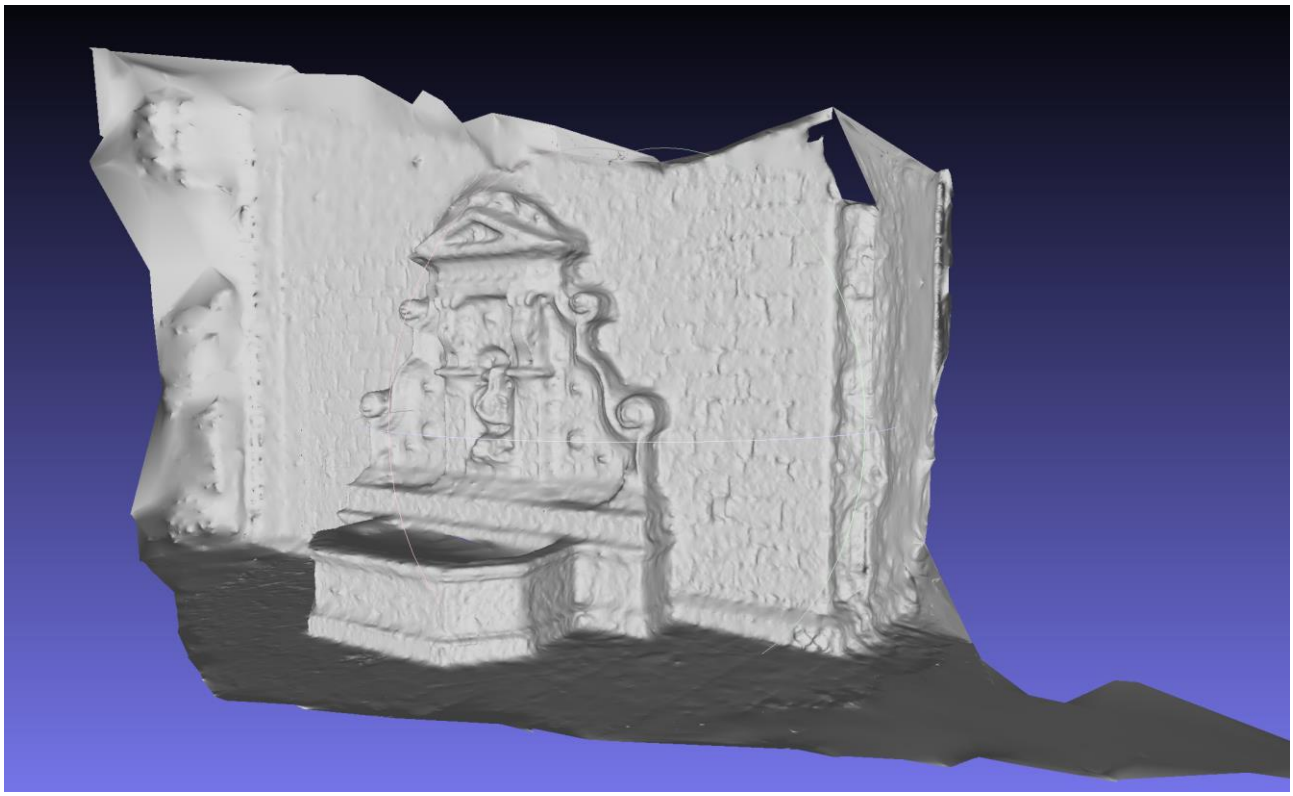

Figura 7 Mesh poligonale del file Fountain-P11

Nome file	Dimensione originale	Dimensione formato
Adaptive_class_density2	118.023 KB	226.678 KB
Class_k_density	94.482 KB	181.684 KB
Density_euclidean	120.002 KB	230.344 KB
Fountain-P11	82.445 KB	221.941 KB
KittiLucaDensity	107.169 KB	205.881 KB
Linear_density	102.695 KB	197.432 KB
Mep	149.909 KB	394.432 KB
Mesh_159 (copia)	85.457 KB	232.371 KB

Come possiamo vedere nella tabella sopra riportata, anche in questo caso notiamo un incremento della memoria utilizzata. Ancora una volta, tale incremento è atteso, in quanto riportiamo informazioni extra rispetto al file testuale originario e, inoltre, i poligoni che compongono lo spazio li rappresentiamo come dati strutturati, con campi a cui diamo dei nomi.

Nome file	Durata conversione
Class_k_density	87,078042 s
Density_euclidean	74,251799 s
Fountain-P11	91,905961 s
KittiLucaDensity	73,609685 s

Linear_density	78,699055 s
Mep	73,498023 s
Mesh_159 (copia)	114,668192 s

Per quanto riguarda la durata delle conversioni, i tempi oscillano tra uno o due minuti, sufficientemente brevi per ammettere una conversione online.


```
{
    "localmap_id": "cube.txt",
    "time": "2020-07-06T18:34:05.184597Z",
    "map_description": "Polygonmesh local map of cube.txt found at
                        https://people.sc.fsu.edu/~jburkardt/data/ply/ply.html",
    "coordinate_system": "relative",
    "list_of_characteristics_point": [],
    "list_of_characteristics_polygon": [],
    "list_of_points": [
        [[-1.0, -1.0, -1.0], []],
        [[1.0, -1.0, -1.0], []],
        [[1.0, 1.0, -1.0], []],
        [[-1.0, 1.0, -1.0], []],
        [[-1.0, -1.0, 1.0], []],
        [[1.0, -1.0, 1.0], []],
        [[1.0, 1.0, 1.0], []],
        [[-1.0, 1.0, 1.0], []]
    ],
    "list_of_polygons": [
        {
            "polygon_characteristics": [],
            "list_of_vertices": [0, 1, 2, 3]
        },
        {
            "polygon_characteristics": [],
            "list_of_vertices": [5, 4, 7, 6]
        },
        {
            "polygon_characteristics": [],
            "list_of_vertices": [6, 2, 1, 5]
        },
        {
            "polygon_characteristics": [],
            "list_of_vertices": [3, 7, 4, 0]
        },
        {
            "polygon_characteristics": [],
            "list_of_vertices": [7, 3, 2, 6]
        },
        {
            "polygon_characteristics": [],
            "list_of_vertices": [5, 1, 0, 4]
        }
```

```
    ]
}
```

