Interpolazione di fotogrammi nei video

Università degli Studi di Catania
Dipartimento di Matematica e Informatica
Corso: Multimedia e Laboratorio
Docenti: Filippo Stanco, Dario Allegra

Michele Ferro

Febbraio 2022

Sommario

Qualsiasi video, che si tratti di un semplice filmato dalla durata di pochi secondi, o che si tratti di un prodotto cinematografico, è caratterizzato da una frequenza di riproduzione o di cattura delle immagini mostrate su schermo. Tale variabile, è in grado di influenzare la percezione di movimento da parte dell'uomo, rendendolo più "fluido" al suo aumentare. La frequenza tipicamente accettata dallo standard cinematografico varia dai 23 ai 30 fotogrammi per secondo o **fps**. Tuttavia, spesso si è in cerca di qualcosa di percettivamente migliore, ma non si è a disposizione di mezzi in grado di catturare filmati con frequenza maggiore a tale standard, e in tali contesti si ricorre alla manipolazione digitale del filmato in post-produzione, e mediante algoritmi di interpolazione di diverso tipo, si è in grado di ottenere un prodotto caratterizzato da una frequenza di fotogrammi (o **frame rate**) maggiore.

Scopo di questo progetto è illustrare alcuni di questi metodi di interpolazione, specificandone pro e contro a seconda dei casi di utilizzo.

Indice

1 Introduzione

Questo progetto, scritto in **Python**, è stato realizzato prendendo diretta ispirazione dalla suite di editing multimediale ***FFmpeg***[3], e in particolare dal suo comando `minterpolate`, il quale, preso in input un filmato caratterizzato da un qualsiasi frame rate, è in grado di restituire un output caratterizzato dal frame rate desiderato dall'utente, che sia maggiore o minore dell'originale.

Nell'emulazione, mediante questo script, del comando sopracitato, non è previsto un solo algoritmo di interpolazione dei fotogrammi, e in base alle sue necessità, l'utente può scegliere tra tre diverse modalità:

- `dup`: il frame mancante viene "riempito" mediante una **replica** del suo predecessore;
- `blend`: il frame mancante viene "riempito" calcolando una **media** tra il suo predecessore ed il suo successore;
- `mci`: il frame mancante viene "riempito" utilizzando un algoritmo di ***motion compensation*** in grado di stimare i vettori di movimento tra il precedente (utilizzato come anchor frame) ed il successivo (utilizzato come target frame).

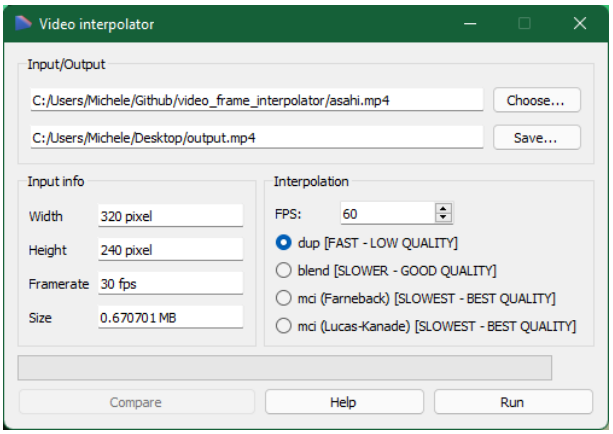
Esse sono state implementate facendo utilizzo della libreria **OpenCV**[4] e verranno analizzate in maniera più dettagliata nei paragrafi successivi.

Inoltre, il software realizzato è compatibile sia con sistemi operativi Windows che GNU/Linux, ed il suo codice sorgente è raggiungibile al presente link:
`https://github.com/nebuchadneZZar01/video_frame_interpolator`.

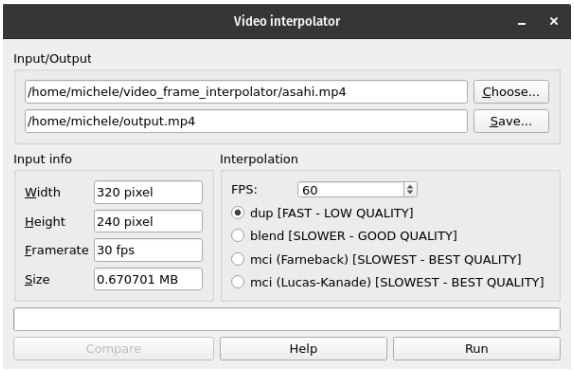
2 GUI

Per il funzionamento di questo script è prevista un'**interfaccia grafica in *Qt5*** (realizzata con l'ausilio della libreria Python `PyQt5`) mediante la quale è possibile caricare il filmato in input in formato `mp4`, scegliere una destinazione di salvataggio (nonché il nome del file di output), applicare la modalità di interpolazione con il framerate desiderato ed infine paragonare il prodotto finale al filmato originale.

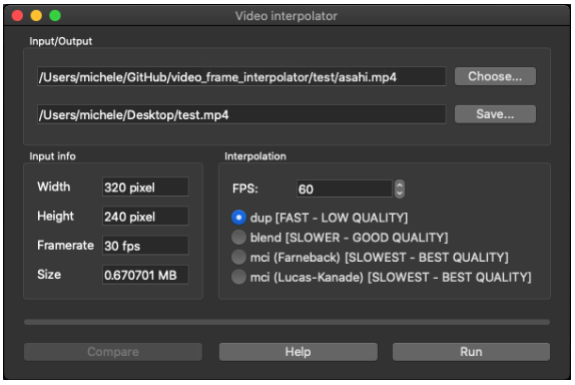
Grazie alla versatilità di PyQt5, e della libreria `pyinstaller`, utilizzata per rendere lo script indipendente dall'ambiente Python installato sulla propria macchina, la GUI permette di avere la stessa esperienza di utilizzo su sistemi GNU/Linux (GTK o Qt), Windows e MacOS.



(a) GUI su Windows 11



(b) GUI su GNU/Linux Pop!_OS



(c) GUI su MacOS Catalina

Figura 1: GUI su vari sistemi

3 Algoritmi di manipolazione

Come già è stato accennato in precedenza, dato un qualsiasi filmato in input, mediante l'interfaccia grafica è possibile scegliere il nuovo frame rate dell'output, sia esso minore o maggiore dell'originale.

In particolare, siano fps_{in} e fps_{out} rispettivamente il frame rate del file in input e quello desiderato dal prodotto dopo la manipolazione eseguita dallo script, vengono distinti i due seguenti casi:

- se $fps_{in} > fps_{out}$ allora i nuovi frame intermedi verranno ottenuti in base ad un processo di **interpolazione**, che varierà a seconda dell'algoritmo utilizzato;
- se $fps_{in} < fps_{out}$ allora i frame intermedi già presenti nel filmato originale verranno scartati mediante un processo di **riduzione**, ed il prodotto risulterà meno fluido.

In entrambi i casi, è importante che la durata del video in input e di quello in output coincidano, e di conseguenza, che il numero di fotogrammi pari a fps_{out} rientri in un'unità di tempo pari esattamente ad un secondo.

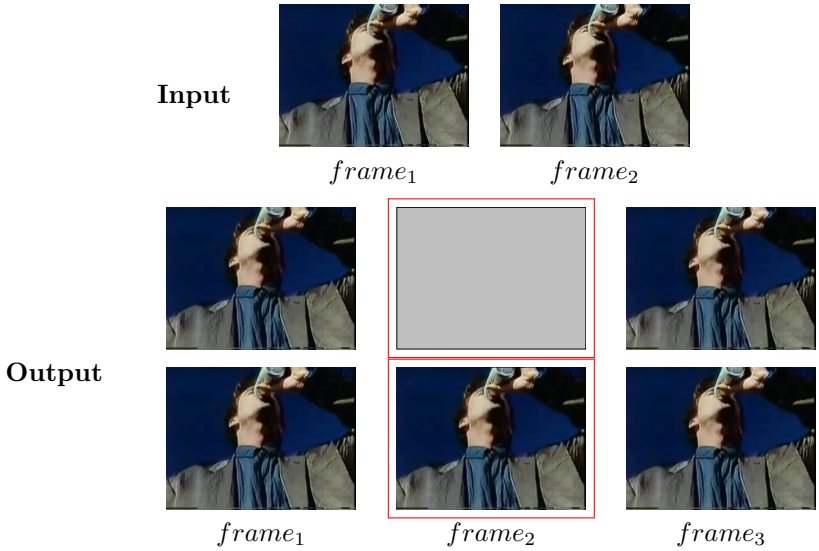


Figura 2: Interpolazione

Figura 3: Riduzione

Per semplicità e per utilità nelle successive trattazioni, dato un video in scala di grigi e di risoluzione $N \times M$, si pone un generico fotogramma $frame_i$ pari alla seguente matrice di dimensione $N \times M$:

$$frame_i = \begin{bmatrix} l_{i_{11}} & \cdots & l_{i_{1M}} \\ \vdots & \ddots & \vdots \\ l_{i_{N1}} & \cdots & l_{i_{NM}} \end{bmatrix} \tag{1}$$

dove i vari coefficienti $l_{i_{xy}}$ (per $x = 1, ..., N$ e $y = 1, ..., M$) corrispondono all'intensità del pixel alle coordinate (x, y) dell'i-esimo fotogramma.

Se nei successivi paragrafi verranno trattati il funzionamento e le leggi matematiche alla base degli algoritmi implementati, il prossimo capitolo li andrà invece a trattare da un punto di vista computazionale, analizzandone le prestazioni su alcuni file di test ed evidenziando pro e contro delle loro rispettive applicazioni.

