

openPyVISION

A.Michelassi 2024 draft vo.4

INDICE

Capitolo 1: L'evoluzione e le sfide della tecnologia video

- 1.1** All'inizio usavamo degli Scan-Converter! **1.2** Funzionamento dei mixer video
 - 1.2** Problemi tecnici legati all'EDID e all'HDCP
 - 1.3** Proiettare una presentazione: una sfida inaspettata
 - 1.4** Soluzioni miste e il ruolo dei mixer grafici
 - 1.5** Scusi, ma chi è lei?
 - 1.6** 24 Fotogrammi al secondo
 - 1.7** Il SONORO
 - 1.8** LA TELEVISIONE
 - 1.9** 25, 30, 50, 60
 - e il 59.99?
 - Lo Smartphone
 - La scelta di 60fps: uno standard per il futuro?
-

Capitolo 2: La matematica delle immagini

- 2.1** Al Jabr of video imaging
 - La Matrice
 - Utilità del sistema di Normalizzazione
 - CIE e Standardizzazione
 - 2.3** Il Grafico di I/O
 - 2.4** Operazioni Semplici
 - 2.5** NumPy
 - NumPy Slice
 - NumPy Stack
 - 2.7** Misurare il Tempo
 - 2.8** Complessità
 - 2.9** NumPy LightSpeed
-

Capitolo 3: Mixing 210

- 3.1 Organizziamo i file
 - 3.2 Invert
 - 3.3 AutoScreen
 - 3.4 Split RGB
 - 3.5 Gamma
 - 3.6 Contrast
 - 3.7 More about Contrast
 - 3.8 Contare i frame
-

Capitolo 4: Mix Video

- 4.1 Mix Bus
 - 4.2 Wipe Left To Right
 - 4.3 Wipe Right To Left
 - 4.4 Wipe Up And Down
 - 4.5 Stinger idea
 - 4.6 Stinger More
 - 4.7 Stinger Optimization
 - 4.8 Stinger in the MixBus
 - 4.9 Dip (coming soon)
-

Capitolo 5: More Input

- 5.1 More Input
 - 5.2 Desktop Capture
 - 5.3 Video Capture
 - 5.4 Video Player
 - 5.5 Playlist
 - 5.6 Matrix Input
 - 5.7 Main Out
 - 5.8 Best Monitor Viewer
-

Capitolo 1: L'evoluzione e le sfide della tecnologia video

1.1 All'inizio usavamo degli Scan-Converter!

Un mixer video è un dispositivo hardware o software fondamentale nella produzione audiovisiva, che permette di commutare tra diverse sorgenti video in modo fluido e impercettibile. L'interfaccia utente dei mixer, pur variando in complessità e funzionalità a seconda del modello, condivide una struttura base comune, permettendo ai professionisti di avere una certa confidenza operativa con la sua interfaccia, senza dover imparare un nuovo strumento partendo da zero.



In contesti come conferenze e convegni, spesso vengono utilizzati anche processori di segnale o mixer grafici. Pur non avendo tutte le funzionalità di un mixer video completo, questi dispositivi consentono di combinare sorgenti video provenienti da computer (presentazioni, immagini, video) con segnali video tradizionali (telecamere, lettori multimediali), offrendo una soluzione più semplice e immediata. Tuttavia, l'integrazione tra segnali RGB (provenienti da computer) e segnali YUV (tipici delle telecamere) ha storicamente rappresentato una sfida a causa delle differenze nei modelli di colore e nella gestione delle informazioni cromatiche.

Il formato YUV, ampiamente utilizzato nei sistemi video, può comportare una perdita di dettagli cromatici, soprattutto nelle componenti rosse e blu dell'immagine, a causa della compressione delle informazioni sul colore. Questa compressione, indicata dalla notazione 4:4:4, 4:2:2, ecc., può

portare a discrepanze tra i colori visualizzati sullo schermo e quelli della scena originale. La notazione 4:4:4, 4:2:2, ecc. indica quanto dell'informazione RGB originale viene catturata e inviata. In un sistema 4:4:4, tutte le componenti (Y, U, V) hanno la stessa risoluzione. Il primo '4' rappresenta la Y o luminanza, essenzialmente un'immagine in bianco e nero usata anche per derivare il verde. In sistemi 4:2:2, le componenti cromatiche (U e V, correlate al blu e al rosso) sono tipicamente compresse alla metà della risoluzione della luminanza. Sistemi 4:2:0 o 4:1:1 comportano una compressione ancora maggiore, con conseguente perdita di qualità più evidente.

Questa compressione può portare a situazioni in cui i colori sullo schermo non corrispondono esattamente a quelli della scena originale. Nonostante il bilanciamento del bianco, ovvero la regolazione del bianco in base alle condizioni di illuminazione, può influenzare la resa cromatica in modo diverso nei sistemi RGB e YUV. Ad esempio, per ottenere una tonalità della pelle naturale sotto determinate luci, potrebbe essere necessario impostare il bilanciamento del bianco in modo da compromettere la riproduzione accurata di altri colori.

1.2 Problemi tecnici legati all'EDID e all'HDCP

Un'altra sfida nell'integrazione di segnali video è legata all'EDID (Extended Display Identification Data) quando si utilizzano interfacce come HDMI o DisplayPort. L'EDID è un protocollo che permette ai display di comunicare le loro capacità al dispositivo sorgente, assicurando la compatibilità del segnale video. Tuttavia, problemi nella comunicazione EDID possono causare incompatibilità o perdita di qualità dell'immagine. Fra i problemi più comuni legati all'EDID si può trovare:

- **Incompatibilità dei Formati:** Se un dispositivo di acquisizione non supporta correttamente l'EDID, potrebbe non essere in grado di interpretare o utilizzare i segnali video provenienti da una sorgente HDMI o DisplayPort. Questo può portare a problemi come segnali mancanti, risoluzioni errate o immagini distorte.
- **Risoluzioni e Frequenze di Aggiornamento:** Alcuni dispositivi di acquisizione potrebbero avere difficoltà a gestire risoluzioni o

frequenze di aggiornamento elevate se il monitor o la sorgente video inviano informazioni errate o non standardizzate tramite l'EDID.

- **Blocco del Segnale:** In alcuni casi, l'EDID può bloccare il segnale video se rileva una discrepanza tra le capacità dichiarate del monitor e quelle effettivamente supportate. Questo può impedire l'acquisizione del segnale da parte di dispositivi di acquisizione esterni.

Agli albori del “digitale” si dava molta enfasi all’EDID perché, mentre prima con i monitor analogici (televisori e proiettori) potevi distribuire liberamente i segnali. Il digitale ha cambiato le regole e lo ha fatto con una strategia di marketing poco chiara che non distingueva nettamente tra FullHD e HDReady, chiamando tutto indistintamente HD. Non tutti i problemi, però derivano solo dall’EDID. Il digitale ha portato con sé anche un sistema di protezione dei contenuti, particolarmente attivo con HDMI e DisplayPort: HDCP (High-bandwidth Digital Content Protection).

HDCP interagisce con EDID nel senso che include informazioni sulla compatibilità HDCP del dispositivo. Prima si poteva liberamente prendere un segnale video e distribuirlo a N elementi che potevano essere televisori, proiettori e registratori. Adesso HDCP analizza l’EDID di ciascuna device per decidere se trasmettere o meno il contenuto protetto. Nel caso di una mancata compatibilità si può incorrere nel blocco del segnale. Se un dispositivo sorgente (come un lettore Blu-ray o una console di gioco) rileva che il dispositivo ricevente non supporta HDCP o non è autorizzato, potrebbe effettivamente bloccare il segnale o ridurne la qualità. Questo non è un’azione diretta dell’EDID, ma piuttosto del sistema HDCP che utilizza alcune informazioni fornite dall’EDID.

- **Registratori e dispositivi di cattura:** Molti dispositivi di registrazione o cattura video professionali sono progettati per essere compatibili con HDCP, permettendo la registrazione di contenuti protetti in determinate circostanze (ad esempio, per uso broadcast legittimo). Tuttavia, dispositivi di cattura consumer potrebbero non avere questa autorizzazione, risultando in uno schermo nero.

Il mercato ha risposto a queste sfide preferendo processori di segnale o mixer grafici per gestire segnali provenienti da apparecchi consumer, usando cavi HDMI, DisplayPort e DVI.

1.3 Proiettare una presentazione: una sfida inaspettata

Proiettare una presentazione, un'operazione apparentemente semplice, è diventata sorprendentemente problematica e dispendiosa nel contesto professionale moderno. I relatori spesso utilizzano i propri PC per le presentazioni, con PowerPoint come software predominante.

PowerPoint è stato originariamente sviluppato da Forethought, Inc. e lanciato nel 1987. Successivamente acquisita da Microsoft, è diventato parte integrante della suite Microsoft Office. Negli anni 2000, Microsoft ha collaborato con Apple e Adobe per sviluppare il formato Office Open XML, che include il formato PPTX introdotto con PowerPoint 2007. Questo formato basato su XML è stato progettato per migliorare la compatibilità, la sicurezza e la gestione dei file rispetto ai formati binari precedenti ([This vs. That](#)).

Tuttavia, PowerPoint, nonostante vanti una certa compatibilità cross-platform, porta con sé una serie di problemi persistenti:

- **Salvataggio incompleto delle informazioni:** Non sempre PowerPoint è in grado di salvare un file con tutte le informazioni disponibili. Potrebbero non essere fisicamente disponibili i video, audio o immagini inclusi nella presentazione, oppure questi potrebbero avere codec incompatibili che ne impediscono la riproduzione. Questo significa che una presentazione preparata meticolosamente su un computer potrebbe non funzionare correttamente su un altro dispositivo.
- **Problemi di formattazione:** L'assenza dello stesso font utilizzato per creare la presentazione può far perdere la formattazione delle pagine, rendendo il risultato inguardabile. Immaginate di vedere una presentazione con i testi fuori posto, tagliati o con caratteri completamente diversi: l'impatto visivo e la professionalità ne risentono gravemente.
- **Problemi hardware specifici:** Oltre ai problemi software, ci sono anche problematiche hardware legate ai PC dei relatori. La scheda grafica di alcuni laptop potrebbe non supportare l'uscita

contemporanea con due risoluzioni diverse, e il monitor integrato potrebbe non essere Full HD. Questo può causare difficoltà nel mirroring del display o nella gestione delle risoluzioni tra il laptop e il proiettore.

