UNIVERSITÁ POLITECNICA DELLE MARCHE

FACOLTÁ DI INGEGNERIA

Corso di Laurea Triennale in
Ingegneria Informatica e dell'Automazione

Simulazione di event cameras per la navigazione di velivoli senza pilota

Simulation of event cameras for unmanned aerial vehicle navigation

Relatore:
Ch.mo Prof. Mancini Adriano

Laureando:
Compagnoni Paolo

Anno Accademico 2018-2019

Indice

Elenco delle figure **4**

1 Introduzione **5**
1.1 Obiettivo . . . 5
1.2 Struttura della tesi . . . 5
1.3 Perché ESIM? . . . 6
1.4 Camere ad eventi . . . 6
1.4.1 Nozioni introduttive . . . 6
1.4.2 Applicazioni . . . 7
1.4.3 Funzionamento . . . 8
1.4.4 Comparazione della larghezza di banda di un pixel DVS e quella di una camera tradizionale . . . 11
1.4.5 Modelli . . . 13
1.4.6 Vantaggi delle camere ad eventi . . . 14
1.4.7 Modello di generazione degli eventi . . . 15

2 Strumenti utilizzati **17**
2.1 Robot Operating System (ROS) . . . 17
2.1.1 Concetti introduttivi . . . 17
2.1.2 Caratteristiche . . . 18
2.1.3 Catkin . . . 18
2.1.4 ROS Master . . . 19
2.1.5 ROS Node . . . 20
2.1.6 ROS Topics . . . 21
2.1.7 ROS Messages . . . 22
2.1.8 Esempio di utilizzo del ROS . . . 22
2.1.9 ROS Launch . . . 25
2.1.10 ROS Packages . . . 26
2.1.11 ROS Services . . . 28
2.1.12 ROS Actions . . . 30
2.1.13 ROS Time . . . 30
2.1.14 ROS Bags . . . 30
2.1.15 Rviz . . . 31

2.1.16 Rqt Image View 32
2.2 Blender e scena 3D utilizzata per la simulazione 33
2.3 Installazione del ROS 34
2.4 Python 35
2.5 FFmpeg 35
2.6 OpenGL 36
2.7 Descrizione del file video usato 36

3 Il simulatore ESIM **37**
3.1 Concetti introduttivi 37
3.2 Adaptive Sampling 38
3.3 Architettura del simulatore 39
3.4 Modi di utilizzo dell'adaptive sampling 40
3.4.1 Metodo basato sul cambiamento di luminosità 40
3.4.2 Metodo basato sullo spostamento del pixel 41
3.5 Vantaggi dell'adaptive sampling 41
3.6 Simulazione dei disturbi e delle non-linearità 42

4 Risultati **43**
4.1 Installazione del simulatore 43
4.2 Simulazione di eventi a partire da un video 45
4.2.1 Concetti preliminari 45
4.2.2 Download del video 45
4.2.3 Pre-processing del video per ESIM 46
4.2.4 Simulazione degli eventi con ESIM 49
4.3 Visualizzare gli eventi simulati 51
4.4 Simulazione di eventi basata su una scena virtuale 52

5 Conclusioni e Sviluppi Futuri **55**
5.1 Conclusioni 55
5.2 Sviluppi futuri delle camere ad eventi 56

Bibliografia **59**

Capitolo 1

Introduzione

1.1 Obiettivo

Questo progetto fornisce gli strumenti sia teorici che pratici per comprendere ed utilizzare nel modo adeguato il simulatore di camere ad eventi quale *ESIM: an Open Event Camera Simulator [4].*

1.2 Struttura della tesi

La tesi mostra il *workflow* necessario per una simulazione con ESIM nelle varie modalità disponibili, partendo prima dalle caratteristiche base dell'ambiente di sviluppo del simulatore per poi arrivare verso l'utilizzo pratico dello stesso. A tale scopo, il lavoro è suddiviso nel modo seguente:

- la prima parte sarà dedicata alle camere ad eventi che rappresentano la tecnologia oggetto della simulazione: ne verranno introdotte le caratteristiche principali, i principi matematici che sono alla base del loro funzionamento ed i vantaggi rispetto alle camere tradizionali;

- nella seconda parte verranno illustrati i principali strumenti software e i materiali multimediali impiegati per ottenere le simulazioni attraverso l'utilizzo di ESIM;

- nella terza parte verrà fornita una panoramica generale sul funzionamento di ESIM che verrà poi messo in pratica nel dettaglio nella quarta parte della tesi, in base alla modalità di utilizzo scelta.;

- nella quarta parte ci si focalizzerà sull'utilizzo pratico del simulatore nelle varie modalità disponibili con tutti gli accorgimenti necessari affinché la simulazione sia effettuata in modo corretto ed accurato;

- la parte finale è incentrata nel trarre le conclusioni alla luce dei risultati ottenuti e ad illustrare i possibili sviluppi futuri nell'impiego delle camere ad eventi;

1.3 Perché ESIM?

Si è scelto di approfondire lo studio di questo simulatore per molteplici aspetti. Uno di questi è dato dal fatto di avere il codice fornito interamente con licenza *open source*, distribuito dall'Università di Zurigo su *Github*. Non è l'unico punto di forza del simulatore. Infatti esso è in grado di fornire numerosi strumenti, come l'integrazione dell'IMU (*Inertial Measurement Unit*), la simulazione del *motion blur* e dei disturbi nel segnale della camera.
Questi aspetti verranno analizzati nel dettaglio nel corso della tesi.

1.4 Camere ad eventi

1.4.1 Nozioni introduttive

Le camere ad eventi sono dei sensori che operano in maniera completamente differente rispetto alle camere attualmente in commercio basate sui *frames* e di fatto rappresentano un cambio radicale nel campo dell'acquisizione delle immagini. Esse infatti non catturano le immagini ad una frequenza di campionamento fissa (*clock*), bensì si basano sulla dinamicità della scena misurando in maniera asincrona le variazioni di luminosità pixel per pixel.
Un evento quindi, è un cambiamento asincrono della luminosità di un pixel che può essere positivo o negativo in relazione ad un aumento o diminuzione della luce che impatta su di esso.
Il risultato è un flusso di eventi che registrano la posizione, tempo e segno della variazione di luminosità.
I vantaggi che queste camere rivoluzionarie offrono sono i seguenti:

- *risoluzione temporale molto alta* (dell'ordine dei microsecondi);
- *range dinamico molto alto* (140db contro i 60db delle camere tradizionali);
- *basso consumo di energia*;
- *alta larghezza di banda dei pixel* (dell'ordine di khz) che si traduce in un ridotto *motion blur* e quindi bassa latenza.

1.4.2 Applicazioni

Si intuisce subito che queste camere possono essere impiegate ampiamente in vari campi dove sono richieste operazioni sotto condizioni di luce scadenti e incontrollabili come ad esempio la *computer vision* e la robotica. Tuttavia esse, per via delle loro caratteristiche innovative, hanno bisogno di nuovi algoritmi e applicazioni per sfruttarne al meglio le caratteristiche.
Malgrado le camere ad eventi siano disponibili solo dal 2008, grandi aziende come Samsung e Prophesee hanno sottolineato un alto interesse commerciale nelle possibilità che esse offrono nella robotica, realtà aumentata e virtuale (AR e VR) e nei videogiochi: più avanti, nella parte finale della tesi, verranno illustrate nel dettaglio le possibili applicazioni future.
Le camere ad eventi possono anche essere utilizzate nei campi della monitorizzazione e della sorveglianza, nel riconoscimento di *gestures* e nel tracciamento di oggetti. Questa nuova tecnologia è molto utile anche nella ricostruzione di immagini HDR (*High Dynamic Range*) e nello SLAM (*Simultaneous Localization and Mapping*).

1.4.2.1 Utilizzo nei droni

Le camere ad eventi trovano un impiego crescente nella navigazione di ***Unmanned aerial vehicles (UAV)*** o **droni** (figura 1.1).
Gli UAV sono dei velivoli capaci di muoversi senza l'ausilio di un pilota umano a bordo. Essi vengono utilizzati in svariati settori come l'agricoltura, sorveglianza, soccorso in caso di terremoti o incendi, spedizione di prodotti e molto altro.

Figura 1.1: Un drone. Immagine tratta da [13]

Un drone viene equipaggiato con una camera ad eventi per migliorarne le capacità di rilevazione di ostacoli ad alta velocità o di oggetti che si muovono con movimenti molto rapidi. Inoltre l'integrazione di una camera ad eventi in un drone permette la visione di oggetti in condizione di luce molto sfavorevoli (come ad esempio una stanza molto buia).

In questo modo, esso è capace di svolgere il processo di costruzione di una mappa di un ambiente e di localizzazione all'interno della mappa stessa, chiamato **SLAM** (*Simultaneous Localization and Mapping*) in maniera molto efficace ed efficiente.

1.4.3 Funzionamento

In questa sezione, verranno analizzati i principi di funzionamento di una camera ad eventi come ad esempio il *Dynamic Vision Sensor* ***(DVS)*** (figura 1.2) che rileva le variazioni della luminosità di ogni pixel in modo asincrono e indipendente. Per luminosità ci si riferirà nel corso della tesi, al logaritmo dell'intensità di luce di ogni pixel.
Ogni pixel contenuto all'interno di una camera ad eventi memorizza quindi il logaritmo dell'intensità di luce che impatta su di esso. Questo accade ogni volta che la camera invia un nuovo evento.

Figura 1.2: Camera DVS. Immagine tratta da [7].

La camera monitora in modo continuo ogni variazione nell'intensità della luce su tutti i suoi pixel. Quando una variazione ha una magnitudo che eccede una certa soglia rispetto al valore memorizzato in precedenza, la camera invia un nuovo evento che viene trasmesso dal chip con la posizione (x,y), il tempo e la polarità (indicato con 1 bit) della variazione ("ON" se la luminosità aumenta, "OFF" se la luminosità diminuisce) (figure 1.3a e 1.3b).
Il flusso di eventi risultante è mostrato nelle figure 1.3e e 1.3f.

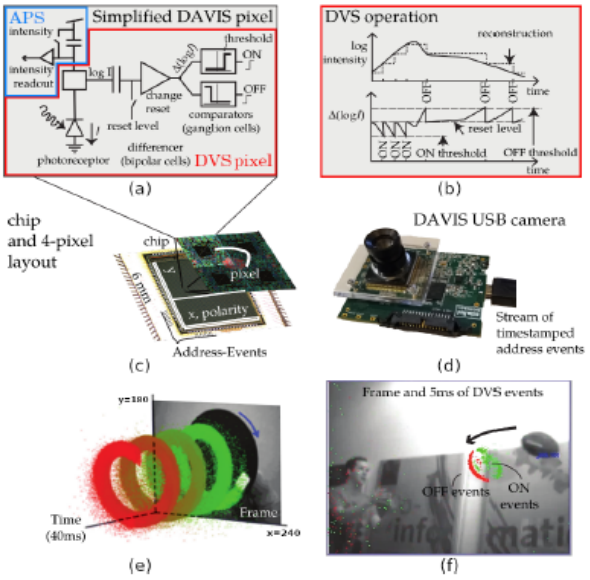

Figura 1.3: Riassunto di una camera DAVIS, che comprende un *dynamic vision sensor* (DVS) e un *active pixel sensor* (APS) nello stesso array di pixel, condividendo lo stesso fotodiodo in ogni pixel. **(a)** Diagramma del circuito semplificato del pixel DAVIS. (Il pixel DVS è in rosso, il pixel APS è in blu). **(b)** Schematizzazione delle operazioni del pixel DVS nel convertire la luce in eventi. **(c)-(d)** Chip DAVIS e camera USB. **(e)** Un quadrato bianco su un disco rotante nero visto dal sensore DAVIS che produce *frames* in scala di grigi e una spirale di eventi nello spazio e nel tempo. Gli eventi nello spazio e nel tempo sono codificati con colori: dal verde (passato) al rosso (presente). **(f)** Frame e eventi sovrapposti in una scena. Immagine tratta da [3, p.2].

Il flusso di eventi viene poi trasmesso dall'array di pixel verso l'esterno attraverso un bus di output condiviso che utilizza un protocollo di trasferimento dei dati chiamato ***AER*** (*Address-Event Rappresentation*). A causa delle numerose variazioni di luminosità che possono accadere durante l'acquisizione di una scena reale, si possono avere delle trasmissioni di dati ad una frequenza che va dai 2 MHz fino ai 1200MHz che potrebbero saturare il bus. Questa larghezza di banda dipende dal chip e dal tipo di hardware utilizzato.
Ogni pixel ha al suo interno un modulatore della frequenza di campionamento dell'intensità della luce. Questo è necessario perché una camera ad eventi è un sensore guidato dai dati: il loro output dipende dalla variazione di luminosità, quindi maggiore è la variazione di luminosità della scena, maggiore sarà la frequenza di campionamento del modulatore presente all'interno di ogni pixel aumentandone la frequenza di invio di dati attraverso il protocollo AER, in modo tale da catturare l'immagine nel modo corretto. Tutti gli eventi sono campionati con una frequenza dell'ordine dei microsecondi e vengono trasmessi con una

latenza molto bassa (dell'ordine di microsecondi). Questi sono alcuni dei fattori che permettono alle camere ad eventi di avere bassa latenza e assenza di *motion blur*.
La quantità di luce che irradia un singolo pixel in una scena è composta da due parti: l'illuminazione della scena stessa e la riflettanza della luce come mostrato nella figura 1.4.

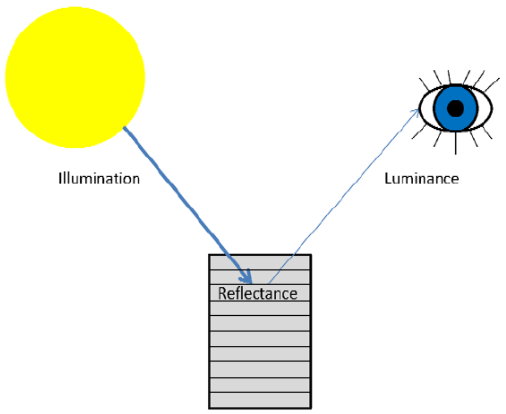

Figura 1.4: Riflettanza. Immagine tratta da [5].