3.1 Interpolazione per duplicazione

Il metodo di interpolazione per duplicazione (o `dup`) è sicuramente quello più banale, sia da un punto di vista concettuale che computazionale.

Nello specifico, si supponga per semplicità che la frequenza del video in output debba essere $fps_{out} = 2 * fps_{in}$, e che si debba stimare il fotogramma mancante $frame_i$; allora, esso sarà ottenuto replicando il suo predecessore $frame_{i-1}$.
Più in generale, si ha:

$$k = \frac{fps_{out}}{fps_{in}} \tag{2}$$

$$frame_i = frame_{i-1} \quad \forall i = k, ..., k * fps_{out} \tag{3}$$

dove k è il generico valore di scala del nuovo frame rate (cioè, $fps_{out} = k * fps_{in}$).

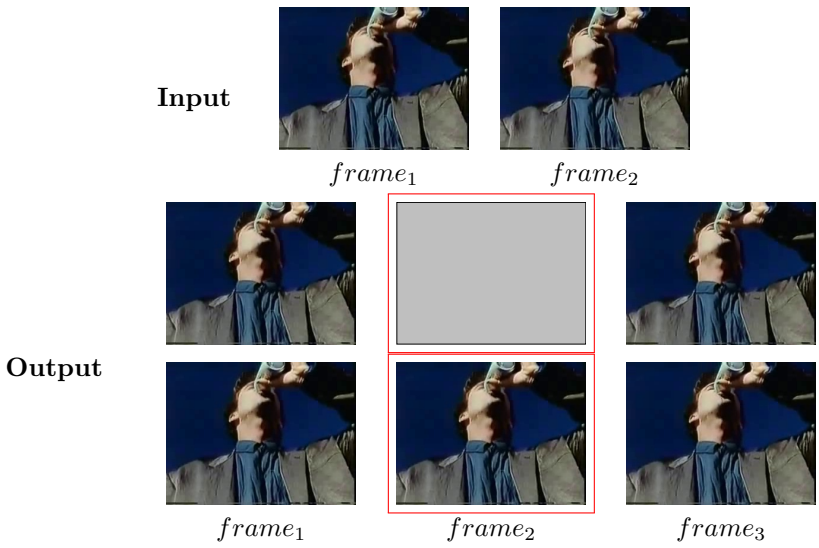


Figura 4: Interpolazione per duplicazione

3.2 Interpolazione per blending

Il processo di interpolazione per blending (o `blend`) opera effettuando una media aritmetica.

In particolare, sia $frame_i$ il frame mancante, e siano $frame_{i-1}$ e $frame_{i+1}$ rispettivamente il suo predecessore ed il suo successore. Allora, si avrà:

$$frame_i = \frac{frame_{i-1} + frame_{i+1}}{2} = \\ = \begin{bmatrix} \frac{l_{i-1_{11}} + l_{i+1_{11}}}{2} & \cdots & \frac{l_{i-1_{1M}} + l_{i+1_{1M}}}{2} \\ \vdots & \ddots & \vdots \\ \frac{l_{i-1_{N1}} + l_{i+1_{N1}}}{2} & \cdots & \frac{l_{i-1_{NM}} + l_{i+1_{NM}}}{2} \end{bmatrix} = \begin{bmatrix} l_{i_{11}} & \cdots & l_{i_{1M}} \\ \vdots & \ddots & \vdots \\ l_{i_{N1}} & \cdots & l_{i_{NM}} \end{bmatrix} \quad (4)$$

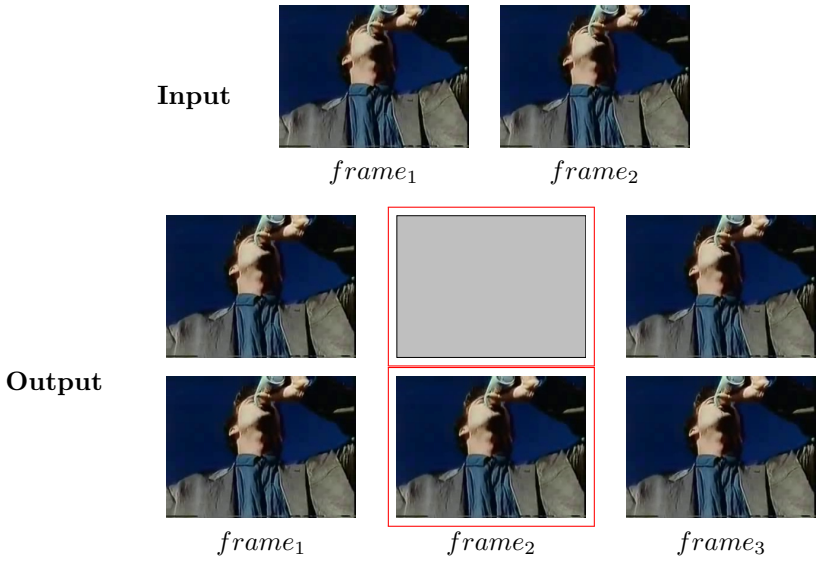


Figura 5: Interpolazione per blending

3.3 Interpolazione attraverso motion compensation

Il metodo di interpolazione attraverso motion compensation (o, più brevemente, **motion interpolation**) è una strategia decisamente diversa da quelle precedentemente trattate, la quale, invece che concentrarsi sui fotogrammi in quanto tali o sui valori di intensità per ogni pixel, opera una stima del movimento da un fotogramma all'altro, per poi sfruttarlo nel corso dell'interpolazione dei fotogrammi mancanti. In realtà, questa strategia viene utilizzata anche nella compressione e nella stabilizzazione video.

In particolare, prima ancora di poter stimare il frame intermedio $frame_i$, dato un **anchor frame** $frame_{i-1}$ ed un **target frame** $frame_{i+1}$, è necessario calcolare i **vettori di movimento** che meglio possono stimare lo spostamento dei punti del soggetto (o della scena in generale) dall'anchor frame al target frame.

L'insieme di questi vettori è rappresentato mediante un pattern (nel caso dei video, bidimensionale) denominato **flusso ottico**, o **optical flow**.

Dato un generico fotogramma, si consideri un pixel di intensità $I(x, y, t)$ al tempo t e di coordinate (x, y): esso, al fotogramma successivo, si sarà mosso di Δx, Δy dopo un tempo Δt; poiché sarà variata

solamente la sua posizione nel tempo, ma non la sua intensità, si ottiene la seguente relazione:

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t) \tag{5}$$

dalla quale, mediante lo sviluppo in serie di Taylor, si ottiene:

$$I(x + \Delta x, y + \Delta y, t + \Delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y + \frac{\partial I}{\partial t}\Delta t + \cdots \tag{6}$$

Troncando poi i termini d'ordine superiore, e dividendo per Δt, si perviene alla seguente uguaglianza:

$$\frac{\partial I}{\partial x}\frac{\Delta x}{\Delta t} + \frac{\partial I}{\partial y}\frac{\Delta y}{\Delta t} + \frac{\partial I}{\partial t} = 0 \tag{7}$$

e ponendo:

- $f_x = \frac{\partial I}{\partial x}$ e $f_y = \frac{\partial I}{\partial y}$;
- $u = \frac{\Delta x}{\Delta t}$ e $v = \frac{\Delta y}{\Delta t}$;
- $f_t = \frac{\partial I}{\partial t}$,

si ottiene infine:

$$f_x u + f_y v + f_t = 0 \tag{8}$$

la quale è denominata **equazione del flusso ottico**, nella quale i termini f_x, f_y e f_t sono i gradienti, mentre u e v sono le incognite, ossia i parametri del vettore di movimento.

Tuttavia, non essendo possibile risolvere un'equazione in due incognite in maniera tradizionale, è necessario ricorrere a diversi metodi di risoluzione. Nello script, sono stati utilizzati il metodo di Gunnar Farnebäck[2] ed il metodo di Lucas-Kanade[1], i quali risolvono due diverse formulazioni del medesimo problema.

3.3.1 Metodo di Gunnar Farnebäck

Il metodo di Gunnar Farnebäck calcola il flusso ottico per ogni punto del frame, in modo da ottene un flusso ottico **denso**, e fa utilizzo dell'**espansione polinomiale**.

La sua idea di base è quella di approssimare alcuni neighbor per ogni pixel mediante il seguente polinomio quadratico:

$$f(x) \sim x^T A x + b^T x + c \tag{9}$$

dove:

- A è una matrice simmetrica;
- b è un vettore;
- c è uno scalare.

Analizzando un caso di **traslazione ideale**, si consideri un fotogramma rappresentato dal seguente polinomio quadratico esatto:

$$f_1(x) = x^T A_1 x + b_1^T x + c_1 \tag{10}$$

A partire da esso, si costruisca il seguente segnale f_2 mediante uno spostamento globale pari a d:

$$\begin{aligned} f_2(x) = f_1(x - d) &= (x - d)^T A_1 (x - d) + b_1^T (x - d) + c_1 = \\ &= x^T A_1 x + (b_1 - 2A_1 d)^T x + d^T A_1 d - b_1^T d + c_1 = \\ &= x^T A_2 x + b_2^T x + c_2 \end{aligned} \tag{11}$$

andando a uguagliare:

$$A_2 = A_1, \tag{12}$$
$$b_2 = b_1 - 2A_1 d, \tag{13}$$
$$c_2 = b_1^T d + c_1. \tag{14}$$

Inoltre, dalla relazione (13) si è in grado di ottenere la traslazione d:

$$2A_1 d = -(b_2 - b_1) \quad \rightarrow \quad d = \frac{1}{2} A_1^{-1}(b_2 - b_1) \tag{15}$$

Ciononostante, realisticamente non è spesso possibile rappresentare un segnale mediante un singolo polinomio, e inoltre andare a relazionare i due fotogrammi attraverso una traslazione globale appare fin troppo semplicistico, considerando che più elementi possono muoversi nella medesima scena, e in direzioni differenti.