Un vantaggio dell'aver convertito file piccoli è la possibilità di poterne riportare uno per intero (nella sezione qui sopra). Questo rappresenta un cubo e, possiamo notare, gli otto vertici riportati nel campo list_of_points e le sei facce quadrate riportate nel campo list_of_polygons. Come abbiamo già anticipato nella Sezione 4.2.4, Polygonmesh è l'unica mappa locale che permette l'assenza di caratteristiche da associare ai suoi elementi atomici, come dimostra l'esempio riportato.

6 Conclusioni e sviluppi futuri

Il principale scopo di questo lavoro di tesi è stato quello di definire un formato di rappresentazione di mappe tridimensionali per il supporto alle operazioni di robot autonomi, in modo tale da proporlo come possibile standard. Abbiamo per prima cosa analizzato gli standard esistenti per la rappresentazione di ambienti in tre dimensioni, individuando quali fossero i loro limiti. Abbiamo, in seguito, evidenziato quali fossero le caratteristiche mancanti che volevamo introdurre con il nostro formato: in primo luogo l'incertezza delle misurazioni riportate nella mappa e, in seguito, la suddivisione dello spazio rappresentato in sottospazi più piccoli in relazione tra di loro attraverso un grafo possibilmente ciclico. Nella tesi abbiamo presentato la nostra proposta per l'eventuale nuovo standard di rappresentazione, riportando e descrivendo gli schemi JSON da noi prodotti. Abbiamo introdotto il concetto di mappa globale e mappa locale e, per ogni tipo di mappa locale, descritto le principali caratteristiche. Infine, abbiamo riportato i risultati sperimentali da noi ottenuti, che dimostrano come il formato da noi definito riesca a rappresentare mappe di dimensioni realistiche. Abbiamo notato come le dimensioni dei file prodotti siano ancora gestibili in ambito robotico, nonostante l'utilizzo di una rappresentazione più verbosa rispetto alle rappresentazioni comunemente utilizzate. Per quanto riguarda i tempi di conversione, invece, abbiamo notato che i risultati variano in base alla tipologia e alla dimensione del file convertito. Negli esempi da noi riportati, Densegrid e Sparsegrid hanno avuto risultati tali per cui è possibile effettuare una conversione online, mentre per quanto riguarda Sparsegrid, ciò avviene solo per file sufficientemente piccoli. Infine, per quanto riguarda Octree, la conversione è ancora più onerosa e, per i risultati che abbiamo osservato, non è consigliata una conversione online.

In questo modo si conclude il nostro lavoro di tesi. Riteniamo che il formato da noi definito abbia ottime potenzialità e sia sufficientemente flessibile e adattabile a future esigenze. Per quanto riguarda eventuali sviluppi futuri, la prima, importante, azione da fare è testare più approfonditamente il formato definito, tramite una serie di nuovi casi d'uso reali con mappe anche più complesse di quelle da noi convertite. In particolare, è sulla mappa globale che si devono concentrare i maggiori sforzi in quanto, come già detto nella Sezione 5.1 è l'unica tipologia di mappa i cui esempi riportati non sono esempi reali ma creati da noi. Inoltre, potrebbe essere interessante provare a definire un formato che riassuma sia il lavoro da noi svolto in ambito tridimensionale sia lo standard IEEE Std 1873^{TM}-2015 per il mondo bidimensionale, in modo da avere un formato univoco che gestisca entrambi i casi. Sia lo standard IEEE Std 1873^{TM}-2015 sia il formato da noi definito definiscono solamente un metodo di rappresentazione per mappe statiche. Rimane, quindi, ancora aperto il problema di come rappresentare mappe dinamiche, sia in due che in tre dimensioni. Nel formato da noi definito vi è presente un campo per riportare la data di creazione di una mappa locale, in modo tale da supportare una versione primitiva di evoluzione della mappa nel tempo. Tuttavia tale meccanismo può non essere sufficiente e dovrà essere oggetto di studio e discussione per raggiungere una sufficiente maturità.

Appendice A Schemi

A.1 Mappa globale

```
{
    "$id": "https://github.com/effelam/JSON_3dMaps/blob/master/globalmap.json",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "globalMap",
    "description": "Implementation of the global map. A global map is a graph
                    whose vertices are local maps and whose (directed) arcs
                    represent the transformation between the coordinate systems of
                    the two connected local maps.",
    "type": "object",

    "properties":{
        "globalmap_id": {"type": "string"},
        "list_of_vertices": {
            "description": "List of all the vertices. Each vertex is a local map,
                            represented by a triple: its id, the type of local
                            map, and the path where the map is stored.",
            "type": "array",
            "items": {"$ref": "#/definitions/vertex"}
        },
        "list_of_arcs": {
            "description": "List of all the arcs. Each arc represents a
                            transformation between two local maps.",
            "type": "array",
            "items": {"$ref": "#/definitions/transformation"}
        }
    },

    "definitions": {
        "vertex" :{
            "type": "object",
            "properties": {
                "vertex_id": {"type": "string"},
                "vertex_type": {
                    "type": "string",
                    "enum": ["densegrid", "octree", "pointcloud", "polygonmesh",
                             "sparsegrid", "other"]
                },
                "vertex_path": {"type": "string"}
            },
            "required": ["vertex_id", "vertex_type"]
        },
```

```
        "transformation": {
            "type": "object",

            "properties": {
                "transformation_id": {"type": "string"},
                "arc_head": {"type": "string"},
                "arc_tail": {"type": "string"},
                "rotation": {
                    "type": "array",
                    "minItems": 4,
                    "maxItems": 4,
                    "items": {"type": "number"}
                },
                "translation": {
                    "type": "array",
                    "minItems": 3,
                    "maxItems": 3,
                    "items": {"type": "number"}
                },
                "uncertainty": {
                    "description": "7x7 uncertainty matrix",
                    "type": "array",
                    "minItems": 28,
                    "maxItems": 28,
                    "items": {"type": "number"}
                }
            },
            "required": ["transformation_id", "arc_head", "arc_tail"]
        }
    },

    "additionalProperties": false,
    "required": ["globalmap_id", "list_of_vertices", "list_of_arcs"]
}
```