La luce incidente su un pixel è il prodotto tra l'illuminazione della scena e la riflettanza delle superfici.
Accade però che la variazione logaritmica dell'intensità di luce della scena dipende soltanto della riflettanza delle superfici in quanto la luminosità della scena è in genere costante e il logaritmo del prodotto corrisponde alla somma dei logaritmi. Quindi le variazioni della riflettanza sono dovute in genere al movimento delle superfici all'interno della scena.

1.4.4 Comparazione della larghezza di banda di un pixel DVS e quella di una camera tradizionale

Come già detto in precedenza, i dati relativi agli eventi vengono trasmessi su un bus che ha una certa larghezza di banda. Quindi nonostante i pixel DVS siano veloci, si potrebbero avere dei problemi nel momento in cui l'immagine varia troppo frequentemente e di conseguenza, il fotoricettore può perdere alcune delle variazioni. Nelle camere tradizionali invece, un concetto speculare si ritrova nel tempo di esposizione.
In fotografia infatti il tempo di esposizione o velocità di otturazione, è il tempo durante il quale la luce passa attraverso l'otturatore della fotocamera per permettere poi ad essa di raggiungere la pellicola (nelle fotocamere analogiche) o il sensore (nelle fotocamere digitali). È un parametro molto importante perché il tempo di esposizione in combinazione con il diaframma (che è un meccanismo che regola la quantità di luce che passa attraverso l'obiettivo), determina la luminosità che è presente nelle foto che vengono scattate. La figura 1.5 mostra una foto sovraesposta mentre la figura 1.6 una sottoesposta.

Figura 1.5: Fotografia sovraesposta. Immagine tratta da [12].

Figura 1.6: Fotografia sottoesposta. Immagine tratta da [12].

La prima fotografia è stata scattata con un tempo di esposizione di 2 millisecondi, mentre la seconda è stata scattata con una velocità di otturazione di un terzo di secondo, pari a circa 330 millisecondi.
Si può quindi tracciare un collegamento tra le due tecnologie: la frequenza nell'alternarsi dei segnali ON e OFF (rispettivamente variazione di luminosità crescente e decrescente) delle camere ad eventi è il reciproco del tempo di esposizione delle camere tradizionali.
Il sistema di misurazione mostrato nella figura 1.7a, è composto da un pixel DVS che cattura la luce emessa da un LED la cui intensità varia in base ad un segnale sinusoidale.
Nella figura 1.7b, vengono graficati il numero di eventi per ciclo di un pixel DVS in funzione della frequenza degli stimoli. Alle basse frequenze, il pixel DVS produce una certa quantità di eventi per ciclo. All'aumentare della frequenza fino a quella di *cut-off* (nella figura è intorno ai 200Hz), ci si accorge che il numero di eventi per ciclo cala drasticamente e quindi alcune variazioni vengono filtrate dal fotoricettore del DVS. La frequenza di *cut-off* è una funzione monotona crescente dell'intensità della luce. Come si nota dalla figura, la curva relativa ad un'illuminazione alta ha un frequenza di taglio più alta rispetto a quella che si ha con un'illuminazione mille volte più bassa. Nel primo caso la larghezza di banda del pixel DVS è di circa 3kHz, equivalente ad un tempo di esposizione di 300 µs mentre nel secondo (intensità di luce 1000 volte minore) la larghezza di banda è ridotta a 300Hz.
Si può notare come il pixel DVS anche le caso peggiore dove la luminosità è 1000 volte inferiore, abbia una frequenza di risposta dieci volte più alta rispetto alla frequenza di Nyquist di 30 Hz relativa ad una camera standard che opera a 60 fps.

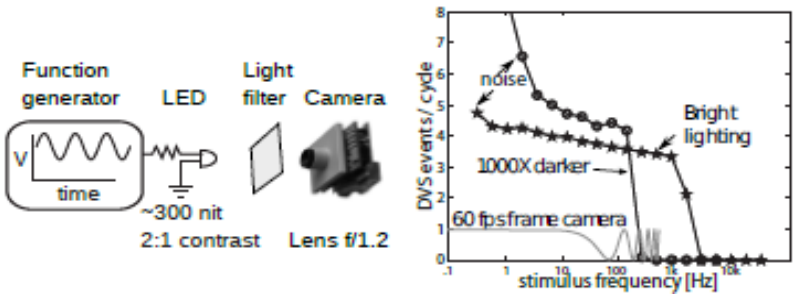

Figura 1.7: **(a)**: Scenario di misurazione del luce attraverso un pixel DVS. In **(b)**: Le risposte delle due tipologie di camere a diversi livelli di illuminazione. Immagine tratta da [3].

1.4.5 Modelli

Il primo modello di camera ad eventi è stata sviluppata da Mahowald e Mead al Caltech negli anni che vanno dal 1986-1992 nel loro tesi di dottorato premiato con il premio Clauser[3]. Il sensore utilizzava il protocollo AER per trasportare gli eventi in uscita ed aveva pixel logaritmici con tre strati di retina Kufler. Malgrado ciò, aveva delle carenze:

- la camera aveva dei pixel molto grandi che la rendevano poco utilizzabile nella pratica;
- ogni superficie di retina avvolta da cavi aveva bisogno di una particolare tensione di bias che doveva essere impostata in modo molto preciso con l'uso di potenziometri;
- ogni pixel aveva una risposta molto differente l'uno con l'altro;

Un modello molto più performante del precedente è la camera DVS. Essa è basata su una struttura a retina di silicio composta da due parti accoppiate tra loro: un fotoricettore a tempo continuo e un circuito di lettura resettato ad ogni campionamento di un pixel. Malgrado molti problemi possano essere risolti con l'analisi degli eventi generati da una camera DVS (ad esempio esaminando i cambiamenti di luminosità), alcuni necessitano un output statico (ad esempio la luminosità "assoluta").
L'**ATIS** (*Asynchronous Time Based Image Sensor*) (Figura 1.8), propone una struttura diversa: ogni pixel è composto da un *subpixel* DVS che a sua volta attiva un altro *subpixel* che legge l'intensità assoluta di luce. Il *trigger* resetta un condensatore attraverso un alto voltaggio. La carica di esso viene fatta diminuire attraverso un fotodiodo. Maggiore è la luce, maggiore è la velocità con il quale il condensatore si scarica. L'intensità letta dall'ATIS fornisce due eventi codificando il tempo che passa tra i due voltaggi di soglia. In questo modo solo il pixel che cambia fornisce i nuovi valori di intensità. Maggiore è il cambiamento dell'illuminazione, più corto è il tempo tra questi due eventi.
L'ATIS ha un notevole range dinamico statico (maggiore di 120dB) ma ad ogni modo presenta degli svantaggi. Il pixel ATIS ha un'area almeno doppia rispetto al pixel DVS e inoltre, nelle scene in oscurità, il tempo tra due eventi potrebbe essere molto lungo e la lettura dell'intensità degli stessi potrebbe essere interrotta da nuovi eventi che si presentano nel frattempo.

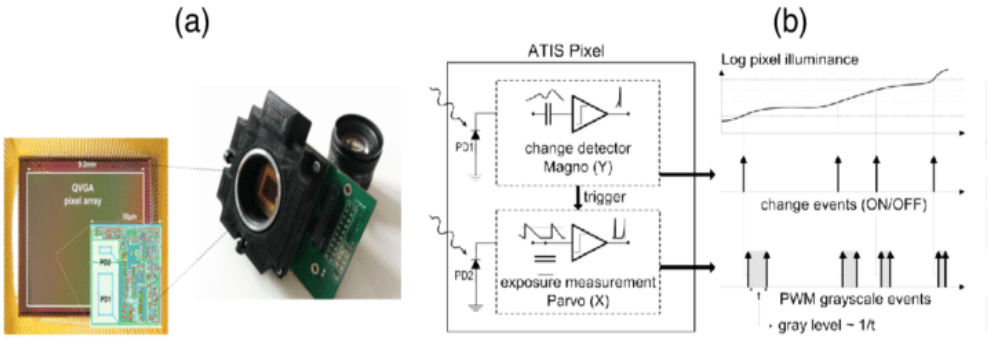

Figura 1.8: **(a)**: Il sensore ATIS. Immagine tratta da [6].

Ultimo ma non per importanza è il **DAVIS** (*Dynamic and Active Pixel Vision Sensor*) che combina un convenzionale *"Active Pixel Sensor"* (**APS**) e un pixel DVS insieme nello stesso pixel. Il vantaggio è che il circuito di lettura aumenta solo del 5% l'area della superficie del pixel DVS mantenendone le dimensioni contenute. Ad ogni modo, la frequenza di lettura dell'APS ha un limitato range dinamico (55dB) e come nelle camere standard, è ridondante se i pixel non cambiano.

1.4.6 Vantaggi delle camere ad eventi

Rispetto alle camere tradizionali, le camere ad eventi offrono dei vantaggi significativi:

- *Alta Risoluzione Temporale*: il monitoraggio dei cambiamenti della luminosità è veloce nella circuiteria analogica, mentre la lettura degli eventi è digitale seguendo un *clock* di 1 Mhz (ad esempio, gli eventi vengono campionati e rilevati con una risoluzione dell'ordine dei microsecondi). Quindi le camere ad eventi riescono ad evitare il *motion blur*, tipico disturbo delle camere tradizionali quando si acquisiscono scene con movimenti molto veloci;

- *Bassa Latenza*: nelle camere ad eventi, ogni pixel lavora in maniera indipendente e quindi non c'è bisogno di aspettare il tempo di esposizione dell'intero frame come accade con le camere standard: appena la variazione della luminosità supera la soglia, essa viene rilevata e l'evento viene trasmesso in un tempo molto ristretto. Infatti la latenza è molto bassa (intorno ai 10 µs);

- *Bassa Energia*: con l'utilizzo di questa tecnologia si evita la trasmissione di dati ridondanti in quanto (come detto sopra) solo le variazioni di luminosità generano la trasmissione di dati e quindi l'energia viene impiegata solo per

processare i pixel in cambiamento.
Per questo motivo, le camere ad eventi hanno consumi molto ridotti (intorno ai 10 µW);

- *High Dynamic Range (HDR).* Rispetto alle camere basate sui *frames*, che hanno un range dinamico di 60dB, le camere ad eventi superano i 120dB che rende possibile l'acquisizione e quindi il riconoscimento di oggetti anche in condizioni di luce molto sfavorevoli sia di giorno che di notte. Questo è possibile grazie al fatto che i fotoricettori lavorano in scala logaritmica e che ogni pixel lavora in maniera indipendente.

1.4.7 Modello di generazione degli eventi

In questa sezione verranno illustrati i modelli matematici che vengono utilizzati dalle camere ad eventi nella generazione degli eventi.
Si indica con $L \doteq \log(I)$ la luminosità, calcolata come il logaritmo della fotocorrente I generata da un fotoricettore quando la luce impatta sul singolo pixel indipendente della camera.
In assenza di disturbi, un evento $e_k \doteq (x_k, t_k, p_k)$ è innescato al pixel $x_k \doteq (x_k, y_k)^\top$ in un tempo t_k non appena l'incremento di luminosità dell'ultimo evento del pixel definito nel modo seguente,

$$\Delta L(x_k, t_k) \doteq L(x_k, t_k) - L(x_k, t_k - \Delta t_k) \tag{1.1}$$

raggiunge la *contrasto di soglia* $\pm C$,

$$\Delta L(x_k, t_k) = p_k C \tag{1.2}$$

dove C > 0, Δt_k è il tempo trascorso dall'ultimo evento dello stesso pixel, e la polarità $p_k \in \{+1, -1\}$ è il segno del cambiamento della luminosità.

1.4.7.1 Il contrasto

La variabile C è determinata dalle correnti di bias del pixel e sono prodotte da un generatore di bias programmato digitalmente nel chip. Il contrasto C può essere stimato tramite la conoscenza di queste correnti di bias.
Nelle tipiche camere DVS, le soglie vengono impostate tra il 10% e il 50% della variazione dell'illuminazione.
Ci sono anche delle applicazioni delle camere DVS con un alto *gain* dei fotoricettori che operano a soglie molto basse (anche dell'1%). Ad ogni modo, esse vengono usate in circostanze molto favorevoli con condizioni meteo ideali e con un'illuminazione molto chiara.
Il limite inferiore del contrasto di soglia è calcolato in base ai disturbi e alla variabilità tra pixel e pixel. Impostare un valore di C troppo basso comporta un

aumento dei disturbi degli eventi che nascono da pixel con valori bassi di C.

1.4.7.2 Eventi e la derivata rispetto al tempo della luminosità

In un piccolo intervallo di tempo Δt_k, un incremento della luminosità può essere approssimato utilizzando l'espansione di Taylor,

$$\Delta L(x_k, t_k) \approx \frac{\partial L}{\partial t}(x_k, t_k)\Delta t_k \tag{1.3}$$

e quindi l'equazione (1.2) può essere formulata nel modo seguente:

$$\frac{\partial L}{\partial t}(x_k, t_k) \approx \frac{p_k C}{\Delta t_k} \tag{1.4}$$

È importante notare che gli eventi della camera DVS sono scatenati da una variazione nella magnitudo della luminosità e non dal superamento della soglia da parte della derivata nel tempo della luminosità.

1.4.7.3 Causa della generazione degli eventi

Assumendo che l'illuminazione sia costante linearizzando l'equazione (1.2) e assumendo che la luminosità sia costante, si può mostrare che gli eventi sono causati dal movimento dei bordi.
Per piccoli valori Δt, l'incremento dell'intensità (equazione (1.2)) può essere approssimato nel modo seguente:

$$\Delta L \approx -\nabla L \cdot v\Delta t \tag{1.5}$$

che è causato da un gradiente della luminosità $\nabla L(x_k, t_k) = (\frac{\partial}{\partial x}L, \frac{\partial}{\partial y}L)^\top$ che si muove con una velocità $v(x_k, t_k)$ sul piano dell'immagine con uno spostamento $\Delta x \doteq v\Delta t$.
Nell'equazione (1.5), il prodotto scalare ci indica che se il movimento è parallelo al bordo nessun evento è generato dal momento in cui $v \cdot \Delta L = 0$. Se invece, il movimento è perpendicolare al bordo, (v $||$ ΔL) gli eventi sono generati con frequenza più alta.