- **Suoni di sistema e aggiornamenti:** Durante una presentazione, è comune sentire suoni di sistema indesiderati o essere interrotti da notifiche di aggiornamenti software. Non c'è niente di peggio che vedere un relatore interrompere il suo discorso per chiudere una finestra di aggiornamento o per disattivare un allarme del sistema.
- **Riavvio improvviso:** Un altro problema potenzialmente catastrofico è il riavvio improvviso del computer, spesso causato da aggiornamenti automatici o errori di sistema. Questo può interrompere una presentazione nel momento meno opportuno, causando disagi sia al relatore che al pubblico.
- **Incompatibilità generiche:** Infine, ci sono le incompatibilità generiche tra hardware e software diversi. Non tutti i laptop sono costruiti allo stesso modo, e le differenze nei driver, nei sistemi operativi e nelle configurazioni possono causare problemi imprevedibili durante la presentazione.

Quindi, perché non lasciare che ogni relatore si porti il proprio computer? La risposta la porta il vento, come direbbe Bob Dylan! Sebbene democratico, sarebbe solo un moltiplicare al quadrato i problemi sopra menzionati, rendendo la situazione ancora più imprevedibile.

La soluzione è ovviamente usare Linux, segnali SDI fino a 100 metri e fibra oltre 150 metri dimenticando completamente tutti quelli che sono stati i problemi di HDCP e EDID, e usare software che permetta di avere immagini animate e scritte compatibili con l'universo! Sarà questo che ci porterà l'intelligenza artificiale? Al momento, però, non è possibile fare una stima per capire quando arriverà questo futuro semplificato. Per risolvere le problematiche si usano tecniche miste, a volte molto audaci, non prive di costi elevati e complessità tecniche.

Per un approfondimento sulla struttura interna dei file PPTX e la collaborazione tra Microsoft, Apple e Adobe, puoi visitare la pagina [Microsoft Learn sul formato PPTX](#).

1.4 Soluzioni miste e il ruolo dei mixer grafici

I mixer grafici risolvono questi problemi? Sì.

Pro: Hanno hardware potenti che permettono di modificare il segnale preso da una sorgente consumer e restituire un segnale compatibile. Questo non significa che la cosa non possa essere fatta con dei convertitori, solo che era un qualcosa che già veniva messo nel budget di spesa delle aziende quando il video era analogico e quindi è rimasto.

Contro: Sono spesso molto costosi e vengono usati non solo per convertire i segnali, ma anche come mixer video a tutti gli effetti, senza offrire le stesse operatività artistiche.

L'uso di queste macchine ha creato due scuole di pensiero:

- **Televisiva:** La grafica diventa un video YUV, diventando solo un altro input del mixer video.
- **Grafica:** Il segnale del mixer video viene inserito nella grafica. Questa operazione presenta vari problemi tecnici, spesso non risolti correttamente, causando difetti visibili nei video incorporati nelle immagini grafiche.

Nonostante i mixer grafici siano comodi, potenti e versatili, eseguire le operazioni tipiche di un mixer video su un processore di segnale è molto complicato e richiede esperienza specifica. D'altro canto, alcuni mixer video hanno procedure che non sono sviluppate in modo user-friendly, risultando un po' intimidatorie, soprattutto all'inizio.

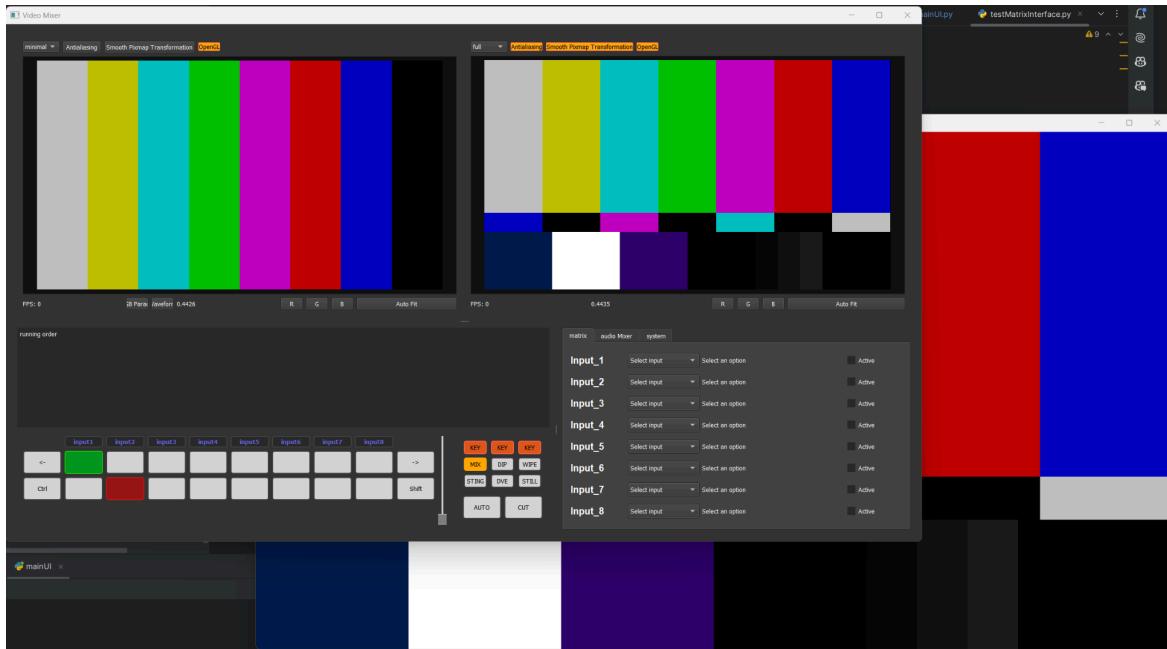
La mia idea è quindi di creare, nei limiti del possibile, un software per il missaggio video che abbia un'interfaccia semplice ma completa per svolgere in modo economico tutte quelle attività che si svolgono in ambito congressuale.

Un software che possa essere utilizzato da chiunque abbia una minima esperienza con i mixer video, ma che abbia anche una vocazione per il compositing live e, più avanti, perché no anche per l'animazione delle telecamere remotate, delle luci e dell'audio. Amo i film muti, ma purtroppo

non sono più molto richiesti, quindi ho cominciato a studiare come sincronizzare l'audio con il video e prima o poi mi deciderò a inserirla.

Non ho trovato un libro solo su questo argomento e non è stato facile mettere insieme queste informazioni. Quello che voglio scrivere è una sorta di introduzione a quello che troverete nel mio codice su GitHub OpenPyVision. Nel panorama della letteratura tecnica e scientifica, raramente si trovano testi che affrontano specificamente la matematica del mixing video. Tutti i testi sono incentrati su immagini statiche. Questo libro si distingue proprio come il celebre "Al-Kitāb al-mukhtaṣar fī ḥisāb al-jabr wa-l-muqābala" di Muhammad ibn Musa al-Khwarizmi. Così come Al-Khwarizmi ha rivoluzionato la matematica introducendo l'algebra e fornendo strumenti per risolvere equazioni complesse, questo testo si propone di fornire una comprensione approfondita e pratica dei problemi legati al video.

L'obiettivo è di discostarsi dagli approcci tradizionali presenti in altri libri sull'argomento, offrendo invece una guida diretta e pratica per comprendere e risolvere i problemi specifici del video mixing. Proprio come "Al-Jabr" ha segnato un punto di svolta nella storia della matematica, questo libro mira a essere un punto di riferimento per chiunque desideri approfondire la matematica applicata al video mixing e alle transizioni video.



1.5 Scusi, ma chi è lei?

La mia passione per il video mixing nasce da un percorso professionale eclettico. Dopo gli studi in regia cinematografica, mi sono specializzato in effetti speciali e animazione, approfondendo software come "Shake" e "Nuke" e seguendo corsi online tenuti da esperti del settore come Steve Wright. Tuttavia, in Italia ho trovato poche opportunità per applicare queste competenze in progetti di grande portata, che spesso richiedevano team numerosi e risorse dedicate.

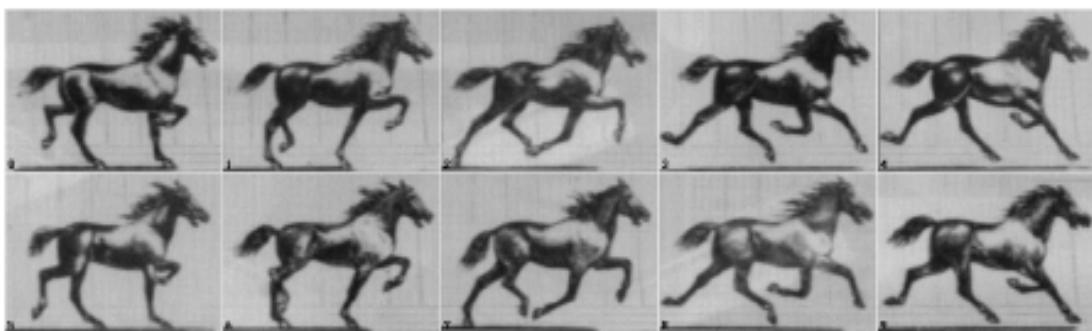
Un momento di svolta è arrivato poco prima della pandemia, quando una produzione televisiva con cui collaboravo ha chiuso i battenti. Ho colto l'occasione per esplorare nuove strade, approfittando del reddito di cittadinanza per approfondire le mie conoscenze nel campo del machine learning e dell'analisi dei dati. Durante questo periodo di studio e sperimentazione, ho iniziato a vedere connessioni tra le mie competenze nel campo degli effetti speciali e le nuove conoscenze acquisite. Ho iniziato a utilizzare librerie di calcolo matriciale, simili a quelle usate nel machine learning, per elaborare immagini e video. Ho anche sperimentato con ChatGPT per automatizzare alcune attività e correggere errori nel codice.

Nel tempo libero, ho sviluppato piccoli progetti personali, tra cui una prima bozza di un programma per il mixaggio video. L'entusiasmo per questo progetto è cresciuto, e ho iniziato a pensare che forse potevo davvero creare qualcosa di utile e innovativo. Ho continuato a lavorare nel settore audiovisivo, cercando di integrare le nuove competenze nei miei progetti. Ho sperimentato con algoritmi di elaborazione delle immagini e ho affinato le mie capacità di programmazione.