A.2 Mappa locale

A.2.1 Densegrid

```
{
    "$id": "https://github.com/effelam/JSON_3dMaps/blob/master/local_maps_JSON/de
nsegrid.json",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "densegrid",
    "description": "Implementation of the local map type densegrid. A densegrid
                    is a parallelepiped divided in equal voxels, which are
                    described by the proper field.",
```

```
    "type": "object",

    "properties":{
        "localmap_id": {"type": "string"},
        "time": {"description": "Time of creation of the map.", "type": "string",
                 "format": "date-time"},
        "map_description": {"type": "string"},
        "coordinate_system": {"type": "string",
                              "enum": ["absolute", "relative"]},
        "absolute_pose": {"description": "Geo-referenced location of the local
                                          map. Required if coordinate_system is
                                          absolute, not required if
                                          coordinate_system is relative."},

        "resolution": {
            "description": "Resolution (length of a single voxel's edges) along
                            the x, y, and z axes, expressed in meters.",
            "type": "array",
            "minItems": 3,
            "maxItems": 3,
            "items": {"type": "number", "exclusiveMinimum": 0}
        },

        "size": {
            "description": "Size of the dense grid, expressed as the number of
                            voxels along the x, y, and z axes.",
            "type": "array",
            "minItems": 3,
            "maxItems": 3,
            "items": {"type": "integer", "exclusiveMinimum": 0}
        },

        "list_of_characteristics": {
            "description": "List of the voxels' characteristics. Each voxel
                            should present a value for each one of these.",
            "type":"array",
            "items": {
                "type": "object",
                "properties": {
                    "C_name": {"type": "string"},
                    "C_description": {"type": "string"},
                    "C_values": {"type": "string"}
                },
                "required": ["C_name"]
            },
            "minItems": 1
        },

        "list_of_voxels": {
```

```
            "description": "List of all the voxels. The first one is the voxel
                            with local coordinate (0,0,0), the second (1,0,0), ...
                            (size[0]-1,0,0), (0,1,0), ..., (size[0]-1,size[1]-1,0),
                            (0,0,1), ..., (size[0]-1,size[1]-1,size[2]-1).
                            Each voxel is a list of characteristics, described in
                            the field characteristics.",
            "type": "array",
            "items": {"$ref": "#/definitions/densegrid_voxel"}
        }
    },

    "definitions": {
        "densegrid_voxel": {
            "description": "A densegrid_voxel is an array of values, one for each
                            characteristic reported in the wanted_characteristics
                            field.",
            "type": "array",
            "minItems": 1
        }
    },

    "if": {
        "properties": {
            "coordinate_system": {"const": "absolute"}
        }
    },
    "then": {
        "properties": {
            "absolute_pose": {
                "type": "object",
                "properties": {
                    "translation": {
                        "type": "array",
                        "items": [
                            {
                                "type": "number",
                                "minimum": -90,
                                "maximum": 90
                            },
                            {
                                "type": "number",
                                "minimum": -180,
                                "maximum": 180
                            },
                            {
                                "type": "number"
                            }
                        ]
                    },
```

```
                    "orientation": {
                        "type": "array",
                        "items": [
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            },
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            },
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            }
                        ]

                    }
                },
                "required": ["translation", "orientation"]
            }
        },
        "required": ["absolute_pose"]
    },
    "else": {
        "properties": {
            "absolute_pose": false
        }
    },

    "additionalProperties": false,
    "required": ["localmap_id", "coordinate_system", "resolution", "size",
                 "list_of_voxels", "list_of_characteristics"]
}
```