Capitolo 2

Strumenti utilizzati

2.1 Robot Operating System (ROS)

2.1.1 Concetti introduttivi

Uno degli strumenti più utilizzati nel corso del lavoro svolto in questa tesi è il ROS (*Robot Operating System*)[14]. In seguito ne verranno illustrate le caratteristiche e componenti principali con alcuni esempi pratici di utilizzo.

Figura 2.1: Logo del ROS. Immagine tratta da [11].

Il ROS è un *framework* utilizzato nello sviluppo e programmazione di robot. È stato sviluppato dall'università di Stanford nel 2007.
È caratterizzato da quattro elementi fondamentali [14]:

- ***Plumbing***: permette l'esecuzione di più programmi contemporaneamente e ne gestisce la comunicazione tra di loro;
- ***Tools***: il *framework* è composto da una serie di strumenti per la visualizzazione dei dati, creazione di interfacce utente (GUI) ed esecuzione di simulazioni complesse;
- ***Capabilities***: sono delle funzionalità già scritte che permettono di svolgere le simulazioni. Ci sono funzioni di *mapping*, controllo, pianificazione e manipolazione dei dati;

- **Ecosistema**: il ROS fornisce dei tutorial molto dettagliati che consentono l'utilizzo ottimale del *framework*. Sia le librerie che gli strumenti di sviluppo sono scritti con diversi linguaggi di programmazione e consente un'organizzazione del codice e dei files in *packages*;

2.1.2 Caratteristiche

Il ROS ha delle caratteristiche molto interessanti che hanno permesso il suo ampio utilizzo nel mondo della robotica. Esso è un *framework*:

- ***peer to peer***: come già anticipato sopra, il *framework* è organizzato in modo tale che i vari processi in esecuzione (*ROS node*, *ROS service* ecc.) comunichino tra di loro attraverso delle API già definite;
- **distribuito**: i programmi possono comunicare tra di loro attraverso una rete di elaboratori e comunicare tra di loro;
- **multilinguaggio**: i moduli che compongono il ROS sono scritti in vari linguaggi di programmazione (C++, Python ecc.);
- **leggero**: ROS sfrutta librerie già scritte nei vari linguaggi di programmazione rendendolo di fatto molto performante;
- **gratis e *open source***: il software è distribuito con licenza libera;

2.1.3 Catkin

Catkin è il *build system* creato appositamente per il ROS che è utilizzato per creare librerie, eseguibili ed interfacce a partire da codice sorgente.
Esso sostituisce un vecchio *build system* esistente (il *rosbuild*).
Catkin combina delle marco del *CMake* e script Python per fornire delle funzionalità aggiuntive al *CMake* stesso. Esso è stato progettato per essere più convenzionale rispetto al *rosbuild*, fornendo una maggiore distribuzione dei *packages*, un migliore supporto *cross-compiling* ed una portabilità più estesa.
Per effettuare il *build*, il *build system* ha bisogno di alcune informazioni come la posizione e le dipendenze del codice sorgente, la definizione delle dipendenze esterne, dove sono posizionate, quali elementi devono essere compilati e dove devono essere installate. Queste informazioni vengono inserite in file di configurazione letti dal *build system*. In un IDE, questi dati vengono memorizzati come metadati che fanno parte del *workspace* o del progetto. Con il *CMake*, viene utilizzato un file .txt chiamato *CMakeLists.xml* mentre in *GNU Make* viene utilizzato un file chiamato *Makefile*. Il *build system* quindi, utilizza queste informazioni per generare i codici eseguibili, librerie ecc.
Il ROS utilizza un proprio *build system* in quanto i *tools* già esistenti come *CMake*, *Autotools* sono troppo complessi da utilizzare per le applicazioni di interesse

del *framework*. Quindi l'obiettivo del Catkin è quello di compilare ed eseguire codice in maniera più semplice e veloce.
Per installare Catkin, bisogna eseguire questo comando da con il terminale Linux (tutto il lavoro della tesi è stato svolto con il sistema operativo Ubuntu 16.04 e il ROS nella sua versione chiamata *kinetic*:

```
1 >sudo apt-get install ros-kinetic-catkin
```

Per creare un *workspace* di Catkin, sono necessari pochi comandi da eseguire con il terminale Linux (i comandi verranno mostrati nel capitolo 4).

Catkin è composto da tre cartelle fondamentali:

- ***src***: contiene il codice sorgente. È la cartella che si utilizza per poter creare, clonare ed editare codice sorgente per generare le risorse di cui abbiamo bisogno;
- ***build***: è la cartella dove il *CMake* viene chiamato per generare i *packages* a partire dai codici sorgente;
- ***devel***: è la cartella dove i *files* compilati sono memorizzati;

Con il comando:

```
1 >catkin clean
```

è possibile pulire l'intero contenuto delle cartelle *build* e *devel*.
Se vogliamo controllare o modificare delle impostazioni di catkin, è possibile tramite il comando:

```
1 >catkin config
```

2.1.4 ROS Master

Il *ROS Master* gestisce la comunicazione tra i vari processi in esecuzione all'interno del *framework*. Esso controlla che tutti i nodi comunichino correttamente tra di loro.
Il *ROS Master* si avvia con il comando:

```
1 >roscore
```

Ogni processo, si registra al suo avvio col *ROS master*.
Nella figura 2.2, viene riportata la schermata principale del *ROS Master*.

```
paolo@paolo-VirtualBox:~$ roscore
... logging to /home/paolo/.ros/log/9f79f614-9186-11ea-8a1e-080027a2bc46/roslaunch-paolo-VirtualBox-3218.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://paolo-VirtualBox:36029/
ros_comm version 1.12.14


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [3242]
ROS_MASTER_URI=http://paolo-VirtualBox:11311/

setting /run_id to 9f79f614-9186-11ea-8a1e-080027a2bc46
process[rosout-1]: started with pid [3255]
started core service [/rosout]
```

Figura 2.2: Schermata di avvio del *ROS Master*.

2.1.5 ROS Node

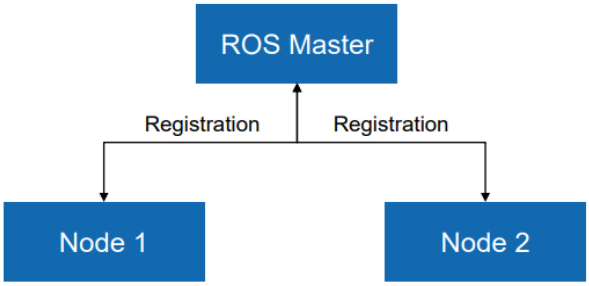

Figura 2.3: *ROS Node* e la comunicazione con il *ROS Master*. Immagine tratta da [14].

Un *ROS Node*, è un processo eseguibile con uno scopo specifico. Sono gestiti individualmente, compilati ed eseguibili. Sono organizzati in *packages*.
Un *ROS Node*, si avvia tramite il seguente comando:

```
1 >rosrun nome_del_package nome_del_rosnode
```

È possibile visualizzare la lista di tutti i nodi attivi con il comando:

```
1 >rosnode list
```

Infine, si possono ottenere le informazioni su un nodo specifico con il comando:

```
1 >rosnode info nome_del_rosnode
```

Ogni nodo è registrato con il *ROS Master* per permettere la comunicazione tra il *ROS Master* e ogni nodo attivo in quel determinato momento.

2.1.6 ROS Topics

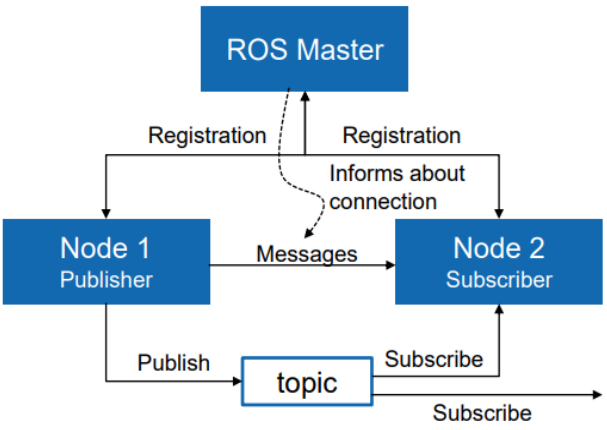

Figura 2.4: *ROS Master*, *Node* e *Topic* in comunicazione tra loro. Immagine tratta da [14].

I nodi comunicano tra di loro mediante l'utilizzo dei *ROS Topics*. Un nodo può svolgere la funzione di *publisher* o di *subscriber*. Tipicamente si ha un solo nodo *publisher* ed n nodi *subscribers*.
In altre parole, un *ROS Topic* è il nome per un flusso di messaggi scambiati dai vari nodi in esecuzione.
Analogamente a quanto visto per i nodi, è possibile ottenere una lista dei *topics* attivi con il comando:

```
>rostopic list
```

Con il seguente comando, è possibile stampare sul terminale i contenuti di un determinato *topic*:

```
>rostopic echo /nome_del_topic
```

Infine, per mostrare le informazioni di un *topic* è necessario avviare il comando:

```
>rostopic info /nome_del_topic
```

Riassumendo, il nodo *publisher* pubblica i suoi contenuti all'interno del *topic* mentre tutti i nodi *subscriber* si iscrivono al *topic* e riceverono i messaggi inviati dal nodo *publisher*. Questo meccanismo è illustrato nella figura 2.4.

2.1.7 ROS Messages

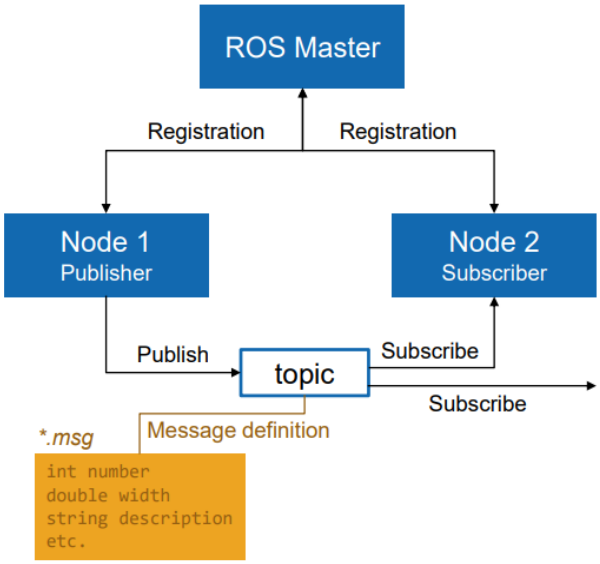

Figura 2.5: *ROS Master*, *Node*, *Topic* e *Message* in comunicazione tra loro. Immagine tratta da [14].

Il *ROS Message* è una struttura dati che definisce il tipo di un determinato *ROS Topic*.
Esso è composto da una struttura nidificata di variabili intere, stringhe, *float*, *boolean* o oggetti come è mostrato nella figura 2.5.
Un *ROS Message* è definito all'interno di un file con estensione **.msg*.
Per visualizzare il tipo di un determinato *ROS Message*, si usa il comando:

```
1 >rostopic type /nome_del_topic
```

Per pubblicare un messaggio su un *topic* invece, bisogna utilizzare il seguente comando:

```
1 >rostopic pub /nome_del_topic type data
```

2.1.8 Esempio di utilizzo del ROS

Utilizzeremo degli strumenti già forniti all'interno del ROS come il nodo *talker* e *chatter* per mettere in pratica i concetti esposti sopra.
Per prima cosa, bisogna avviare il *ROS Master* come il comando *roscore*.
Successivamente, si avvia il nodo *talker* che è un nodo *publisher* che come detto sopra, pubblicherà dei messaggi su un *ROS Topic*. Il nodo *talker* pubblica dei messaggi in modo ripetitivo composti da una stringa "Hello World" seguita da

un numero progressivo che aumenta ad ogni ripetizione.
Il nodo *talker* è contenuto all'interno del *package roscpp_tutorials*.
Per avviare il nodo *talker*, bisogna eseguire il seguente comando:

```
>rosrun roscpp_tutorials talker
```

Di seguito nella figura 2.6 ne è mostrato l'output nel terminale.

```
paolo@paolo-VirtualBox:~$ rosrun roscpp_tutorials talker
[ INFO] [1589018558.132263863]: hello world 0
[ INFO] [1589018558.232775006]: hello world 1
[ INFO] [1589018558.332426611]: hello world 2
[ INFO] [1589018558.433054536]: hello world 3
[ INFO] [1589018558.532541126]: hello world 4
[ INFO] [1589018558.632971178]: hello world 5
[ INFO] [1589018558.732378947]: hello world 6
[ INFO] [1589018558.832972936]: hello world 7
[ INFO] [1589018558.932558613]: hello world 8
[ INFO] [1589018559.033474123]: hello world 9
[ INFO] [1589018559.132743579]: hello world 10
[ INFO] [1589018559.233121546]: hello world 11
[ INFO] [1589018559.332670427]: hello world 12
[ INFO] [1589018559.433269484]: hello world 13
[ INFO] [1589018559.532833169]: hello world 14
[ INFO] [1589018559.632428594]: hello world 15
[ INFO] [1589018559.733555231]: hello world 16
[ INFO] [1589018559.833065298]: hello world 17
[ INFO] [1589018559.932891271]: hello world 18
[ INFO] [1589018560.033208334]: hello world 19
```

Figura 2.6: Output del nodo *talker*.

Il nodo i*talker* pubblica i suoi messaggi su un *topic* chiamato *chatter* di tipo *string*.

Infatti, digitando il comando (figura 2.7)

```
>rosnode info /talker
```

si può vedere che tra le *Publications* del nodo, è presente */chatter* seguito dall'indicazione sul tipo del *topic*, ovvero *[std_msgs/String]*.

```
paolo@paolo-VirtualBox:~$ rosnode info /talker
--------------------------------------------------------------------------------
Node [/talker]
Publications:
 * /chatter [std_msgs/String]
 * /rosout [rosgraph_msgs/Log]

Subscriptions: None

Services:
 * /talker/get_loggers
 * /talker/set_logger_level
```

Figura 2.7: Informazioni del nodo *talker*.