L'idea di openPyVISION è nata dalla volontà di unire le mie passioni e competenze per creare un software di video mixing accessibile, intuitivo e potente, che potesse rispondere alle esigenze dei professionisti del settore e semplificare il loro flusso di lavoro.

1.6 24 Fotogrammi al secondo

Perché il cinema utilizza 24 fotogrammi al secondo (fps), mentre i segnali video ne usano 25, 30, 50 o addirittura 60? La risposta si trova in un affascinante viaggio attraverso la storia del cinema e della televisione, un percorso che ci porta dalle lanterne magiche del XVII secolo agli smartphone di oggi.



Il cinema è un'invenzione con una storia ricca e complessa, caratterizzata da numerosi pionieri in tutto il mondo che, forse contemporaneamente, prendevano spunto da idee o macchine viste magari nelle fiere internazionali. Questo periodo ha visto anche molte battaglie legali per stabilire chi fosse il vero inventore del cinema. I fratelli Auguste e Louis Lumière sono spesso considerati i pionieri del cinema, in quanto il 28 dicembre 1895 organizzarono la prima proiezione pubblica di film con il loro

cinematografo. In questa occasione mostraron brevi documentari come "L'uscita dalle officine Lumière" e "L'arrivo di un treno alla stazione di La Ciotat". Il loro approccio era orientato verso il realismo e la documentazione della vita quotidiana.

Parallelamente, negli Stati Uniti, Thomas Alva Edison e il suo assistente William Kennedy Laurie Dickson svilupparono il Kinetoscopio e il Kinetografo negli anni 1890. Il Kinetoscopio permetteva a una sola persona di vedere un breve filmato attraverso un oculare. Anche se le invenzioni di Edison erano innovative, la sua visione del cinema era più individualista rispetto ai Lumière, che avevano un approccio collettivo con le proiezioni pubbliche. Forse, storicamente le due invenzioni che hanno portato a queste idee sono state la lanterna magica e lo zoetrope.

A metà del '600 esisteva la cosiddetta "lanterna magica" che proiettava immagini dipinte su un lastrino di vetro sulle pareti, illuminato da una candela a olio. Una sorta di diapositiva la cui invenzione viene attribuita a Christiaan Huygens, ma si trovano riferimenti a frati e gesuiti che ne parlavano nei loro scritti, quindi qualcosa di simile poteva essere già esistito. Lo zoetrope era un cilindro con una serie di immagini e, se fatto girare a una certa velocità, si percepivano le immagini come animate. Il prassinoscopio combinava questi due principi. Aveva un cilindro centrale con l'immagine, una candela all'interno del cilindro e le immagini venivano proiettate su un cilindro esterno semitrasparente.

Ma perché 24 fps? Durante l'era del cinema muto, i film venivano girati a velocità variabili, spesso tra 16 e 20 fps. Le prime macchine da presa avevano una manovella per avanzare la pellicola e un meccanismo che aiutava a mantenere una velocità costante, ma si poteva comunque aumentare o diminuire la velocità sia in fase di ripresa che di esecuzione. Aumentare la velocità dava un effetto comico alla scena, mentre rallentare aumentava la suspense. Questa scelta derivava probabilmente da considerazioni empiriche



basate su queste macchine manuali vendute come giochi per l'intrattenimento o mostrate nelle fiere e nelle feste di paese, che ricreavano l'illusione del movimento.

Con l'avvento del sonoro, fu necessario standardizzare la velocità di proiezione per garantire la sincronizzazione dell'audio, e 24 fps si rivelò essere una velocità adeguata per ottenere un movimento fluido e mantenere i costi della pellicola gestibili. Con l'avvento del sonoro, fu necessario standardizzare la velocità di proiezione per garantire la sincronizzazione dell'audio. 24 fps si rivelò essere una velocità adeguata per ottenere un movimento fluido e mantenere i costi della pellicola gestibili.

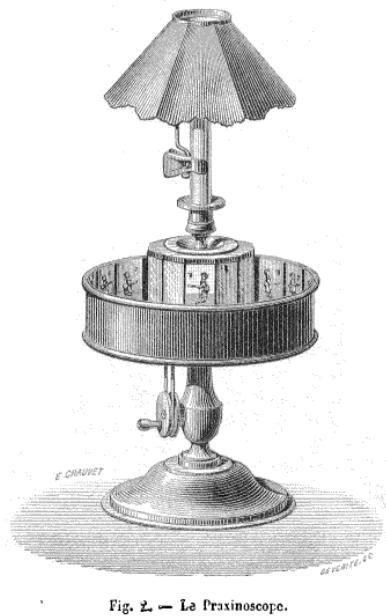
1.7 Il SONORO

Non è chiaro chi ebbe per primo l'idea di trasmettere un disco alla radio e in quale paese divenne popolare per primo, forse in Inghilterra o negli Stati Uniti. Di certo, furono i gestori delle prime stazioni radio a intuire il potenziale di questa nuova invenzione.

Nei primi decenni del cinema, i film erano muti, accompagnati da musica dal vivo eseguita in sala. Questo limitava l'esperienza cinematografica, poiché il pubblico non poteva sentire dialoghi o effetti sonori. Nonostante alcuni sostenevano che i film muti rappresentassero una forma d'arte, i grandi investitori non erano completamente d'accordo.

La trasmissione di musica via radio divenne rapidamente popolare. Le stazioni radiofoniche iniziarono a trasmettere musica regolarmente e a realizzare show dal vivo, come i radio romanzi, raggiungendo un pubblico vasto e diversificato. Questo cambiò il panorama dell'intrattenimento, rendendo la musica accessibile a chiunque possedesse una radio.

Questa popolarità della radio scatenò una competizione tra imprenditori di vari settori dell'intrattenimento, inclusi i produttori cinematografici. Il pubblico si abituò a sentire voci e musica trasmesse in tempo reale, e questa aspettativa si trasferì al cinema, dove iniziò a emergere la domanda di film che combinassero immagini e suoni. La svolta arrivò nel 1927 con il film "Il cantante di jazz" (The Jazz Singer), prodotto dalla Warner Bros. Questo film utilizzava il sistema Vitaphone, che



sincronizzava una colonna sonora registrata su disco con le immagini proiettate. "Il cantante di jazz" presentava segmenti di dialoghi e canzoni sincronizzati, offrendo al pubblico una nuova esperienza cinematografica.

Il successo de "Il cantante di jazz" segnò l'inizio dell'era del cinema sonoro. Le case cinematografiche adottarono rapidamente la nuova tecnologia, trasformando il modo in cui i film venivano prodotti e consumati. I film sonori permisero una narrazione più ricca e coinvolgente, con dialoghi, musica e effetti sonori che arricchivano l'esperienza visiva. Questa transizione non fu priva di sfide: gli studi dovettero affrontare problemi tecnici come l'insonorizzazione dei set e l'uso di microfoni ingombranti, mentre molti attori del muto faticarono ad adattarsi. L'impatto di

queste innovazioni si estese oltre l'intrattenimento: governi e regimi autoritari riconobbero il potenziale strategico delle nuove tecnologie.

1.8 LA TELEVISIONE

Le sperimentazioni tecnologiche non si limitarono solo al campo cinematografico e radiofonico. Stati e governi riconobbero l'importanza strategica di queste innovazioni. In Italia, ad esempio, la radio divenne essenziale per le comunicazioni militari e le prime sperimentazioni della trasmissione televisiva furono interrotte.

In Germania, invece, il regime nazista di Adolf Hitler sfruttò le nuove tecnologie per la propaganda. Durante i Giochi Olimpici del 1936, la Germania sperimentò le prime trasmissioni televisive, utilizzando la tecnologia per promuovere l'immagine del regime e dimostrare la sua supremazia tecnica.

Inizialmente, la televisione era meccanica. La sua invenzione viene spesso accreditata a Paul Nipkow, un ingegnere tedesco, che sviluppò un

disco meccanico con fori realizzati a spirale dall'esterno verso l'interno. Mentre il disco girava, ogni foro passava davanti all'immagine, permettendo alla luce di passare attraverso in sequenza. Dietro il disco, una cellula fotoelettrica catturava la luce che passava attraverso i fori. L'intensità della luce variava a seconda della luminosità di ciascun punto dell'immagine scansionata. La cellula fotoelettrica convertiva queste variazioni di luce in segnali elettrici di intensità variabile. I segnali elettrici prodotti potevano essere trasmessi via cavo o onde radio. Per ricostruire l'immagine, un dispositivo ricevente utilizzava un disco di Nipkow identico, sincronizzato con quello di trasmissione. Il segnale ricevuto modulava una fonte di luce (come una lampada al neon) dietro il disco ricevente. Mentre il disco girava, la luce modulata passava attraverso i fori, ricostruendo l'immagine originale riga per riga su uno schermo.

Tuttavia, la televisione meccanica presentava molte limitazioni. L'immagine trasmessa era piccola, la necessità di sincronizzazione precisa tra trasmettitore e ricevitore era critica e spesso problematica, e la scansione meccanica limitava la velocità di trasmissione, compromettendo la qualità dell'immagine in movimento. Nonostante queste limitazioni, il disco di Nipkow fu fondamentale per lo sviluppo dei primi sistemi televisivi meccanici e aprì la strada alle tecnologie di scansione elettronica che seguirono, come il tubo iconoscopio di Vladimir Zworykin negli anni '20.

Fu solo con l'avvento del tubo catodico che la televisione cominciò a diventare un prodotto commercialmente fruttuoso. Le radio commerciali, come la RCA (Radio Corporation of America), giocarono un ruolo chiave nello sviluppo della televisione elettronica negli Stati Uniti. Mentre in Europa si combatteva, negli Stati Uniti la sperimentazione e la diffusione delle televisioni proseguivano, seppur limitate a pochi pionieri e appassionati di tecnologia. La maggior parte delle famiglie americane non possedeva ancora un televisore. Le trasmissioni televisive erano disponibili solo in alcune città, principalmente dove erano presenti stazioni sperimentali. Nel 1939, durante la Fiera Mondiale di New York, RCA presentò al pubblico la televisione elettronica, segnando un momento cruciale nella storia della televisione.