Per far sì che l'algoritmo possa ricoprire un caso più generico, è necessario anzitutto applicare l'espansione a entrambi i fotogrammi, ottenendo così i coefficienti di espansione $A_1(x)$, $b_1(x)$ e $c_1(x)$ relativi al $frame_1$, e i coefficienti $A_2(x)$, $b_2(x)$ e $c_2(x)$ relativi al $frame_2$.
Se in teoria, secondo quanto constatato in precedenza mediante la (13), si dovrebbe ottenere l'uguaglianza $A_1(x) = A_2(x)$, all'atto pratico si ha:

$$A(x) = \frac{A_1(x) + A_2(x)}{2}, \tag{16}$$

e inoltre

$$\Delta b(x) = -\frac{1}{2}(b_2(x) - b_1(x)) \tag{17}$$

attraverso la quale si può ricavare, partendo dall'uguaglianza (13):

$$A(x)d(x) = \Delta b(x), \tag{18}$$

dove $d(x)$ non indica più un movimento globale, ma uno spostamento che può variare in verso e direzione, a seconda degli elementi che si muovono nella scena.

Tuttavia, sebbene l'equazione (18) precedentemente ottenuta possa essere risolta punto per punto, nella pratica il risultato avrebbe un'eccessiva quantità di errore, risultando in un flusso ottico troppo rumoroso.
Di conseguenza, si cerca di trovare $d(x)$ assumendo che il motion field varii lentamente, andando a minimizzare la seguente quantità per ogni intorno I di x:

$$\sum_{\Delta x \in I} w(\Delta x) ||A(x + \Delta x)d(x) - \Delta b(x - \Delta x)||^2 \tag{19}$$

dove $w(\Delta x)$ è la funzione peso per i punti dell'intorno. Il minimo è poi ottenuto mediante:

$$d(x) = \left(\sum w A^T A\right)^{-1} \sum w A^T \Delta b \tag{20}$$

nella quale sono stati trascurati gli indici per una maggiore leggibilità. Pertanto, il minimo valore sarà esattamente dato da

$$e(x) = \left(\sum w \Delta b^T \Delta b\right) - d(x)^T \sum w A^T \Delta b \tag{21}$$

Ciò indica che prima vengono calcolati $A^T A$, $A^T \Delta b$ e $\Delta b^T \Delta b$ punto per punto e poi questi vengono pesati mediante w prima di risolvere l'equazione per trovare lo spostamento. L'inverso del minimo valore $e(x)$ può poi essere utilizzato per calcolare l'intervallo di confidenza, con piccoli valori che indicano un'ampia confidenza.

In realtà, la stima dello spostamento può ulteriormente essere raffinata, supponendo a priori la conoscenza di uno spostamento $\tilde{d}(x)$ (arrotondato all'intero più vicino) dal primo al secondo fotogramma, così da ottenere la variazione $x + \tilde{d}(x)$ nel polinomio associato al secondo fotogramma. Quindi, sostituendo nella (16) e nella (17) si ottiene:

$$A(x) = \frac{A_1(x) + A_2(\tilde{x})}{2}, \tag{22}$$

$$\Delta b(x) = -\frac{1}{2}(b_2(\tilde{x}) - b_1(x)) + A(x)\tilde{d}(x), \tag{23}$$

dove:

- $\tilde{x} = x + \tilde{d}(x)$;
- i primi due termini in Δb sono coinvolti nel calcolo del movimento residuo;
- l'ultimo termine in Δb aggiunge nuovamente lo spostamento a priori;
- se $\tilde{d} = 0$, come da aspettativa le due equazioni riportano rispettivamente alla (16) e alla (17).

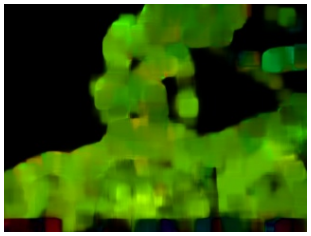
Si distinguono quindi, due diversi approcci di questo metodo di stima del movimento:

- **approccio iterativo**: vengono utilizzati gli stessi coefficienti dell'espansione polinomiale in tutte le iterazioni e allo stesso modo, pertanto vengono calcolati una sola volta; tuttavia, se alla prima iterazione gli spostamenti sono troppo ampi, la stima dello spostamento risultante nei passi successivi non migliorerà, rendendo il metodo inconcludente.
- **approccio multi-scala**: partendo da una scala imprecisa, così da ottenere una stima approssimata dello spostamento, lo si propaga attraverso scale via via più precise, così da ottenere stime di movimento a loro volta sempre più precise.

Si noti come in ambedue gli approcci, lo spostamento calcolato per ogni passo, viene considerato uno spostamento a priori nel passo successivo.

(a) Anchor frame

(b) Target frame



(c) Rappresentazione dei vettori di movimento in HSV

Figura 6: Esempio di ricerca del flusso ottico con metodo di Farnebäck

3.3.2 Metodo di Lucas-Kanade

Partendo dal presupposto che tutti i pixel nell'intorno di un dato pixel p hanno tutti un movimento similare (breve e quasi costante) da un fotogramma al successivo, il metodo di Lucas-Kanade viene applicato **a blocchi**, così da associare ad ogni blocco di pixel un singolo vettore di movimento, ed ottenere così un flusso ottico **sparso**.

Attraverso questa assunzione, l'equazione (8) del flusso ottico attraverso un vettore (u, v) viene modificata così da considerare tutti i pixel di un blocco centrato in p; si ottiene così il seguente sistema:

$$\begin{aligned} &f_x(q_1)u + f_y(q_1)v = -f_t(q_1) \\ &f_x(q_2)u + f_y(q_2)v = -f_t(q_2) \\ &\vdots \\ &f_x(q_n)u + f_y(q_n)v = -f_t(q_n) \end{aligned} \tag{24}$$

dove:

- q_1, q_2, …, q_n sono i pixel dentro il blocco;
- $f_x(q_i)$, $f_y(q_i)$, $f_t(q_i)$ (per $i = 1, \ldots, n$) sono i gradienti del fotogramma f rispetto alle posizioni x, y e al tempo t, calcolate per il pixel q_i.

Si giunge quindi alla forma matriciale $Ah = b$, dove

$$A = \begin{bmatrix} f_x(q_1) & f_y(q_1) \\ f_x(q_2) & f_y(q_2) \\ \vdots & \vdots \\ f_x(q_n) & f_y(q_n) \end{bmatrix}, \qquad h = \begin{bmatrix} u \\ v \end{bmatrix}, \qquad b = \begin{bmatrix} -f_t(q_1) \\ -f_t(q_2) \\ \vdots \\ -f_t(q_n) \end{bmatrix}$$

Essendo un sistema con più equazioni (esattamente n) che incognite (due, le componenti del vettore colonna $h = (u, v)^T$), il metodo di Lucas-Kanade applica il **criterio dei minimi quadrati**, così da risolvere il sistema:

$$A^T Ah = A^T b \quad \Longleftrightarrow \quad h = (A^T A)^{-1} A^T b \tag{25}$$

dove A^T è la trasposta della matrice A. Si ottiene così la seguente matrice 2×2:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^n f_x(q_i)^2 & \sum_{i=1}^n f_x(q_i) f_y(q_i) \\ \sum_{i=1}^n f_y(q_i)^2 f_x(q_i) & \sum_{i=1}^n f_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum_{i=1}^n f_x(q_i) f_t(q_i) \\ \sum_{i=1}^n f_y(q_i) f_t(q_i) \end{bmatrix} \tag{26}$$

Affinché l'equazione (25) sia **risolvibile**, la matrice $A^T A$ dovrebbe essere **invertibile**, o i suoi **autovalori** dovrebbero soddisfare la condizione $\lambda_1 \geq \lambda_2 \geq 0$; inoltre, è necessario che λ_2 non sia troppo piccolo, onde evitare rumore. Inoltre, se il rapporto $\frac{\lambda_1}{\lambda_2}$ è troppo grande, allora il punto p è un **edge**. Pertanto, affinché questo metodo funzioni bene, è necessario che λ_1 e λ_2 siano grandi abbastanza da avere simile magnitudine.

Un altro aspetto da non trascurare, è che nella (25) ad ogni pixel è data la stessa importanza, è possibile affinare ulteriormente la stima introducendo un **peso**; partendo dalla (25), si ottiene quindi:

$$A^T WAh = A^T WB \quad \Longleftrightarrow \quad h = (A^T WA)^{-1} A^T Wb \tag{27}$$

dove W è una matrice diagonale di dimensione $n \times n$ contenente i pesi $W_{ii} = w_i$ associati all'equazione del pixel q_i, dove tipicamente il peso w_i è dato dalla funzione Gaussiana della distanza tra il pixel q_i ed il pixel centrale p; ciò riporta alla seguente soluzione:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^n w_i f_x(q_i)^2 & \sum_{i=1}^n w_i f_x(q_i) f_y(q_i) \\ \sum_{i=1}^n w_i f_y(q_i)^2 f_x(q_i) & \sum_{i=1}^n w_i f_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum_{i=1}^n w_i f_x(q_i) f_t(q_i) \\ \sum_{i=1}^n w_i f_y(q_i) f_t(q_i) \end{bmatrix} \tag{28}$$

Quindi, per tracciare il movimento attraverso questo metodo, è necessario che il vettore di flusso sia iterativamente applicato e ricalcolato, finché non viene raggiunta una soglia vicina allo zero, indicando

quindi che le finestre del fotogrammi sono molto simili; effettuando quest'operazione per ogni finestra, il punto può essere tracciato attraverso una sequenza di immagini, finché non è oscurato (e quindi la sua intensità è nulla) o non esce fuori dal fotogramma.