A.2.2 Octree

```
{
    "$id": "https://github.com/effelam/JSON_3dMaps/blob/master/local_maps_JSON/octree.json",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "octree",
    "description": "Implementation of the local map type octree. An octree
                    represents the space with an octree, that is a tree. Each
```

```
                node of this tree that is not a leaf has exactly 8 children.
                Each node (including leaves) has the characteristics listed
                in the corresponding field.",
    "type": "object",

    "properties":{
        "localmap_id": {"type": "string"},
        "time": {"description": "Time of creation of the map.", "type": "string",
                 "format": "date-time"},
        "map_description": {"type": "string"},
        "coordinate_system": {"type": "string",
                              "enum": ["absolute", "relative"]},
        "absolute_pose": {"description": "Geo-referenced location of the local
                                          map. Required if coordinate_system is
                                          absolute, not required if
                                          coordinate_system is relative."},

        "size": {
            "description": "Size of the space, expressed in meters along the x,
                            y, and z axes.",
            "type": "array",
            "minItems": 3,
            "maxItems": 3,
            "items": {"type": "number", "exclusiveMinimum": 0}
        },

        "max_depth": {
            "description": "Maximum number of levels of the tree. If not
                            specified the tree is fully explored.",
            "type": "integer",
            "exclusiveMinimum": 0
        },

        "list_of_characteristics": {
            "description": "List of the nodes' characteristics. Nodes will
                            present a value for some of these.",
            "type":"array",
            "items": {
                "type": "object",
                "properties": {
                    "C_name": {"type": "string"},
                    "C_description": {"type": "string"},
                    "C_values": {"type": "string"}
                },
                "required": ["C_name"]
            },
            "minItems": 1
            },
```

```
        "tree": {
            "description": "The actual tree. The first node is the root.",
            "$ref": "#/definitions/node"
        }
    },

    "definitions": {
        "node": {
            "type": "object",

            "properties": {
                "node_characteristics": {
                    "type": "array",
                    "minItems": 1
                },
                "node_children": {
                    "anyOf": [
                        {
                            "type": "array",
                            "minItems": 0,
                            "maxItems": 0
                        },
                        {
                            "type": "array",
                            "minItems": 8,
                            "maxItems": 8,
                            "items": {"$ref": "#/definitions/node"}
                        }
                    ]
                }
            },
            "required": ["node_children", "node_characteristics"]
        }
    },

    "if": {
        "properties": {
            "coordinate_system": {"const": "absolute"}
        }
    },
    "then": {
        "properties": {
            "absolute_pose": {
                "type": "object",
                "properties": {
                    "translation": {
                        "type": "array",
                        "items": [
                            {
```

```
                                "type": "number",
                                "minimum": -90,
                                "maximum": 90
                            },
                            {
                                "type": "number",
                                "minimum": -180,
                                "maximum": 180
                            },
                            {
                                "type": "number"
                            }
                        ]
                    },
                    "orientation": {
                        "type": "array",
                        "items": [
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            },
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            },
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            }
                        ]

                    }
                },
                "required": ["translation", "orientation"]
            }
        },
        "required": ["absolute_pose"]
    },
    "else": {
        "properties": {
        "absolute_pose": false
        }
    },

    "additionalProperties": false,
    "required": ["localmap_id", "coordinate_system", "size", "octree",
```

```
                "list_of_characteristics"]
}
```

A.2.3 Pointcloud

```
{
    "$id": "https://github.com/effelam/JSON_3dMaps/blob/master/local_maps_JSON/po
intcloud.json",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "pointcloud",
    "description": "Implementation of the local map type pointcloud. A pointcloud
                is a list of points, each one that has a coordinate and a list
                of characteristics.",
    "type": "object",

    "properties":{
        "localmap_id": {"type": "string"},
        "time": {"description": "Time of creation of the map.", "type": "string",
                 "format": "date-time"},
        "map_description": {"type": "string", "maxLength": 250},
        "coordinate_system": {"type": "string",
                              "enum": ["absolute", "relative"]},
        "absolute_pose": {"description": "Geo-referenced location of the Local
                                          map. Required if coordinate_system is
                                          absolute, not required if
                                          coordinate_system is relative."},

        "list_of_characteristics": {
            "description": "List of the points' characteristics. Each node should
                            present a value for each one of these.",
            "type":"array",
            "items": {
                "type": "object",
                "properties": {
                    "C_name": {"type": "string"},
                    "C_description": {"type": "string"},
                    "C_values": {"type": "string"}
                },
                "required": ["C_name"]
            },
            "minItems": 1
        },

        "list_of_points": {
            "description": "List of all the points. Each point is a pair: the
                            first element is its coordinates and the second one is
                            a list of its characteristics as listed in the
```

```
                        characteristic field.",
            "type": "array",
            "items": {"$ref": "#/definitions/point"}
        }
    },

    "definitions": {
        "point": {
            "description": "A point is a pair. The first element represents the
                        coordinates (x,y,z) in the space and the second
                        element is an array of values, one for each
                        characteristic reported in the wanted_characteristics
                        field.",
            "type": "array",
            "items": [
                {
                    "type": "array",
                    "items": {"type": "number"},
                    "minItems": 3,
                    "maxItems": 3
                },
                {
                    "type": "array",
                    "minItems": 1
                }
            ]
        }
    },


    "if": {
        "properties": {
            "coordinate_system": {"const": "absolute"}
        }
    },
    "then": {
        "properties": {
            "absolute_pose": {
                "type": "object",
                "properties": {
                    "translation": {
                        "type": "array",
                        "items": [
                            {
                                "type": "number",
                                "minimum": -90,
                                "maximum": 90
                            },
                            {
```

```
                                "type": "number",
                                "minimum": -180,
                                "maximum": 180
                            },
                            {
                                "type": "number"
                            }
                        ]
                    },
                    "orientation": {
                        "type": "array",
                        "items": [
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            },
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            },
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            }
                        ]

                    }
                },
                "required": ["translation", "orientation"]
            }
        },
        "required": ["absolute_pose"]
    },
    "else": {
        "properties": {
        "absolute_pose": false
        }
    },

    "additionalProperties": false,
    "required": ["localmap_id", "coordinate_system", "list_of_points",
                 "list_of_characteristics"]
}
```