1.9 25, 30, 50, 60

Nel 1940, le prime trasmissioni televisive erano in bianco e nero e avvenivano completamente in diretta, poiché i videoregistratori non erano ancora stati sviluppati. Le immagini erano spesso riprese da proiezioni in formato 16mm, e in quel periodo iniziava a emergere il primo mercato di contenuti creati specificamente per la televisione. Viene sviluppato nel giro di breve tempo il National Television System Committee uno standard per le trasmissioni 525 linee di risoluzione e una frequenza di 30 fotogrammi al secondo (fps), con una scansione interlacciata a 60 Hz.

Il formato delle immagini sia al cinema che in televisione era il rapporto d'aspetto 4:3 (1.33:1). Questo rapporto d'aspetto fu adottato inizialmente nel cinema perché il sonoro veniva stampato di lato sulla pellicola, riducendo la larghezza disponibile per l'immagine stessa. Le prime televisioni adottarono questo stesso rapporto d'aspetto per diverse ragioni. In primo luogo, era compatibile con i contenuti cinematografici esistenti, facilitando la trasmissione di film e cortometraggi già prodotti. Inoltre, il formato 4:3 risultava pratico ed efficiente nella trasmissione e visualizzazione delle immagini su schermi televisivi dell'epoca.

Queste scelte tecnologiche e di formato hanno gettato le basi per lo sviluppo successivo della televisione come mezzo di comunicazione di massa, influenzando la produzione di contenuti e la progettazione degli apparecchi televisivi per molti anni a venire. Mentre gli Stati Uniti e l'America latina hanno adottato lo standard di 60 Hz per la distribuzione dell'energia elettrica, e questo ha effettivamente influenzato la scelta dei 30 fps (frames per second) per le trasmissioni televisive. Questa relazione è dovuta al desiderio di sincronizzare il refresh dello schermo con la frequenza della rete elettrica per evitare interferenze visibili.

In Europa è più complessa e varia a seconda del paese. Non è del tutto accurato affermare che lo standard di 50 Hz sia stato universalmente adottato solo dopo il Piano Marshall. Molti paesi europei avevano già adottato il 50 Hz come standard prima della Seconda Guerra Mondiale. Per esempio, la Germania aveva standardizzato a 50 Hz già negli anni '20. Il Regno Unito aveva iniziato a standardizzare a 50 Hz negli anni '30, completando il processo negli anni '50. In effetti, la standardizzazione in

Europa è stata un processo graduale che è iniziato prima della guerra, si è interrotto ed è continuato nel dopoguerra. Il Piano Marshall ha certamente contribuito alla ricostruzione e modernizzazione dell'Europa post-bellica ma non ha dettato degli standard unici, probabilmente perché era rivolto anche a recuperare parte delle tecnologie e delle strutture già create prima della guerra.

Nei primi giorni della televisione, si trasmetteva effettivamente a 30 fotogrammi interi al secondo (negli Stati Uniti), un sistema noto come scansione progressiva. Tuttavia, nei primi televisori la luminosità dell'immagine non era stabile e tendeva a calare. Per risolvere questi problemi, gli ingegneri svilupparono il sistema di scansione interlacciata. Invece di trasmettere 30 immagini complete al secondo, il sistema interlacciato divideva ogni fotogramma in due "campi" - uno contenente le righe dispari e l'altro le righe pari dell'immagine. Questi campi venivano trasmessi alternativamente a una frequenza di 60 campi al secondo (negli USA), o 50 campi al secondo in Europa. Da qui la differenza 25 e 30 progressivi, 50 e 60 interlacciati, poiché però l'immagine era tracciata linea dopo linea si diceva di 525 linee di risoluzione.

1.9.1 – Perchè 59.99?

Mentre gli americani nel 1939 andavano al cinema a vedere *Gone with the Wind* a colori in Europa e in Asia si fermava tutto. Se in Italia il 3 gennaio del 1954 alle ore 11:00, comincia la prima trasmissione regolare in bianco e nero, gli stati uniti sono a un passo dal trasmettere regolarmente a colori.

L'introduzione delle trasmissioni a colori negli Stati Uniti fu un capolavoro di ingegneria che portò al cambio di frequenza da 60 Hz a 59.94 Hz. La sfida era mantenere la compatibilità con i televisori in bianco e nero, permettendo loro di ricevere e decodificare il segnale senza problemi. La modulazione del segnale di crominanza (che contiene le informazioni sul colore) doveva essere tale da non interferire con il segnale di luminanza (che contiene le informazioni di luminosità) e con il segnale audio. La frequenza del colore era legata alla frequenza di scansione orizzontale e al frame rate del segnale televisivo. Riducendo leggermente la frequenza di frame da 60 Hz a 59.94 Hz, si poteva ottenere una migliore separazione spettrale tra il segnale di crominanza e il segnale audio, riducendo l'interferenza. Il valore

esatto di 59.94 Hz (più precisamente, 59.94005994005994 Hz) fu scelto per mantenere una precisa relazione con la frequenza del colore e la scansione orizzontale. La frequenza della sottoportante del colore fu fissata a 3.579545 MHz, un valore derivato dalla divisione di 3.579545 MHz per 1000, che è un multiplo della frequenza di campo orizzontale, ottenendo così un'interferenza minima.

1.9.2 – Lo Smartphone

Dal 1955 a oggi ci sono state molte invenzioni storicamente rilevanti, con vari attori che hanno cercato di accaparrarsi la fetta di mercato migliore. Tra queste, l'innovazione forse più rivoluzionaria, paragonabile per impatto al cinema e alla televisione, è lo smartphone. I personal computer, in alcuni paesi, sono riusciti a migliorare significativamente l'alfabetizzazione digitale, il digital divide. Tuttavia, nonostante Windows e internet abbiano aperto una finestra sul mondo, non sono riusciti completamente a rendere l'accesso alla tecnologia universale e immediato.

In questo contesto, gli smartphone hanno avuto un successo maggiore, riducendo le difficoltà intrinseche dei computer e offrendo una soluzione sempre a portata di mano. Hanno semplificato l'accesso alle informazioni e alla comunicazione, diventando strumenti indispensabili nella vita quotidiana e trasformando radicalmente il modo in cui le persone interagiscono con la tecnologia e tra di loro. Stanno introducendo nuove sfide e nuovi modi di guardare le immagini - si possono guardare sia orizzontalmente che verticalmente e permettono l'interazione diretta dell'utente. Lo streaming video e audio on-demand ha trasformato il modo in cui consumiamo i media. Il gaming mobile è diventato un'importante industria di intrattenimento. I nuovi formati di contenuto, come le storie verticali e i video brevi, sono diventati popolari grazie agli smartphone. Movimenti come il citizen journalism e i contenuti generati dagli utenti hanno democratizzato la produzione di notizie e intrattenimento.

Hanno reso possibile il lavoro da remoto, offrendo flessibilità lavorativa. Le app per la gestione del tempo e la produttività aiutano le persone a organizzare meglio le loro vite. Tuttavia, i confini tra vita lavorativa e personale sono diventati più sfumati, e sono emerse nuove professioni legate al mobile, come gli sviluppatori di app e gli influencer. Si

stanno evolvendo per diventare sempre più leggeri, pieghevoli indossabili per integrare sempre più la realtà aumentata e virtuale. Probabilmente l'intelligenza artificiale diventerà sempre più centrale e possiamo immaginarla come il mix fra lanterna magica e prassinoscopio che ha portato a ispirare il cinema in passato.

Ovviamente raccolgono una grande quantità di dati personali, rendendo la privacy e la sicurezza una preoccupazione crescente. La cybersecurity e la protezione dei dati personali sono diventate essenziali. Possono essere utilizzati per la sorveglianza e il tracciamento delle attività, e la diffusione di disinformazione e fake news è facilitata dalla loro ubiquità. Oltre a questo contribuiscono al consumo energetico e all'impronta di carbonio. La produzione e lo smaltimento dei rifiuti elettronici sono problemi significativi nonostante ci sono sforzi per rendere la produzione di smartphone più sostenibili.

La mia non è una materia considerata da studio universitario, ma è di interesse collettivo. Mentre prima era un qualcosa che facevano solo le grandi emittenti e i network, adesso avere una sorta di studio televisivo in casa è alla portata di tutti. Tuttavia, su come sia possibile mettere insieme le immagini o perché si faccia in un modo piuttosto che in un altro, non si trova facilmente in un libro di testo. Questo è il mio tentativo di raccontare come risolvere alcuni dei problemi legati al produrre immagini usando quella che è stata l'altra grande rivoluzione insieme agli smartphone, ma non ha suscitato purtroppo lo stesso clamore. L'Open source!

1.9.3 Licenza MIT

Copyright (c) 2024 Alessio Michelassi Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2: La Al Jabr delle immagini

2.1 Introduzione gentile

La matematica dietro i video non è difficile da comprendere, ma ha una sua complessità e richiede risorse a livello hardware. Per comprendere bene il funzionamento di un mixer video, inizieremo definendo come i computer vedono le immagini e come possono sommarle e moltiplicarle insieme. Questo ci porterà direttamente al nocciolo del problema: la velocità di esecuzione.

2.1.1 La matrice

In informatica, la gestione delle immagini è una disciplina conosciuta come **Computer Generated Imagery (CGI)**. Un'immagine a colori può essere rappresentata come una matrice di pixel, dove ogni pixel è un vettore di tre valori RGB (rosso, verde, blu). Ogni pixel ha un valore che va da 0 a 1; 0 indica l'assenza del colore e 1 la presenza massima del colore.

0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0

In modo più semplice, possiamo dire che un'immagine a colori è composta da tre matrici: una per il verde, una per il rosso e una per il blu. Possiamo visualizzare queste matrici con un esempio ridotto a 8x8 pixel:

Zero rappresenta il nero, mentre 1 rappresenta il bianco. 0.5 è il grigio medio. Con questo sistema, possiamo creare immagini in bianco e nero, tre per la precisione, ognuna delle quali rappresenta la quantità o la mappa di rosso, verde e blu presente nell'immagine.



La prima notizia è che nei computer non esistono immagini a colori ma solo matrici di numeri. Del colore e di come crearlo partendo da questi valori se ne occupano le “device” a cui mandiamo queste informazioni. All’interno dei computer, ogni singola immagine è una matrice e ogni punto che la definisce è detto **pixel**, rappresentato da tre valori che vanno da 0 a 1. Anche se abbiamo usato una matrice di 8x8, un’immagine HD è una matrice di 1920x1080 pixel. Che si consideri un vettore RGB o tre matrici, è solo un modo formale di nominarlo.