(a) Anchor frame

(b) Target frame

(c) Rappresentazione dei vettori di movimento in HSV

Figura 7: Esempio di ricerca del flusso ottico con metodo di Lucas-Kanade

3.4 Riduzione

Viene infine trattato l'ultimo metodo di manipolazione dei filmati, totalmente differente dalle strategie differenti in quanto diverso anche il suo obbiettivo d'utilizzo.
Difatti, questa strategia consiste in una vera e propria riduzione del numero di fotogrammi che compone l'input, in modo da aver sì un prodotto qualitativamente peggiore – in quanto i movimenti risulteranno meno "fluidi" –, ma allo stesso tempo anche meno oneroso in termini di spazio d'archiviazione.

In realtà, concettualmente, questa strategia è abbastanza semplice rispetto alle precedenti: basti pensare, per esempio al caso in cui si ha un input da $30fps$ e che si desideri un prodotto da $15fps$, allora dati i fotogrammi $frame_{i-1}$, $frame_i$ e $frame_{i+1}$, basterà semplicemente scartare il fotogramma intermedio $frame_i$.

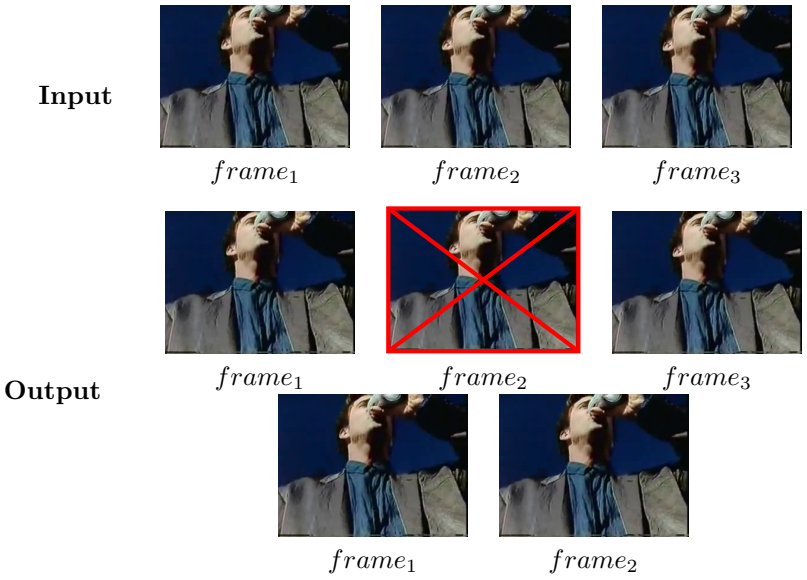


Figura 8: Riduzione

4 Test e risultati

Tutti i test sono stati effettuati su una macchina con le seguenti specifiche:

- **CPU**: Intel Core i7 7th gen. (7700HQ);
- **RAM**: 16 GB;
- **Sistema Operativo**: GNU/Linux Manjaro KDE;
- **Kernel Linux**: `5.16.5-1-MANJARO`.

In particolare, verranno considerati i tre seguenti file di test:

- `rgb.mp4`: semplice filmato dalla **durata** di 49 secondi, il quale mostra in sequenza i colori (puri) bianco, rosso, verde, blu e nero; la sua **risoluzione** in pixel è 1920×1080 ed il suo **peso** è di 10,4 MB;
- `bouncing ball.mp4`: filmato dalla **durata** di 35 secondi che mostra un cerchio bianco che si muove su uno sfondo nero, rimbalzando tra i bordi dello schermo; la sua **risoluzione** in pixel è 1920×1080 ed il suo **peso** è di 944 KB;
- `asahi.mp4`: filmato dalla **durata** di 15 secondi, il quale presenta uno spot pubblicitario giapponese degli anni '80 con protagonista Mel Gibson; la sua **risoluzione** in pixel è 320×240 ed il suo **peso** è 654 KB;

inoltre i tre file hanno tutti un **frame rate** pari a 30 fps.
Nei test sono state effettuate tutte **interpolazioni** a 60 fps e **riduzioni** a 5 fps, sebbene siano comunque possibili altre configurazioni.

Si noti la diversa natura dei tre filmati presi in esame, così da poter essere in grado di considerare non solo le differenze prestazionali dei singoli algoritmi applicati su essi, ma anche in quale contesto risulterebbe essere più vantaggioso, da un punto di vista qualitativo, utilizzare un metodo di interpolazione piuttosto che uno di natura differente.

4.1 Duplicazione

Come già accennato nel precedente capitolo, l'algoritmo di duplicazione è sicuramente non solo quello concettualmente più semplice, ma anche quello meno oneroso da un punto di vista computazionale; infatti, in tutti i tre casi, il tempo impiegato per avere un file di output è inferiore al minuto.
Cionondimeno, è importante fare delle considerazioni relative all'utilizzo delle risorse rispettivamente ai file sui quali l'algoritmo è stato eseguito.

`rgb.mp4` Su questo file è stato registrato, ignorando il picco iniziale del caricamento del file in input (tra il secondo 5 ed il secondo 10), un utilizzo della CPU oscillante attorno al 100%; analizzando attentamente il grafico, è possibile notare che tale picco resta stabile per circa 23 secondi, effettiva durata della manipolazione del filmato per mezzo dell'algoritmo. L'alto utilizzo della memoria, approssimativamente vicino ai 9000 MB, è invece dovuto non solo all'alta risoluzione del filmato (per un totale di esattamente 2.073.600 pixel per fotogramma), ma alla presenza di uno statico bianco puro dalla durata di circa 10 secondi, decisamente più pesante di altri colori (che in complessivamente, contribuiscono ad un peso del file di molto maggiore, se paragonato agli altri due).

(a) Anchor Frame (b) Frame interpolato (c) Target frame

Figura 9: Esempio di interpolazione per duplicazione su `rgb.mp4`

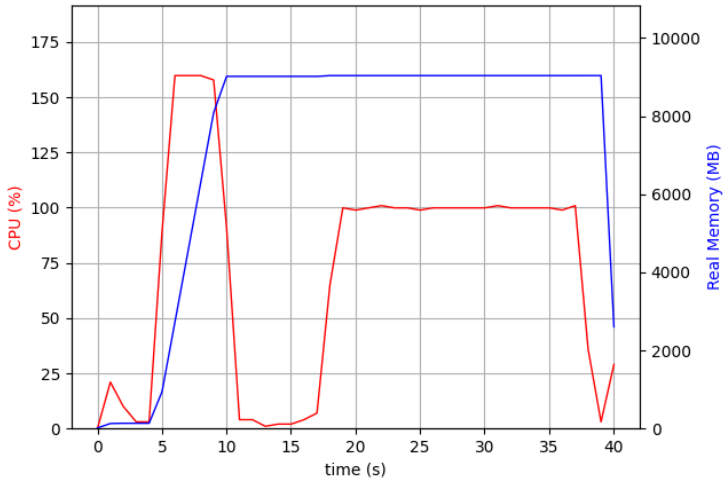
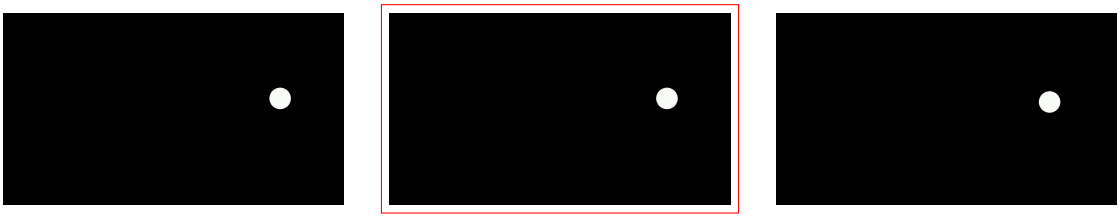


Figura 10: Grafico prestazionale dell'algoritmo di duplicazione su `rgb.mp4`

`bouncing ball.mp4` Anche in questo caso, ignorando il picco di caricamento del file, è possibile notare un utilizzo della CPU del 100% nel corso della produzione dell'output; cionondimeno, è possibile durare non solo l'inferiore durata del processo, di circa 16/17 secondi, ma anche un inferiore utilizzo della memoria, dovuto al minor peso del file ed al suo basso contenuto di colori (sola presenza del bianco su sfondo nero).



(a) Anchor frame (b) Frame interpolato (c) Target frame

Figura 11: Esempio di interpolazione per duplicazione su `bouncing ball.mp4`

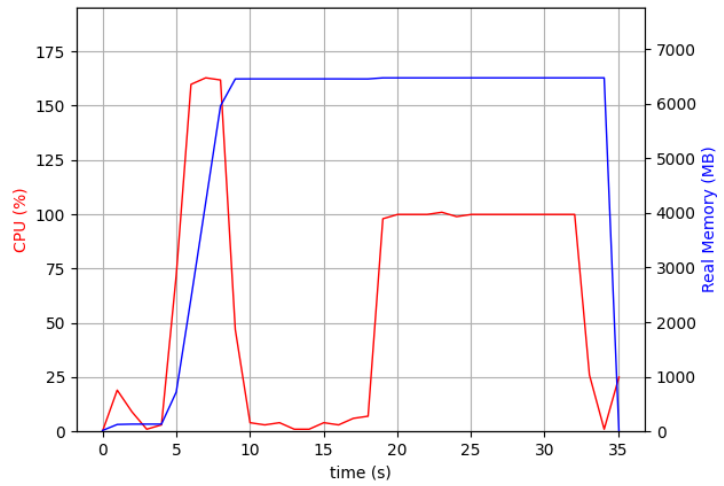


Figura 12: Grafico prestazionale dell'algoritmo di duplicazione su `bouncing ball.mp4`

`asahi.mp4` Su questo file, l'algoritmo di duplicazione ha impiegato il minor tempo nella produzione dell'output, di appena 3 secondi, e con un blando utilizzo delle risorse CPU; si notano infatti un solo picco pari a circa il 37%, ed un utilizzo della memoria pari a 250 MB. È importante notare come la minor risoluzione (con un totale di 76.800 pixel per fotogramma) ed il minore peso complessivo del file abbiano contribuito al minor impatto sulle risorse, nel corso dell'esecuzione dell'algoritmo.