A.2.4 Polygonmesh

```
{
    "$id": "https://github.com/effelam/JSON_3dMaps/blob/master/local_maps_JSON/po
lygonmesh.json",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "polygonmesh",
    "description": "Implementation of the local map type polygonmesh. In this
                    type of map the 3D world is described by a collection of 2D
                    polygons. Each polygon has a value for each characteristic
                    listed in the wanted_characteristics field.",
    "type": "object",

    "properties":{
        "localmap_id": {"type": "string"},
        "time": {"description": "Time of creation of the map.", "type": "string",
                 "format": "date-time"},
        "map_description": {"type": "string"},
        "coordinate_system": {"type": "string",
                              "enum": ["absolute", "relative"]},
        "absolute_pose": {"description": "Geo-referenced location of the local
                                          map. Required if coordinate_system is
                                          absolute, not required if
                                          coordinate_system is relative."},

        "list_of_characteristics": {
            "description": "List of the wanted characteristics. Each node should
                            present a value for each one of these.",
            "type":"array",
            "items": {
                    "type": "object",
                    "properties": {
                        "C_name": {"type": "string"},
                        "C_description": {"type": "string"},
                        "C_values": {"type": "string"}
                    },
                    "required": ["C_name"]
            }
        },

    "list_of_points": {
        "description": "List of all the points.",
        "type": "array",
        "items": {"$ref": "#/definitions/point"}
    },

        "list_of_polygons": {
            "description": "List of all the polygons.",
            "type": "array",
```

```
            "items": {"$ref": "#/definitions/polygon"}
        }
    },

    "definitions": {
        "point": {
            "description": "It represents a point. Each point is represented as a
                            pair of arrays. The first one represents its
                            coordinates, as 3 real numbers. The second one is a
                            collection of the values of the wanted characteristics
                            for that point.",
            "type": "array",
            "items": [
                {
                    "type": "array",
                    "items": {"type": "number"},
                    "minItems": 3,
                    "maxItems": 3
                },
                {
                    "type": "array"
                }
            ]
        },

        "polygon": {
            "type": "object",
            "properties": {

                "polygon_characteristics": {
                    "description": "List of the characteristics of this polygon.
                                    These are taken from the
                                    list_of_characteristcs_polygon field",
                    "type": "array"
                },

                "list_of_vertices": {
                    "description": "The list of points that defines the current
                                    polygon.",
                    "type": "array",
                    "items": {“type": "integer"},
                    "minItems": 3
                }
            },
            "required": ["list_of_points"]
        }
    },

    "if": {
```

```
        "properties": {
            "coordinate_system": {"const": "absolute"}
        }
    },
    "then": {
        "properties": {
            "absolute_pose": {
                "type": "object",
                "properties": {
                    "translation": {
                        "type": "array",
                        "items": [
                            {
                                "type": "number",
                                "minimum": -90,
                                "maximum": 90
                            },
                            {
                                "type": "number",
                                "minimum": -180,
                                "maximum": 180
                            },
                            {
                                "type": "number"
                            }
                        ]
                    },
                    "orientation": {
                        "type": "array",
                        "items": [
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            },
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            },
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            }
                        ]

                    }
                },
```

```
                "required": ["translation", "orientation"]
            }
        },
        "required": ["absolute_pose"]
    },
    "else": {
        "properties": {
        "absolute_pose": false
        }
    },
    "additionalProperties": false,
    "required": ["localmap_id", "coordinate_system", "list_of_points",
                 "list_of_polygons", "list_of_characteristics_point",
                 "list_of_characteristics_polygon"]
}
```

A.2.5 Sparsegrid

```
{
    "$id": "https://github.com/effelam/JSON_3dMaps/blob/master/local_maps_JSON/sparsegrid.json",
    "$schema": "http://json-schema.org/draft-07/schema#",

    "title": "sparsegrid",
    "description": "Implementation of the local map type sparsegrid. A sparsegrid
                    is a parallelepiped divided in smaller voxels, like
                    densegrids. Unlike densegrids, in sparsegrids only voxels with
                    at least one non-null characteristic are explicitly reported
                    in the voxels field.",
    "type": "object",

    "properties": {
        "localmap_id": {"type": "string"},
        "time": {"description": "Time of creation of the map.", "type": "string",
                 "format": "date-time"},
        "map_description": {"type": "string"},
        "coordinate_system": {"type": "string",
                              "enum": ["absolute", "relative"]},
        "absolute_pose": {"description": "Geo-referenced location of the local
                                          map. Required if coordinate_system is
                                          absolute, not required if
                                          coordinate_system is relative."},

        "resolution": {
            "description": "Resolution (length of a single voxel's edges) along
                            the x, y, and z axes, expressed in meters.",
            "type": "array",
            "minItems": 3,
            "maxItems": 3,
```

```
            "items": {
                "type": "number",
                "exclusiveMinimum": 0
            }
        },

        "size": {
            "description": "Size of the grid, expressed as the maximum index of
                            sparse voxels nx, ny, and nz along the x, y, and z
                            axes.",
            "type": "array",
            "minItems": 3,
            "maxItems": 3,
            "items": {
                "type": "integer",
                "exclusiveMinimum": 0
            }
        },

        "list_of_characteristics": {
            "description": "List of the voxels' characteristics. Each voxel
                            should present a value for each one of these.",
            "type":"array",
            "items": {
                "type": "object",
                "properties": {
                    "C_name": {"type": "string"},
                    "C_description": {"type": "string"},
                    "C_values": {"type": "string"}
                },
                "required": ["C_name"]
            },
            "minItems": 1
        },

        "list_of_voxels": {
            "description": "List of all the voxels that have at least one
                            non-null characteristic. Each voxel is a pair: the
                            first element is its coordinates in the grid and the
                            second one is a list of its characteristics as listed
                            in the characteristic field.",
            "type": "array",
            "items": {"$ref": "#/definitions/sparsegrid_voxel"}
        }
    },

    "definitions": {
        "sparsegrid_voxel": {
            "description": "A sparsegrid_voxel is a pair. The first element
```

```
                        represents the coordinates (x,y,z) in the 3D grid and
                        the second element is an array of values, one for each
                        characteristic reported in the wanted_characteristics
                        field.",
            "type": "array",
            "items": [
                {
                    "type": "array",
                    "minItems": 3,
                    "maxItems": 3,
                    "items": {"type": "integer", "minimum": 0}
                },
                {
                    "type": "array",
                    "minItems": 1
                }
            ]
        }
    },

    "if": {
        "properties": {
            "coordinate_system": {"const": "absolute"}
        }
    },
    "then": {
        "properties": {
            "absolute_pose": {
                "type": "object",
                "properties": {
                    "translation": {
                        "type": "array",
                        "items": [
                            {
                                "type": "number",
                                "minimum": -90,
                                "maximum": 90
                            },
                            {
                                "type": "number",
                                "minimum": -180,
                                "maximum": 180
                            },
                            {
                                "type": "number"
                            }
                        ]
                    },
                    "orientation": {
```

```
                        "type": "array",
                        "items": [
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            },
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            },
                            {
                                "type": "number",
                                "minimum": 0,
                                "maximum": 360
                            }
                        ]

                    }
                },
                "required": ["translation", "orientation"]
            }
        },
        "required": ["absolute_pose"]
    },
    "else": {
        "properties": {
            "absolute_pose": false
        }
    },

    "additionalProperties": false,
    "required": ["localmap_id", "coordinate_system", "list_of_voxels",
                 "resolution", "size", "list_of_characteristics"]
}
```