Il *topic chatter* non ha in questo momento alcun *subscriber* come si può vedere dalla figura 2.8 ma ha un nodo *publisher*, ovvero *talker*.

Digitando il comando

```
>rosrun info /chatter
```

visualizziamo le informazioni del *topic chatter*.

```
paolo@paolo-VirtualBox:~$ rostopic info /chatter
Type: std_msgs/String

Publishers:
 * /talker (http://paolo-VirtualBox:32949/)

Subscribers: None
```

Figura 2.8: Informazioni del *topic chatter*.

Se digitiamo il comando:

```
>rostopic type /chatter
```

riceviamo in output da terminale il tipo del *topic chatter*: *std_msgs/String*.

È possibile visualizzare sul terminale la frequenza con cui vengono emessi i messaggi nel *topic chatter*. Ciò è possibile tramite il comando:

```
>rostopic hz /chatter
```

Ora è necessario avviare il secondo nodo chiamato *listener* che fungerà da nodo *subscriber*. Si avvia con il seguente comando:

```
>rosrun roscpp_tutorials listener
```

Una volta avviato, il *topic chatter* avrà *listener* come nodo *subscriber*, come mostrato nella figura 2.9.

```
paolo@paolo-VirtualBox:~$ rostopic info /chatter
Type: std_msgs/String

Publishers:
 * /talker (http://paolo-VirtualBox:41887/)

Subscribers:
 * /listener (http://paolo-VirtualBox:38987/)
```

Figura 2.9: Output del nodo *listener*.

Listener mostra una stringa "I heard:" seguita dal messaggio pubblicato sul *topic chatter* dal nodo *publisher talker* (Figura 2.10).

```
paolo@paolo-VirtualBox:~$ rosrun roscpp_tutorials listener
[ INFO] [1589022137.357044173]: I heard: [hello world 218]
[ INFO] [1589022137.457571174]: I heard: [hello world 219]
[ INFO] [1589022137.557584496]: I heard: [hello world 220]
[ INFO] [1589022137.657545023]: I heard: [hello world 221]
[ INFO] [1589022137.757543642]: I heard: [hello world 222]
[ INFO] [1589022137.857511628]: I heard: [hello world 223]
[ INFO] [1589022137.957738358]: I heard: [hello world 224]
[ INFO] [1589022138.058468827]: I heard: [hello world 225]
[ INFO] [1589022138.158599356]: I heard: [hello world 226]
[ INFO] [1589022138.256926302]: I heard: [hello world 227]
[ INFO] [1589022138.356937456]: I heard: [hello world 228]
[ INFO] [1589022138.456824613]: I heard: [hello world 229]
[ INFO] [1589022138.557172289]: I heard: [hello world 230]
[ INFO] [1589022138.656571890]: I heard: [hello world 231]
[ INFO] [1589022138.757250059]: I heard: [hello world 232]
[ INFO] [1589022138.859637363]: I heard: [hello world 233]
[ INFO] [1589022138.957161171]: I heard: [hello world 234]
```

Figura 2.10: Output del nodo *listener*.

L'ultimo comando di questo esempio, permette di pubblicare un messaggio sul *topic chatter* di tipo string "ciao":

```
>rostopic pub /chatter std_msgs/String "data:'ciao'"
```

Questo è l'output sul terminale (è ancora in esecuzione in nodo *talker* quindi il nostro messaggio compare in mezzo a quelli inviati dal nodo *talker*):

```
[ INFO] [1589026127.509239156]: I heard: [hello world 488]
[ INFO] [1589026127.608185123]: I heard: [hello world 489]
[ INFO] [1589026127.708541422]: I heard: [hello world 490]
[ INFO] [1589026127.744074695]: I heard: [data:'ciao']
[ INFO] [1589026127.808995401]: I heard: [hello world 491]
```

Figura 2.11: Output del nodo *listener*.

2.1.9 ROS Launch

ROS Launch è un *tool* che permette di eseguire più nodi. Sono scritti in XML con estensione **.launch*.
Se non è già stato fatto, all'avvio di un *ROS Launch* viene avviato anche il *ROS Master*.
Il comando per eseguire un file *.launch* è:

```
>roslaunch nome_del_file.launch
```

Se il file *.launch* è all'interno di un *package*, si usa:

```
>roslaunch nome_package nome_del_file.launch
```

2.1.9.1 Struttura ed esempio di un file launch

Un esempio di un file *.launch* in grado di permettere l'esecuzione del nodo *listener* e *talker* è il seguente:

```
<launch>
    <node name="listener" pkg="roscpp_tutorials" type="listener"
        output="screen"/>
    <node name="talker" pkg="roscpp_tutorials" type="talker"
        output="screen"/>
</launch>
```

dove:

- **<launch>**: è il *tag* che identifica un *launch file*;
- **<node>**: è un *tag* che specifica un nodo da lanciare;
- **name**: identifica il nome del nodo;
- **pkg**: identifica il nome del *package* che contiene il nodo;
- **type**: identifica il tipo di nodo (deve corrispondere ad un eseguibile con lo stesso nome);
- **output**: specifica dove depositare i messaggi di log;

2.1.10 ROS Packages

Il software che compone il ROS è organizzato in *packages*. Essi contengono codici sorgente, *launch files*, file di configurazione, definizione di messaggi, dati e documentazioni.
Un *package* che richiede altri componenti per poter essere compilato ha bisogno di dichiarare le dipendenze con il seguente comando:

```
> catkin_create_pkg nome_del_package {dipendenze}
```

Un *package* può essere organizzato nel modo indicato in figura:

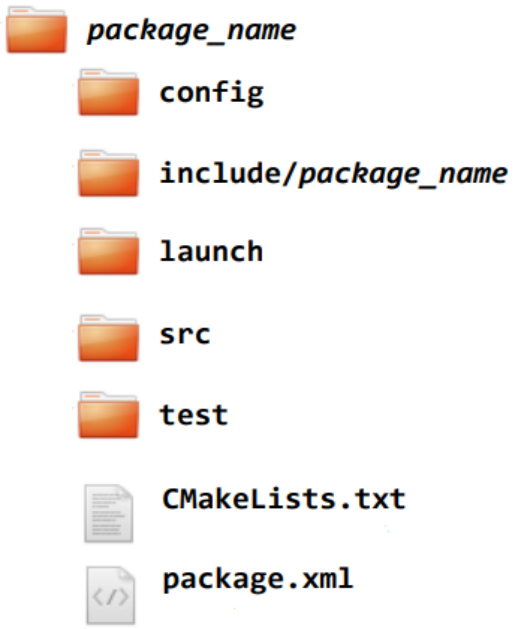

Figura 2.12: Esempio di un *package*. Immagine tratta da [14].

2.1.10.1 package.xml

Il *package.xml* definisce le proprietà del *package* come il nome, autori, versione e dipendenze con altri *package*.

2.1.10.2 CMakeLists.txt

Il *CMakeLists.txt* rappresenta il file di input che viene mandato al *CMakeBuild system*. Esso contiene:

- la versione del CMake richiesta (*cmake_minimum_required(VERSION ...)*);
- il nome del *package* (*project()*);
- indicazioni su dove trovare gli altri *packages Cmake* o Catkin necessari per la compilazione (*find_package()*);
- generatori di messaggi, servizi e azioni (*add_message_files()*, *add_service_files()* e *add_action_files()*);
- l'invocazione nella generazione di messaggi (*generate_messages()*);
- specifiche sui *package build info export* (*catkin_package()*);

- librerie o eseguibili da compilare (*(add_ library() add_ executable() target_ link_ libraries()*);
- test per la compilazione (*catkin_ add_ gtest()*);
- metodi per installare regole (*install()*);

2.1.11 ROS Services

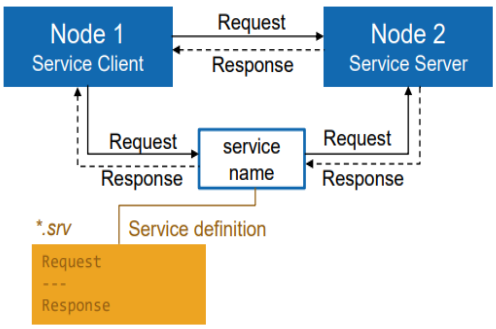

Figura 2.13: *ROS Service* in comunicazione con due nodi. Immagine tratta da [14].

La comunicazione *client/server* tra nodi è realizzata con il *ROS Service* attraverso l'utilizzo di due elementi principali:

- il *service server* che promuove il servizio;
- il *service client* che usufruisce del servizio;

I *ROS Services* vengono definiti in file di estensione *.srv.

Per ottenere la lista dei servizi disponibili, bisogna eseguire questo comando da terminale:

```
> rosservice list
```

In analogia con gli altri strumenti presentati fino ad ora, per visualizzare il tipo di servizio si utilizza il comando:

```
>rosservice type /nome_del_servizio
```

Per chiamare un servizio fornito da un *ROS Service* si utilizza:

```
>rosservice call /nome_del_servizio args
```

2.1.11.1 Esempio di utilizzo di un ROS Service

In questo esempio, verrà utilizzato un servizio molto semplice che permette la somma di due numeri a e b chiamato *add_two_ints*.

Per prima cosa si procede all'avvio del *service server* assicurandosi di aver prima eseguito il comando *roscore*:

```
>rosrun roscpp_tutorials add_two_ints_server
```

Ora il servizio è disponibile nella lista dei servizi attivi quindi posso vederne il tipo con il comando:

```
>rosservice type /add_two_ints
```

Un aspetto molto importante da tenere in considerazione del *ROS Service* è bisogna conoscere gli argomenti da fornire in input e ciò che viene restituito in output. Possiamo vederli con questo comando:

```
>rossrv show roscpp_tutorials/TwoInts
```

Il risultato è il seguente:

```
paolo@paolo-VirtualBox:~$ rossrv show roscpp_tutorials/TwoInts
int64 a
int64 b
---
int64 sum
```

Figura 2.14: Informazioni del servizio *add_two_ints*.

Si nota come nella prima parte sono indicate le variabili da fornire in argomento e nella seconda gli elementi forniti in output.
Quindi con il seguente comando, si chiama il servizio *add_two_ints* passandogli come argomento due numeri:

```
>rosservice call /add_two_ints "a:10     b:5"
```

È importante premere il tasto TAB per comporre nel modo corretto il comando.
Ciò che accade nel terminale è mostrato nella figura sottostante:

```
paolo@paolo-VirtualBox:~$ rosservice call /add_two_ints "a: 10
b: 5"
sum: 15
```

Figura 2.15: Risultato della chiamata del servizio *add_two_ints*.

2.1.12 ROS Actions

Le *ROS Actions* sono simili alle chiamate dei *ROS Services* ma forniscono delle funzionalità aggiuntive come:

- cancellare un task;
- ricevere un feedback sul progresso dell'attività;

Essi vengono memorizzati in files con estensione *.action*.

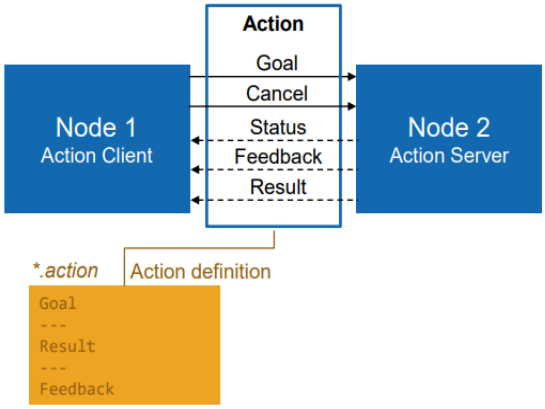

Figura 2.16: Comunicazione tra nodi e *ROS Actions*. Immagine tratta da [14].

2.1.13 ROS Time

Il *ROS Time* è uno strumento molto utile nelle simulazioni quando bisogna simulare l'andamento nel tempo di alcuni comportamenti.
Per lavorare con un *clock* simulato, si utilizza:

```
>rosparam set use_sim_time true
```

2.1.14 ROS Bags

L'ultimo elemento molto importante che verrà sfruttato in seguito nel capitolo dedicato alla generazione di eventi è il *ROS Bag*.
Un *Bag* è un formato utilizzato dal ROS per immagazzinare i messaggi. Sono in formato binario e sono memorizzati con estensione *.bag*.
Per registrare tutto ciò che viene inserito all'interno di tutti i *ROS Topics*, si utilizza il seguente comando:

```
>rosbag record --all
```

Mentre se vogliamo registrare tutti i messaggi che sono stati inviati in *topics* specifici, si utilizza:

```
>rosbag record topic_1 topic_2 topic_3
```

La registrazione viene interrotta con l'utilizzo di *ctrl+C*.
I file .bag vengono memorizzati con una data d'inizio e l'ora di memorizzazione ad esempio:
2020-02-07-10-30-00.bag.