(a) Anchor frame (b) Frame interpolato (c) Target frame

Figura 13: Esempio di interpolazione per duplicazione su `asahi.mp4`

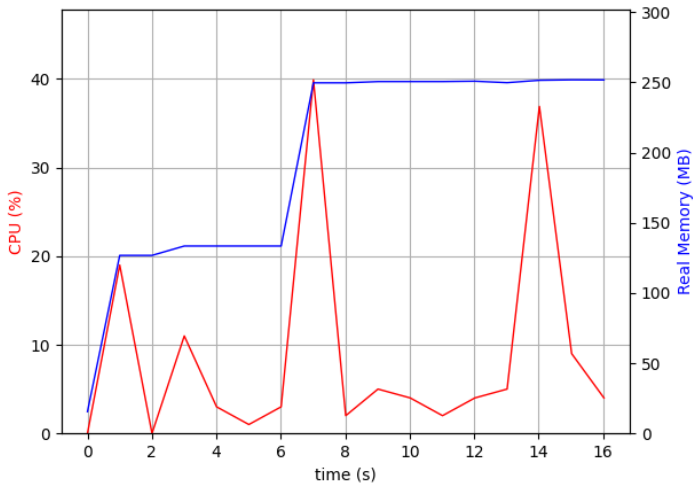


Figura 14: Grafico prestazionale dell'algoritmo di duplicazione su `asahi.mp4`

Generalmente, da un punto di vista qualitativo, l'algoritmo ha avuto un discreto impatto su tutti i tre file: difatti, i tre output risultano essere leggermente più fluidi dei filmati originali, ma solo effettuando un'effettiva comparativa di tipo full-reference. Infatti, il fatto che i fotogrammi interpolati siano un semplice duplicato dei precedenti, non incide particolarmente sulle transizioni.

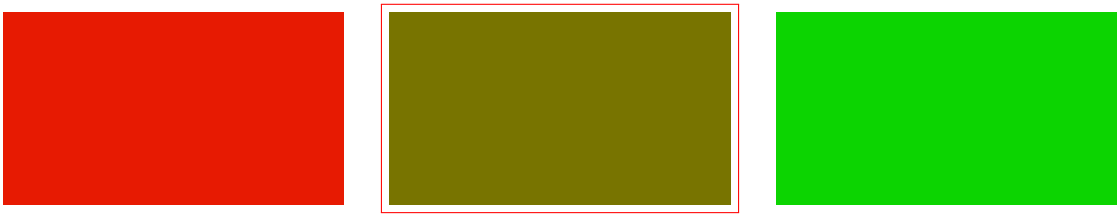
Di seguito, vengono riportati i pesi dei file prima dopo il processo di interpolazione (è da considerarsi anche la compressione).

File	Peso originale	Peso interpolazione
`rgb.mp4`	10,4 MB	8,02 MB
`bouncing ball.mp4`	944 KB	8,03 MB
`asahi.mp4`	654 KB	1,44 MB

4.2 Blending

Da un punto di vista computazionale, la strategia di blending si è rivelata essere generalmente più pesante della precedente, ma comunque in grado di offrire un accettabile compromesso tra utilizzo delle risorse, tempo impiegato alla produzione dell'output (in ogni caso superiore alle tempistiche viste nell'analisi del metodo di duplicazione), e qualità visiva.

`rgb.mp4` È possibile notare, nella fascia del grafico che denota la produzione dell'output, due diversi utilizzi delle risorse: in un primo tempo, è possibile infatti notare un utilizzo abbastanza costante della CPU, con oscillazioni attorno al 100% (con un costante aumento della memoria utilizzata nel tempo); successivamente, si nota un'oscillazione dell'utilizzo della CPU che varia tra il 20% ed il 55%, unito ad un impatto sulla memoria più costante, che si ferma sui 14 GB. Il tutto, per un'esecuzione totale pari a circa un quarto d'ora. È importante notare come la causa di questi due momenti sia da attribuire alla natura del filmato: infatti, per circa i suoi tre quarti, il video è composto da passaggi di colore (bianco, rosso, verde e blu), per poi passare ad un nero totale, che ha un impatto decisamente minore sulle risorse nel corso dell'interpolazione, visto che le operazioni matematiche sono minimizzate.
Nella Figura 15 è mostrato un esempio dell'interpolazione partendo da un fotogramma e dal suo successivo.



(a) Anchor frame (b) Frame interpolato (c) Target frame

Figura 15: Esempio di interpolazione per blending su `rgb.mp4`

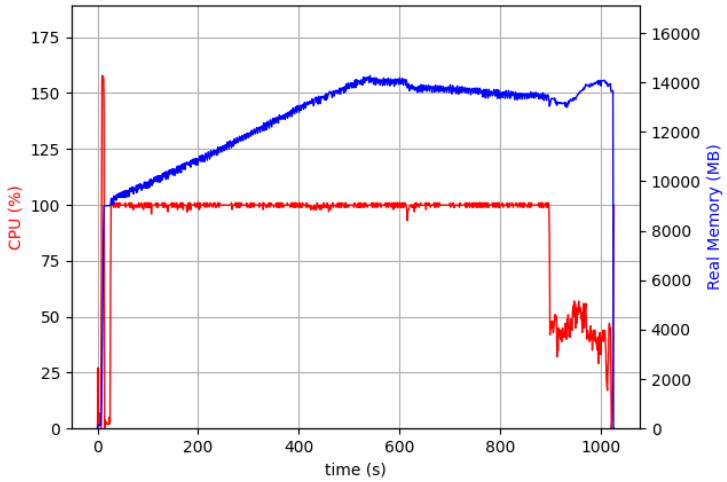
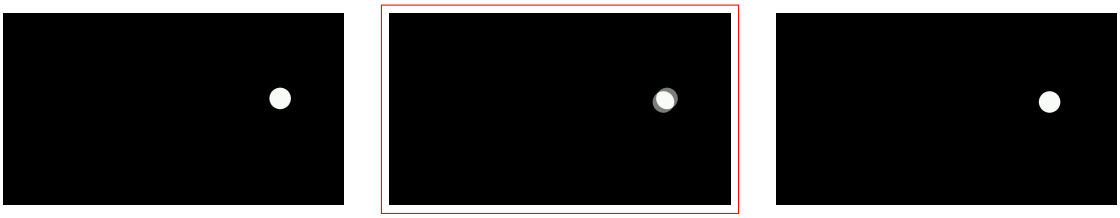


Figura 16: Grafico prestazionale dell'algoritmo di blending su `rgb.mp4`

`bouncing ball.mp4` In questo caso, la produzione ha impiegato circa dieci minuti, durante i quali l'utilizzo della CPU è rimasto stabile attorno al 100%, mentre l'impatto sulla memoria ha avuto un aumento costante nel tempo fino ad arrivare ad approssimativamente 12,5 GB. Ciò è dovuto al fatto che, il filmato originale presenta un numero di pixel anch'esso costante nel tempo in base al colore (un cerchio bianco di un certo raggio, sempre uguale, che su muove su uno sfondo nero).
In Figura 17 è mostrato un esempio di interpolazione su questo file.



(a) Anchor frame (b) Frame interpolato (c) Target frame

Figura 17: Esempio di interpolazione per blending su `bouncing ball.mp4`

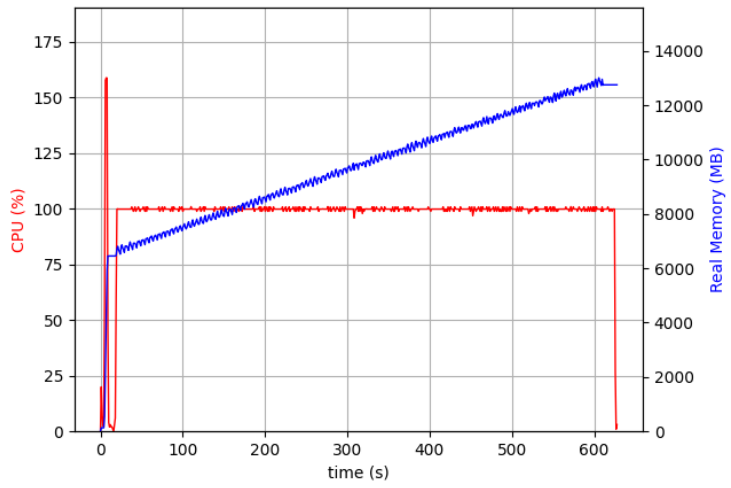


Figura 18: Grafico prestazionale dell'algoritmo di blending su `bouncing ball.mp4`

`asahi.mp4` La produzione dell'output, partendo da questo file, è stata decisamente la meno duratura nell'utilizzo del metodo di blending. Infatti, dal grafico è possibile notare che la computazione ha impiegato appena dieci secondi, nettamente inferiore rispetto a quanto impiegato avendo in input i due file precedenti, durante i quali la CPU ha lavorato con un carico mediamente pari al 100%; le considerazioni sull'utilizzo della memoria non sono dissimili dai due file precedenti, ma lo è il suo carico: infatti, nonostante vi sia stato un effettivo costante aumento, l'utilizzo massimo è pari a poco più di appena 350 MB, da attribuirsi all'inferiore risoluzione rispetto agli altri video presi in esame. In Figura 19 è mostrato un esempio di interpolazione su questo file.