Appendice B Tabelle campi

B.1 Mappa globale

Global map				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
Globalmap_id	Identificativo univoco della mappa	Stringa	Obbligatorio	
List_of_vertices	Lista dei vertici che compongono la mappa. Rappresentano le singole mappe locali	Array di oggetti di tipo Vertex	Obbligatorio	
List_of_arcs	Lista degli archi che collegano i vertici. Ogni arco rappresenta una trasformazione da una mappa locale all'altra	Array di oggetti di tipo Transformation	Obbligatorio	

Vertex				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
Vertex_id	Identificativo univoco del vertice	Stringa	Obbligatorio	È pari al relativo globalmap_id della mappa associata
Vertex_type	Tipo di mappa rappresentata dal vertice	Stringa	Obbligatorio	Può avere solo valori Densegrid, Octree, Pointcloud, Polygonmesh, Sparsegrid, Other
Vertex_path	Percorso assoluto al file dell'istanza della mappa locale rappresentata dal vertice	Stringa	Opzionale	Se non presente, il vertice non rappresenta una mappa locale

Transformation				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
Transformation_id	Identificativo univoco della trasformazione	Stringa	Obbligatorio	
Head	Identificativo della mappa di arrivo della trasformazione	Stringa	Obbligatorio	Ogni vertice è una coppia di stringhe, la prima che rappresenta il tipo di mappa locale e il secondo il nome della mappa

Tail	Identificativo della mappa di partenza della trasformazione	Stringa	Obbligatorio	
Rotation	Rappresenta la rotazione applicata al sistema di riferimento di partenza	Array di quattro numeri	Opzionale	
Traslation	Rappresenta la traslazione applicata al sistema di riferimento di partenza	Array di tre numeri	Opzionale	
Uncertainty	Rappresenta l'incertezza associata alla trasformazione	Array di 28 numeri	Opzionale	È una matrice 7x7 simmetrica, di cui vengono riportati solamente i 28 valori unici

B.2 Mappa locale

Local map				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
Localmap_id	Identificativo univoco della mappa locale	Stringa	Obbligatorio	
Time	Ora e data di creazione della mappa	Data	Opzionale	Ogni vertice è una coppia di stringhe, la prima che rappresenta il tipo di mappa locale e il secondo il nome della mappa
Map_description	Una descrizione di cosa rappresenti questa mappa	Stringa	Opzionale	
Coordinate_system	Specifica se il sistema di riferimento della mappa in questione è localizzato tramite delle coordinate geografiche o meno	Stringa	Obbligatorio	Può avere solamente come valori *absolute* o *relative*
Absolute_pose	Riporta le coordinate associate al sistema di riferimento della mappa in questione	Array di coppie di array	Obbligatorio se il campo precedente ha valore *absolute*, opzionale altrimenti	Il primo array di ogni elemento rappresenta latitudine, longitudine e altitudine del sistema di riferimento, mentre il secondo la rotazione effettuata

				attorno all'asse verticale

B.2.1 Densegrid

Densegrid				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
Resolution	Lunghezza dei lati di un voxel, in metri	Array di tre numeri	Obbligatorio	
Size	Numero di voxel presenti su ogni asse	Array di tre numeri	Obbligatorio	
List_of_characteristics	Lista delle caratteristiche da assegnare ad ogni voxel	Array di stringhe	Obbligatorio	
List_of_voxels	Lista dei voxel che compongono la mappa	Array di oggetti di tipo Densegrid_voxel	Obbligatorio	Vi è un ordinamento specifico. Il primo voxel ha coordinate [0,0,0] e in seguito verranno elencati tutti i voxel aumentando prima la coordinata x, fino al voxel in coordinata [x_{max}, 0, 0], dopo la coordinata y, raggiungendo il voxel [x_{max}, y_{max},0] e infine la coordinata z fino al voxel [x_{max}, y_{max}, z_{max},]

Densegrid_voxel				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
	Lista dei valori di tutte le caratteristiche assegnate a questo voxel	Array	Obbligatorio	Non ha nome

B.2.2 Octree

Octree				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
Size	Dimensione dello spazio rappresentato, in metri	Array di tre numeri	Obbligatorio	
Max_depth	Massima profondità dell'albero	Intero	Opzionale	Se questo valore è assente si è permesso di espandere ogni nodo fino al nodo foglia
List_of_characteristics	Lista delle caratteristiche da assegnare ai nodi	Array di stringhe	Obbligatorio	
Tree	La radice dell'albero	Node	Obbligatorio	