Per visualizzare le informazioni di un bag si utilizza il comando:

```
>rosbag info nome_del_file.bag
```

Per leggere un file bag e stamparne tutti i suoi contenuti sul terminale si utilizza:

```
>rosbag play nome_del_file.bag
```

Esistono delle opzioni di visualizzazione che possono essere fornite dopo il comando *play*, ad esempio:

```
>rosbag play --rate=0.5 nome_del_file.bag
```

2.1.15 Rviz

Rviz (figura 2.17) è uno strumento che permette la visualizzazione 3D in ROS. Rviz si sottoscrive ai *topics* e ne visualizza i messaggi contenuti. Esso permette di visualizzare gli elementi di una scena 3D da più visuali della camera.
Si lancia con il seguente comando:

```
>rosrun rviz rviz
```

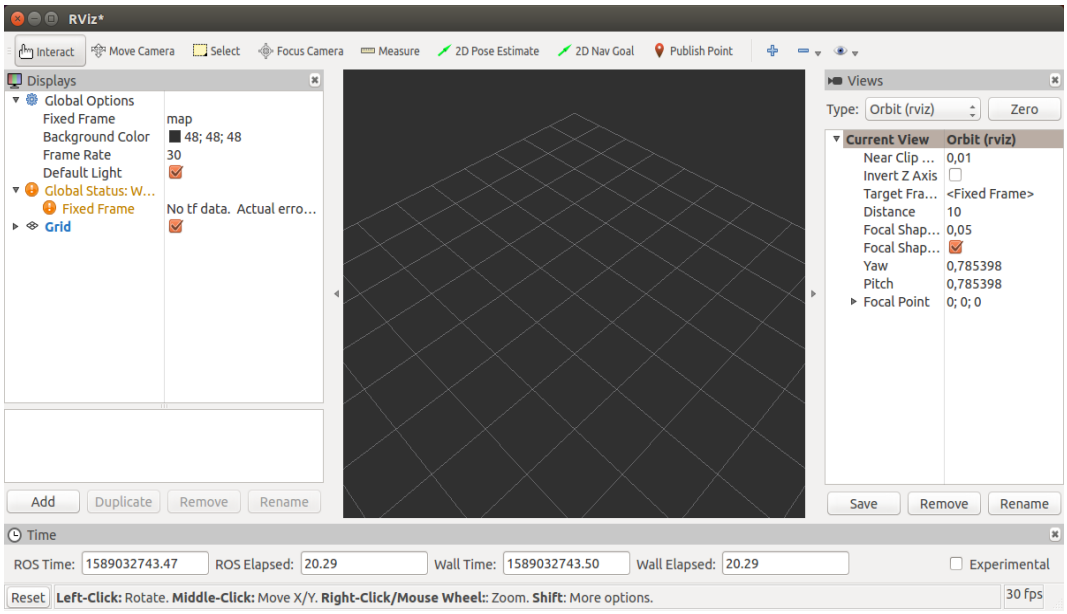

Figura 2.17: Schermata iniziale di Rviz.

2.1.16 Rqt Image View

Rqt Image View è un altro strumento di visualizzazione che verrà impiegato ampiamente nel capitolo 4 per le simulazioni effettuate da ESIM. Si lancia con il seguente comando:

```
1 >rosrun rqt_image_view
```

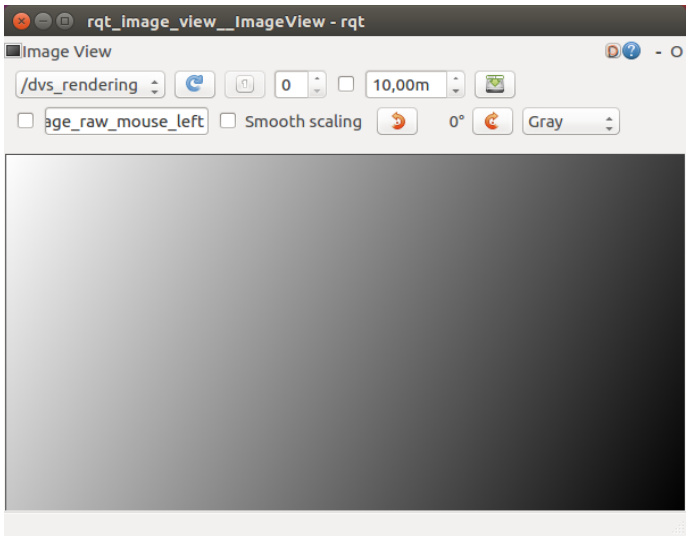

Figura 2.18: Schermata iniziale di Rqt Image View.

.

2.2 Blender e scena 3D utilizzata per la simulazione

In seguito verrà proposta una simulazione nella generazione di eventi a partire da uno scenario in 3D creato con il programma Blender. La scena comprende una ricostruzione di una stanza arredata in 3D come mostrato nella figura 2.19.

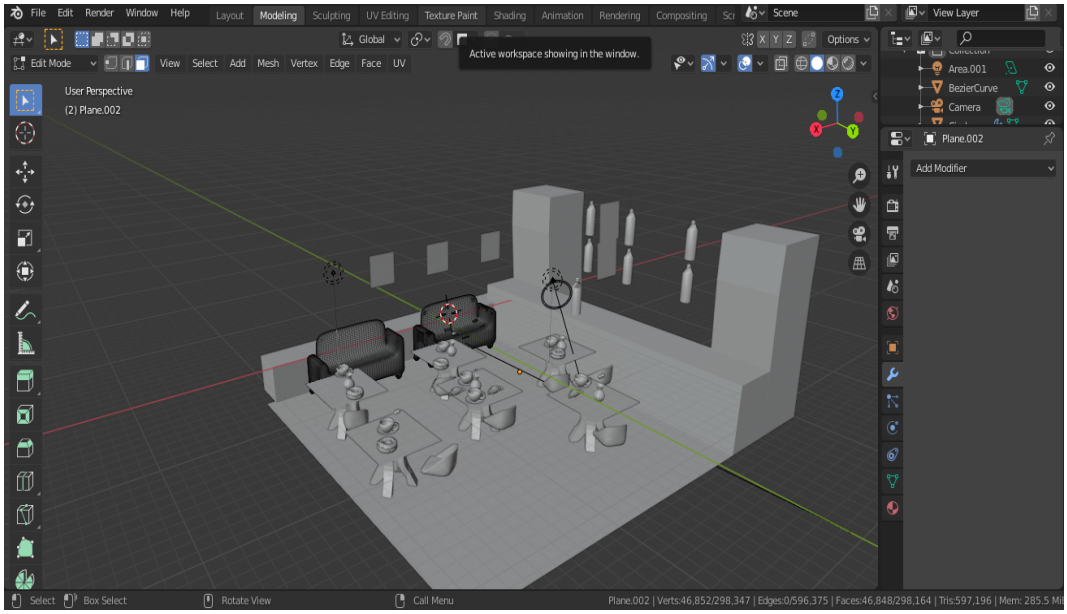

Figura 2.19: Scena in 3D creata con Blender. Modello 3D tratto da [2].

2.3 Installazione del ROS

In questa sezione verrà illustrata la procedura di installazione del ROS.
Per lo svolgimento del lavoro oggetto della tesi si è utilizzata la versione chiamata *Kinetic*, compatibile soltanto con le versioni 15.10 e 16.01 di Ubuntu.
Il primo passo è impostare il proprio PC per accettare il software che proviene dal *packages.ros.org* utilizzando il comando:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(
    lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Successivamente, si impostano le chiavi con:

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-
    key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Prima di procedere con l'installazione vera e propria, ci assicuriamo che l'indice del *package* Debian sia aggiornato:

```
sudo apt-get update
```

Avviamo l'installazione completa di ROS con il seguente comando:

```
sudo apt-get install ros-kinetic-desktop-full
```

Fatto ciò è conveniente impostare le variabili d'ambiente del ROS in modo tale da richiamarle automaticamente nella sessione *bash* ogni volta che essa è avviata. Per fare questo, si digitano i seguenti comandi:

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Arrivati a questo punto si è installato tutto ciò che necessario per avviare i *packages* fondamentali del ROS ma se si vuole permettere la creazione di *workspaces custom*, bisogna installare questo *tool* tramite il seguente comando da terminale:

```
sudo apt install python-rosdep python-rosinstall python-rosinstall-
    generator python-wstool build-essential
```

Prima di iniziare ad usare i *tools* presenti nel ROS, c'è bisogno di inizializzare il *rosdep* che permette di installare le dipendenze del codice che si vuole compilare ed è fondamentale per eseguire i componenti del ROS. *Rosdep* si installa con il seguente comando:

```
sudo apt install python-rosdep
```

e si inizializza con i seguenti comandi:

```
sudo rosdep init
rosdep update
```

2.4 Python

Figura 2.20: Il logo di Python. Immagine tratta da [1].

Per effettuare la simulazione degli eventi a partire da un video con ESIM, è necessario utilizzare il linguaggio di programmazione Python per creare un piccolo script di nome *generate_stamps_file.py*.

2.5 FFmpeg

Figura 2.21: Logo FFmpeg. Immagine tratta da [9]

Nella simulazione degli eventi a partire da un file video con ESIM, è stato utile l'utilizzo di un software per l'elaborazione di file video chiamato *FFmpeg*.
In particolare è stato utilizzato per estrapolare i *frames* in formato *.png* dai due video generati da ESIM dopo la simulazione, ossia quello con i *frames* APS e quello che contiene la simulazione degli eventi.

2.6 OpenGL

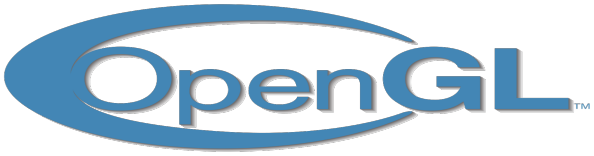

Figura 2.22: Logo OpenGL. Immagine tratta da [10]

Nel capitolo 4, verrà effettuata una simulazione di eventi a partire da una scena virtuale creata con Blender. Si utilizzerà il simulatore ESIM insieme al motore grafico *OpenGL*. L'obiettivo sarà quello di abbinare le immagini della scena virtuale renderizzate dal motore grafico con la relativa simulazione degli eventi generata da una camera in movimento nella scena utilizzando il simulatore ESIM.

2.7 Descrizione del file video usato

Nel capitolo 4, è stato utilizzato un file video in formato .mp4 registrato con la camera ferma fissata in alto, mentre alcune persone camminano da sinistra verso destra fermandosi per un paio di secondi al centro della scena per poi tornare a camminare verso destra uscendo dall'inquadratura.

Capitolo 3

Il simulatore ESIM

3.1 Concetti introduttivi

Come abbiamo ampiamente approfondito nei capitoli precedenti, le camere ad eventi hanno caratteristiche innovative che aprono le porte a nuove sfide nel campo della ricerca nella progettazione e realizzazione di nuovi algoritmi che ne accentuino le loro straordinarie possibilità di utilizzo.
Purtroppo il loro prezzo sul mercato è ancora elevato ed hanno una risoluzione molto bassa (come il sensore DAVIS346 che costa circa 6000$ ed ha una risoluzione di 346X260).
Inoltre, le attuali camere ad eventi hanno notevoli limitazioni come ad esempio un rapporto segnale/rumore molto basso e necessitano di complesse configurazioni e di conseguenza richiedono l'intervento di esperti per il loro corretto utilizzo.
Questi problemi rendono difficile la comprensione del vero potenziale delle camere ad eventi per l'impiego in vari campi, come ad esempio la robotica.
Inoltre in parallelo sono stati fatti numerosi passi avanti nel *deep learning* che hanno portato ad una crescita esponenziale della domanda di dati. Per soddisfare questa domanda, in passato sono stati creati diversi simulatori come (CARLA, Microsoft Airsim, UnrealCV)[4] che sono stati impiegati anche per le camere standard. Tuttavia nel campo della ricerca c'è ancora la necessità di un simulatore delle camere ad eventi che sia accurato, efficiente e scalabile.
A causa delle differenze notevoli nel funzionamento delle camere ad eventi rispetto al quelle tradizionali, lo sviluppo di un simulatore non è triviale. Una delle caratteristiche di rilievo risiede nella quantità di dati in uscita da una camera ad eventi che dipende dal movimento nella scena in esame. Le camere standard registrano sempre la stesso numero di *frames* ad una frequenza stabilita indipendentemente dal tipo di scenario ripreso. Questo porta ad un cambio radicale nella progettazione di un simulatore che deve tenere conto di questa frequenza variabile di dati.
ESIM combina un simulatore di eventi con i motori grafici per consentire una si-

mulazione accurata mediante l'utilizzo dell'*adaptive sampling* che verrà analizzato nella sezione seguente.

3.2 Adaptive Sampling

Nella progettazione del simulatore non è possibile considerare l'analisi a tempo continuo del segnale visivo di ogni pixel.
In passato, lavori precedenti avevano cercato di ovviare a questo problema campionando il segnale visivo in modo sincrono attuando un'interpolazione lineare tra i campioni utilizzando un' alta frequenza di campionamento con l'obiettivo di ricostruire in maniera approssimata il segnale. Questo consentiva di convertire un segnale visivo continuo in uno discreto con un certo grado di approssimazione.
In ESIM viene ripreso questo concetto nella simulazione degli eventi ma con una differenza significativa: invece di scegliere un *framerate* di *rendering* arbitrario e quindi campionare i *frames* uniformemente nel tempo con un *framerate* scelto, si è scelto di campionare i *frames* in maniera adattiva, modulando la frequenza di campionamento in base alle dinamiche del segnale visivo su cui effettuare la simulazione.

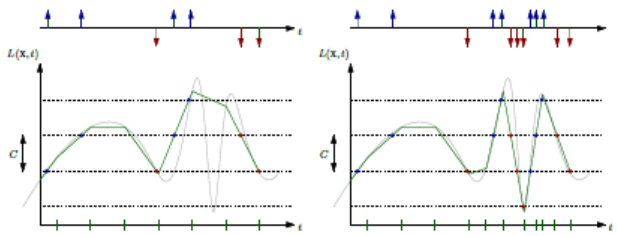

Figura 3.1: Confronto tra *Uniforme Sampling* (grafico a sinistra) e *Adaptive Sampling* (grafico a destra). Immagine tratta da [4].

Come si evince dalla figura 3.1, entrambe le strategie producono eventi simili quando il segnale visivo varia poco. Nel momento in cui quest'ultimo aumenta in maniera considerevole, l'*adaptive sampling* produce una simulazione molto più accurata dell'*uniform sampling* in quanto il *framerate* cambia con il variare del segnale visivo.
Questo cambio di approccio necessita una precisa interazione tra motore grafico e simulatore di eventi.

3.3 Architettura del simulatore

L'architettura di ESIM è illustrata nella figura 3.2 e come si può notare, accoppia in modo stretto il simulatore ad eventi con il *rendering engine*. Questo consente all'ESIM di elaborare i *frames* basandosi sulla dinamica del segnale visivo.

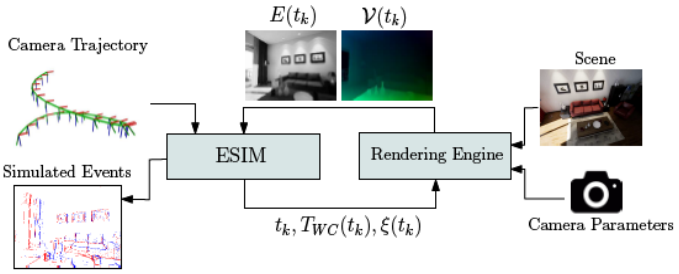

Figura 3.2: Architettura di ESIM. Immagine tratta da [4].