(a) Anchor frame (b) Frame interpolato (c) Target frame

Figura 19: Esempio di interpolazione per blending su `asahi.mp4`

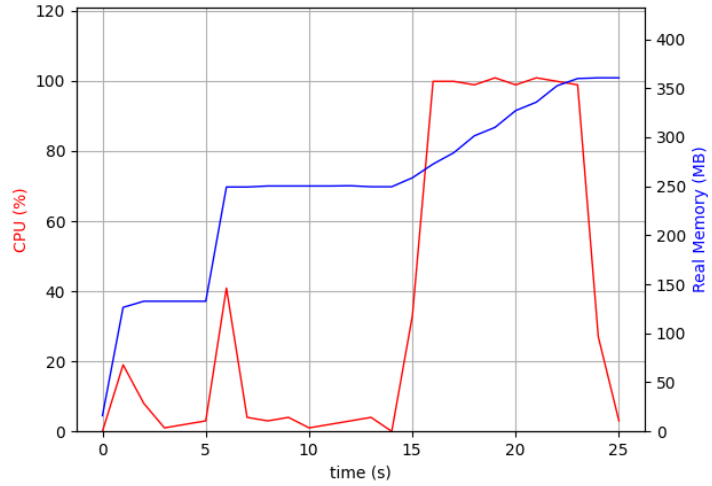


Figura 20: Grafico prestazionale dell'algoritmo di blending su `asahi.mp4`

Di seguito si illustrano inoltre i rispettivi pesi dei file prima e dopo l'applicazione dell'interpolazione mediante blending.

File	Peso originale	Peso interpolazione
`rgb.mp4`	10,4 MB	8,10 MB
`bouncing ball.mp4`	944 KB	11,00 MB
`asahi.mp4`	654 KB	1,91 MB

4.3 Motion compensated interpolation secondo Farnebäck

È già stato accennato come la motion compensation sia computazionalmente più pesante, tra tutti i metodi descritti in questa palese, e le analisi sperimentali ne concederanno una diretta dimostrazione. Per quanto riguarda la motion compensation di Gunnar Farnebäck, i migliori risultati visivi sono stati ottenuti esclusivamente in quei filmati che presentano soggetti i movimenti, al prezzo di una maggior durata generale della computazione.

`rgb.mp4` Tra i tre file, questo è sicuramente quello sul quale l'algoritmo di Farnebäck ha avuto i peggiori risultati sperimentali. Anche in questo caso, data la natura del video possiamo suddivitere la computazione in due momenti temporali: il primo, durante il quale l'interpolazione è stata effettuata sui colori bianco, rosso, verde e blu, e il secondo momento, durante il quale l'interpolazione è stata effettuata su un nero puro. Durante la prima fase si ha avuto un utilizzo totale della CPU oscillante tra il $125\% - 160\%$ circa, con aumenti lineari della memoria intervallati da momenti di stasi, fino a raggiungere un picco di 14 GB; il secondo momento, è quello durante il quale i calcoli sono stati effettuati su un nero puro, e quindi minimizzati a causa della sua natura. Sperimentalmente, è importante notare come, al prezzo di un utilizzo così intenso delle risorse, il risultato sul presente filmato sia stato praticamente identico a quello ottenuto mediante l'algoritmo di duplicazione, visto che in esso non è presente alcun tipo di movimento.

(a) Anchor frame (b) Frame interpolato (c) Target frame

Figura 21: Esempio di motion compensated interpolation di Farnebäck su `rgb.mp4`

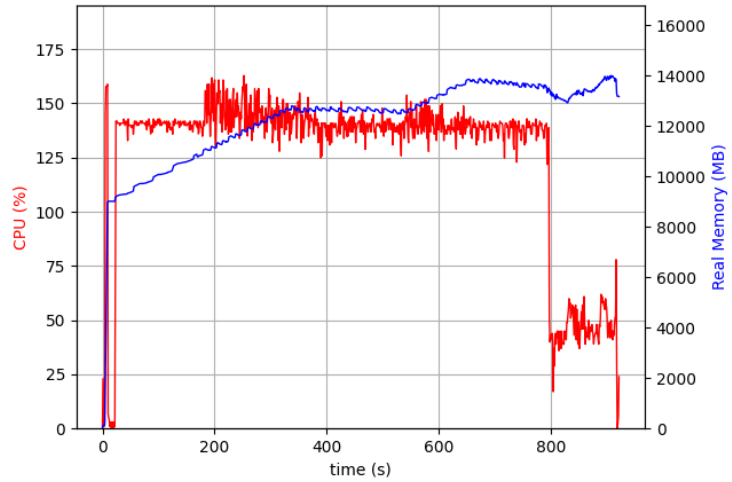
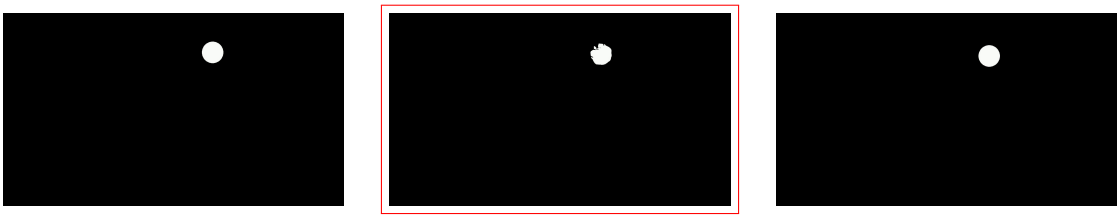


Figura 22: Grafico prestazionale della motion compensated interpolation di Farnebäck su `rgb.mp4`

`bouncing ball.mp4` In questo caso, la CPU ha computato per circa dieci minuti (durata minore rispetto al file precedente, al netto della sua risoluzione, a causa del suo contenuto visivo), prima di dare in output un risultato: in particolare, il carico su di essa ha oscillato tra $125\% - 175\%$ con un aumento dell'utilizzo in memoria mediamente lineare. Da notare, tuttavia, è che l'interpolazione presenta degli artefatti grafici, visto che alcuni pixel neri hanno coperto parte del cerchio.



(a) Anchor frame (b) Frame interpolato (c) Target frame

Figura 23: Esempio di motion compensated interpolation di Farnebäck su `bouncing ball.mp4`

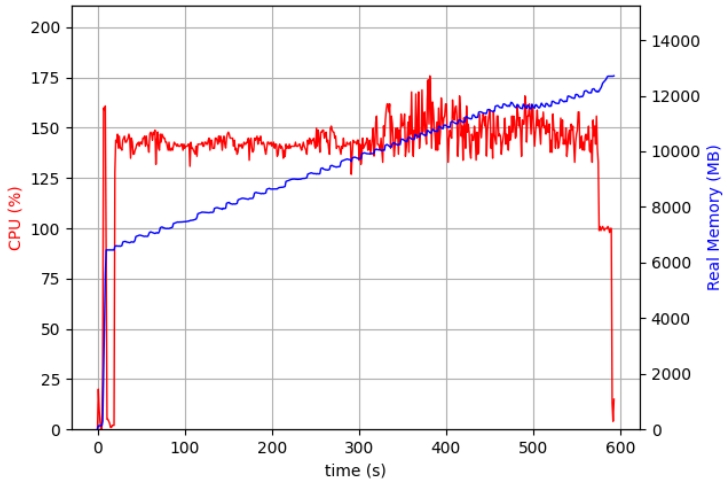


Figura 24: Grafico prestazionale della motion compensated interpolation di Farnebäck su `bouncing ball.mp4`

`asahi.mp4` Anche nel caso dell'interpolazione mediante il metodo di Farnebäck, questo file è risultato essere quello con la computazione meno duratura. Con un carico sulla CPU del 100%, e con un lineare aumento del peso in memoria fino a 350 MB, l'algoritmo ha infatti impiegato appena dieci secondi nel produrre l'output; come sempre, la causa di ciò è da ritrovarsi nella minore risoluzione del filmato rispetto ai precedenti due. In ogni caso, è bene notare come, tra i tre, l'output di questo file è risultato essere quello qualitativamente migliore da un punto di vista visivo: infatti, l'interpolazione è stata effettuata prendendo come riferimento i movimenti dell'attore, mediante una parametrizzazione esaustiva dei vettori di movimento per ogni pixel.

(a) Anchor frame (b) Frame interpolato (c) Target frame

Figura 25: Esempio di motion compensated interpolation di Farnebäck su `asahi.mp4`

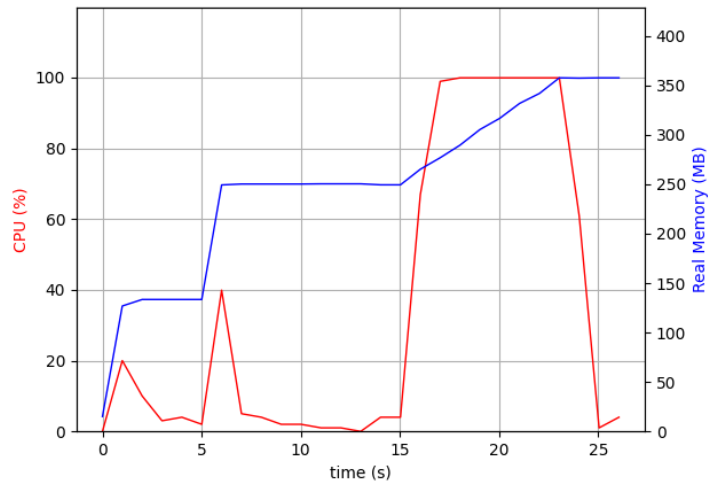


Figura 26: Grafico prestazionale della motion compensated interpolation di Farnebäck su `asahi.mp4`

Di seguito si illustrano inoltre i rispettivi pesi dei file prima e dopo l'applicazione della motion compensated interpolation di Farnebäck.

