UNIVERSITÁ POLITECNICA DELLE MARCHE

FACOLTÁ DI INGEGNERIA

Corso di Laurea Triennale in
Ingegneria Informatica e dell'Automazione

Progettazione e Sviluppo di un modulo software in ROS per gestione di camera eventi

Design and Development of a module in the ROS ecosystem to manage event based cameras

Relatore:
Dott. Mancini Adriano

Laureando:
Zonneveld Giacomo

Correlatore:
Galdelli Alessandro

Anno Accademico 2019-2020

Indice

Elenco delle figure **4**

1 Introduzione **5**
1.1 Motivazioni ed obiettivi 6
1.2 Struttura della Tesi 7

2 Visual Object Tracking **9**
2.1 VOT con camere frame-based 10
2.2 VOT con camere ad eventi 13

3 Strumenti e Metodi **17**
3.1 Strumenti Utilizzati 17
3.1.1 Sensore DAVIS346 17
3.1.2 Software DV 18
3.1.3 ROS e catkin 19
3.1.4 Repositories 20
3.1.5 Python e librerie 21
3.2 Analisi 22
3.2.1 Acquisizione dati e analisi della loro struttura 22

4 Sviluppo Modulo Software **27**
4.1 Progettazione 27
4.2 Implementazione 30
4.2.1 converter1.py 30
4.2.2 converter2.py 33
4.2.3 converter3.py 33
4.3 Analisi delle prestazioni 34
4.4 Applicazione algoritmo EKLT 35

5 Conclusioni e Sviluppi Futuri **37**
5.1 Conclusioni 37
5.2 Sviluppi futuri 38

Appendice A Termini tecnici **39**
A.1 Dynamic Range 39
A.2 Inertial Measurement unit IMU 39
A.3 Frame per secondo (fps) e immagine di intensità 39
A.4 Rilevamento delle features 40

Bibliografia **41**

Capitolo 1

Introduzione

Al giorno d'oggi i bisogni della Computer Vision e della Robotica stanno spingendo verso la produzione di nuovi sensori in grado di affrontare sfide complesse come ad esempio la stima dei movimenti ad alta velocità e il tracciamento di oggetti in movimento in scene con illuminazione scarsa o eccessiva.

I sensori attualmente utilizzati sono sensori di tipo ***frame-based***. Il loro funzionamento consiste in una continua e costante generazione di **immagini di intensità** (Appendice A.3) (o *frame*), cioè matrici di dati i cui valori rappresentano la scala di un determinato colore e quindi la sua intensità. Ciascuna di queste immagini corrisponde alla digitalizzazione della scena ripresa. Il numero di immagini prodotte in un certo intervallo di tempo dipende sia dalla velocità di acquisizione del sensore, sia dal tempo di esposizione. La velocità di acquisizione assume un valore costante e dipendente dalla qualità del sensore stesso, mentre il tempo di esposizione dipende dalla luminosità della scena (scene poco illuminate richiedono tempi di esposizione più lunghi rispetto a scene ben illuminate).
Queste camere tendono ad essere poco efficaci nella raccolta di informazioni in scene in cui si hanno movimenti ad alta velocità, del sensore stesso o di oggetti nella scena circostante, e in cui il *range dinamico* è elevato (Appendice A.1). Di fatti, come già detto, la raccolta è condizionata dalla velocità di acquisizione del sensore che risulta essere limitata ad un valore fisso espresso in frame al secondo (Appendice A.3). Quindi, se un oggetto si muove ad una velocità maggiore rispetto a quella di acquisizione, si avrà un'inevitabile perdita di informazioni dovuta al tempo cieco tra un'immagine e l'altra e al possibile effetto *blur* (o sfocatura). Altre conseguenze potrebbero riguardare la sfocatura della scena La velocità fissa di acquisizione determina una raccolta costante di informazioni dalla scena indipendentemente da ciò che accade. Questo implica che, anche in assenza di scenari interessanti come il movimento di un oggetto, vengano comunque prodotti dati non necessari con conseguente spreco di memoria. Questi dati, processati poi dagli algoritmi, determinano inevitabilmente l'aumento del tempo di elaborazione. Le performance di questo sensore sono anche fortemente condizionate dalla luminosità della scena dalla quale verranno prodotte immagini "bruciate",

o sovraesposte, in corrispondenza di picchi elevati di luminosità e poco nitide in corrispondenza di condizioni di scarso illuminamento.

L'innovativo sensore utilizzato per lo studio, oggetto di questa tesi, rientra invece nella categoria delle **camere ad eventi**. La sua introduzione nel campo della Computer Vision consente, a partire dalle sue caratteristiche, di colmare le mancanze delle camere *frame-based*. La novità di questi sensori consiste nell'introduzione di un nuovo tipo di dato, l'**evento**, il quale rappresenta il cambiamento di luminosità in un determinato punto nella scena il quale rappresenta il cambiamento di luminosità in un determinato punto *(x,y)* nella scena. Questo cambiamento di luminosità corrisponde al verificarsi di un movimento nel punto stesso. In questo modo il sensore è in grado di raccogliere i dati in maniera asincrona, cioè non costante, e questo fa sì che, in assenza di movimento, non vengano raccolti dati provenienti dagli elementi statici della scena. Come evidente, tutti i problemi legati allo spreco di memoria e al tempo di elaborazione vengono risolti, difatti il volume di dati è direttamente proporzionale alla quantità di movimenti nella scena. La registrazione del cambiamento di luminosità consente di superare le difficoltà legate all'illuminazione della scena e in più, avendo il sensore un tempo di latenza pressoché nullo, si avrà una reattività molto elevata favorendo così l'acquisizione di oggetti in movimento ad alta velocità.

1.1 Motivazioni ed obiettivi

Il presente lavoro di tesi si propone di valutare il funzionamento del sensore mettendolo in azione in vari tipi di ambienti, in posizione statica e dinamica, di analizzarne il flusso dei dati prodotto così da poter comprendere come interagire con gli stessi e infine di effettuare dei test con un algoritmo di tracciamento degli oggetti creato ad hoc per tale tipo di sensori. A tale scopo si è reso necessario produrre un modulo software per poter effettuare la conversione dal file prodotto dal sensore, contenente i dati raccolti, ad un file utilizzabile in ambiente *Robot Operating System* (*ROS*) nel quale è eseguito l'algoritmo.
Il codice sviluppato consente quindi di sottoporre a vari algoritmi di analisi i dati raccolti dal sensore. Sarà quindi possibile semplificare la fase di testing di algoritmi già presenti e non ancora sviluppati e consentire al sensore di diventare un punto fondamentale nel futuro della *Computer Vision* andando a soddisfare le sempre crescenti richieste di questo campo.

1.2 Struttura della Tesi

La parte essenziale del progetto di tesi consiste nello sviluppo di un modulo software in ambiente *ROS* che consenta la conversione dei dati provenienti da una camera ad eventi. Inizialmente si introduce il concetto di *Visual Object Tracking* (*VOT*) e se ne mostrano alcune applicazioni sia con camere standard sia con camere ad eventi. In seguito si introducono tutti gli strumenti utilizzati per lo sviluppo del software e i metodi evidenziando gli textitstep che hanno portato alla produzione dello stesso. Successivamente viene posto rilievo agli step che hanno portato alla produzione del software stesso consistenti in analisi del sensore, individuazione di un metodo di *Object Tracking* e quindi progettazione e sviluppo del programma. Infine si mostrano i risultati e i possibili sviluppi futuri del progetto. A chiudere si inserisce un'appendice in cui si riporta la spiegazione di alcuni termini tecnici usati con frequenza.
Quindi la struttura dell'elaborato è individuabile nei seguenti punti:

- esposizione di metodi di *Object Tracking* con *sensori standard* e *sensori ad eventi*;
- introduzione degli strumenti e dei metodi;
- analisi e sviluppo del modulo software;
- conclusioni;
- appendice.

Capitolo 2

Visual Object Tracking

Il tracciamento visivo degli oggetti è un campo della *Computer Vision* che, grazie ai suoi molteplici usi in diverse applicazioni reali, ha reso possibile il raggiungimento di obiettivi, un tempo impensabili, come ad esempio la guida autonoma dei veicoli [1], la robotica medica e industriale, il riconoscimento facciale, i sistemi per le diagnosi mediche, l'assistenza alle persone non vedenti ed altro.

La *Visual Object Tracking* è il processo che consiste nell'identificazione di un oggetto o di una regione di interesse in una sequenza di dati. A tale scopo si è resa necessaria l'implementazione di metodi di rilevamento e selezione delle *features*(fare riferimento ad Appendice A.4), cioè di metodi di raccolta di informazioni provenienti da un'immagine che portino al raggiungimento dell'obiettivo di tracciare il movimento degli oggetti in una data scena. Grazie al loro tracciamento all'interno delle sequenze di immagini è possibile riconoscere gli oggetti, definiti da una serie di punti, linee, contorni o oggetti, e tracciarne il loro movimento.
Di seguito vengono brevemente mostrati alcuni approcci alla problematica con *sensori standard* e con *camere ad eventi*.

2.1 VOT con camere frame-based

I video, che sono sequenze di immagini, forniscono importanti informazioni per comprendere il contenuto delle scene rappresentate, tuttavia l'estrazione di oggetti continua ad essere una grande sfida per la Computer Vision.
Alcuni algoritmi di *Object Tracking*, basati sul tracciamento per rilevamento degli oggetti, stanno guadagnando grande popolarità grazie alla loro efficacia in termini di prestazioni e grazie alla semplicità nell'effettuare le classificazioni degli oggetti. Questo tipo di algoritmi considera il tracking come processo di rilevamento dell'oggetto in fotogrammi di immagini consecutive effettuato addestrando un classificatore binario alla distinzione dell'oggetto dallo sfondo. In genere un classificatore richiede dati di addestramento per funzionare correttamente. Questi dati vengono generati in real-time [2] durante il tracciamento e il classificatore si aggiorna così da migliorare le prestazioni di tracciamento. A questo punto si entra nella fase di generazione ed etichettatura dei campioni come trattato in [3] dove si considera come esempio positivo la posizione stimata dell'oggetto in ogni immagine e si considerano come esempi negativi le posizioni attorno all'oggetto. A questo punto viene aggiornato il classificatore. Durante il tracciamento, il classificatore trova la posizione dell'oggetto massimizzando il punteggio di classificazione in una regione locale, normalmente intorno alla posizione trovata nella cornice precedente, utilizzando il metodo della finestra scorrevole. La figura 2.1 mostra il processo di aggiornamento e tracciamento.