Il primo elemento di fondamentale importanza è la ***sensor trajectory***. È una funzione regolare τ che mappa ad ogni instante di tempo t, una posa, il *twist* (velocità angolare e lineare) e l'accelerazione del sensore.
Si denota la posa del sensore espressa in alcuni *frame inerziali* W con $T_{WB} \in SE$, dove SE definisce un gruppo di movimenti rigidi in 3D, il *twist* è definito con $\xi(t) = ({}_W v(t), {}_B \omega_{WB}(t))$, e la sua accelerazione è espressa (nel *frame inerziale*) con ${}_B a(t)$.
Un altro elemento utilizzato nel simulatore è il ***rendering engine***.
Il *renderer* è una funzione reale che mappa ad ogni istante di tempo t un'immagine renderizzata della scena in base alla posa corrente del sensore. Il *renderer* è parametrizzato dall'ambiente ε, dal *sensor trajectory* all'interno dell'ambiente τ e dalla configurazione Θ.
La variabile ε controlla le geometrie della scena e le sue dinamiche (potrebbe rappresentare ad esempio una strada con dei veicoli in movimento oppure una sala da pranzo).
La variabile τ rappresenta la traiettoria della camera nell'ambiente che può essere generata da un modello dinamico di un robot o da un set di input di controllo.
La variabile Θ rappresenta la configurazione del sensore di visione simulata che include dei parametri della camera come la grandezza del sensore, lunghezza focale e parametri di distorsione.
Nell'architettura sono compresi anche il valore della soglia di contrasto C di una camera ad eventi e il tempo di esposizione di una camera tradizionale.
Il *renderer* fornisce anche il *motion field* $\nu(t_k)$.
Riassumendo, al tempo t_k, il simulatore ESIM campiona una nuova posa della camera $T_{WC}(t_k)$ *e il twist* $\xi(t_k)$ a partire dalla traiettoria definita dall'utente e

li passa al *rendering engine* che renderizza una nuova mappa d'irradianza $E(t_k)$ e una mappa del *motion field* $\nu(t_k)$ che serviranno per produrre la simulazione degli eventi.
Il *motion field* viene usato per calcolare il cambiamento previsto di luminosità che a sua volta permette di scegliere il tempo di *rendering* successivo t_{k+1}.

3.4 Modi di utilizzo dell'adaptive sampling

3.4.1 Metodo basato sul cambiamento di luminosità

Sotto le ipotesi delle superfici lambertiane, si considera una espansione di Taylor al primo ordine della luminosità:

$$\frac{\partial \mathcal{L}(x;t_k)}{\partial t} \simeq -\langle \nabla \mathcal{L}(x;t_k), \nu(x;t_k) \rangle \tag{3.1}$$

dove $\nabla\mathcal{L}$ è il gradiente della luminosità dell'immagine e ν è il *motion field*. Quindi $\nabla\mathcal{L} \simeq \frac{\partial \mathcal{L}(x;t_k)}{\partial t}\Delta t$ è la variazione (positiva o negativa) di luminosità prevista al pixel $\mathbf{x}$ e al tempo t_k durante un dato intervallo di tempo Δt.
Si assicura che per ogni pixel $\mathbf{x} \in \Omega$, $|\Delta\mathcal{L}| \leq C$. Questo significa che la variazione di luminosità è limitata dalla soglia di contrasto desiderata C della camera ad eventi.
L'istante di tempo t_{k+1} viene calcolato nel modo seguente:

$$t_{k+1} = t_k + \lambda_b C \left| \frac{\partial \mathcal{L}}{\partial t} \right|_m^{-1} \tag{3.2}$$

dove: $\left|\frac{\partial \mathcal{L}}{\partial t}\right|_m = max_{x\in\Omega} \left|\frac{\partial \mathcal{L}(x;t_k)}{\partial t}\right|$ è il massimo variazione prevista della luminosità nel piano dell'immagine Ω, e $\lambda_b \leq 1$ è un parametro che controlla lo scambio tra la velocità di *rendering* e l'accuratezza.
L'equazione (3.2) impone quindi che il *framerate* del *rendering* cresca proporzionalmente con la massima variazione assoluta di luminosità prevista nell'immagine. Tuttavia, non c'è garanzia che il segnale sia stato campionato correttamente, a causa di effetti non lineari. Per far fronte a questo problema, si può scegliere un valore di $\lambda_b < 1$. Nelle simulazioni che verranno approfondite nel capitolo successivo, $\lambda_b = 0.5$.

3.4.2 Metodo basato sullo spostamento del pixel

Esiste una strategia più semplice per adottare l'*adaptive sampling*. Essa si basa sull'assunzione che il massimo spostamento di un pixel tra due *frames* successivi è limitato.
Questo metodo può essere adottato scegliendo un tempo di campionamento successivo t_{k+1} come segue:

$$t_{k+1} = t_k + \lambda_v \left|\nu(x_k; t_k)\right|_m^{-1} \tag{3.3}$$

dove $|\nu| = max_{x\in\Omega} |\nu(x; t_k)|$ è la magnitudo massima del *motion field* al tempo t_k, e $\lambda_v \leq 1$. Come detto sopra, si sceglie λ_v per mitigare l'effetto delle non-linearità presenti. Nelle simulazioni adottate in questa tesi, $\lambda_v = 0.5$.

3.5 Vantaggi dell'adaptive sampling

In questa sezione verrà effettuato un confronto tra l'*adaptive sampling* e uno schema di campionamento a frequenza fissata (figura 3.3).

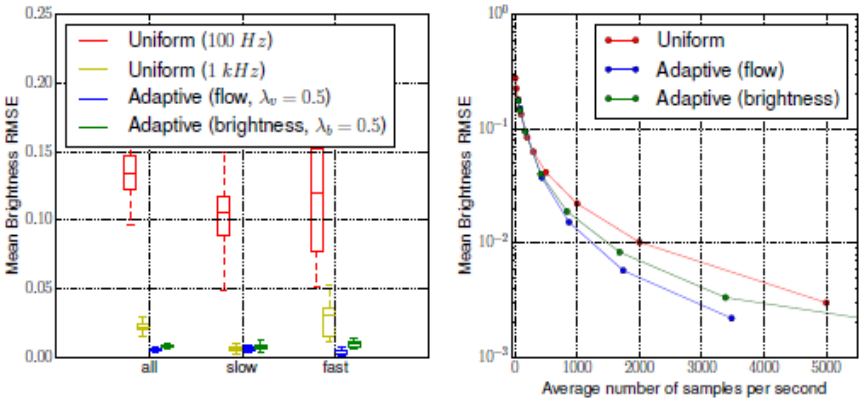

Figura 3.3: Confronto tra varie metodologie di campionamento. Immagine tratta da [4].

Nella figura 3.3 a sinistra, si ha l'RMSE (*Root Mean Standard Error*) del segnale di luminosità in funzione del *framerate* per ogni strategia di campionamento ("*all*" per indicare che i tutti i campioni sono analizzati, "*slow*" per indicare un *framerate* basso" e "*fast*" per un *framerate* alto) elaborato nell'intera sequenza di valutazione.
Nella figura 3.3 a destra viene rappresentato l'RMSE del segnale di luminosità ricostruito in funzione del numero medio di campioni al secondo elaborati per ogni strategia di campionamento, esaminati nell'intera sequenza di valutazione.

3.6 Simulazione dei disturbi e delle non-linearità

Nel simulatore sono implementati dei modelli standard di disturbo per le camere ad eventi.
I modello sono ottenuti campionado il contrasto in base alla variabile $\mathcal{N}(C, \sigma_C)$, dove σ_C controlla l'ammontare del disturbo impostato dall'utente.

Capitolo 4

Risultati

4.1 Installazione del simulatore

Per procedere all'installazione dell'ESIM è fondamentale aver installato il ROS (come illustrato nel capitolo 2).
Come prima operazione da compiere, è necessario installare *catkin_tools* con il seguente comando:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu `
    lsb_release -sc` main" > /etc/apt/sources.list.d/ros-latest.list'
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
sudo apt-get update
sudo apt-get install python-catkin-tools
```

Successivamente, bisogna creare un *workspace* fatto esclusivamente per contenere il simulatore. Si effettua con i seguenti comandi:

```
mkdir -p ~/sim_ws/src && cd ~/sim_ws
catkin init
catkin config --extend /opt/ros/kinetic --cmake-args -
    DCMAKE_BUILD_TYPE=Release
```

Per procedere oltre, c'è bisogno di installare *vcstools*:

```
sudo apt-get install python-vcstool
```

Quindi si clona il *repository* importato (ricordandosi di creare ed usare una chiave SSH per il proprio account *Github*) e si procede poi all'avvio di *vsctools*:

```
cd src/
git clone git@github.com:uzh-rpg/rpg_esim.git
vcs-import < rpg_esim/dependencies.yaml
```

Di seguito, vengono installati vari software utili per il simulatore : *pcl_ros*, *glm*, *glfw*:

```
sudo apt-get install ros-kinetic-pcl-ros
sudo apt-get install libglfw3 libglfw3-dev
sudo apt-get install libglm-dev
```

Opzionalmente, può essere installato il *trajectory server* con il seguente comando:

```
sudo apt-get install ros-kinetic-hector-trajectory-server
```

Ora, è utile disabilitare alcune dipendenze non necessarie:

```
cd ze_oss
touch imp_3rdparty_cuda_toolkit/CATKIN_IGNORE \
      imp_app_pangolin_example/CATKIN_IGNORE \
      imp_benchmark_aligned_allocator/CATKIN_IGNORE \
      imp_bridge_pangolin/CATKIN_IGNORE \
      imp_cu_core/CATKIN_IGNORE \
      imp_cu_correspondence/CATKIN_IGNORE \
      imp_cu_imgproc/CATKIN_IGNORE \
      imp_ros_rof_denoising/CATKIN_IGNORE \
      imp_tools_cmd/CATKIN_IGNORE \
      ze_data_provider/CATKIN_IGNORE \
      ze_geometry/CATKIN_IGNORE \
      ze_imu/CATKIN_IGNORE \
      ze_trajectory_analysis/CATKIN_IGNORE
```

e compilare il nodo *event_camera_simulator* con:

```
catkin build esim_ros
```

Infine, si crea un alias per il *workspace* in modo tale che esso può essere richiamato velocemente ad ogni avvio della *bash*:

```
echo "source ~/sim_ws/devel/setup.bash" >> ~/setupeventsim.sh
chmod +x ~/setupeventsim.sh
```

Aggiungiamo le modifiche al file *.bashrc* con la seguente riga di comando:

```
alias ssim='source ~/setupeventsim.sh'
```

Quindi adesso, con il comando *ssim* si inizializzerà il *workspace* del simulatore. Ciò è fondamentale per lavorare con il simulatore ESIM.

4.2 Simulazione di eventi a partire da un video

4.2.1 Concetti preliminari

In questa sezione si procede alla simulazione di eventi a partire da un video descritto nel capitolo 2 di questa tesi.

Nella figura 4.1, è schematizzato l'intero processo: dopo aver opportunamente elaborato il video con FFmpeg estrapolando tutti i frames del video salvandoli in files .png con il nome dato dal numero del *frame* in ordine progressivo, si avvia uno *script* Python per generare il file *images.csv* che contiene una lista formata da due colonne che associa l'istante temporale del *frame* con il suo relativo file .png.

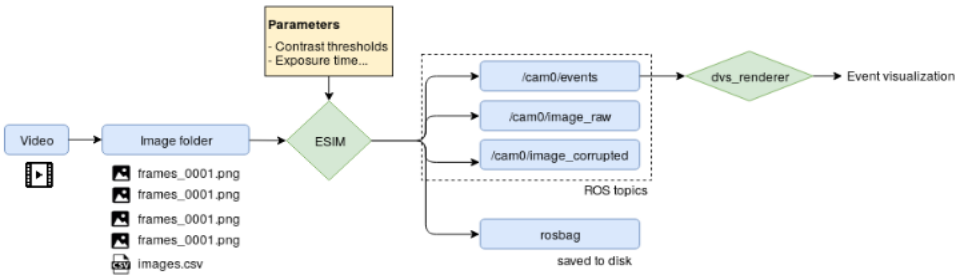

Figura 4.1: Schematizzazione del processo di generazione degli eventi e dei *frames* APS da parte di ESIM. Immagine tratta da [8].

Una volta che ESIM ha generato gli eventi e i *frames* APS simulati, vengono avviati i *ROS Topics* con gli elementi necessari per la loro visualizzazione. Gli eventi simulati sono salvati opportunamente in un *ROS Bag* chiamato *out.bag*. Infine viene avviato il *dvs_renderer* che avvia la simulazione a schermo degli eventi.

4.2.2 Download del video

Se non si ha a disposizione un video per effettuare una simulazione di prova, si può usare un software chiamato *youtube-dl* che permette il download dei video da *YouTube*.
Come video di prova, si è scelto un video della *National Geographic* che è girato a 1200 *frames* per secondo che poi verrà ridimensionato per agevolare la simulazione.
Per prima cosa, si crea una nuova cartella chiamata *example* all'interno della cartella *tmp*. Successivamente ci si sposta all'interno di essa:

```
1 mkdir -p /tmp/example
2 cd /tmp/example
```

Installiamo il software *youtube-dl*:

```
pip install youtube-dl
```

Scarichiamo il video con il comando:

```
youtube-dl https://youtu.be/THA_5cqAfCQ -o cheetah
```

Il video verrà salvato all'interno della cartella *example*

Adesso si utilizzano gli strumenti offerti da FFmpeg per ridurre la durata del video: verrà creato un secondo file video in formato *.mkv* che contiene soltanto la parte del video che va dal minuto 2'07 e finisce al minuto 2'47'.
Per fare ciò si utilizza il seguente comando:

```
ffmpeg -i cheetah.mkv -ss 00:02:07 -t 00:00:40 -async 1 -strict -2
    cheetah_cut.mkv
```

Con *-i* si indica il video da fornire in input ad FFmpeg (in questo caso *cheetah.mkv*. Con *-ss* si indica da quale punto del video iniziare a salvare il video. Con *-t*, si indica la durata della lettura del video in input (in questo caso si salveranno 40 secondi a partire dal minuto *2'07*.

Successivamente si ridimensiona il video per agevolare la simulazione video:

```
ffmpeg -i cheetah_cut.mkv -vf scale=640:-1 -crf 0 cheetah_sd.mkv
```

4.2.3 Pre-processing del video per ESIM

Ora verranno effettuate le operazioni di pre-processamento del video descritte nel capitolo 2, che porteranno alla creazione della cartella *frames* che conterrà tutti i *frames* in png del video stesso e il relativo file *images.csv*.
Le operazioni sopra descritte, si effettuano con i seguenti comandi:

```
mkdir frames
ffmpeg -i video_test.mkv frames/frames_%010d.png
```

Manca la creazione del file *images.csv* che verrà descritto nella sezione successiva.

4.2.3.1 Lo script "generate_stamps_file.py"

Come anticipato sopra, questo script in python genera un file in formato .csv che è formato da due colonne: la prima contiene l'istante di tempo relativo ad un determinato *frame* del video fornito in input, mentre la seconda fornisce il nome del file .png associato al *frame*.
Il codice Python è riportato di seguito:

```
import argparse
from os import listdir
from os.path import join


if __name__ == "__main__":

    parser = argparse.ArgumentParser(
        description='Generate "images.csv" for ESIM DataProviderFromFolder')

    parser.add_argument('-i', '--input_folder', default=None, type=str,
                        help="folder containing the images")
    parser.add_argument('-r', '--framerate', default=1000, type=float,
                        help="video framerate, in Hz")

    args = parser.parse_args()

    images = sorted(
        [f for f in listdir(args.input_folder) if f.endswith('.png')])

    print('Will write file: {} with framerate: {} Hz'.format(
        join(args.input_folder, 'images.csv'), args.framerate))
    stamp_nanoseconds = 1
    dt_nanoseconds = int((1.0 / args.framerate) * 1e9)
    with open(join(args.input_folder, 'images.csv'), 'w') as f:
        for image_path in images:
            f.write('{},{}\n'.format(stamp_nanoseconds, image_path))
            stamp_nanoseconds += dt_nanoseconds

    print('Done!')
```

Inizialmente vengono importate le librerie usate dallo *script* come *argparse*, *listdir* e *join*. La libreria *argparse* consente di creare interfacce *user-friendly* da riga di comando.
Con *listdir* si hanno degli strumenti per gestire liste delle cartelle del *filesystem*.
Il primo passo consiste nel definire un oggetto *parser*, richiamando la classe *ArgumentParser* che crea l'oggetto richiedendo nel costruttore la definizione dell'attributo *description*, che contiene una breve descrizione dello *script*.

È necessario indicare attraverso il metodo *add_ argument* il modo in cui deve essere fatto il *parsing* di ogni singolo argomento fornito da riga di comando. Nel nostro caso ogni argomento sarà preceduto da *-i* e *-r*.
Nella riga 11 del codice *generate_ stamps_ file.py* infatti, vengono indicati:

- la stringa che precede l'argomento di cui si vuole effettuare il *parsing* (ad esempio -i);
- il nome della variabile su cui verrà salvato il valore fornito da riga di comando (–input_ folder);
- il valore di *default* nel caso in cui il valore non venga specificato dall'utente (nel nostro caso *none*;
- il tipo di argomento (indicato con *type*;
- una piccola stringa di aiuto, indicata con *help*;

Il metodo *parse_ args()*, converte le stringhe fornite nell'argomento in oggetti e li assegna come attributi di un *namespace*.

Con:

```
1 images = sorted(
2         [f for f in listdir(args.input_folder) if f.endswith('.png')
    ])
```

si crea una variabile *f* in cui viene salvata una lista di tutti i nomi dei file .png contenuti all'interno della cartella indicata da *args.input_ folder*.
dt_ nanoseconds è una variabile che contiene l'intervallo di tempo che intercorre tra un frame in png e l'altro: nel caso in esame, si acquisiscono 30 *frames* al secondo quindi viene diviso un secondo per la variabile *args.framerate* (30) e poi viene convertito in nanosecondi. Infine viene scritto il file.csv accostando ad ogni riga del file stesso, l'istante di tempo incrementato del valore contenuto nella variabile *dt_ nanoseconds* e il relativo nome del file .png che contiene il *frame*.

La chiamata dello *script* nel nostro caso sarà fatta quindi in questo modo:

```
1 python scripts/generate_stamps_file.py -i /tmp/example/frames -r
    30.0
```

Il file *images.csv* è mostrato in figura :

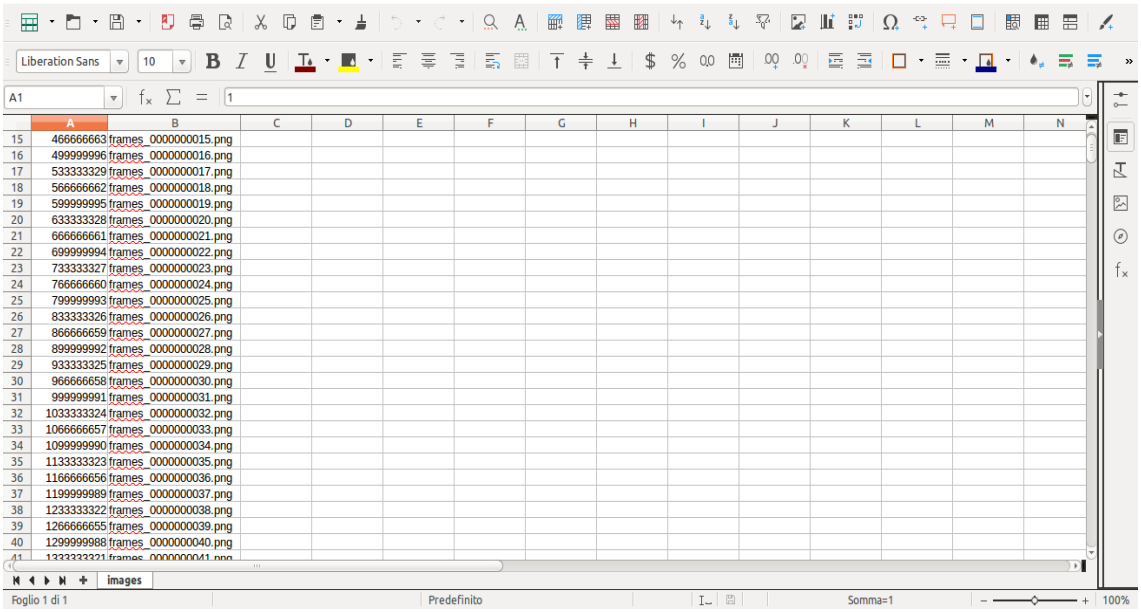

Figura 4.2: Parte del file *images.csv*

.

4.2.4 Simulazione degli eventi con ESIM

Con il seguente comando, viene iniziato il processo di creazione degli eventi con ESIM (bisogna assicurarsi di aver avviato il *ROS Master* attraverso il comando *roscore*):

```
rosrun esim_ros esim_node \
 --data_source=2 \
 --path_to_output_bag=/tmp/out.bag \
 --path_to_data_folder=/tmp/video_test/frames \
 --ros_publisher_frame_rate=60 \
 --exposure_time_ms=10.0 \
 --use_log_image=1 \
 --log_eps=0.1 \
 --contrast_threshold_pos=0.15 \
 --contrast_threshold_neg=0.15
```

Viene eseguito un nodo chiamato *esim_node* del *package esim_ros* in cui vengono indicati i seguenti argomenti:

- –data_source=2 dice all'ESIM di leggere dati da una cartella che contiene immagini. Esso guarderà all'interno di una file .csv che associa gli istanti temporali alle immagini da leggere;

- –path_to_data_folder fornisce il percorso alla cartella dove sono memorizzati i file .csv e i *frames* in png;
- –path_to_output_bag fornisce il percorso dove salvare il file .bag;
- –ros_publisher_frame_rate indica il *framerate* del sensore APS simulato (in fps);
- –exposure_time_ms fornisce il tempo di esposizione del *frame* del sensore APS (in millisecondi);
- –use_log_image dice all'ESIM di operare nel dominio a intensità logaritmica. Quindi ogni immagine verrà convertita nel modo seguente: $L = \log(\frac{I}{255+eps})$, dove *eps* è impostato con il parametro *–log_ eps*;
- –contrast_threshold_pos e –contrast_threshold_neg impostano i valori della soglia di contrasto (positiva e negativa). Un valore più basso comporta una sensitività più alta (e quindi molti eventi). Un valore più alto comporta una sensitività più bassa (e quindi meno eventi);

4.3 Visualizzare gli eventi simulati

Dopo aver avviato l'ESIM con il comando mostrato prima, gli eventi vengono memorizzati all'interno del file */tmp/out.bag*. Per visualizzarli, si apre una nuova finestra del terminale avviando un nodo chiamato *dvs_renderer* come segue:

```
rosrun dvs_renderer dvs_renderer events:=/cam0/events
```

Successivamente si apre un'altra finestra del terminale per riprodurre a loop (*-l*) il file *out.bag* (figura 4.3). Il comando è il seguente:

```
rosbag play /tmp/out.bag -l
```

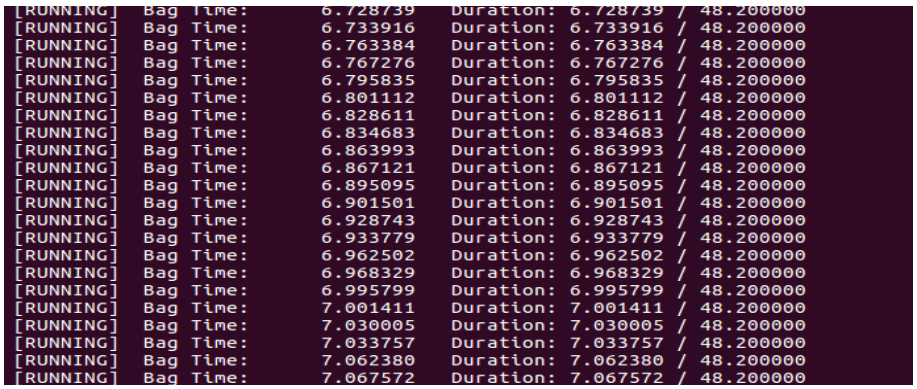

Figura 4.3: Screenshot del comando che esegue la riproduzione del contenuto del file *out.bag*.

Infine, è possibile visualizzare gli eventi simulati aprendo *rqt_image_view* in un'altra finestra del terminale:

```
rqt_image_view /dvs_rendering
```

Viene aperta una finestra di *rqt_image_view* dove è possibile selezionare tra la visualizzazione dei soli eventi (figura 4.4) oppure quella dei *frames* APS (figura 4.5).

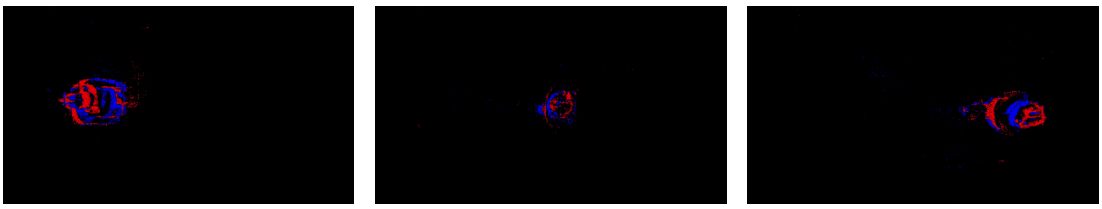

Figura 4.4: Tre frames della simulazione degli eventi.

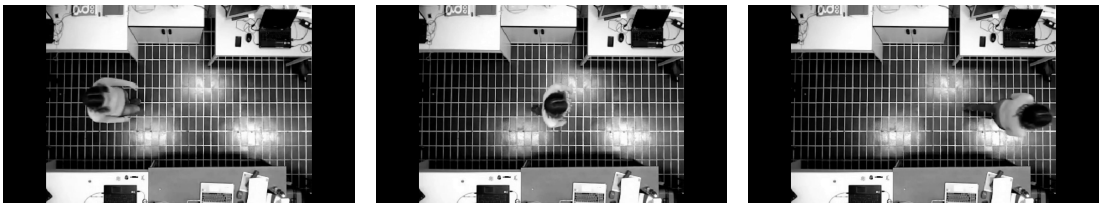

Figura 4.5: Tre frames APS della simulazione.

4.4 Simulazione di eventi basata su una scena virtuale

L'ultima simulazione proposta in questa tesi riguarda l'utilizzo di una scena virtuale implementata tramite l'utilizzo di Blender e del motore grafico OpenGL. Per ottenere la simulazione, bisogna editare il file *cfg/opengl.conf* che è mostrato di seguito:

```
--v=0
--random_seed=2
--data_source=0
--path_to_output_bag=/tmp/out.bag

--contrast_threshold_pos=0.5

--contrast_threshold_neg=0.5
--contrast_threshold_sigma_pos=0
--contrast_threshold_sigma_neg=0

--exposure_time_ms=12.0
--use_log_image=1
--log_eps=0.001

--calib_filename=/home/paolo/sim_ws/src/rpg_esim/
    event_camera_simulator/esim_ros/cfg/calib/pinhole_mono.yaml
--renderer_scene=/home/paolo/Scrivania/scena_virtuale.obj

--renderer_zmax=20
--renderer_type=2
--renderer_zmin=0.3
--renderer_zmax=40.0

--trajectory_type=0
--trajectory_length_s=10.0
--trajectory_sampling_frequency_hz=5
--trajectory_spline_order=5
--trajectory_num_spline_segments=10
--trajectory_lambda=0.01
--trajectory_multiplier_x=1.8
--trajectory_multiplier_y=1.8
```

```
--trajectory_multiplier_z=0.5
--trajectory_multiplier_wx=0.80
--trajectory_multiplier_wy=0.80
--trajectory_multiplier_wz=0.80
--trajectory_offset_z=0.0