La seconda notizia è che i numeri dentro ai computer non esistono realmente. Vengono rappresentati usando un sistema chiamato **discretizzazione**. Questo significa che i valori vengono suddivisi in unità discrete, o passi, che il computer può manipolare. I computer utilizzano un sistema binario, composto solo da 0 e 1, chiamati **bit** (binary digit). Questi bit vengono combinati per memorizzare e manipolare tutti i tipi di dati, inclusi numeri e testo.

Per capire come vengono creati i numeri interi, possiamo immaginare uno spazio di memoria composto da 8 celle (o bit), ognuna delle quali può essere 0 o 1. Questo è noto come un **byte**.

$$\begin{aligned} |0|0|0|0|0|0|0|0| &= 0 \\ |0|0|0|0|0|0|0|1| &= 1 \\ |0|0|0|0|0|0|1|0| &= 2 \\ |0|0|0|0|0|0|1|1| &= 3 \\ \dots \\ |1|1|1|1|1|1|1|0| &= 254 \\ |1|1|1|1|1|1|1|1| &= 255 \end{aligned}$$

Quindi un byte è l’insieme di 8 celle o bit che contengono un valore da 0 a 1 e questo blocco di bit viene chiamato registro. In questo sistema, o

rappresenta il nero e 255 rappresenta il bianco, mentre 127 è il grigio medio. I computer non hanno più registri a 8 bit ormai da più di 20 anni; attualmente i registri sono a 64 bit, ma si trovano anche a 256 e anche a 512 bit. Se in effetti può sembrare un po' limitante usare solo 8 bit per rappresentare il colore, possiamo dire che Photoshop, ad esempio, usa 16 bit per dare all'utente la possibilità di ottenere la massima qualità dell'immagine.

I pionieri che si posero per primi queste domande furono probabilmente i ricercatori della Kodak. Nonostante la pellicola continuasse ad essere considerata una costante, avevano capito che c'era un futuro nel digitalizzare le immagini per creare effetti visivi, restaurare pellicole e per fare color correction utilizzando i computer. Mentre 8 bit erano pochi e 16 bit erano decisamente troppi, trovarono sistemi alternativi come ad esempio l'uso dei 10 bit.

Tuttavia, la tendenza negli anni è sempre stata quella di ridurre le dimensioni dei file, questo perché gli hard disk vengono venduti a euro/byte. Ecco un esempio della differenza in termini di dimensioni dei file:

- Un'immagine 1920x1080 a 8 bit occupa circa 6.2 MB.
- Un'immagine 1920x1080 a 10 bit occupa circa 7.75 MB.
- Un'immagine 1920x1080 a 16 bit occupa circa 12.44 MB.

Nel caso di un video a 60 fotogrammi al secondo, dobbiamo moltiplicare tutto per 60, il che ci dà:

- $6.2 \text{ MB} * 60 = 372 \text{ MB}$ al secondo per un video a 8 bit.
- $7.75 \text{ MB} * 60 = 465 \text{ MB}$ al secondo per un video a 10 bit.
- $12.44 \text{ MB} * 60 = 746.4 \text{ MB}$ al secondo per un video a 16 bit.

Un documento interessante è quello pubblicato nel 1995 da Kodak per la conversione 10 bit a 8 bit, dove spiegava ai tecnici come convertire le immagini da 10 bit dal formato cinematografico per il video a 8 bit. [Kodak Cineon Document](#)

Cineon è stato un sistema di digitalizzazione sviluppato dalla Kodak negli anni '90 per la post-produzione cinematografica. Includeva sia l'hardware che il software necessari per la scansione di pellicole

cinematografiche e la loro conversione in formato digitale. Fu introdotto in commercio ad uso delle case di produzione negli anni '90 e era un sistema progettato per mantenere una qualità elevata durante il processo di digitalizzazione, permettendo di effettuare la post-produzione in formato digitale e poi ritrasferire il filmato su pellicola per la distribuzione. Questo processo è noto come "Digital Intermediate".

2.1.3 Utilità del Sistema di Normalizzazione

Quindi, OK. 255 è bianco a 8 bit, se è a 16 bit diventa 65535, ma come detto ci sono spazi colore a 10 bit (1024) e a volte a 12 bit (4096). In questo modo, se il bianco e il grigio medio hanno valori sempre diversi, diventa estremamente facile confondersi e commettere errori. Per risolvere questo problema, si usa il sistema di normalizzazione.

La normalizzazione dei dati consente di dire che, indipendentemente dalla risoluzione del colore o dalla profondità di bit del sistema, 0 è il nero e 1 è il bianco. Quindi, se ho un'immagine a 10 bit, non devo confondermi nel fare calcoli perché il bianco è 1024; rimane semplicemente 1. Questo approccio facilita l'implementazione di algoritmi di elaborazione delle immagini in modo coerente e scalabile.

In Python, possiamo utilizzare la libreria **NumPy** (che verrà approfondita nel Capitolo 3) per lavorare con matrici e creare immagini. NumPy è una potente libreria che consente di eseguire operazioni vettoriali e matriciali in modo efficiente, rendendola ideale per la manipolazione delle immagini. Quando creo una matrice, devo dichiarare in anticipo quale sarà la sua forma e la sua profondità di bit.

Lo spazio colore più semplice da capire è l'sRGB, che ha una profondità di 8 bit. Per creare un'immagine semplice, possiamo utilizzare NumPy per generare una matrice di valori. Ecco un esempio di codice che crea un'immagine casuale:

```
import numpy as np
import matplotlib.pyplot as plt

height = 8
width = 8
```

```
# Creiamo una matrice con 3 canali r,g,b a 8 bit
image = np.random.randint(0, 256, (height, width, 3), dtype=np.uint8)

# Visualizza l'immagine
plt.figure(figsize=(10, 6)) # Imposta le dimensioni della figura
plt.imshow(image)
plt.axis('off') # Nasconde gli assi
plt.title('Immagine a colori casuale')
plt.show()
```

Numpy ci permette di creare immagini, matplotlib le visualizza in modo super easy e in questo caso creiamo una matrice con dei valori casuali dentro. L'abbiamo visualizzata, ma possiamo anche stamparla semplicemente scrivendo `print(image)`.



Affronteremo più avanti nel testo Numpy, mentre faremo finta di sapere già cosa fanno i 4 comandi matplotlib che stiamo usando anche perchè noi li useremo fondamentalmente per visualizzare in modo rapido semplici immagini in bianco e nero e a colori.

Il bello di numpy è che ci permette di vedere cosa c'è dentro un'immagine semplicemente usando il comando `print`.

```
image = np.random.randint(0, 256, (height, width, 3), dtype=np.uint8)
print(image)
```

```
[[[ 58 253 158]
 [ 14 184 121]
 [235 165 140]
 [134 134 110]
 [148 118 192]
 [ 48 255  64]
 [ 57 238 180]
 [186 113 183]] ...
```

Nella stampa possiamo vedere i dati delle matrici rgb che Numpy raggruppa in modo un po particolare. E' la rappresentazione che usa Numpy per accellerare i calcoli?

Utilizzando questo script però, possiamo divertirci a rappresentare visivamente i dati presenti nelle tre matrici nel modo che ci aspetteremmo di vedere.

```
import numpy as np

import matplotlib.pyplot as plt
import pandas as pd

height = 8
width = 8

# Creiamo una matrice con 3 canali r,g,b a 8 bit
image = np.random.randint(0, 256, (height, width, 3), dtype=np.uint8)

# Estraiamo i canali rosso, verde e blu
green_channel = image[:, :, 1]
red_channel = image[:, :, 0]
blue_channel = image[:, :, 2]

# Creiamo dataframes pandas per visualizzare le matrici
df_green = pd.DataFrame(green_channel)
df_red = pd.DataFrame(red_channel)
df_blue = pd.DataFrame(blue_channel)
# Stampa delle matrici
print("GREEN")
print(df_green)
print("\nRED")
print(df_red)
```

```

print("\nBLUE")
print(df_blue)

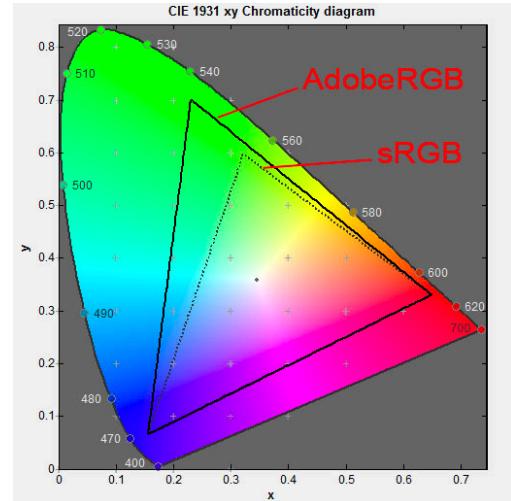
# Visualizziamo l'immagine con matplotlib
plt.figure(figsize=(10, 6))
plt.imshow(image)
plt.axis('off')
plt.title('Immagine a colori casuale')
plt.show()

```

GREEN								RED								BLUE										
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7			
0	173	51	183	21	102	160	184	165	0	205	8	185	239	156	185	44	205	0	254	107	112	66	236	195	115	165
1	179	99	165	86	65	191	96	174	1	204	254	201	220	106	242	35	125	1	235	151	63	83	222	69	113	45
2	222	81	227	65	81	201	158	252	2	93	200	238	103	85	214	250	21	2	76	67	90	114	180	21	159	33
3	184	127	102	42	50	174	247	122	3	126	54	213	158	228	160	71	49	3	161	174	135	188	209	106	251	202
4	166	196	168	113	8	250	148	186	4	241	27	61	51	125	117	245	75	4	108	37	59	108	171	116	22	108
5	230	234	21	46	211	37	52	174	5	157	170	218	115	141	75	245	47	5	40	67	57	85	145	116	132	177
6	182	223	72	237	175	237	211	253	6	120	157	40	120	5	87	203	192	6	95	85	109	118	74	53	255	118
7	157	59	189	163	142	236	39	101	7	197	245	200	145	235	203	161	243	7	113	216	69	222	31	163	134	123

2.2 - CIE e Standardizzazione

L'sRGB è uno spazio colore a 8 bit ed è considerato uno standard ancora oggi molto diffuso e utilizzato. Si basa su uno studio chiamato CIE 1931. La CIE, o **Commission Internationale de l'Éclairage** (Commissione Internazionale per l'Illuminazione), è un'organizzazione internazionale che si occupa di standardizzazione nel campo della luce, dell'illuminazione, del colore e degli spazi colore. Fondata nel 1913, la CIE è la principale autorità internazionale sulla luce, l'illuminazione, il colore e gli spazi colore. Il suo scopo principale è standardizzare e fornire procedure di misurazione nel campo dell'illuminazione e del colore.