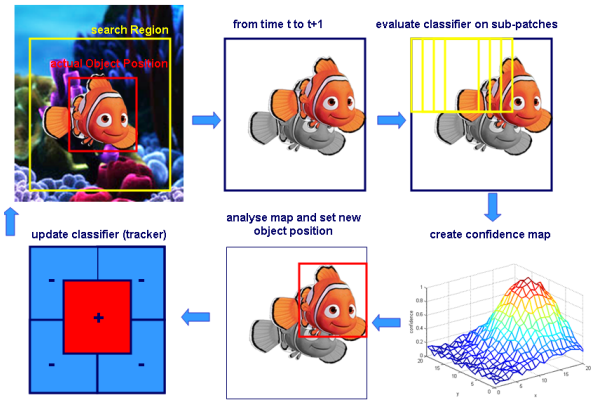

Figura 2.1: Esempio di aggiornamento classificatore e tracciamento.
Fonte: Grabner et al [3]

Molti metodi [4],[5],[6] fanno uso di *reti neurali convoluzionali* (**CNN**) addestrate che consentono il riconoscimento degli oggetti effettuando un'estrazione delle caratteristiche e facendo confronti con modelli addestrati con oggetti-esempio. Essendo addestrati in maniera supervisionata, i modelli richiedono un gran numero di etichette che definiscano i vari possibili oggetti. Perciò sono necessari modelli diversi per ogni campo di applicazione. Ad esempio nell'am-

bito della guida autonoma(figura 2.2) si hanno modelli contenenti etichette che rappresentano persone, macchine, semafori, strisce pedonali...

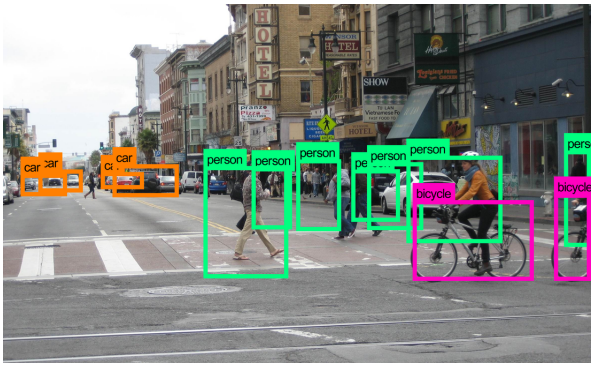

Figura 2.2: Esempio di riconoscimento oggetti con CNN.
Fonte: https://industrywired.com/the-era-of-computer-vision-is-here/

Da questo discende che per avere un'applicazione efficiente è necessario avere modelli di dimensioni dell'ordine di decine di terabyte.

Altri algoritmi di tracciamento invece non richiedono alcun tipo di addestramento, favorendo quindi una semplice applicazione del metodo proposto senza doversi munire dei modelli addestrati. Molti metodi si basano sul processo di *image registration* [7]. Nella visione artificiale, gli insiemi di dati acquisiti su un oggetto in movimento sono in sistemi di coordinate diversi da immagine a immagine. Tramite il processo di *image registration* è possibile trasformare differenti insiemi di dati, presenti in diversi insiemi di coordinate, in un unico sistema di coordinate. Ad esempio questi set di dati possono rappresentare lo stesso oggetto visto da prospettive diverse. È quindi possibile evidenziare ogni cambiamento in termini di dimensioni, forma o posizione degli oggetti così da consentirne il tracciamento. La figura 2.3 mostra il risultato che si ottiene applicando tale tecnica. Le tecniche che implementano la image registration sono però costose.

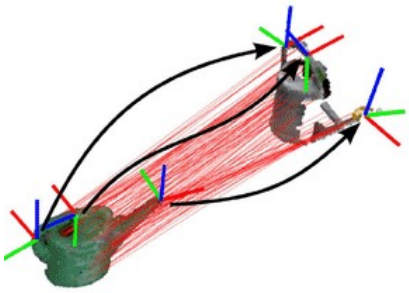

Figura 2.3: Applicazione image registration
Fonte: Stückler et al [8]

Un altro metodo è quello di *Kanade-Lucas-Tomasi*[9] (**KLT**) che seleziona e memorizza le *features* ottimali per il tracciamento, trascurando invece le *features* non funzionali all'obiettivo. È stato implementato con l'obiettivo di superare il problema del costo computazionale della *image registration*. Questo metodo si rivela efficiente se gli oggetti nella scena si muovono solo per piccole dimensioni della finestra temporale di osservazione. Il *KLT* si rivela come tecnica ideale per il *tracking delle features* da un'immagine all'altra, lavorando direttamente sulle informazioni dell'intensità spaziale. Il risultato ottenuto dall'applicazione di questo metodo può essere visto nell'esempio rappresentato in figura 2.4. Una volta individuate le caratteristiche nella prima immagine, queste verranno seguite nei frame successivi, ottenendo quindi un tracciato come in figura.

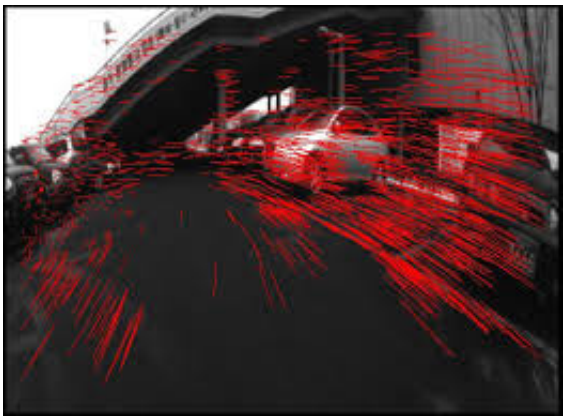

Figura 2.4: Esempio di tracking KLT
Fonte: http://www.cs.cmu.edu/ 16385/s17/Slides/15.1_Tracking__KLT.pdf

Infine si pone l'attenzione su un algoritmo di *spostamento medio basato su una funzione di trasformazione in scala invariante* (*Scale-invariant feature transform*, **SIFT**) che consente l'inseguimento degli oggetti in scenari reali eseguendo due misure in contemporanea [10]. Le *features*, rilevate e pesate sulla funzione, vengono utilizzate per far corrispondere la regione di interesse, contenente l'oggetto, tra le varie immagini. Nel frattempo lo *spostamento medio* (o *mean shift*) viene utilizzato per trovare le corrispondenze analizzando istogrammi di colore. Da queste due misure si estraggono le distribuzioni di probabilità che vengono valutate al fine di effettuare una stima della probabilità, che sia la più elevata possibile, di trovare regioni simili tra loro. In questo modo, come asserito dagli ideatori, la stima parallela consente di avere prestazioni comunque buone di tracking anche se una delle due misure diventa instabile. Nella figura 2.5 possiamo vedere i risultati di tracciamento ottenuti con lo spostamento medio in prima riga, con la corrispondenza delle caratteristiche SIFT in seconda riga e con entrambe le tecniche combinate in ultima riga.

Come accennato nell'introduzione, questi metodi di tracciamento si rivelano molto efficaci in situazioni più o meno controllate, mentre rivelano delle carenze

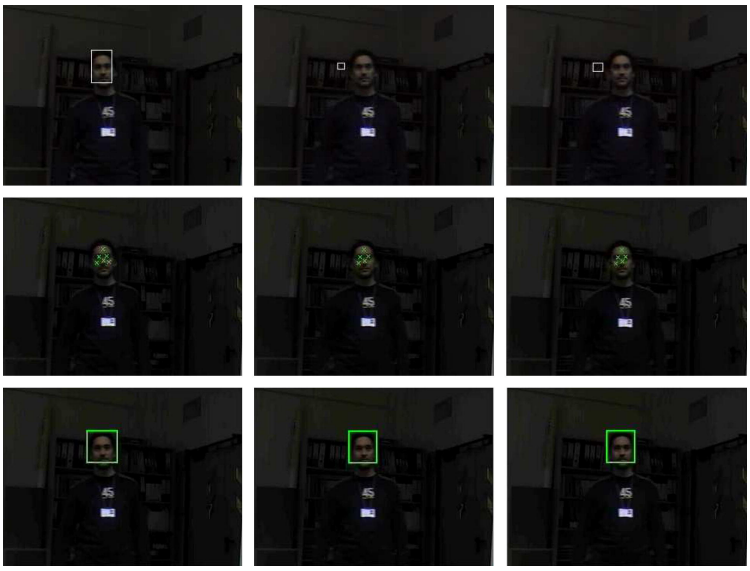

Figura 2.5: Esempio di mean shift, SIFT e loro combinazione
Fonte: Zhou et al [10]

in condizioni di scarsa o elevata illuminazione e nel caso in cui l'oggetto si muova a grande velocità. Difatti non è possibile effettuare tracciamenti nel tempo cieco tra un'immagine e l'altra e inoltre risultano costosi in quanto elaborano le informazioni da tutti i pixel anche se nella scena non sono presenti movimenti. Di contro le camere ad eventi acquisiscono solo informazioni rilevanti per il tracciamento e rispondono in modo asincrono riempendo così il tempo cieco tra un'immagine e l'altra.

2.2 VOT con camere ad eventi

Con l'avvento delle *camere ad eventi* stanno emergendo nuovi lavori che, avvalendosi degli eventi, propongono soluzioni ottimali ai sopra citati problemi legati all'uso delle *camere standard*. Ogni **evento** rappresenta il cambiamento di luminosità di un pixel ad un instante di tempo e viene raccolto se il cambiamento supera una soglia di base. Tale soglia consente di escludere possibili disturbi. Ogni k evento può essere indicato tramite la seguente formula $e_k = (x_k, y_k, t_k, p_k)$. Come si evince dalla formula, ogni evento sarà caratterizzato dalla posizione in coordinate rappresentate da una coppia di valori interi *(x,y)* del pixel attivato, dall'istante di tempo t in cui si verifica con formato *timestamp Unix* in microsecondi e da un bit p per la polarità di tipo booleano che indica se la luminosità del pixel è aumentata (*true*) o se è diminuita (*false*).