--simulation_minimum_framerate=20.0
--simulation_imu_rate=1000.0
--simulation_adaptive_sampling_method=1
--simulation_adaptive_sampling_lambda=0.5
--simulation_post_gaussian_blur_sigma=0.3

--ros_publisher_frame_rate=30.0
--ros_publisher_depth_rate=30.0
--ros_publisher_optic_flow_rate=30.0
--ros_publisher_pointcloud_rate=10
--ros_publisher_camera_info_rate=10
```

Il file contiene tutte le impostazioni utili per effettuare la simulazione, che comprende:

- il movimento della camera nei tre assi (traiettoria);
- i valori dell'*adaptive sampling*;
- i valori della soglia di contrasto positiva e negativa;
- la calibrazione della camera simulata;
- il percorso dove risiede il file *.obj* della scena virtuale;
- il percorso dove memorizzare il file bag di output (*out.bag*);

Una volta che le impostazioni sono state definite, bisogna avviare la simulazione con il seguente comando:

```
roslaunch esim_ros esim.launch config:=cfg/opengl.conf
```

Ora la simulazione è stata avviata, tuttavia non è ancora stato eseguito Rviz che permette la visualizzazione della simulazione stessa.
Per fare ciò, basta eseguire le seguenti righe di comando dal terminale:

```
roscd esim_visualization
rviz -d cfg/esim.rviz
```

Si può utilizzare anche *r Rqt* per visualizzare la simulazione con più dettagli:

```
roscd esim_visualization
rqt --perspective-file cfg/esim.perspective
```

Di seguito, si propone uno *screenshot* della simulazione (figura 4.6):

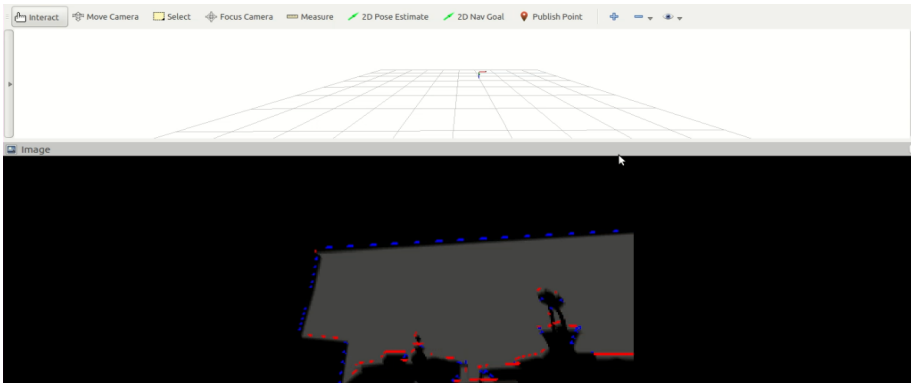

Figura 4.6: Simulazione con OpenGL della scena virtuale.

Capitolo 5

Conclusioni e Sviluppi Futuri

5.1 Conclusioni

Alla luce degli *screenshot* riportati alla fine del capitolo 4, le simulazioni effettuate con ESIM sono andate a buon fine e hanno dimostrato le potenzialità che questo simulatore offre in termini di:

- **affidabilità**: il simulatore è molto stabile e non presenta particolari problemi nel suo utilizzo;
- **leggerezza**: ESIM si è dimostrato incredibilmente fluido ed efficiente durante le operazioni, utilizzando le risorse dell'elaboratore in modo intelligente e limitato;
- **efficacia**: la simulazione si avvicina molto al funzionamento reale delle camere ad eventi, non solo in termini di verosimiglianza nel *output* video, ma anche nella quantità di impostazioni disponibili (IMU, traiettorie della camera, calibrazione ecc.) che la rendono molto utile nello sviluppo di algoritmi;
- **ampia gamma di scelta**: con questo simulatore ci sono molti strumenti per poter generare eventi (video, scene virtuali ecc.) utilizzando diversi motori grafici come *UnrealEngine* e *OpenGL*;
- **documentazione**: la quantità e la qualità della documentazione disponibile rende l'ESIM facile e intuitivo nell'utilizzo pratico.

5.2 Sviluppi futuri delle camere ad eventi

Le camere ad eventi rappresentano un cambio di paradigma nell'acquisizione delle informazioni visuali. Quindi alla luce di questo si sono aperte nuove sfide nella progettazione di nuovi metodi di elaborazione dei dati acquisiti e di estrapolazione delle informazioni per sbloccarne i loro vantaggi e potenzialità.
Nello specifico bisognerà porre attenzione:

- sul differente *output* spazio-temporale: l'*output* delle camere ad eventi è molto differente da quello delle camere standard: gli eventi sono asincroni e spazialmente sparsi, mentre nelle camere tradizionali le immagini sono sincrone e dense. Quindi tutti gli algoritmi di visione sviluppati per le camere tradizionali non sono direttamente applicabili per le camere ad eventi;
- sul differente effetto dei disturbi e della dinamica: tutti i sensori di visione hanno dei disturbi causati dai circuiti a *transistor* e hanno delle non-linearità. Questo è vero anche per le camere ad eventi dove il processo di quantizzazione temporale del contrasto è complesso;
- sul differente rilevamento fotometrico: le camere tradizionali forniscono informazioni in scala di grigi. Al contrario, ogni evento contiene informazioni binarie (aumento/decremento) sul cambiamento della luminosità. La variazione della luminosità non dipende soltanto dalla variazione della luminosità della scena, ma anche del passato e presente movimento tra la scena e la camera;

La ricerca futura deve quindi trovare risposte in merito alle seguenti domande:

- qual è il miglior modo per estrarre le informazioni dagli eventi per uno specifico compito?
- come il disturbo e gli effetti non lineari devono essere modellati per estrarre al meglio le informazioni dagli eventi?

In futuro, le camere ad eventi avranno un utilizzo sempre più presente nella vita di tutti i giorni con la loro implementazione nell'industria di massa (ad esempio *smartphone*, *tablet* e guida autonoma) imprimendo un aumento esponenziale della qualità delle immagini acquisite in condizioni di luce o di movimento molto sfavorevoli.

Ringraziamenti

Sono giunto alla parte forse più importante della tesi che riassume in poche righe la mia infinita riconoscenza verso tutte le persone che mi sono state vicine in questo lungo cammino.
Voglio ringraziare la mia famiglia che mi è stata vicina e che soprattutto che non mi ha mai messo ulteriori pressioni quando le cose non stavano andavano nel verso giusto.
Se oggi sono qui a scrivere la tesi di laurea è merito anche dei miei amici che mi hanno dato i consigli giusti nei momenti in cui ho pensato veramente di non farcela.
Grazie a Lorenzo che è stato oltre che un amico, il mio coinquilino anconetano. Senza di te l'intero viaggio universitario avrebbe avuto sicuramente un cielo più scuro sopra la mia testa, insegnandomi che, dopo due battute e una birretta, il sorriso e la voglia di tornare a combattere tornano sempre.
Un grazie enorme a Carlo con il quale ho condiviso e condivido tuttora, le fatiche e complicazioni varie nel far combaciare gli impegni musicali quando sei ingarbugliato tra esami e lezioni universitarie. Diciamo che mentre Lorenzo scaccia le nuvole, tu sei quello che lascia andare via le mie emozioni riportandomi ad essere lucido quando le cose non vanno per il verso giusto.
Un grazie anche a Giuseppe con cui lavoro e con cui sfido quotidianamente le mie capacità e con cui cerco di migliorarmi per il bene di un progetto a cui entrambi teniamo molto.
Grazie a Niccolò, Marino, Nicholas, Niko, Alessio e Simone per sostenermi sempre ogni volta che ne ho bisogno e riconosco che non è una cosa scontata.
Ringrazio Marco, che ha creduto in me per la realizzazione di piccoli lavori informatici di cui sono molto orgoglioso e spero di poter continuare ancora a contribuire al tuo splendido progetto.
Ringrazio i miei compagni di corso Matteo, Simone, Elia e Savio con cui ho condiviso le gioie e dolori della vita universitaria, fatta di esami e progetti infiniti da portare a termine. Siete stati davvero fondamentali nel farmi scivolare via la pressione che avevo addosso durante le lezioni ed esami.
Ringrazio i miei nonni che desideravano davvero tanto vedermi un giorno con quella corona di alloro. Devo tanto a loro e pagherei oro per poterli vedere sorridere e festeggiare con me, ma sono sicuro che lo stanno già facendo in qualche

modo.
Un ringraziamento speciale va ai dipendenti e colleghi del Comune di Sant'Egidio alla Vibrata con cui ho lavorato durante il mio anno nel Servizio Civile Nazionale. Non avrei mai potuto pensare di poter coniugare lavoro e studio in quel periodo senza la vostra collaborazione.

Bibliografia

[1] Python Software Foundation. Python logo.

[2] Free3D. Coffee shop model.

[3] Guillermo Gallego, Tobi Delbrück, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew J. Davison, Jörg Conradt, Kostas Daniilidis, and Davide Scaramuzza. Event-based vision: A survey. *CoRR*, abs/1904.08405, 2019.

[4] Henri Rebecq, Daniel Gehrig, and Davide Scaramuzza. ESIM: an open event camera simulator. *Conf. on Robotics Learning (CoRL)*, October 2018.

[5] ResearchGate. Illustration of how illumination times reflectance equals luminance. the light that reaches the eye (luminance) is a function of illumination (light source) and the percentage of that illumination that is reflected off the paper surface (reflectance). notice that the light from the sun (designated by the down arrow) is more intense than the light reflected off the paper (designate by the up arrow) because a certain percentage of light is absorbed by the paper.

[6] ResearchGate. A motion-based feature for event-based pattern recognition - scientific figure on researchgate.

[7] ResearchGate. Vertical landing for micro air vehicles using event-based optical flow - scientific figure on researchgate.

[8] UZH. Simulating events from a video, 2020.

[9] Wikipedia. Ffmpeg — wikipedia, l'enciclopedia libera, 2020.

[10] Wikipedia. opengl — wikipedia, l'enciclopedia libera, 2020.

[11] Wikipedia. Robot operating system — wikipedia, l'enciclopedia libera, 2020.

[12] Wikipedia. Tempo di esposizione — wikipedia, l'enciclopedia libera, 2020.

[13] Wikipedia. Unmanned aerial vehicle — wikipedia, l'enciclopedia libera, 2020.

[14] ETH Zürich. Programming for robotics - ros.

Elenco delle figure

1.1 Un drone. Immagine tratta da [13] 7
1.2 Camera DVS. Immagine tratta da [7]. 8
1.3 Riassunto di una camera DAVIS, che comprende un *dynamic vision sensor* (DVS) e un *active pixel sensor* (APS) nello stesso array di pixel, condividendo lo stesso fotodiodo in ogni pixel. **(a)** Diagramma del circuito semplificato del pixel DAVIS. (Il pixel DVS è in rosso, il pixel APS è in blu). **(b)** Schematizzazione delle operazioni del pixel DVS nel convertire la luce in eventi. **(c)-(d)** Chip DAVIS e camera USB. **(e)** Un quadrato bianco su un disco rotante nero visto dal sensore DAVIS che produce *frames* in scala di grigi e una spirale di eventi nello spazio e nel tempo. Gli eventi nello spazio e nel tempo sono codificati con colori: dal verde (passato) al rosso (presente). **(f)** Frame e eventi sovrapposti in una scena. Immagine tratta da [3, p.2]. 9
1.4 Riflettanza. Immagine tratta da [5]. 10
1.5 Fotografia sovraesposta. Immagine tratta da [12]. 11
1.6 Fotografia sottoesposta. Immagine tratta da [12]. 11
1.7 **(a)**: Scenario di misurazione del luce attraverso un pixel DVS. In **(b)**: Le risposte delle due tipologie di camere a diversi livelli di illuminazione. Immagine tratta da [3]. 12
1.8 **(a)**: Il sensore ATIS. Immagine tratta da [6]. 14

2.1 Logo del ROS. Immagine tratta da [11]. 17
2.2 Schermata di avvio del *ROS Master*. 20
2.3 *ROS Node* e la comunicazione con il *ROS Master*. Immagine tratta da [14]. 20
2.4 *ROS Master*, *Node* e *Topic* in comunicazione tra loro. Immagine tratta da [14]. 21
2.5 *ROS Master*, *Node*, *Topic* e *Message* in comunicazione tra loro. Immagine tratta da [14]. 22
2.6 Output del nodo *talker*. 23
2.7 Informazioni del nodo *talker*. 23
2.8 Informazioni del *topic chatter*. 24

2.9 Output del nodo *listener*. 24
2.10 Output del nodo *listener*. 25
2.11 Output del nodo *listener*. 25
2.12 Esempio di un *package*. Immagine tratta da [14]. 27
2.13 *ROS Service* in comunicazione con due nodi. Immagine tratta da [14]. 28
2.14 Informazioni del servizio *add_two_ints*. 29
2.15 Risultato della chiamata del servizio *add_two_ints*. 29
2.16 Comunicazione tra nodi e *ROS Actions*. Immagine tratta da [14]. 30
2.17 Schermata iniziale di Rviz. 32
2.18 Schermata iniziale di Rqt Image View. 32
2.19 Scena in 3D creata con Blender. Modello 3D tratto da [2]. 33
2.20 Il logo di Python. Immagine tratta da [1]. 35
2.21 Logo FFmpeg. Immagine tratta da [9] 35
2.22 Logo OpenGL. Immagine tratta da [10] 36

3.1 Confronto tra *Uniforme Sampling* (grafico a sinistra) e *Adaptive Sampling* (grafico a destra). Immagine tratta da [4]. 38
3.2 Architettura di ESIM. Immagine tratta da [4]. 39
3.3 Confronto tra varie metodologie di campionamento. Immagine tratta da [4]. 41

4.1 Schematizzazione del processo di generazione degli eventi e dei *frames* APS da parte di ESIM. Immagine tratta da [8]. 45
4.2 Parte del file *images.csv* 49
4.3 Screenshot del comando che esegue la riproduzione del contenuto del file *out.bag*. 51
4.4 Tre frames della simulazione degli eventi. 51
4.5 Tre frames APS della simulazione. 52
4.6 Simulazione con OpenGL della scena virtuale. 54