Node				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
Node_characteristics	Lista dei valori di tutte le caratteristiche assegnate a questo nodo	Array	Obbligatorio	
Node_children	Lista dei nodi figli di questo nodo	Array di oggetti di tipo Node vuoto o con esattamente otto elementi	Obbligatorio	

B.2.3 Pointcloud

Pointcloud				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
List_of_characteristics	Lista delle caratteristiche da assegnare ad ogni punto	Array di stringhe	Obbligatorio	
List_of_points	Lista dei punti	Array di oggetti di tipo Point	Obbligatorio	

Point				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
	Coordinate del punto	Array di tre numeri	Obbligatorio	Nessun nome
	Lista dei valori di tutte le caratteristiche assegnate a questo punto	Array	Obbligatorio	Nessun nome

B.2.4 Polygonmesh

Polygonmesh				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
List_of_characteristics_point	Lista delle caratteristiche da assegnare ai vertici	Array di stringhe	Obbligatorio	
List_of_characteristics_polygon	Lista delle caratteristiche da assegnare ai poligoni	Array di stringhe	Obbligatorio	
List_of_points	Lista dei punti	Array di oggetti di tipo Points		
List_of_polygons	Lista dei poligoni	Array di oggetti di tipo Polygon		

Polygon				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
Polygon_characteristics	Lista dei valori di tutte le caratteristiche assegnate a questo poligono	Array	Opzionale	
List_of_vertices	Lista dei punti vertici di questo poligono	Array di interi che rappresentano l'indice del punto in list_of_points	Obbligatorio	Devono essere almeno tre

Point				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
	Coordinate del punto	Array di tre numeri	Obbligatorio	Nessun nome
	Lista dei valori di tutte le caratteristiche assegnate a questo punto	Array	Obbligatorio	Nessun nome

B.2.5 Sparsegrid

Sparsegrid				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
Resolution	Lunghezza dei lati di un voxel, in metri	Array di tre numeri	Obbligatorio	
Size	Numero di voxel presenti su ogni asse	Array di tre numeri	Obbligatorio	
List_of_characteristics	Lista delle caratteristiche da assegnare ad ogni voxel	Array di stringhe	Obbligatorio	
List_of_voxels	Lista dei voxel che compongono la mappa	Array di oggetti di tipo Sparsegrid_voxel	Obbligatorio	

Sparsegrid_voxel				
Nome campo	Descrizione	Tipo di dato	Obbligatorietà	Note
	Coordinate del voxel nella griglia	Array di tre interi	Obbligatorio	Nessun nome
	Lista dei valori di tutte le caratteristiche assegnate a questo voxel	Array	Obbligatorio	Non ha nome

Bibliografia

[1] B. Wang e S. Qin, «Multi-robot environment exploration based on label maps building via recognition of frontiers,» *2014 International Conference on Multisensor Fusion and Information Integration for Intelligent Systems (MFI),* pp. 1-9, 2014.

[2] D. Claes, F. Oliehoek, H. Baier e K. Tuyls, «Decentralised Online Planning for Multi-Robot Warehouse,» in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, San Paulo, International Foundation for Autonomous Agents and Multiagent Systems, 2017, pp. 492-500.

[3] «IEEE Standard for Robot Map Data Representation for Navigation,» *1873-2015 IEEE Standard for Robot Map Data Representation for Navigation,* pp. 1-54, 2015.

[4] «IEEE/RAS Standard for Robot Map Data Representation for Navigation,» [Online]. Available: https://bit.ly/2BEMsa4. [Consultato in Febbraio 2020].

[5] OGC, «Geography Markup Language (GML) 2.0,» [Online]. Available: https://bit.ly/3ivvbkj. [Consultato in Giugno 2020].

[6] OGC, «GML Introduction,» [Online]. Available: https://www.w3.org/Mobile/posdep/GMLIntroduction.html. [Consultato in Giugno 2020].

[7] OGC, «CityGML,» [Online]. Available: https://www.3dcitydb.org/3dcitydb/citygml/. [Consultato in Giugno 2020].

[8] OGC, «OGC IndoorGML,» [Online]. Available: http://docs.opengeospatial.org/is/14-005r5/14-005r5.html. [Consultato in Giugno 2020].

[9] web 3D consortium, «What is X3D?,» [Online]. Available: https://www.web3d.org/x3d/what-x3d. [Consultato in Giugno 2020].

[10] web 3D consortium, «X3D: Architecture and base components,» [Online]. Available: https://bit.ly/3grMSPH. [Consultato in Giugno 2020].

[11] Google, «Google Maps,» [Online]. Available: https://www.google.it/maps. [Consultato in Giugno 2020].

[12] Firaxis Games, «Civilization VI,» [Online]. Available: https://civilization.com/it-IT/. [Consultato in Giugno 2020].

[13] Atm, «Metro Milano,» [Online]. Available: https://www.atm.it/it/Pagine/default.aspx. [Consultato in Giugno 2020].

[14] «XML,» [Online]. Available: https://www.w3.org/XML/. [Consultato in Giugno 2020].

[15] «JSON,» [Online]. Available: https://www.json.org/json-en.html. [Consultato in Giugno 2020].

[16] «OGC Standards,» [Online]. Available: https://www.ogc.org/standards/. [Consultato in Giugno 2020].

[17] OGC, «Geography Markup Language,» [Online]. Available: http://www.ogc.org/standards/gml. [Consultato in Giugno 2020].

[18] OGC, «InfraGML,» [Online]. Available: https://www.ogc.org/standards/infragml. [Consultato in Giugno 2020].