Gli spazi a colori del CIE sono stati creati utilizzando i dati di una serie di esperimenti, in cui i soggetti dei test umani hanno regolato i colori primari rosso, verde e blu per trovare una corrispondenza visiva a un secondo colore puro. Gli esperimenti originali furono condotti a metà degli

anni '20 e sono andati avanti nel tempo nel tentativo di semplificare la matematica dietro le quinte.

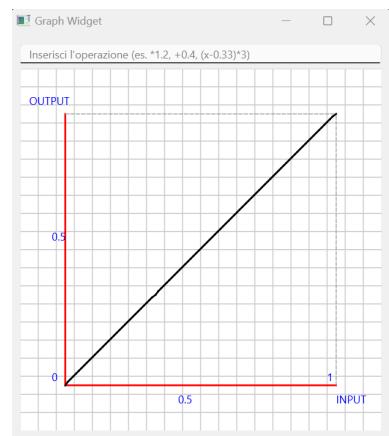
Il grafico sostanzialmente ci fa vedere l'intero spettro visivo con due triangoli che mostrano lo spazio sRGB e quello a 16 bit proposto da Adobe con Photoshop. Lo spazio Adobe RGB è più ampio e più simile, ma non identico, a quello che un essere umano riesce a percepire. Tuttavia, vari fisici studiando la materia si sono accorti di come la percezione visiva sia ingannata da una serie di fattori conosciuti anche come illusioni ottiche. Per questo motivo, il sistema sRGB continua a essere molto utilizzato.

Potete approfondire le illusioni ottiche o visive in questa sezione di Wikipedia: [Optical Illusion](#)

sRGB fra l'altro ci darà una mano a semplificare i calcoli nei nostri esempi e implementazioni. Questo è particolarmente rilevante per lo streaming su internet, dove i codec utilizzati per la trasmissione operano generalmente a 8 bit.

2.3 - Il Grafico di I/O

Quando studiavo effetti speciali, c'erano due libri considerati fondamentali perlomeno da chi mi insegnava questa materia e che spesso li descrivevano come la Bibbia e il Vangelo del settore. Il primo è "The Art and Science of Digital Compositing" di Ron Brinkman, autore del software "Shake". Il secondo è "Digital Compositing for Film and Video" di Steve Wright, un veterano dell'industria degli effetti speciali.



Entrambi mostrano che è possibile creare un grafico chiamato I/O (Input/Output) in cui sugli assi X e Y si rappresentano i valori dell'immagine prima e dopo la modifica. Supponiamo che 0 rappresenti l'immagine non modificata e 1 l'immagine completamente modificata. Avremo quindi una retta che va da (0,0) a (1,1) nel caso in cui l'immagine non è stata modificata in nessun modo.

Possiamo costruire uno script in python per mostrare cosa avviene a un'immagine ogni volta che modifichiamo un'immagine con una certa operazione.

```
import sys
import numpy as np
from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QLineEdit
from PyQt6.QtGui import QPainter, QPen, QColor
from PyQt6.QtCore import Qt

class GraphWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Graph Widget')
        self.setGeometry(100, 100, 400, 450)

        self.layout = QVBoxLayout()
        self.setLayout(self.layout)

        self.input_box = QLineEdit(self)
        self.input_box.setPlaceholderText("Inserisci l'operazione (es. *1.2, +0.4, (x-0.33)*3)")
        self.input_box.textChanged.connect(self.update_graph)
        self.layout.addWidget(self.input_box)

        self.graph_widget = GraphDrawingWidget()
        self.layout.addWidget(self.graph_widget)

    def update_graph(self, text):
        self.graph_widget.update_curve(text)

class GraphDrawingWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.setFixedSize(400, 400)

        # Define colors
        self.gridColor = QColor(200, 200, 200)
        self.axisColor = QColor(255, 0, 0)
        self.lineColor = QColor(0, 0, 0)
        self.dotLineColor = QColor(155, 155, 155)
        self.textColor = QColor(0, 0, 255)
        self.expression = 'x'
        self.curve = np.linspace(0, 1, 100)
        self.update_curve(self.expression)

    def update_curve(self, expression):
        self.expression = expression
        x = np.linspace(0, 1, 100)
        try:
            y = eval(self.expression)
            self.curve = np.clip(y, 0, 1)
        except Exception as e:
            self.curve = x # If there's an error, revert to the identity curve
        self.update()

    def paintEvent(self, event):
        painter = QPainter(self)
```

```

painter.setRenderHint(QPainter.RenderHint.Antialiasing)

# Background
painter.fillRect(self.rect(), Qt.GlobalColor.white)

# Draw the grid
painter.setPen(QPen(self.gridColor, 1, Qt.PenStyle.SolidLine))
for x in range(0, self.width(), 20):
    painter.drawLine(x, 0, x, self.height())
for y in range(0, self.height(), 20):
    painter.drawLine(0, y, self.width(), y)

# Draw the axes
painter.setPen(QPen(self.axisColor, 2, Qt.PenStyle.SolidLine))
painter.drawLine(50, self.height() - 50, self.width() - 50, self.height() - 50) # X axis
painter.drawLine(50, self.height() - 50, 50, 50) # Y axis

# Draw labels and ticks
painter.setPen(QPen(self.textColor, 2))
painter.setFont(painter.font())
painter.drawText(self.width() - 50, self.height() - 30, 'INPUT')
painter.drawText(10, 40, 'OUTPUT')
painter.drawText(35, self.height() - 55, '0')
painter.drawText(self.width() - 60, self.height() - 55, '1')
painter.drawText(35, (self.height() - 50) // 2 + 15, '0.5')
painter.drawText((self.width() - 50) // 2, self.height() - 30, '0.5')

# Draw the curve
painter.setPen(QPen(self.lineColor, 2, Qt.PenStyle.SolidLine))
for i in range(1, len(self.curve)):
    start_x = 50 + (self.width() - 100) * (i - 1) / (len(self.curve) - 1)
    end_x = 50 + (self.width() - 100) * i / (len(self.curve) - 1)
    start_y = self.height() - 50 - (self.height() - 100) * self.curve[i - 1]
    end_y = self.height() - 50 - (self.height() - 100) * self.curve[i]
    painter.drawLine(int(start_x), int(start_y), int(end_x), int(end_y))

# Draw dashed lines
pen = QPen(self.dotLineColor, 1, Qt.PenStyle.DashLine)
painter.setPen(pen)
painter.drawLine(50, 50, self.width() - 50, 50) # Line from (0,1) to (1,1)
painter.drawLine(self.width() - 50, self.height() - 50, self.width() - 50, 50) # Line from (1,0) to (1,1)

painter.end()

def main():
    app = QApplication(sys.argv)
    widget = GraphWidget()
    widget.show()
    sys.exit(app.exec())

if __name__ == '__main__':
    main()

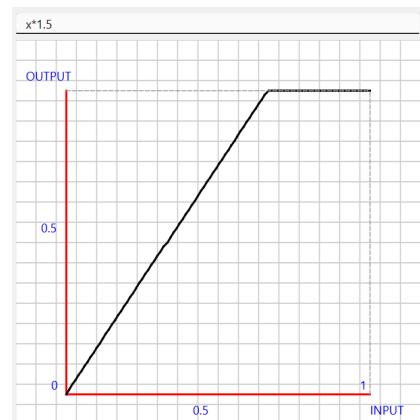
```

2.4 - Operazioni Semplici

Moltiplicando tutti i pixel per un valore, ad esempio 1.5, si aumenta la luminosità dell'immagine. Questa operazione è chiamata "Gain". In termini matematici:

$$y = x \times \text{gain}$$

Dove x è il valore iniziale del pixel e **gain** è il fattore di moltiplicazione. Quando osserviamo il grafico, vediamo che la retta aumenta la sua pendenza, riflettendo l'aumento di luminosità. Tuttavia, tutti i valori dei pixel sono limitati a un massimo di 1, quindi se un'operazione produce un valore superiore a 1, questo viene ridotto a 1, fenomeno chiamato "clamping". Lo stesso avviene per i valori che scendono sotto lo zero. Questo effetto è più evidente quando la risoluzione del colore (o "latitudine di posa" nei tempi della pellicola) è limitata. Tuttavia, il dettaglio nelle parti più scure dell'immagine viene mantenuto poiché i valori vicini allo zero rimangono simili.



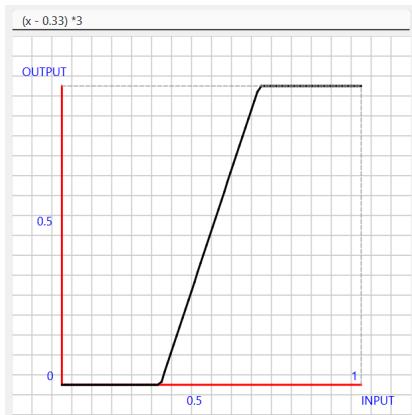
L'operazione di "Brightness", analogamente al "Gain", avviene sommando un valore costante a tutti i pixel dell'immagine:

$$y = x + \text{brightness}$$

In questo caso, la retta nel grafico si sposta verso l'alto, mostrando chiaramente un clamping sia verso l'alto che verso il basso.

Una delle operazioni più utilizzate per migliorare un'immagine è il cambiamento del contrasto, che rappresenta la differenza fra le parti più chiare e scure dell'immagine. Aumentando il contrasto, le aree più scure diventano più scure e le aree più luminose diventano più luminose.