Un metodo molto interessante è stato proposto in merito al *processo di localizzazione e mappatura simultanea* (*Simultaneous localization and mapping*,

SLAM) di un drone che si muove in un ambiente sconosciuto e che risulta capace di orientarsi all'interno della mappa [11]. Sul drone sono stati installati una camera ad eventi, una camera standard e un'unità di misura inerziale, o *IMU* (vedi Appendice A.2). Per verificare i risultati dell'algoritmo implementato, il drone è stato fatto volare, a basse ed elevate velocità, all'interno della stessa stanza sottoposta a scarsa e buona illuminazione. Per la *feature tracking* è stato utilizzato il metodo *FAST*, metodo che individua gli angoli nella scena, applicato sia alle *immagini di intensità* prodotte dalla *camera standard*, sia a quelle virtuali ottenute dalla *camera ad eventi*. Tramite il metodo *KLT*, le *features* vengono tracciate in maniera indipendente. Inizialmente vengono considerate tutte come *features candidate* e vengono seguite per diverse immagini. Quando una *feature* può essere triangolata efficacemente, questa viene convertita in *feature persistente* e viene associata al punto di riferimento in 3D sulla mappa.

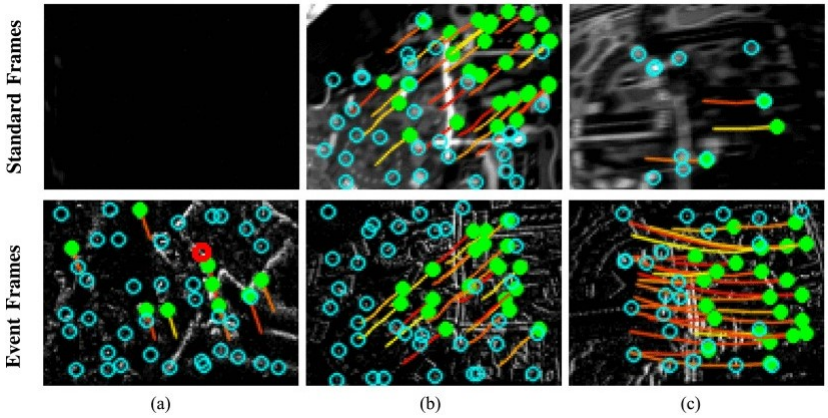

Figura 2.6: SLAM. Fonte: R. Vidal et al [11]

Nella figura 2.6 sono mostrati esempi di *feature tracking* in determinate condizioni ed estratte da immagini di intensità (Standard Frames) oppure da immagini virtuali create con gli eventi (Event Frames). La colonna (a) rappresenta il tracciamento dell'acquisizione fatta con ambiente scarsamente illuminato, la colonna (b) con ambiente ben illuminato e a velocità di volo moderata e la colonna (c) con la sfocatura dovuta al volo con velocità elevate. I punti verdi rappresentano le *features persistenti*, mentre quelli blu corrispondono alle *features candidate*. Dalla figura è possibile notare quanto sia più efficace l'uso degli eventi per il tracking.

Un altro metodo che estende il *tracking delle features* agli eventi è il metodo **EKLT** in cui il tracciamento delle *features* è basato sia sulle immagini di intensità che sugli eventi estendendo il metodo *KLT*. Questo permette di costruire tracciati con elevata risoluzione temporale sfruttando da un lato la rappresentazione fotometrica delle *immagini di intensità* che non dipende dalla direzione del movimento, e dall'altro l'alta risoluzione temporale data dalla raccolta asincrona degli eventi. Se *KLT* effettua una stima del flusso ottico, basandosi sul presupposto che il flusso ottico sia constante in un intorno del pixel considerato, allo stesso

modo il metodo *EKLT* applica questa stima sia ai frames che agli eventi, andando a tracciare le *features* su frames sincronizzati con gli eventi. Come asserito dagli ideatori, questa scelta consente di avere prestazioni ottimali.

Capitolo 3

Strumenti e Metodi

3.1 Strumenti Utilizzati

3.1.1 Sensore DAVIS346

La *camera ad eventi* utilizzata è la *DAVIS346* prodotta dall'azienda *IniVation*. Il sensore ha dimensioni 346x260 pixels, alimentato tramite porta USB 3.0 micro in modo tale da consentire il trasferimento dello stream dei dati ad alte velocità. Tra le caratteristiche più rilevanti troviamo un *High Dynamic Range*(Appendice A.1) pari a 120 dB contro i 55-60 dB delle camere standard, una bassa latenza (1 microsecondo) e una capacità di raccolta eventi pari a 12 milioni di eventi al secondo.

Figura 3.1: Event-camera DAVIS346
Fonte: http://rpg.ifi.uzh.ch/CED.html

Come già accennato nell'introduzione, questo sensore è in grado di raccogliere dati in modo asincrono consentendo quindi un elevato risparmio di dati da salvare e da elaborare. Se nella *camere standard frame-based* vengono prodotti dati sotto forma di una serie di immagini di intensità (o *frames*), in scala di grigi o a colori, le *Event Camera* producono invece dati principalmente sotto forma di eventi. Essendo l'output un flusso di eventi asincroni, le *camere ad eventi* riescono a

percepire i movimenti con una latenza esigua, ma non forniscono sufficienti informazioni sull'ambiente. Al contrario le *camere frame-based*, basate sulle immagini di intensità, forniscono una misurazione dell'intensità per ogni pixel, ma con una latenza non trascurabile (10-20 ms) [12]. Quindi questi due tipi di informazioni risultano essere complementari tra loro e per questo motivo la *DAVIS346* oltre a produrre gli eventi, fornisce anche le immagini di intensità. Oltre a questi due tipi di dati, che saranno utilizzati nello sviluppo del progetto, vengono prodotte anche informazioni riguardanti la dinamica del sensore contenute nel tipo di dato *Inertial Measurement unit* (*IMU*) e informazioni riguardanti l'attivazione dei pixels contenute in *Triggers*.

3.1.2 Software DV

Per la gestione del sensore la casa produttrice fornisce il software *DV* che consente, tramite moduli, di registrare le acquisizioni e riprodurle. Una volta collegato al calcolatore, il sensore verrà riconosciuto dal software che consentirà immediatamente di iniziare a registrare la scena. Oltre ai moduli standard per la registrazione e la riproduzione, ce ne sono altri che permettono la manipolazione dei dati raccolti. Ad esempio è possibile applicare filtri agli eventi con l'obiettivo di diminuire il numero di disturbi (falsi eventi) oppure migliorare il contrasto dei frames. Questo software è stato installato in ambiente *Windows 10* per preferenza personale, anche se risulta essere disponibile per *Linux* e *MacOS*.

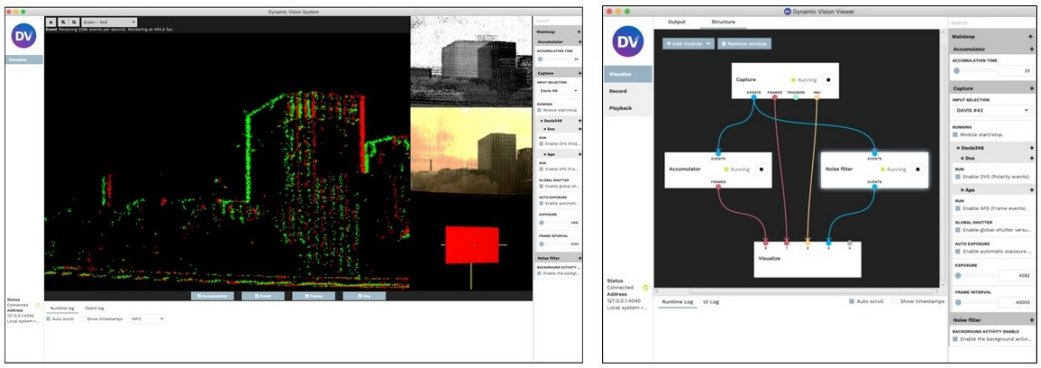

Figura 3.2: Software DV
Fonti: https://inivation.com/inivation-announces-launch-of-dv-software-to-accelerate-development-of-event-based-vision-applications/
https://inivation.com/inivation-announces-launch-of-dv-software-1-0/

3.1.3 ROS e catkin

ROS è un insieme di librerie software e strumenti open-source che aiutano a costruire applicazioni per robot. Vengono forniti algoritmi, drivers e tools che consentono il controllo a basso livello dei dispositivi collegati al sistema[13]. Una delle caratteristiche più importanti di questo sistema, punto chiave dello sviluppo del progetto, è la realizzazione della comunicazione tra processi tramite messaggi. Questo tipo di comunicazione consente la trasmissione di informazioni tra nodi in modo asincrono implementando l'incapsulamento dei dati. I messaggi sono in formato standard fornito da *ROS* stesso, oppure possono essere creati ad hoc per tipi di dati particolari non ancora supportati [14]. Ogni tipo di messaggio viene identificato univocamente dal **topic** che tramite una stringa consente di assegnare un'identità al tipo di dato da inviare. In questo modo, se ad esempio sono presenti più nodi che si aspettano tipi di dati diversi che vengono prodotti contemporaneamente, grazie ai *topics*, ogni nodo sarà in grado di raccogliere solo i dati che hanno quel identificativo richiesto. In questo modo viene implementata la comunicazione asincrona tra i vari nodi. I file proprietari di questo sistema hanno estensione **bag**. Al loro interno si andrà a salvare o leggere i dati raccolti sotto forma di messaggi. Per rendere operativo *ROS* è necessario effettuare un'installazione le cui linee guida sono ben dettagliate nella sua documentazione [15]. Un'importante menzione a riguardo va fatta per *catkin*, che è il compilatore ufficiale di *ROS* e viene utilizzato per la gestione di tutti i pacchetti di *ROS*, tra cui quelli creati dal programmatore per lo sviluppo di applicativi software in grado di assolvere determinate richieste. La versione di *ROS* utilizzata è la versione *Kinetic Kame*, rilasciata nel 2016 e sulla quale si basano gli algoritmi di supporto al sensore e di tracciamento degli oggetti. Attualmente *ROS* è in grado di funzionare esclusivamente in ambiente *Linux*.