File	Peso originale	Peso interpolazione
`rgb.mp4`	10,4 MB	8,02 MB
`bouncing ball.mp4`	944 KB	10,5 MB
`asahi.mp4`	654 KB	1,88 MB

4.4 Motion compensated interpolation secondo Lucas-Kanade

Trattandosi di un algoritmo di compensazione per flussi ottici sparsi, il metodo di Lucas-Kanade è risultato essere sperimentalmente il metodo più oneroso tra tutti, persino più del metodo di Farnebäck. Infatti, in due casi è stata ampiamente superata un'ora, prima di avere un effettivo prodotto in output.

`rgb.mp4` Se il metodo di Farnebäck è risultato essere svantaggioso nell'interpolazione di questo filmato, il metodo di Lucas-Kanade si rivela esserlo ancor di più: si tratta infatti della computazione più duratura misurata sperimentalmente (approssimativamente due ore e un quarto), e con un utilizzo delle risorse non indifferente; il carico sulla CPU oscilla tra il $620\% - 800\%$, ed il peso sulla memoria ammonta a 14 GB. Un prezzo elevatissimo da pagare, considerando che, a causa dell'assenza di movimenti nel filmato, il risultato finale è praticamente identico – anche in questo caso – a quello ottenuto mediante il metodo di duplicazione.

(a) Anchor frame (b) Frame interpolato (c) Target frame

Figura 27: Esempio di motion compensated interpolation di Lucas-Kanade su `rgb.mp4`

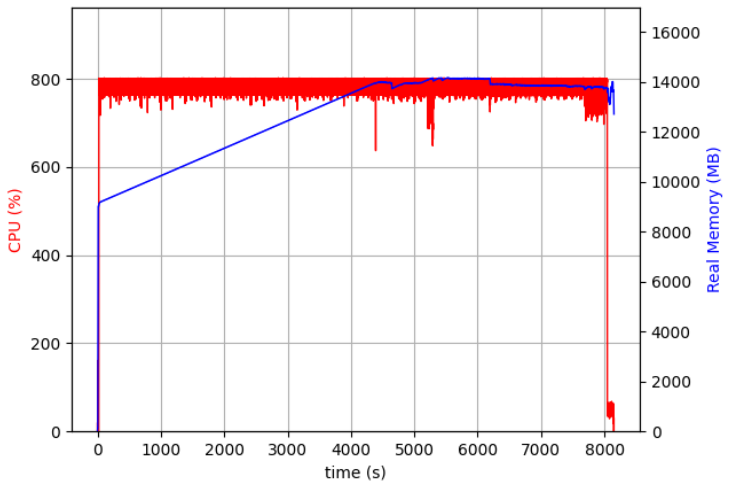
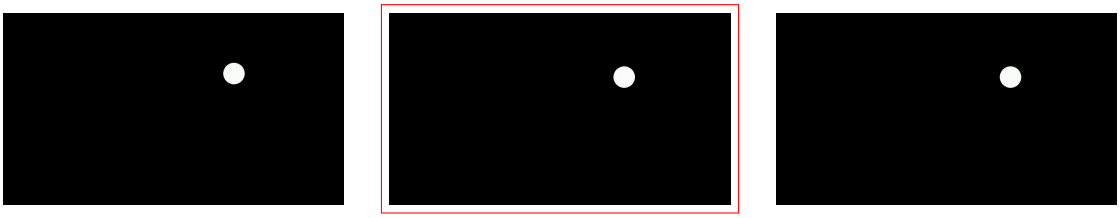


Figura 28: Grafico prestazionale della motion compensated interpolation di Lucas-Kanade su `rgb.mp4`

`bouncing ball.mp4` Anche in questo caso è stato registrato un ampio utilizzo delle risorse: la computazione è infatti terminata dopo circa un'ora e tre quarti, con un carico sulla CPU tra il $690\% - 800\%$ ed un peso in memoria pari a quasi 13 GB. Ciononostante, i risultati sono decisamente più apprezzabili rispetto al file precedente: oltre a vantare una maggiore fluidità, la parametrizzazione relativa ha ridotto gli artefatti grafici, che invece erano visibili nell'output ottenuto mediante l'algoritmo di Farnebäck. Ciò è da ricondursi al fatto che nel video sono presenti solamente due aree di colore.



(a) Anchor frame (b) Frame interpolato (c) Target frame

Figura 29: Esempio di motion compensated interpolation di Lucas-Kanade su `bouncing ball.mp4`

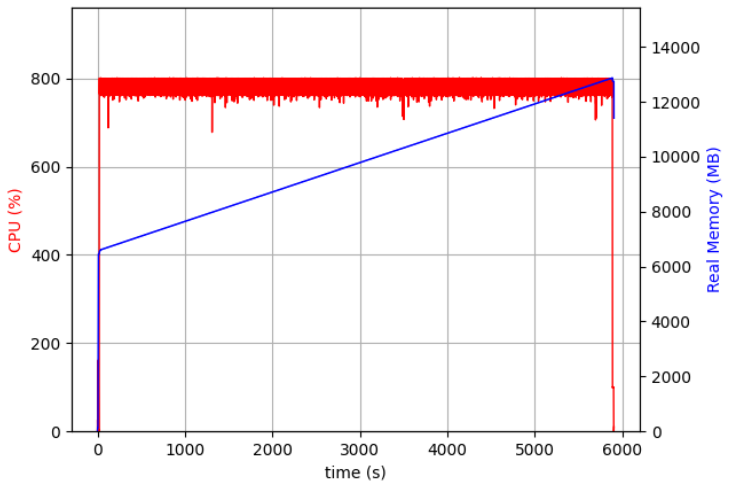


Figura 30: Grafico prestazionale della motion compensated interpolation di Lucas-Kanade su `bouncing ball.mp4`

`asahi.mp4` Sebbene tra i test eseguiti su questo filmato, il metodo di Lucas-Kanade sia quello che più ha impiegato nella sua computazione, si è comunque rivelato essere quello meno duraturo se paragonato ai due test analizzati in precedenza. La computazione del filmato in output è infatti durata poco più di quattro minuti, e l'utilizzo delle risorse si è rivelato essere in linea a quanto detto in precedenza, con un carico medio sulla CPU del 800% circa, e con un costante aumento del peso in memoria, fino a raggiungere i 350 MB.
Tuttavia in questo caso, nonostante la maggior fluidità dei movimenti dell'attore, è possibile notare degli artefatti grafici nel frame interpolato.

(a) Anchor frame (b) Frame interpolato (c) Target frame

Figura 31: Esempio di motion compensated interpolation di Lucas-Kanade su `bouncing ball.mp4`

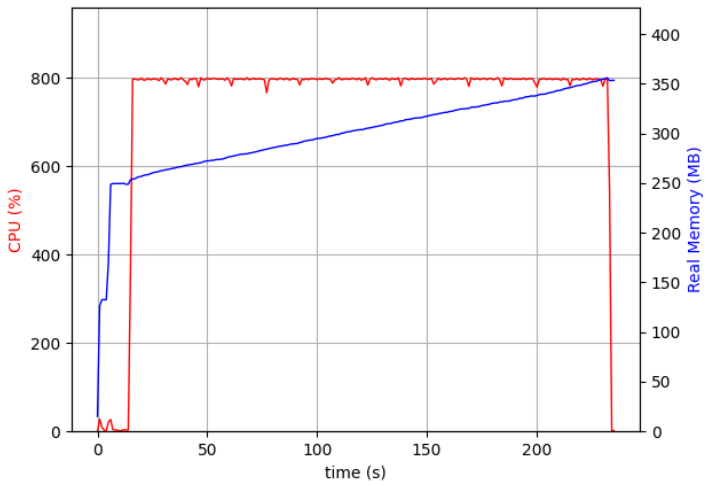


Figura 32: Grafico prestazionale della motion compensated interpolation di Lucas-Kanade su `asahi.mp4`

Di seguito si illustrano inoltre i rispettivi pesi dei file prima e dopo l'applicazione della motion compensated interpolation di Lucas-Kanade.

File	Peso originale	Peso interpolazione
`rgb.mp4`	10,4 MB	8,02 MB
`bouncing ball.mp4`	944 KB	9,34 MB
`asahi.mp4`	654 KB	2,12 MB

4.5 Riduzione

Come già affrontato in precedenza, questo algoritmo è decisamente differente da quelli presentati in precedenza; infatti, piuttosto che interpolare un fotogramma intermedio, esso lo va a scartare, dando in output un filmato caratterizzato da un frame rate minore, e di conseguenza anche da un peso minore rispetto all'input, com'è possibile notare nella tabella alla fine di questo semicapitolo.
Generalmente infatti, questo algoritmo si è rivelato essere non solo quello meno oneroso in termini di risorse, ma anche quello più veloce su tutti e tre i file di test. Si ricorda che da filmati a 30 fps, si è passati a filmati a 5 fps.