Al contrario, diminuirlo riduce la luminosità delle aree luminose e rende più chiare le aree scure. Non esiste uno standard universale su come



debba essere implementato un operatore di contrasto. Un metodo, come mostra Ron Brinkman nel suo "The Art & Science of Digital Compositing", è sottrarre un valore costante e quindi moltiplicare il risultato per un'altra costante:

$$y = (x - 0.33) \times 3$$

Anche in questo caso c'è un clamping, ma se l'immagine è migliorata e non ha perdite significative, questo è un sistema veloce che

permette di lavorare sull'immagine con operazioni di somma e moltiplicazione.

L'ideale è ammorbidire gli spigoli in modo che il grafico assuma una forma a "S". Questo tipo di contrasto è detto polarizzato o "biased". Oltre a mantenere tutti i dati all'interno del dominio, migliora sensibilmente il risultato finale:

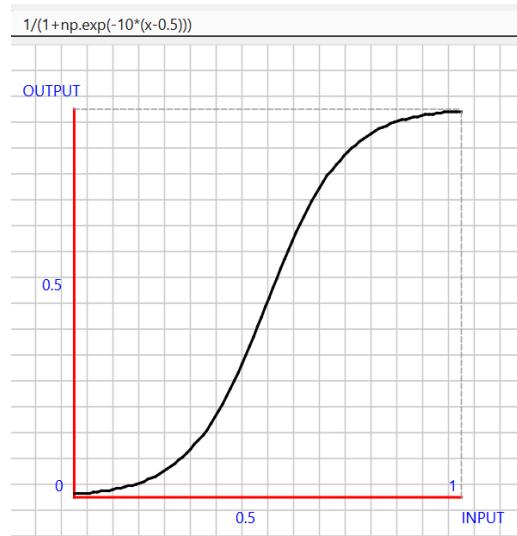
$$y = \frac{1}{1 + \exp(-10 \times (x - 0.5))}$$

Non vi preoccupate se ha un aspetto difficile, vedremo che ci sono algoritmi per calcolare questa cosa molto velocemente e semplicemente.

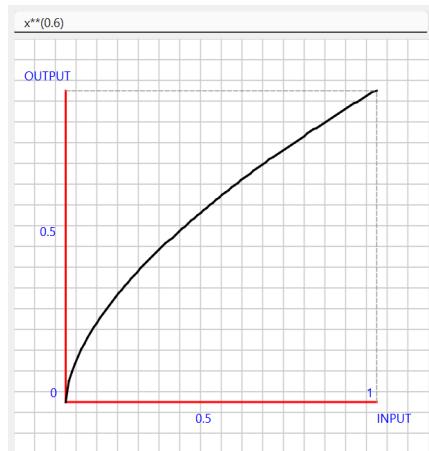
La correzione gamma è un'altra operazione fondamentale nell'elaborazione delle immagini:

$$y = X^{\frac{1}{\gamma}}$$

dove **y** è l'output, **x** è l'input, e **gamma** è il fattore di correzione. Con una calcolatrice, è facile verificare come a differenti valori di gamma si associano differenti valori di luminosità per ogni pixel. Per esempio, un



pixel che ha un valore iniziale di 0.5 finirà con un valore di circa 0.665 se utilizziamo un gamma di 1.7. La vera ragione della popolarità dell'operatore di gamma diventa evidente quando si esamina cosa succede ai valori estremi, cioè 0 e 1, dell'immagine. In altre parole, la "Gamma Correction" ha effetto solo sui pixel non bianchi o non neri, aumentando la luminosità senza mandare in clamp i pixel.

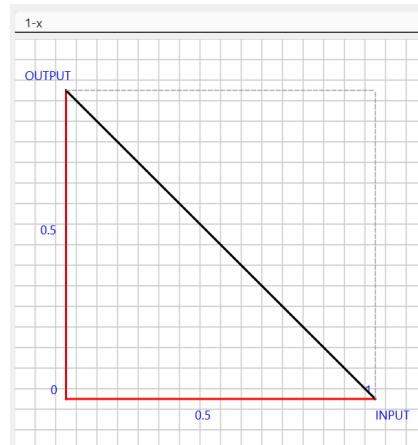


Un'altra operazione utile è l'inversione dei colori. Moltiplicando per -1 un'immagine e sottraendo 1, si ottiene il negativo:

$$y=1-x$$

In pellicola, alcune operazioni venivano fatte usando il negativo per poi trasformarlo in positivo. Alcune espressioni matematiche non sono altro che la trasposizione di ciò che veniva fatto in laboratorio.

Per capire come applicare queste operazioni alle immagini, dobbiamo approfondire come le immagini vengono gestite, e possiamo farlo molto semplicemente usando NumPy.



2.5 - NUMPY

[NumPy](#) è un pacchetto fondamentale per il calcolo scientifico in Python. Si installa utilizzando il comando `pip install numpy` ed è una libreria esterna che permette di lavorare con array multidimensionali come vettori e matrici. NumPy offre un ampio assortimento di routine per operazioni veloci su array, tra cui manipolazione matematica, logica, di forma, ordinamento, selezione, I/O, trasformazioni discrete di Fourier, algebra lineare di base, operazioni statistiche di base, simulazioni casuali e molto altro.

E' stato originariamente creato da Travis Oliphant nel 2005, come estensione della libreria *Numeric*, per fornire un framework più potente e flessibile per il calcolo numerico. Da allora, è diventato uno degli strumenti più utilizzati nella comunità scientifica e di data science.

Prima di mostrare quanto è veloce NumPy, vediamo cosa offre e come può essere usato per creare immagini.

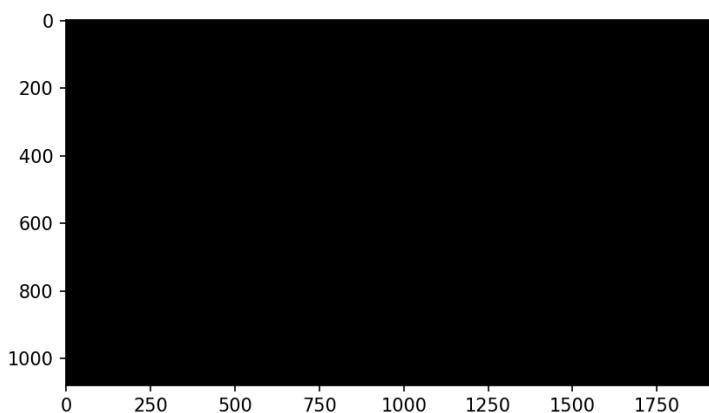
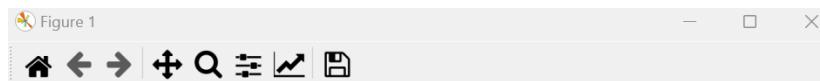
```
import numpy as np

blackFrame = np.zeros((1080, 1920, 3), dtype=np.uint8)
```

Il comando **np.zeros** crea una matrice di zeri delle dimensioni specificate. In questo caso, creiamo una matrice 1080x1920 con 3 canali (rosso, verde e blu) e assegnamo il tipo di dato **uint8**, che rappresenta un intero senza segno a 8 bit (valori positivi tra 0 e 255)

```
# usa matplotlib per mostrare l'immagine
import matplotlib.pyplot as plt

plt.imshow(blackFrame)
plt.show()
```



Il comando **np.ones** crea una matrice di 1 delle dimensioni specificate. Moltiplichiamo per 255 per ottenere un frame completamente bianco.

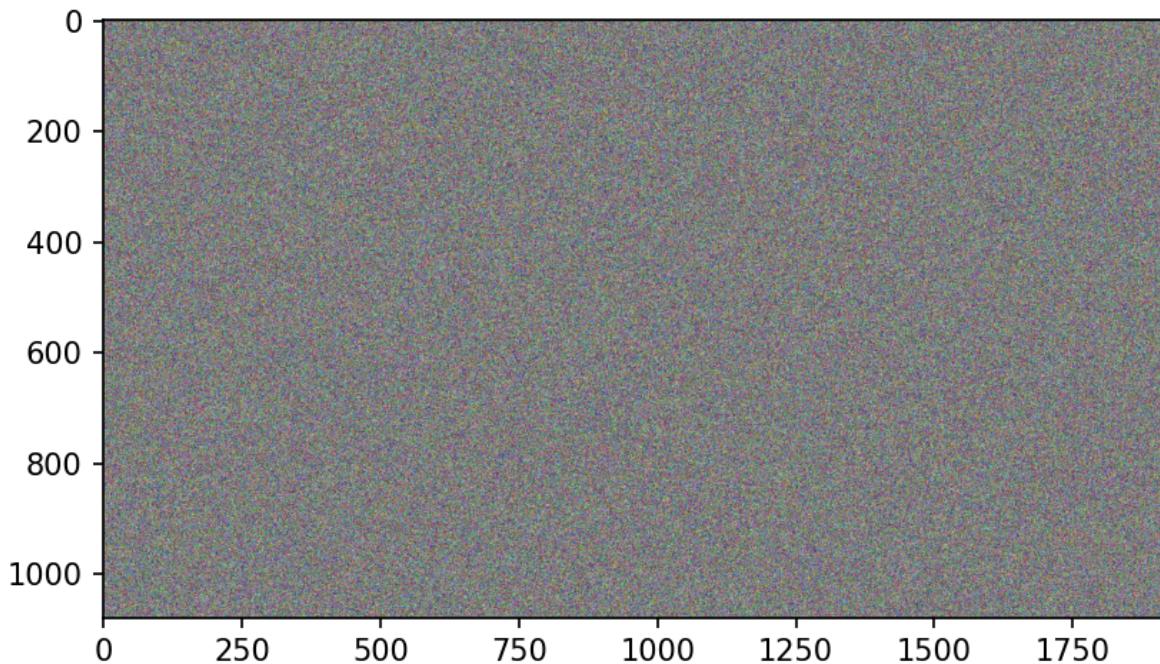
```
# Creazione di un frame nero con dimensioni 1080x1920 e 3 canali (RGB)
oneFrame = np.ones((1080, 1920, 3), dtype=np.uint8) * 255
```

Un'altra opzione per ottenere lo stesso risultato è **np.full**, che crea una matrice delle dimensioni specificate riempita con il valore 255.

```
# Creazione di un frame bianco
whiteFrame = np.full((1080, 1920, 3), 255, dtype=np.uint8)
```