Figura 3.3: Logo ROS. Fonte: https://www.ros.org/

3.1.3.1 Windows, Linux e Oracle VirtualMachine VirtualBox

Inizialmente è stato scelto come ambiente di lavoro *Windows 10*, all'interno del quale è stato installato il software *DV*. Essendo però tutti gli algoritmi inerenti la *Visual Object Tracking* sviluppati e funzionanti in *ROS*, scegliere come ambiente *Linux* è stato un passaggio obbligato. Al fine di evitare inutili complicazioni dovuti all'installazione di *Linux* sulla macchina, si è deciso di utilizzare una macchina virtuale così da semplificare tutta la fase di setup. La versione di *ROS* richiesta è compatibile con *Linux Xenial 16.04*. Il software utilizzato per la creazione della macchina virtuale è *Oracle VirtualMachine VirtualBox* in versione 6.1.8 r137981.

Per questo ambiente virtuale sono state messe a disposizione le seguenti risorse: 8GB di memoria centrale, 4 processori, 128 MB di memoria video e 30 GB di memoria secondaria. La macchina fisica è invece caratterizzata da processore *Intel Core i7* di nona generazione, 16 GB di memoria centrale, 4 GB di memoria dedicata e 512 GB di memoria secondaria allo stato solido.

3.1.4 Repositories

In tutta la fase di studio, sia nella parte di analisi che nella parte di progettazione e implementazione, si è fatto uso di alcuni repository contenenti codice utilizzabile per vari scopi, tutti pubblicati dal Dipartimento di Informatica dell'*Università di Zurigo*, il quale ha anche messo a disposizione dei dataset [16] in vari formati, compreso il formato *bag*, per l'analisi del funzionamento del sensore.

3.1.4.1 rpg_dvs_ros

Il repository *rpg_dvs_ros* [17],[18],[19],[12] mette a disposizione i drivers per il sensore. Il suo uso ha consentito la riproduzione tramite player dedicato dei file *bag* in modo tale da poter effettuare dei test sui file convertiti tramite il codice implementato e di verificarne la corretta esecuzione. Il player si comporta come un nodo di tipo *ROS* che attende messaggi in arrivo. Tramite comando *ROS*: *"rosbag play file.bag"*, i messaggi all'interno del file vengono inviati sul canale di comunicazione aperto dal player che quindi ne riproduce il contenuto. Nella figura 3.4 possiamo vedere il risultato dell'esecuzione del player: abbiamo una riproduzione degli eventi, rappresentati dai pixel blu e rossi che rappresentano il valore della polarità degli eventi (blu = *true*, rosso = *false*). Nell'elenco a discesa sarà possibile decidere il *topic* rilevato dal player da mostrare.

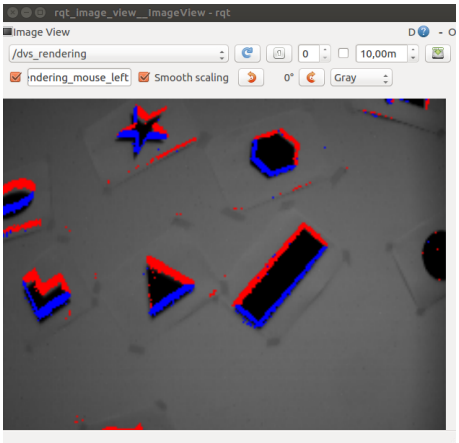

Figura 3.4: Player

3.1.4.2 rpg_davis_simulator

Il repository *rpg_ davis_ simulator* [20],[21] è stato utilizzato come base di partenza per progettare la strategia di raccolta dei dati in un file *bag*. Difatti in questo repository viene implementato un simulatore della camera ad eventi che genera un dataset grazie ad un motore di rendering e lo concretizza in un file *bag* costituito da messaggi riguardanti eventi, informazioni di calibrazione della camera, immagini di intensità e altri tipi di dati non necessari allo scopo dell'oggetto di studio.

3.1.4.3 rpg_eklt

Un altro repository utilizzato è *rpg_ eklt* [22]. All'interno di esso è contenuto il codice che implementa la tecnica *EKLT* [23]. L'algoritmo implementato sfrutta la complementarità tra *eventi* e *immagini di intensità* per il rilevamento delle *features*. Considerando che la frequenza delle immagini è condizionata dalla velocità del sensore, sfruttando gli eventi, sarà possibile tenere traccia degli oggetti anche in quei tempi compresi tra un'immagine e l'altra. Quindi le prime *features* vengono estratte dai *frame*, poi vengono registrate utilizzando solo gli *eventi*. A questo punto l'algoritmo riesce a creare un tracciato delle *features* con un'elevata risoluzione temporale.

3.1.4.4 rpg_feature_tracking_analysis

Questo è l'ultimo repository pubblicato dall'*Università di Zurigo* che si è considerato [24]. Al suo interno viene fornito un metodo per la valutazione del tracciamento delle *features* basato sugli *eventi* ottenuto tramite il metodo *EKLT*. Il codice fornito è stato utilizzato per verificare che l'output prodotto dal modulo software, oggetto di tesi, fosse corretto. Difatti un mancato o esiguo tracciamento dei dati avrebbe denotato un errore in fase di progettazione.

3.1.4.5 simbamford/AedatTools/PyAedatTools

Durante la ricerca di un tool di conversione da file *aedat4* a file *bag* è stato trovato il repository *simbamford/AedatTools/PyAedatTools* [25] che nel codice *ExportRosbag.py* propone un metodo di conversione basato però su file *aedat3*. Nonostante sia incompatibile con formato *aedat4*, a seguito dello studio della tecnica di raccolta, questa è stata implementata nel codice *converter3.py* a scopo di confronto.

3.1.5 Python e librerie

Per l'implementazione dei codici è stato scelto come interprete *Python* per la sua semplicità e per la ricchezza di *API* in merito alla *Computer Vision*. La casa

costruttrice del sensore mette a disposizione l'*API* ***python-dv*** [26] che, tramite i suoi metodi, consente una facile interazione con il sensore. Questa API richiede che l'interprete sia nella versione 3.5. Un'altra API molto importante è ***rospy*** che consente di sfruttare le funzionalità di *ROS* in particolar modo per la generazione di file *bag* e per la comunicazione con scambio di messaggi. Un'*API* largamente utilizzata per lo sviluppo di applicativi operanti nell'ambito della *Computer Vision* è ***OpenCV***. Al suo interno sono contenuti metodi per l'interazione con le immagini di intensità e per l'applicazione, su di esse, di algoritmi di tracciamento degli oggetti. L'ultima API che si menziona è ***CVBridge*** [27] che consente la conversione da immagini di intensità, gestite da *OpenCV*, a messaggi *ROS* e viceversa come rappresentato in figura 3.5. Questa *API* è compatibile con interprete *Python* fino alla versione 2.7. Per questo motivo, spiegato in dettaglio successivamente, si è utilizzata l'*API* ***vision-opencv*** [28] compatibile con *Python 3.x*. In ambiente *Windows 10* si è deciso di utilizzare come *IDE* il software *Pycharm*, mentre in ambiente *Linux*, al fine di risparmiare memoria, si è utilizzato un editor di testo.

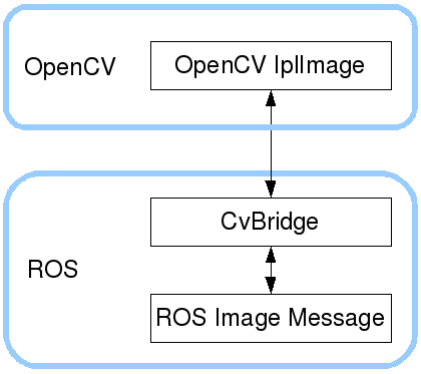

Figura 3.5: Schema di funzionamento CVBridge
Fonte: ROS [27]

3.2 Analisi

3.2.1 Acquisizione dati e analisi della loro struttura

Inizialmente si è deciso di partire dall'analisi del funzionamento del sensore. Sono state fatte delle acquisizioni di prova in ambiente aperto illuminato da luce naturale e in ambiente chiuso illuminato da luce artificiale con il sensore in movimento e in posizione fissa. Dall'analisi delle registrazioni effettuate è stato possibile fare delle importanti considerazioni. Scene illuminate da luce artificiale mostrano la comparsa di disturbi, visibile in figura 3.6 (d), e cioè di falsi eventi, in corrispondenza delle lampade al neon. Probabilmente questo è dovuto al fatto che la luce

viene emessa ad una certa frequenza e la velocità di variazione della luminosità in un dato punto nella lampada risulta essere minore rispetto alla velocità di acquisizione degli eventi da parte del sensore. Questi disturbi però vengono a cessare nel momento in cui il sensore non riprende direttamente le lampade, ad esempio in un ambiente illuminato di luce naturale, vedi figura 3.6 (b).

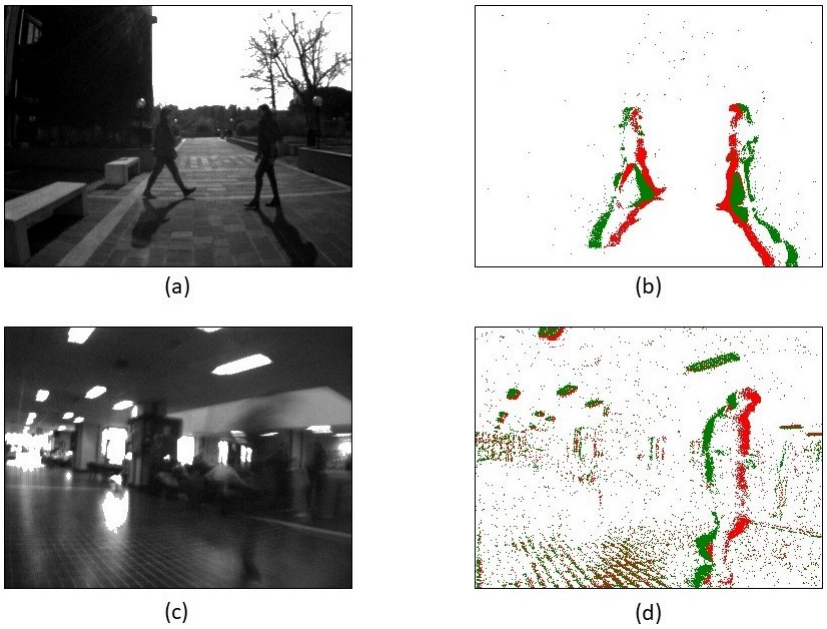

Figura 3.6: Frammenti di acquisizione in formato frame ed eventi

Un'altra osservazione riguarda il volume dei dati. Come asserito da Lichtsteiner et al [19], le dimensioni dei file prodotti risultano essere direttamente proporzionali al movimento della scena. Come conseguenza diretta avremo che file contenenti acquisizioni con sensore in movimento, a parità di tempo, avranno dimensioni molto più elevate rispetto ad acquisizioni con sensore in posizione fissa. Per comprendere questa osservazione, si noti la differenza del numero di pixel colorati tra figura 3.6(b) e figura 3.6(d). Altra conseguenza diretta è che, se nella scena non abbiamo movimenti, il sensore non produce dati ad eccezione di possibili disturbi o rumore, quindi potremo dire che il sensore è in una condizione di stand-by da cui esce autonomamente ogni volta che viene rilevato un cambiamento di luminosità e quindi un movimento di un oggetto nella scena ripresa. Un'ultima osservazione interessante riguarda le acquisizioni con il sensore in movimento. Per effettuarle, il sensore è stato collegato al calcolatore alimentato a batteria. Dopo aver disattivato le impostazioni di risparmio energetico, si è iniziato ad acquisire la scena. Dall'analisi della registrazione è emerso che, necessitando il sensore di un'alimentazione potente, si ha un calo di *frame rate* con conseguente perdita di informazioni sotto forma di *frames*, vedi figura 3.6(c). Gli *eventi* invece non hanno subito nessun rallentamento o perdita, vedi figura