[19] OGC, «GeoSPARQL,» [Online]. Available: https://www.ogc.org/standards/geosparql. [Consultato in Giugno 2020].

[20] G. Gröger e L. Plumer, «CityGML - Interoperable semantic 3D city model,» *ISPRS Journal of Photogrammetry and Remote Sensing,* vol. 71, pp. 12-33, 2012.

[21] Gröger e all., «OGC City Geography Markup Language (CityGML) Encoding Standard, Version 2.0.0. Open Geospatial Consortium,» 2012.

[22] OGC, «Berlin 3D Portal provides CityGML data for 550,000 buildings,» [Online]. Available: https://www.ogc.org/blog/2196. [Consultato in Giugno 2020].

[23] «Semantic 3D City Model of Berlin,» [Online]. Available: https://www.3dcitydb.org/3dcitydb/visualizationberlin/. [Consultato in Giugno 2020].

[24] «3D City Database for CityGML,» [Online]. Available: https://bit.ly/38w26QV. [Consultato in Giugno 2020].

[25] K.-J. Li, G. Conti, E. Konstantinidis, S. Zlatanova e P. Bamidis, «OGC IndoorGML: A Standard Approach for Indoor Maps,» in *Geographical and Fingerprinting Data to Create Systems for Indoor Positioning and Indoor/Outdoor Navigation*, Cambridge, Academic Press, 2019, pp. 187-207.

[26] T. Becker, C. Nagel e T. H. Kolbe, «A Multilayered Space-Event Model for Navigation in Indoor Spaces,» in *3D Geo-Information Sciences. Lecture Notes in Geoinformation and Cartography*, Berlino, Springer, 2009, pp. 61-77.

[27] N. Nurseitov, M. Paulson e R. Reynolds, «Comparison of JSON and XML Data Interchange Formats: A Case Study,» *Comparison of JSON and XML data interchange formats: A case study. 22nd International Conference on Computer Applications in Industry and Engineering,* pp. 157-162, 2009.

[28] «Ajv: Another JSON Schema Validator,» [Online]. Available: https://bit.ly/2ZBLYJT. [Consultato in Febbraio 2020].

[29] «Ajv: $data reference,» [Online]. Available: https://bit.ly/31L9LJT. [Consultato in Febbraio 2020].

[30] J. Yan-Bin, «Quaternions and Rotations,» Settembre 2013. [Online]. Available: https://stanford.io/2NZxG01. [Consultato in Giugno 2020].

[31] S. Sivčev, M. Rossi, E. Omerdic, J. Coleman, G. Dooly e D. Toal, «Collision Detection for Underwater ROV Manipulator Systems,» *Sensors,* vol. 18, 2018.

[32] A. Hornung, K. Wurm, M. Bennewitz, C. Stachniss e W. Burgard, «OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees,» *Auton Robot,* vol. 34, pp. 189-206, 2013.

[33] M. Botsch, L. Kobbelt e M. Pauly, Polygon Mesh Processing, New York: Taylor & Francis Inc, 2010.

[34] Python, «Python Dictionary,» [Online]. Available: https://docs.python.org/3/tutorial/datastructures.html#dictionaries. [Consultato in Giugno 2020].

[35] «Pandas,» [Online]. Available: https://pandas.pydata.org/. [Consultato in Giugno 2020].

[36] F. Lamonaca, «JSON Schema,» 02 2020. [Online]. Available: https://github.com/effelam/JSON_3dMaps. [Consultato in Giugno 2020].

[37] University of Freiburg, «OctoMap 3D Scan dataset,» [Online]. Available: https://bit.ly/2VNyA3Y. [Consultato in Aprile 2020].

[38] D. Munoz, J. A. Bagnell, N. Vandapel e M. Hebert, «Contextual Classification with Functional Max-Margin Markov Networks,» *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR),* Giugno 2009.

[39] T. Hackel, N. Savinov, L. Ladicky, J. D. Wegner, K. Schindler e M. Pollefeys, «SEMANTIC3D.NET: A new large-scale point cloud classification benchmark,» *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences,* Vol. %1 di %2IV-1-W1, pp. 91-98, 2017.

[40] «Ply Files: an ASCII Polygon Format,» [Online]. Available: https://people.sc.fsu.edu/~jburkardt/data/ply/ply.html. [Consultato in Febbraio 2020].

[41] L. Morreale, A. Romanoni e M. Matteucci, «Dense 3D Visual Mapping via Semantic Simplification,» *2019 International Conference on Robotics and Automation (ICRA),* pp. 6891-6897, 2019.

[42] A. Romanoni, A. Delaunoy, M. Pollefeys e Matteucci Matteo, «Automatic 3D Reconstruction of Manifold Meshes via Delaunay Triangulation and Mesh Sweeping,» in *WACV 2016: IEEE Winter Conference on Applications of Computer Vision*, Lake Placid, NY, 2016.

[43] A. Romanoni, D. Fiorenti e M. Matteucci, «Mesh-based 3D textured urban mapping,» *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS),* pp. 3460-3466, 2017.

[44] C. Strecha, W. Hansen, L. Van Gool, P. Fua e U. Thoennessen, «On benchmarking camera calibration and multi-view stereo for high resolution imagery,» in *IEEE Conference on Computer Vision and Pattern Recognition*, Anchorage, AK, 2008.

[45] A. Geiger, P. Lenz e R. Urtasun, «Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite,» *Conference on Computer Vision and Pattern Recognition (CVPR),* p. 3354–3361, 2012.

[46] «Object File Format (.off),» [Online]. Available: https://shape.cs.princeton.edu/benchmark/documentation/off_format.html. [Consultato in Giugno 2020].