`rgb.mp4` Escludendo come di consueto il picco iniziale, dovuto al caricamento del file in input, si ha un solo picco di utilizzo della CPU del 100%, con una durata circa tre secondi. L'utilizzo della memoria raggiunge invece un picco massimo di poco più 10 GB.

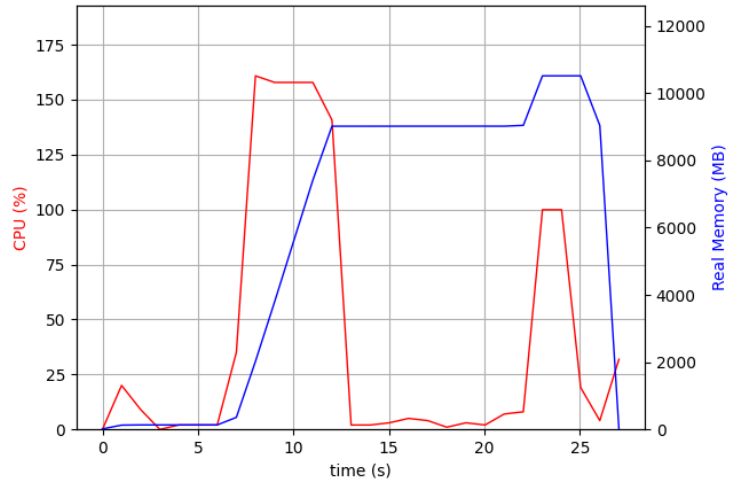


Figura 33: Grafico prestazionale della riduzione a 5 fps su `rgb.mp4`

`bouncing ball.mp4` In questo caso, la computazione è durata circa un paio di secondi, con un impatto massimo sulla CPU pari a circa l'80% ed un peso in memoria pari a poco meno di 8 GB.

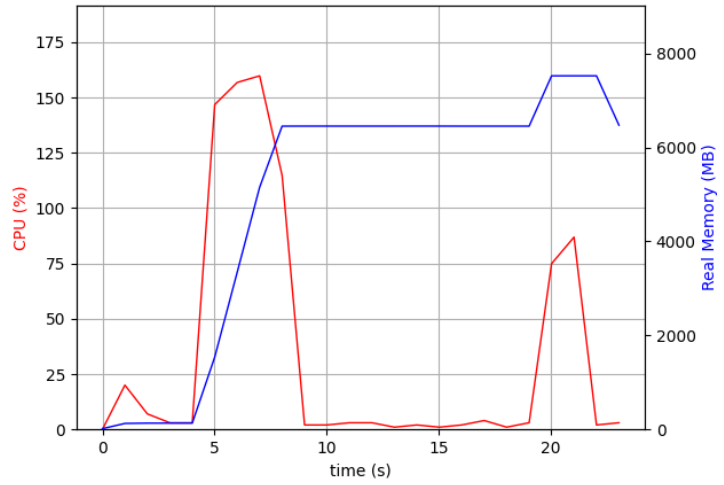


Figura 34: Grafico prestazionale della riduzione a 5 fps su `bouncing ball.mp4`

`asahi.mp4` Le prestazioni migliori sono state registrate su questo file, in quanto quello caratterizzato dalla risoluzione minore: infatti, anche in questo caso la computazione è durata circa un paio di secondi, ma l'utilizzo delle risorse è stato il meno cospicuo: l'utilizzo della CPU ammonta ad appena il 10%, mentre il peso sulla memoria è di poco superiore a 250 MB, nel corso dell'operazione.

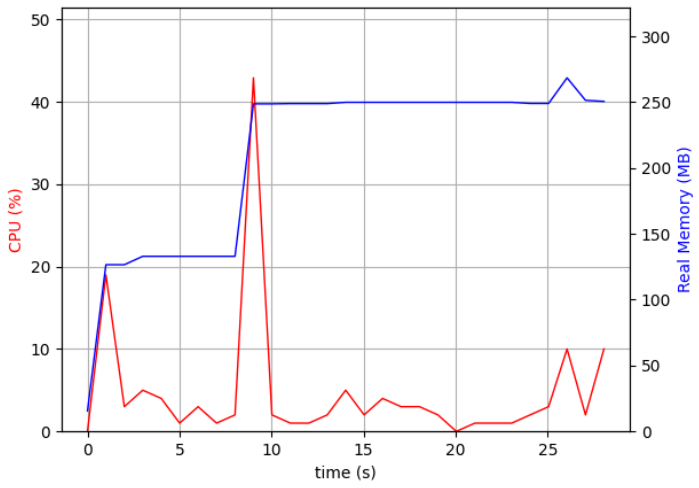


Figura 35: Grafico prestazionale della riduzione a 5 fps su `asahi.mp4`

Di seguito si illustrano inoltre i rispettivi pesi dei file prima e dopo l'applicazione della riduzione a 5 fps.

File	Peso originale	Peso interpolazione
`rgb.mp4`	10,4 MB	774 KB
`bouncing ball.mp4`	944 KB	1,01 MB
`asahi.mp4`	654 KB	401 KB

4.6 Considerazioni sui risultati

Traendo le somme dalle analisi effettuate in precedenza, è possibile fare ulteriori considerazioni sui vari algoritmi di manipolazione, e in quali contesti si potrebbe rivelare più proficuo utilizzarne uno piuttosto che un altro.

Il metodo di duplicazione si potrebbe quasi definire un algoritmo "lazy", in grado di offrire in ogni caso discreti risultati, con il minimo sforzo computazionale e al minor tempo possibile.

Il metodo di blending potrebbe effettivamente offrire il miglior compromesso tra risultato, risorse e tempo impiegato nella produzione nell'output, ma realisticamente, il vero miglior risultato viene offerto nel caso in cui si hanno delle schermate interamente caratterizzate da colori puri: in tal caso, infatti, la transizione da un colore all'altro risulterebbe meno netta di un passaggio istantaneo, e una transizione caratterizzata da diverse sfumature di colore intermedie risulterebbe decisamente più gradevole all'occhio umano.

Relativamente alla motion compensated interpolation, si può dire che in entrambi i casi, è decisamente svantaggioso andare ad utilizzare la stima del flusso ottico in filmati che non sono caratterizzati da un qualsiasi tipo di movimento. Al prezzo di un eccessivo utilizzo delle risorse, e di un lungo tempo di produzione dell'output, si avrebbe un risultato praticamente identico a quello che si otterrebbe andando ad utilizzare il meno oneroso metodo di duplicazione (ciò è anche evidenziato all'uguale peso dell'output risultato dal file `rgb.mp4` nel caso della duplicazione e di entrambi i metodi di motion compensation).
Cionondimeno, è importante distinguere le prestazioni del metodo di Farnebäck dal metodo di Lucas-

Kanade nell'applicazione su video che presentano dei movimenti: il primo infatti è in grado di produrre dei video visivamente migliori e privi di artefatti grafici nel caso in cui le immagini siano complesse, e non caratterizzate da uniche tinte di colore, in quanto in caso contrario si avranno degli artefatti grafici com'è possibile notare in Figura 23; viceversa, il metodo di Lucas-Kanade si rivela essere più vantaggioso nel caso in cui il filmato sia caratterizzato da uniche tinte di colore, in quanto nel caso di fotogrammi più eterogeneo si avrebbero degli artefatti visivi com'è possibile notare in Figura 31.

Detto ciò, è possibile riassumere i quattro algoritmi di interpolazione presentati, distinguendone pro e contro.

DUPLICAZIONE

Vantaggi:

- breve tempo impiegato nella computazione;
- lieve utilizzo delle risorse.

Svantaggi:

- discreti risultati visivi in generale.

BLENDING

Vantaggi:

- miglior compromesso tra qualità visiva e tempo di computazione;
- assenza di artefatti grafici.

Svantaggi:

- altri metodi possono visivamente fare di meglio.

MOTION COMPENSATION DI GUNNAR FARNEBÄCK

Vantaggi:

- alta qualità di interpolazione nel caso in cui i fotogrammi non siano omogenei.

Svantaggi:

- oneroso in termini di risorse e lunghe tempistiche di computazione in caso di filmati in alta risoluzione; inoltre, nel caso di immagini statiche, i risultati sono i medesimi del metodo di duplicazione (svantaggio condiviso col metodo di Lucas-Kanade);
- artefatti grafici nel caso di fotogrammi con aree di colore omogenee.

MOTION COMPENSATION DI LUCAS-KANADE

Vantaggi:

- alta qualità di interpolazione nel caso in cui i fotogrammi siano omogenei.

Svantaggi:

- oneroso in termini di risorse e lunghe tempistiche di computazione in caso di filmati in alta risoluzione; inoltre, nel caso di immagini statiche, i risultati sono i medesimi del metodo di duplicazione (svantaggio condiviso col metodo di Lucas-Kanade);
- artefatti grafici nel caso di fotogrammi con pixel eterogenei.

Riferimenti

[1] J. Bouguet. «Pyramidal implementation of the Lucas-Kanade feature tracker». In: *Intel Corporation, Microprocessor Research Labs* (2000). URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.185.585.

[2] G. Farnebäck. «Two-Frame Motion Estimation Based on Polynomial Expansion». In: *Linköping University, Computer Vision Laboratory* (Giugno 2003). URL: https://www.researchgate.net/publication/225138825_Two-Frame_Motion_Estimation_Based_on_Polynomial_Expansion.

[3] FFmpeg. *A complete, cross-platform solution to record, convert and stream audio and video.* URL: https://www.ffmpeg.org/. (ultimo accesso: 29/01/2022).

[4] OpenCV. *Open Source Computer Vision Library.* URL: https://opencv.org/. (ultimo accesso: 30/01/2022).