3.6(d). Questo dimostra che, se si riuscirà a sfruttare a pieno l'informazione contenuta negli eventi, non solo avremo minori tempi di elaborazione, avendo un volume minore di dati, ma avremo anche minori necessità in termini energetici. A seguito di un attento studio della documentazione del sensore, è emerso che le macro-categorie di dati accennate in 3.1.1 sono raccolte all'interno di una struttura dati di tipo *Google Flatbuffers*. Questa struttura dati consente la creazione di uno schema custom per lo store dei dati stessi. La figura 3.7 intuitivamente mostra al lettore l'organizzazione generale dei dati.

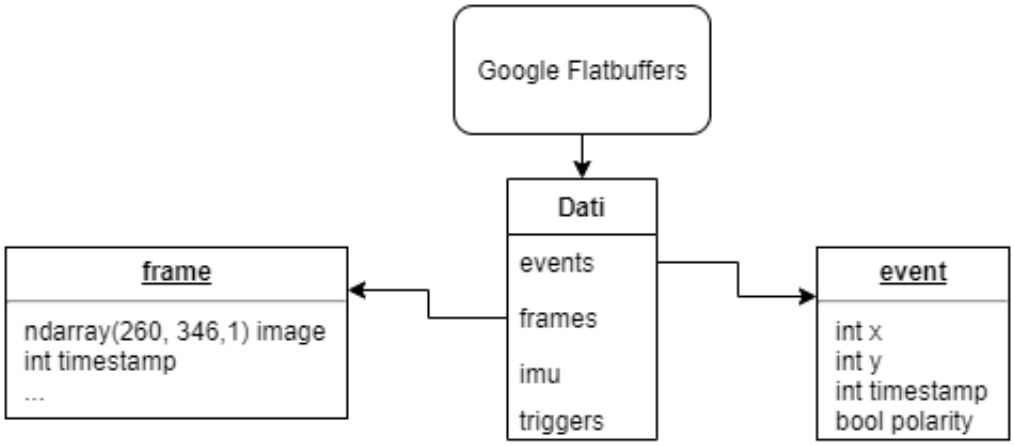

Figura 3.7: Diagramma struttura dei dati

3.2.1.1 Object Tracking

Una volta ottenuti i dati, ci si è posti il problema di poter effettuare un tracciamento degli oggetti in movimento nella scena. A questo scopo si è andati alla ricerca di algoritmi di tracciamento che potessero assolvere alla richiesta. Essendo un nuovo sensore, la letteratura è ancora in fase di definizione. Molte pubblicazioni propongono metodi di tracciamento senza però fornire codice sorgente da testare. Quindi, prima di procedere con la ricerca di un algoritmo adatto, si è cercato di implementare un codice che effettuasse un tracking basato sull'ordine delle sequenze di polarità che si verificano in ogni pixel in un determinato intervallo di tempo dell'ordine dei millisecondi. Da questo tentativo il risultato prodotto non distingueva i vari oggetti sulla scena raccogliendoli tutti quanti. Il presupposto errato, su cui si basava l'algoritmo, era che la polarità potesse indicare la direzione del movimento. Al contrario, qualsiasi sia la direzione di spostamento dell'oggetto, le sequenze di polarità, a meno di errori o disturbi, saranno sempre *false-true*. Quindi, in mancanza di altre idee, si è proceduto effettuando una ricerca più approfondita di un algoritmo con codice disponibile.

3.2.1.2 EKLT

Essendo il metodo *EKLT* l'unico pubblicato con codice eseguibile a libera disposizione, è stato utilizzato come esempio da cui partire per effettuare un tracciamento dei dati raccolti in fase di analisi. Per poter testare il metodo si è reso necessario installare una macchina virtuale con *Linux Xenial 16.04* su cui installare *ROS*. Successivamente è stato clonato il repository *rpg_ eklt* seguendo le indicazioni fornite nella documentazione ed è stato lanciato il codice con un file di esempio. All'atto pratico, il risultato dell'algoritmo è un file *.txt* contenente le informazioni necessarie per ricostruire il *tracking delle features*. Tra le informazioni abbiamo un identificatore della *feature*, il tempo in formato *timestamp Unix* e le coordinate rappresentanti il pixel. Una volta ottenuto questo file, si sottopone al codice di valutazione del tracking che ne mostra visivamente i risultati.

Nella figura 3.8 (a) possiamo vedere una cattura della riproduzione del file *bag* di partenza dove è rappresentata l'*immagine d'intensità* nello sfondo e gli *eventi* con colore blu e rosso. Nella figura 3.8 (b) possiamo vedere il risultato del tracciamento effettuato. I tracciati verdi sono prodotti dal metodo *EKLT*, quelli viola dal metodo *KLT*. Si evidenzia dai risultati che il tracciato verde risulta essere più preciso rispetto a quello viola. La grande quantità di caratteristiche tracciate è legata al fatto che l'acquisizione della scena è stata ottenuta utilizzando il sensore in movimento, perciò è tutta la scena ad essere in movimento, quindi i cambiamenti di luminosità registrati saranno di un numero molto elevato.

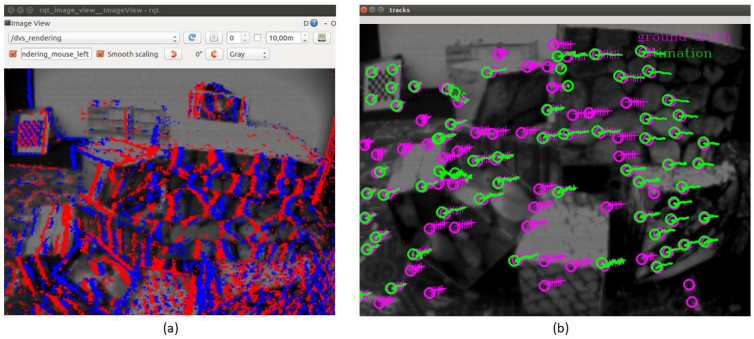

Figura 3.8: Diagramma struttura dei dati

Una volta appurata l'efficacia del metodo, valutata tramite l'esecuzione del codice contenuto nel repository ***rpg_ eklt*** (3.1.4.3) utilizzando vari file di esempio [16], si è cercato di applicarlo alle acquisizioni fatte in precedenza.

Capitolo 4

Sviluppo Modulo Software

4.1 Progettazione

Non essendo presente un tool in grado di convertire file da estensione *.aedat4*, nativa del sensore, all'estensione *.bag*, si è reso necessario produrre un modulo software in grado di assolvere questo compito. Il codice sviluppato è scaturito dall'analisi della struttura del file di esempio messo a disposizione nel repository *rpg_ eklt* [22] e dall'analisi di punti chiave del codice utilizzato per la simulazione del sensore DAVIS, contenuto nel repository *rpg_ davis_ simulator* [20]. Per prima cosa, lanciando da terminale il comando *"rosbag info path_file_ bag"* si ottiene il risultato contenuto in figura 4.1.

```
jack@jack-VirtualBox:~$ rosbag info /home/jack/Scaricati/esempio/boxes_6dof.bag

path:        /home/jack/Scaricati/esempio/boxes_6dof.bag
version:     2.0
duration:    59.8s
start:       Jul 19 2016 17:10:32.27 (1468941032.27)
end:         Jul 19 2016 17:11:32.05 (1468941092.05)
size:        1.7 GB
messages:    107249
compression: none [1262/1262 chunks]
types:       dvs_msgs/EventArray       [5e8beee5a6c107e504c2e78903c224b8]
             geometry_msgs/PoseStamped [d3812c3cbc69362b77dc0b19b345f8f5]
             sensor_msgs/CameraInfo    [c9a58c1b0b154e0e6da7578cb991d214]
             sensor_msgs/Image         [060021388200f6f0f447d0fcd9c64743]
             sensor_msgs/Imu           [6a62c6daae103f4ff57a132d6f95cec2]
topics:      /dvs/camera_info   32518 msgs    : sensor_msgs/CameraInfo
             /dvs/events         1794 msgs    : dvs_msgs/EventArray
             /dvs/image_raw      1298 msgs    : sensor_msgs/Image
             /dvs/imu           59685 msgs    : sensor_msgs/Imu
             /optitrack/davis   11954 msgs    : geometry_msgs/PoseStamped
jack@jack-VirtualBox:~$
```

Figura 4.1: Informazioni file bag di esempio

Analizzando le informazioni ottenute, si possono notare i tipi di messaggi e i *topics* ad essi associati con il numero di messaggi salvati per ogni *topic*. A questo punto quindi si è reso necessario progettare un programma in grado di

estrarre i dati dal file originale in formato *.aedat4* e di inserirli all'interno di messaggi seguendo la struttura predefinita. Nella documentazione *ROS* si legge che tutti i tipi di messaggi, escluso *dvs_msgs/EventArray*, sono supportati nativamente, quindi sarà sufficiente effettuare una chiamata al costruttore dell'oggetto messaggio. Il messaggio *dvs_msgs/EventArray* è un messaggio custom, creato appositamente per incapsulare l'innovativo tipo di dato evento. Questo è stato preso dal repository *rpg_dvs_ros* (3.1.4.1). Come vediamo dalla figura 4.2., *EventArray* è costituito da tre attributi: *height*, *width*, *events*. I primi due rappresentano le dimensioni del sensore, rispettivamente altezza e larghezza. L'ultimo attributo è un array di oggetti messaggio di tipo *Event*. Osservando la struttura del messaggio vediamo che esso è costituito da quattro attributi: *x*, *y*, *ts*, *polarity*. I primi due rappresentano le coordinate del pixel nel quale si verifica l'*evento*, *ts* è il *timestamp* e *polarity* la polarità.

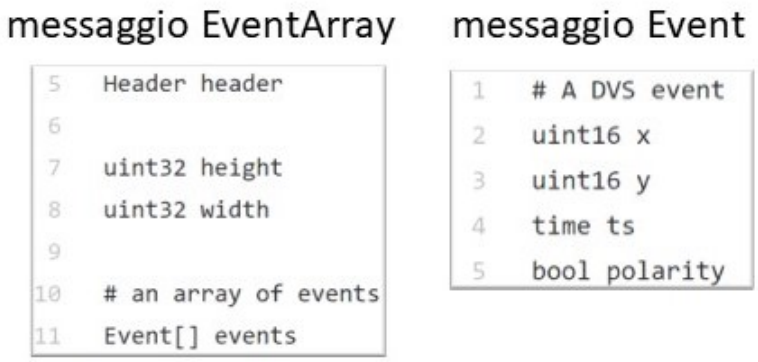

Figura 4.2: Messaggi custom per eventi

All'interno del codice che implementa il simulatore, invece, si è studiato il meccanismo di invio dei messaggi a partire dalla generazione del suo output che difatti viene pubblicato sotto forma di messaggi *ROS*, che sarà costituito da eventi, informazioni sul sensore, immagini di intensità in scala di grigi, la posizione del sensore e una mappa di profondità. Gli ultimi due tipi di dati non vengono considerati visto che sono basati sull'uso di due o più sensori. Nell'implementazione del simulatore, in particolare risalta la gestione delle immagini di intensità che, per essere convertite in messaggio *ROS*, hanno bisogno di una conversione da immagine di tipo *numpy array* a immagine di tipo messaggio. Per effettuare la conversione si utilizza l'*API CVBridge* che consente di fare da ponte tra *OpenCV* e *ROS*. In merito a questo passaggio sono sorti dei problemi non trascurabili riguardanti la compatibilità di *CVBridge* con la versione di *Python* utilizzata. Come già detto in precedenza, l'API che consente la gestione di file *aedat4* richiede un interprete *Python* che abbia almeno versione 3.5. *CvBridge* invece è supportato fino al massimo alla versione 2.7. Per risolvere questo problema si è

inizialmente pensato di creare un sistema del tipo *Client-Server*, dove il *Server*, script con interprete *Python 2.7* avrebbe ricevuto dal *Client*, script con interprete *Python 3.5*, tramite messaggio, il *frame* estratto dal file *aedat4* e, una volta eseguita la conversione utilizzando i metodi dell'*API*, avrebbe ritornato indietro un messaggio con l'immagine correttamente convertita. Per poter fare questo però sarebbe stato necessario serializzare un oggetto che avrebbe dovuto rappresentare il messaggio contenente il frame convertito. Questa soluzione è stata messa da parte nel caso non si fosse trovata nessuna alternativa. La soluzione al problema è stata trovata all'interno dei vari forum sul settore della *Computer Vision* dove si propone di installare in un nuovo ambiente di lavoro, creato con *catkin*, una versione dell'*API OpenCV*, chiamata *vision_opencv*, modificata in modo tale da essere utilizzabile con interprete *Python 3.x* e contenente al suo interno i metodi dell'*API CVBridge*. Un'altra problematica abbastanza complessa ha riguardato la raccolta degli *eventi*. Come evidenziato nella descrizione del sensore, vengono raccolti fino a 12 milioni di eventi al secondo. Quindi bisognava scegliere un tempo di raccolta degli eventi per produrre poi più messaggi. Scegliere un intervallo di tempo troppo breve avrebbe comportato un numero non abbastanza elevato di eventi per array, mentre un intervallo troppo grande avrebbe causato una saturazione della scena. A seguito di vari test, analiticamente si è individuato come valore accettabile 3ms.

4.2 Implementazione

Per lo sviluppo del tool si sono seguite tre strade differenti in modo tale da valutarne l'efficienza ed il tempo di esecuzione. Per tutti i tre le implementazioni, osservando la struttura del messaggio *EventArray* (vedi figura 3.7), si è utilizzato un array contenente gli *eventi* destinati alla pubblicazione del messaggio che viene azzerato ad ogni invio del messaggio.

4.2.1 converter1.py

Per quanto riguarda *converter1.py*, a seguito della creazione dello stream dei dati dal file *aedat4*, si è deciso di raccogliere *frames* ed *eventi* in **maniera sincronizzata** rispettando i tempi dei suddetti *frames* ed *eventi*. Questa scelta consente, non solo di mantenere una logica di raccolta che si avvicina il più possibile al reale funzionamento del sensore, ma consente anche, a scelta dell'utente, di aprire uno stream dei dati dal file *aedat4* verso un programma che effettui il tracciamento dei dati senza dover necessariamente effettuare la conversione fisica in file di tipo *bag*, creando un nodo ad hoc per la pubblicazione del contenuto del file originale sotto forma di messaggi. A questo scopo si fa utilizzo di 2 indici definiti con **k** e **i**, entrambi inizializzati a zero.

L'**indice k** assume valore 0 oppure 1 e indica se il frame correntemente preso in carico è il primo oppure no. Difatti, all'arrivo del primo frame (*k=0*), viene salvato il tempo del frame in una variabile, si impone poi *k=1* e si pubblica il messaggio contenente il frame. Una volta inviato il messaggio, viene preso in carico il secondo frame. L'*indice k*, indicando a questo punto che il *primo frame* è già stato considerato, consentirà di salvare il tempo del *frame attuale* in un'altra variabile, tenendo però in memoria il tempo del *frame precedente*.

L'**indice i** assume valore 1 oppure 2 e indica se l'evento preso in considerazione è il primo della serie di eventi inviati in ogni singolo messaggio oppure no. Quindi se *i=1*, viene salvato il tempo dell'evento in una variabile. A questo punto, per la raccolta di *eventi*, si utilizza la variabile che indica il tempo del *frame attuale*. Quindi verranno raccolti gli *eventi* compresi nell'intervallo che va dal *frame precedente* al *frame attuale*. Per l'estremo iniziale dell'intervallo non c'è bisogno di specificare il tempo del frame precedente in quanto la struttura *Google Flatbuffer* consente un'unica iterazione (vedi 3.2), quindi non ci sarà il rischio di riprendere eventi già raccolti in precedenza. Una volta stabilito l'intervallo di raccolta degli eventi, questi vengono pubblicati dividendoli in più messaggi usando un *tempo di integrazione* definito da utente. Potrebbe accadere che, all'ultima raccolta, all'interno dell'intervallo sopra detto, il *tempo di integrazione* non sia esaurito. Per questo motivo è stato inserita una *condizione if* alla fine che valuta se l'array di appoggio degli *eventi* ha una dimensione non nulla. Se la dimensione è maggiore di 0, viene pubblicato un ulteriore messaggio con i restanti *eventi*. In questo modo si cerca di ridurre al minimo la perdita di informazione. A questo

punto si passa al prossimo frame e con un nuovo intervallo si ripercorrono gli stessi passi. Inoltre per ogni frame vengono inviate, come messaggio *camera_info*, le informazioni riguardanti le dimensioni del frame, rispettivamente larghezza e altezza. Per chiarezza si esplicita il fatto che nella prima iterazione tra eventi e frame c'è lo scostamento di un frame.

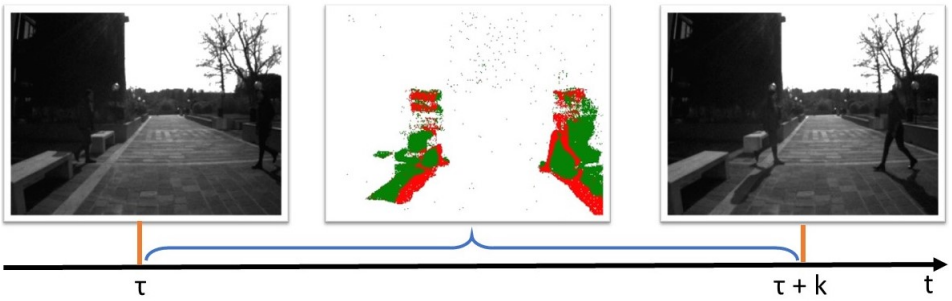

Figura 4.3: Immagine esemplificativa del metodo implementato

La figura 4.3 mostra due frame consecutivi, ai tempi τ e $\tau + k$, e gli eventi registrati nell'intervallo di tempo.

Algorithm 1 converter1.py

```
1:  procedure CONVERSIONE(delta)
2:      Apertura stream dati
3:      for frame in aedat4 do
4:          if k = 0 then
5:              t_frame_attuale = frame.time
6:              k = 1
7:          if k! = 0 then
8:              t_frame_precedente = t_frame_attuale
9:              t_frame_attuale = frame.time
10:         if write = True then
11:             Apri stream dati su file
12:         tempo_frame = frame.time
13:         Invio messaggio ROS per frame
14:         if k! = 0 then
15:             for e in events do
16:                 if i = 1 then
17:                     tempo_primo_ev = e.time
18:                     i = 2
19:                 if e.time<t_frame_attuale then
20:                     Raccolta eventi
21:                     if e.time − tempo_primo_ev > delta then
22:                         i = 1
23:                         Invio messaggio ROS per eventi
24:                 else
25:                     Invio messagio ROS per eventi
26:                 break
27:             Invio messaggio ROS camera_info
28:     Chiusura stream dati
```

4.2.2 converter2.py

Questo codice effettua una **raccolta separata** di *frames* ed *eventi*. Prima vengono inviati tutti i messaggi relativi ai *frames*, poi tutti i messaggi relativi agli *eventi* distribuiti rispettando il *tempo di integrazione* definito da utente. Per quanto riguarda la raccolta degli *eventi*, viene salvato il tempo di ogni *primo evento* di un nuovo messaggio da inviare. Effettuando quindi una sottrazione tra il tempo dell'*evento attuale* e il tempo del *primo evento* si stabilisce se si è all'interno dell'intervallo definito dal *tempo di integrazione* e quindi se l'*evento* va raccolto oppure se è arrivato il momento di pubblicare il messaggio e di iniziare a raccogliere il contenuto del prossimo messaggio. Come appare evidente, questa implementazione non consente la pubblicazione dei messaggi senza creare il file *bag* che quindi sarà un passaggio obbligato.

Algorithm 2 converter2.py

```
1:  procedure CONVERSIONE2(delta)
2:      Apertura stream dati
3:      for frame in aedat4 do
4:          Invio messaggio ROS per frame
5:          Invio messaggio ROS camera_info
6:      for event in aedat4 do
7:          if i = 1 then
8:              tempo_primo_ev = event.time
9:              i=2
10:         Raccolta eventi
11:         if event.time − tempo_primo_ev > delta then
12:             Invia messaggio ROS per eventi
13:             i=1
14:     Chiusura stream dati
```

4.2.3 converter3.py

Questo codice è un riadattamento dell'implementazione proposta sul repository *simbamford/AedatTools* (3.1.4.5), creata anni fa per file di tipo *aedat3*, quindi nella versione precedente del sensore e non compatibile con la versione attuale. L'implementazione propone una **raccolta separata** di *eventi* e *frame*, come in *converter2.py*, ma non considera il *tempo di integrazione*. Come parametro per la raccolta degli eventi viene invece utilizzato un numero costante che identifica il *numero di eventi per ogni messaggio*. Questa scelta, a seguito di test, risulta essere poco affidabile in quanto non considera la mutevolezza dei casi acquisiti con il sensore. Potrebbe difatti verificarsi che, in una scena con pochi movimenti, per la verità esigui, vengano inviati insieme *eventi* che si verificano a distanza di

tempo rilevante. Allo stesso modo, se nella scena sono presenti molti movimenti, all'interno di uno stesso istante di tempo verrebbero inviati più messaggi.

Algorithm 3 converter3.py

```
1:  procedure CONVERSIONE3
2:      numEventi = 25000
3:      Apertura stream dati
4:      for frame in aedat4 do
5:          Invio messaggio ROS per frame
6:          Invio messaggio ROS camera_info
7:      for event in aedat4 do
8:          eventiRaccolti +1
9:          if eventiRaccolti = numEventi then
10:             Invia messaggio ROS per eventi
11:     Chiusura stream dati
```

4.3 Analisi delle prestazioni

Per effettuare l'analisi delle prestazioni sono stati presi in considerazione come parametri di analisi il *tempo di elaborazione* espresso in MB/s e il *numero di eventi totali raccolti*, in modo tale da valutare possibili perdite di informazioni. A tale scopo è stato preso come file di esempio un file *aedat4* con durata 62 secondi e di dimensione 313.6 MB. Tramite un semplice script sono stati individuati il numero totale dei frame (1551) e il numero totale degli eventi (21'533'311). In tabella si riportano i risultati ottenuti.

	converter1.py	converter2.py	converter3.py
N. Eventi	21529678	21533311	21525000
N. Frames	1551	1551	1551
Prestazioni	0.21 MB/s	0.143 MB/s	0.177 MB/s

Ciò che emerge dal test è che *converter1.py* ha un tempo di esecuzione inferiore agli altri, con prestazioni intorno a 0.21 MB/s, contro i 0.143 MB/s di *converter2.py* e i 0.177 MB/s di *converter3.py*. Di contro però, *converter1.py* presenta una perdita di *eventi* pari a circa 0.0168%, nessuna perdita in *converter2.py* e 0.0038% in *converter3.py*.
Da questa analisi sorge quindi spontanea una considerazione. La scelta del codice più adatto dipende dalle priorità dell'utente. Se la riproduzione, senza bisogno di convertire fisicamente il file, è preferita, la scelta ricadrà su *converter1.py*, nonostante una perdita esigua di informazioni. Se invece la priorità è l'avere la totalità delle informazioni a discapito della perdita della funzionalità sopra detta, allora

la scelta ricadrà su *converter2.py*. *converter3.py* non andrebbe mai utilizzato per le considerazioni fatte in fase di spiegazione del funzionamento (4.2.3)

4.4 Applicazione algoritmo EKLT

Come accennato in precedenza, a causa della mancanza di algoritmi di tracciamento disponibili, si è stati obbligati ad utilizzare il codice fornito con la pubblicazione *EKLT* per verificare la corretta conversione. Di seguito vengono illustrati i risultati ottenuti dall'applicazione dell'algoritmo al file convertito tramite i tre diversi software implementati al fine di mettere in luce eventuali differenze. La figura 4.4 mostra due tipologie di grafico per ogni codice. I grafici (a), (b), (c) mostrano una rappresentazione spazio-tempo in 3D delle *features tracciate* utilizzando il metodo *EKLT* (in blu) e il metodo *KLT* (in verde). I grafici (d), (e), (f) mostrano invece gli *errori di tracciamento* riscontrati nell'applicazione del metodo *EKLT*. Quindi avremo nella prima colonna i risultati corrispondenti all'applicazione di *converter1.py*, la seconda colonna corrispondente a *converter2.py* e la terza colonna corrispondente al *converter3.py*.

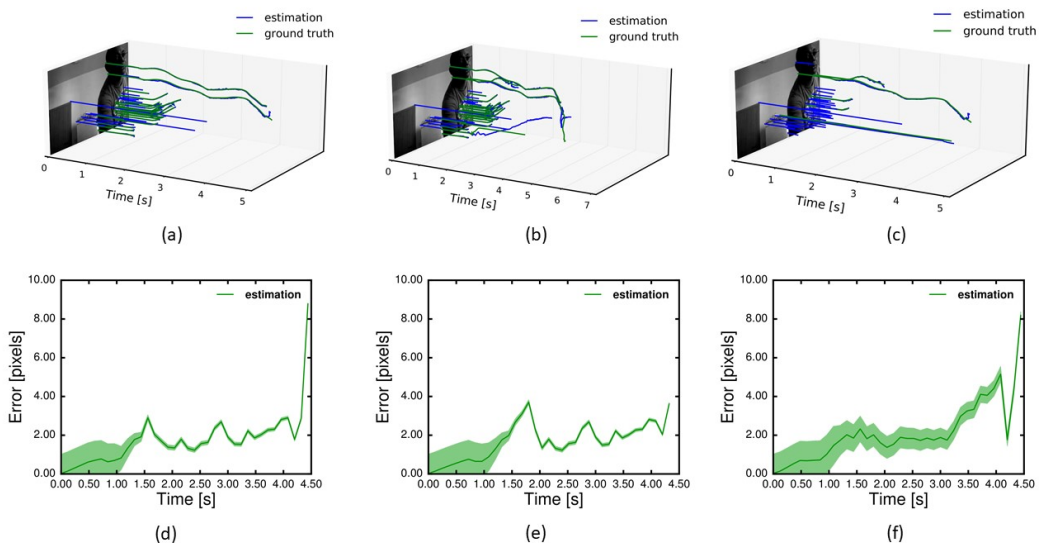

Figura 4.4: Risultati applicazione metodo EKLT

Non risultano significative differenze tra l'uso del file prodotto da *converter1.py* e il file prodotto da *converter2.py* se non per gli ultimi istanti di tempo in cui il metodo *EKLT* produce più *errori nel tracciamento delle features* del file prodotto da *converter1.py*. Confrontando invece i grafici (a) e (d) con (b) e (e) e con i grafici (c) e (f) emergono differenze non trascurabili sia dal punto di vista del numero e della qualità delle *features* tracciate, sia dal punto di vista degli *errori* prodotti.

Capitolo 5

Conclusioni e Sviluppi Futuri

5.1 Conclusioni

Il presente elaborato ha illustrato l'attività di ricerca effettuata sul sensore *Davis346* che ha portato alla progettazione e sviluppo di un *modulo software* che consente di applicare tecniche di *Visual Object Tracking* implementate in ambiente *ROS* alle acquisizioni prodotte tramite il sensore stesso.
Si è partiti dall'illustrazione di alcune tecniche di *Visual Object Tracking* utilizzate su dati prodotti da *camere standard*. Sono poi state mostrate le innovative tecniche applicate ai dati prodotti da *camere ad eventi* mettendo in evidenza la capacità delle stesse di risolvere i problemi riscontrati nell'uso delle precedenti. L'assenza di codici utilizzabili, essendo ancora la letteratura agli inizi, ha obbligato all'uso dell'algoritmo *EKLT*. Quindi si è reso necessario produrre un *modulo software* che consentisse l'applicazione dell'algoritmo, implementato in ambiente *ROS*, ai dati prodotti dal sensore tramite software *DV*. Il progetto di tesi propone quindi tre metodi di conversione, due creati appositamente e uno riadattato da implementazioni precedenti. La scelta del metodo ricade sull'utente che dovrà decidere in base a prestazioni, integrità dei dati e funzionalità aggiuntive come lo streaming dei dati senza effettuare la conversione. Grazie alle notevoli funzionalità messe a disposizione dall'ambiente *ROS*, probabilmente molte tecniche future saranno realizzate al suo interno e sarà quindi utile avere a disposizione un modulo software in grado di far interagire il sensore con tale ambiente e quindi con i vari algoritmi implementati.

5.2 Sviluppi futuri

I moduli software implementati consentono la conversione dei dati relativi a immagini di intensità e agli eventi trascurando i dati riguardanti l'*Unità di Misura Inerziale* (*IMU*) e altri possibili tipi di dato ottenibili dai dati già in possesso. Quindi, in caso di necessità, sarebbe possibile aggiungere al codice la gestione di altri tipi di dato da inviare per messaggio.
Delle migliorie attuabili al codice già esistente potrebbero essere quelle di risolvere in *converter1.py* il problema per cui inizialmente viene inviato per messaggio il primo frame, il secondo frame e poi gli eventi contenuti tra i due frame e di risolvere il problema della perdita di dati che, per quanto esigua, potrebbe influire sul corretto funzionamento dei vari algoritmi utilizzati.
Una funzionalità aggiuntiva potrebbe essere lasciare all'utente la possibilità di scegliere il tempo di inizio e di fine dell'acquisizione da convertire.
In ultimo si potrebbe verificare l'utilità della funzione di riproduzione del file originale senza dover effettuare la conversione e, se necessario, utilizzare i metodi contenuti nell'*api python-dv* per effettuare lo stream dei dati provenienti dal sensore in tempo reale.

Appendice A

Termini tecnici

A.1 Dynamic Range

Il *dynamic range*, o *range dinamico*, è un intervallo identificato dal rapporto tra valore massimo e valore minimo di una grandezza fisica analizzata in un dato contesto. Essendo un rapporto tra grandezze equivalenti, il suo valore è adimensionale, ma per indicare che il valore rappresenta non la grandezza fisica, ma il suo livello si usa indicare come unità di misura il *Bell*. Essendo però la scala *Bell* troppo compressa, in definitiva si rappresenta il *dynamic range* in *decibel dB*. Nel caso analizzato in questo studio di tesi, la grandezza fisica presa in considerazione è la luminosità. Quindi per scene con zone contraddistinte da elevata luminosità e bassa luminosità parleremo di *High Dynamic Range* ***HDR***.

A.2 Inertial Measurement unit IMU

L'*unità di misura inerziale*, o ***IMU***, è una scheda che consente di misurare la dinamica dell'oggetto sulla quale è installata, nel nostro caso il sensore. Nel particolare vengono misurate l'accelerazione, la velocità angolare, la temperatura e l'orientamento. Ogni misura viene fatta su un sistema di 3 assi, quindi ogni grandezza sarà espressa nelle tre coordinate. Da queste informazioni raccolte è possibile stabilire se l'oggetto è in movimento, se sta ruotando o se è inclinato rispetto all'orizzontale.

A.3 Frame per secondo (fps) e immagine di intensità

Unità di misura comunemente utilizzata per rappresentare la velocità di riproduzione dei fotogrammi di un filmato. Nel caso dei sensori indica la velocità di riproduzione delle immagini di intensità. Come accennato nell'introduzione,

un'immagine di intensità corrisponde ad una matrice di dati in cui ogni posizione della matrice assume un valore contenuto in una scala di colore che ne indica l'intensità. Per avere un esempio concreto si faccia riferimento alla figura A.1

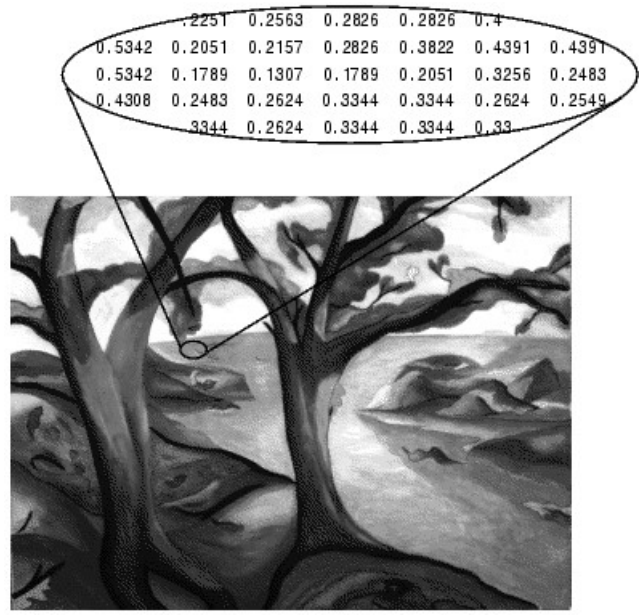

Figura A.1: Esempio immagine di intensità
Fonte:
http://indico.ictp.it/event/a08187/session/121/contribution/75/material/0/1.pdf

A.4 Rilevamento delle features

Con *features*, o caratteristiche, si intendono parti di immagine come punti, linee, contorni o oggetti. Grazie al loro tracciamento all'interno delle sequenze di immagini è possibile riconoscere gli oggetti, definiti da queste caratteristiche, e tracciarne il loro movimento. Con *feature detection*, o *rilevamento delle features*, si indica un insieme di metodi di raccolta di informazioni provenienti da un'immagine che portino al raggiungimento di un obiettivo computazionale in un certo ambito. Quando si sottopongono immagini a questi metodi, il risultato sarà un insieme di caratteristiche individuate, come ad esempio punti, linee, contorni o oggetti all'interno dell'immagine, che consentirà di risolvere il problema proposto come ad esempio il riconoscimento degli oggetti o il tracciamento del loro movimento.

Bibliografia

[1] V. A. Laurense, J. Y. Goh, and J. C. Gerdes. Path-tracking for autonomous vehicles at the limit of friction. In *2017 American Control Conference (ACC)*, pages 5586–5591, 2017.

[2] R. T. Collins, Yanxi Liu, and M. Leordeanu. Online selection of discriminative tracking features. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(10):1631–1643, 2005.

[3] Helmut Grabner, Christian Leistner, and Horst Bischof. Semi-supervised on-line boosting for robust tracking. In *Proceedings of European Conference on Computer Vision*. ., 2008.

[4] Luca Bertinetto, Jack Valmadre, Joao Henriques, Andrea Vedaldi, and Philip Torr. Fully-convolutional siamese networks for object tracking. volume 9914, pages 850–865, 10 2016.

[5] Anfeng He, Chong Luo, Xinmei Tian, and Wenjun Zeng. A twofold siamese network for real-time object tracking. pages 4834–4843, 06 2018.

[6] Bo Li, Junjie Yan, Wei Wu, Zhu Zheng, and Xiaolin Hu. High performance visual tracking with siamese region proposal network. pages 8971–8980, 06 2018.

[7] https://en.wikipedia.org/wiki/image_registration.

[8] Jörg Stückler, Max Schwarz, and Sven Behnke. Mobile manipulation, tool use, and intuitive interaction for cognitive service robot cosero. *Frontiers in Robotics and AI*, 3, 11 2016.

[9] C. Tomasi and Takeo Kanade. Shape and motion from image streams: a factorization method – part 3 detection and tracking of point features. Technical Report CMU-CS-91-132, Carnegie Mellon University, Pittsburgh, PA, April 1991.

[10] Huiyu Zhou, Yuan Yuan, and Chunmei Shi. Object tracking using sift features and mean shift. *Computer Vision and Image Understanding*, 113(3):345–352, March 2009.

[11] A. R. Vidal, H. Rebecq, T. Horstschaefer, and D. Scaramuzza. Ultimate slam? combining events, images, and imu for robust visual slam in hdr and high-speed scenarios. *IEEE Robotics and Automation Letters*, 3(2):994–1001, 2018.

[12] C. Brandli, R. Berner, M. Yang, S. Liu, and T. Delbruck. A 240 × 180 130 db 3 µs latency global shutter spatiotemporal vision sensor. *IEEE Journal of Solid-State Circuits*, 49(10):2333–2341, 2014.

[13] http://wiki.ros.org/ros/introduction.

[14] http://wiki.ros.org/ros/tutorials/creatingmsgandsrv.

[15] http://wiki.ros.org/ros/installation.

[16] http://rpg.ifi.uzh.ch/davis_data.html.

[17] https://github.com/uzh-rpg/rpg_dvs_ros.

[18] Elias Mueggler, Basil Huber, and Davide Scaramuzza. Event-based, 6-dof pose tracking for high-speed maneuvers. 09 2014.

[19] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. A 128x128 120db 15us latency asynchronous temporal contrast vision sensor. 43:566–576, 01 2007.

[20] https://github.com/uzh-rpg/rpg_davis_simulator.

[21] Elias Mueggler, Henri Rebecq, Guillermo Gallego, Tobi Delbruck, and Davide Scaramuzza. The event-camera dataset and simulator: Event-based data for pose estimation, visual odometry, and slam. *The International Journal of Robotics Research*, 36(2):142–149, 2017.

[22] https://github.com/uzh-rpg/rpg_eklt.

[23] Daniel Gehrig, Henri Rebecq, Guillermo Gallego, and Davide Scaramuzza. Eklt: Asynchronous photometric feature tracking using events and frames. *International Journal of Computer Vision*, pages 1–18, 08 2019.

[24] https://github.com/uzh-rpg/rpg_feature_tracking_analysis.

[25] https://github.com/simbamford/aedattools/tree/master/pyaedattools.

[26] https://gitlab.com/inivation/dv/dv-python.

[27] http://wiki.ros.org/cv_bridge.

[28] https://github.com/ros-perception/vision_opencv.

[29] Elias Mueggler, Henri Rebecq, Guillermo Gallego, Tobi Delbruck, and Davide Scaramuzza. The event-camera dataset and simulator: Event-based data for pose estimation, visual odometry, and slam. *The International Journal of Robotics Research*, 36, 11 2016.

Elenco delle figure

2.1 Esempio di aggiornamento classificatore e tracciamento.
Fonte: Grabner et al [3] 10
2.2 Esempio di riconoscimento oggetti con CNN.
Fonte: https://industrywired.com/the-era-of-computer-vision-is-here/ 11
2.3 Applicazione image registration
Fonte: Stückler et al [8] 11
2.4 Esempio di tracking KLT
Fonte: http://www.cs.cmu.edu/ 16385/s17/Slides/15.1_Tracking__KLT.pdf 12
2.5 Esempio di mean shift, SIFT e loro combinazione
Fonte: Zhou et al [10] 13
2.6 SLAM 14

3.1 Event-camera DAVIS346
Fonte: http://rpg.ifi.uzh.ch/CED.html 17
3.2 Software DV
Fonti: https://inivation.com/inivation-announces-launch-of-dv-software-to-accelerate-development-of-event-based-vision-applications/
https://inivation.com/inivation-announces-launch-of-dv-software-1-0/ 18
3.3 Logo ROS. Fonte: https://www.ros.org/ 19
3.4 Player 20
3.5 Schema di funzionamento CVBridge
Fonte: ROS [27] 22
3.6 Frammenti di acquisizione in formato frame ed eventi 23
3.7 Diagramma struttura dei dati 24
3.8 Diagramma struttura dei dati 25

4.1 Informazioni file bag di esempio 27
4.2 Messaggi custom per eventi 28
4.3 Immagine esemplificativa del metodo implementato 31
4.4 Risultati applicazione metodo EKLT 35

A.1 Esempio immagine di intensità
Fonte: http://indico.ictp.it/event/a08187/session/121/contribution/75/material/0/1.pdf