Il comando **np.random.randint** genera numeri interi casuali nell'intervallo specificato (0-255) per creare un frame con rumore casuale.

```
# Creazione di un frame con rumore casuale
randomFrame = np.random.randint(0, 256, (1080, 1920, 3), dtype=np.uint8)
```



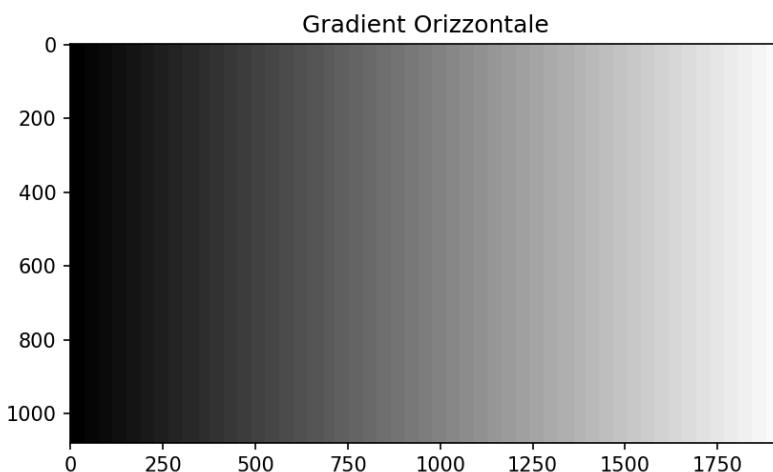
np.linspace permette di creare una lista di numeri ordinati in modo uniforme. Ad esempio, per creare un gradiente che va da zero a 255:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Gradienti orizzontali
gradient_h = np.linspace(0, 255, 1920, dtype=np.uint8)
gradient_h = np.tile(gradient_h, (1080, 1))

plt.imshow(gradient_h, cmap='gray')
plt.title('Gradient Orizzontale')
plt.show()
```

Si crea quindi prima una serie di valori lineari fra 0 e 255 e poi si impila la serie usando tile, che crea 1080 ripetizioni.



La stessa cosa si può fare in verticale.

```
gradient_v = np.linspace(0, 255, 1080, dtype=np.uint8)
gradient_v = np.tile(gradient_v, (1920, 1)).T
```

In questo caso si crea l'immagine verticale, poi una volta impilati i valori si fa il transpose della matrice in modo da farla diventare da 1080x1920 a 1920x1080.

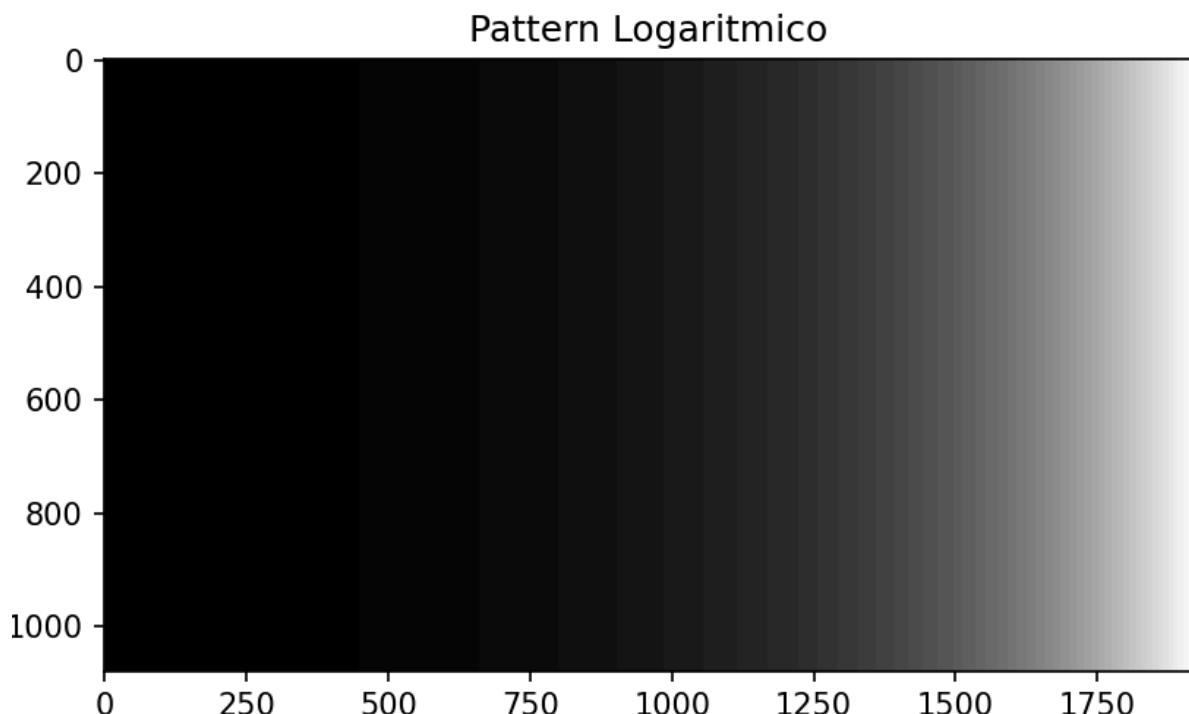
np.logspace fa la stessa cosa ma creando una serie di valori logaritmici:

```
import numpy as np
import matplotlib.pyplot as plt

# Creare una mappa di intensità logaritmica
```

```
log_space = np.logspace(0, 2, 1920, dtype=np.float32)
log_space = np.tile(log_space, (1080, 1))

plt.imshow(log_space, cmap='gray', norm=plt.Normalize(vmin=log_space.min(),
vmax=log_space.max()))
plt.title('Pattern Logaritmico')
plt.show()
```



2.5.2 - NUMPY SLICE

NumPy ha un sistema per manipolare gli array simile a quello che usiamo con Python, ad esempio nelle liste, chiamato slicing.

array[start:stop:step]

Gli slice permettono di accedere e modificare sottoinsiemi di array senza creare copie inutili dei dati, rendendo il codice più veloce e leggibile.

Nel caso in cui si omette **start**, l'array parte dal primo elemento; **stop** indica l'ultimo elemento, mentre **step** indica il passo fra ogni elemento (di default è 1). **[: : 2]** significa dal primo all'ultimo ogni due.

Per esempio:

```
# Creazione della matrice 8x8 bianca
whiteMat = np.ones((8, 8), dtype=np.uint8) * 255

# Creazione della matrice 8x8 nera
blackMat = np.zeros((8, 8), dtype=np.uint8)

# Creazione di un'immagine vuota 8x8 (per contenere metà dell'una e
metà dell'altra)
combinedMat = np.zeros((8, 8), dtype=np.uint8)

# Combinazione delle due matrici
combinedMat[:, :4] = whiteMat[:, :4]
combinedMat[:, 4:] = blackMat[:, 4:]

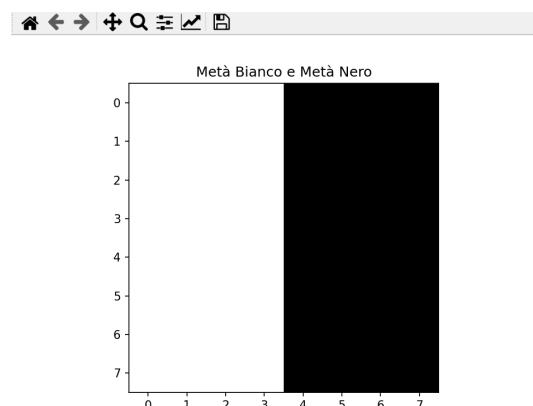
# Visualizzazione dell'immagine combinata
plt.imshow(combinedMat, cmap='gray')
plt.title('Metà Bianco e Metà Nero')

plt.show()
```

L'operazione **[:, :4]** seleziona tutte le righe e le prime 4 colonne dell'array.

Parlando di slicing, può far comodo ricordarci che il primo elemento è lo 0 e il secondo è l'1. A volte questo può confondere, soprattutto quando si lavora con matrici. Per esempio, **array[0, 0]** accede all'elemento nella prima riga e prima colonna, mentre **array[1, 2]** accede all'elemento nella seconda riga e terza colonna.

Un altro modo di usare lo slicing è inserire una virgola al posto dello stop. In questo caso si specifica uno slice a due dimensioni **[rigaStart:rigaStop, colonnaStart:colonnaStop]**. È un po' anti-intuitivo perché in questo caso



si indicano il numero di righe e colonne al posto della posizione. Per esempio, `array[:, 4:]` seleziona tutte le righe e le colonne dalla quarta in poi.

Questi due metodi possono sembrare simili nella scrittura, ma è importante non confonderli: uno si usa per accedere a elementi specifici, mentre l'altro si utilizza per selezionare intervalli di righe e colonne.

Più avanti vedremo come questa cosa ci permetta di fare la dissolvenza wipe fra due immagini per ora limitiamoci a creare un generatore di barre E.B.U. ([European Broadcast Union](#)).

```
from matplotlib import pyplot as plt
import numpy as np

colors = [
    (192, 192, 192), # Grigio
    (192, 192, 0), # Giallo
    (0, 192, 192), # Ciano
    (0, 192, 0), # Verde
    (192, 0, 192), # Magenta
    (192, 0, 0), # Rosso
    (0, 0, 192), # Blu
    (0, 0, 0) # Nero
]

width, height = 1920, 1080
# Crea una matrice vuota
bar_height = height
bar_width = width // len(colors)
bars = np.zeros((bar_height, width, 3), dtype=np.uint8)

# Riempie ogni sezione con il colore corrispondente
for i, color in enumerate(colors):
    bars[:, i * bar_width:(i + 1) * bar_width, :] = color

plt.imshow(bars)
plt.title('Barre colorate')

plt.show()
```



np.arange è una funzione di NumPy che genera un array contenente valori equidistanti all'interno di un intervallo specificato. È simile alla funzione **range** di Python, ma restituisce un array NumPy invece di un oggetto range

```
numpy.arange([start, ]stop, [step, ]dtype=None, *, device=None, like=None)
```

Dove *start* (opzionale) è l'inizio dell'intervallo e ha come valore predefinito 0 mentre *stop* è la fine dell'intervallo - *L'intervallo non include questo valore* - *step* (opzionale) è invece la spaziatura tra i valori (1 è il valore di default) e *dtype* (opzionale) è Il tipo di dati dell'array risultante. Se non specificato, il tipo di dati è inferito dagli altri argomenti.

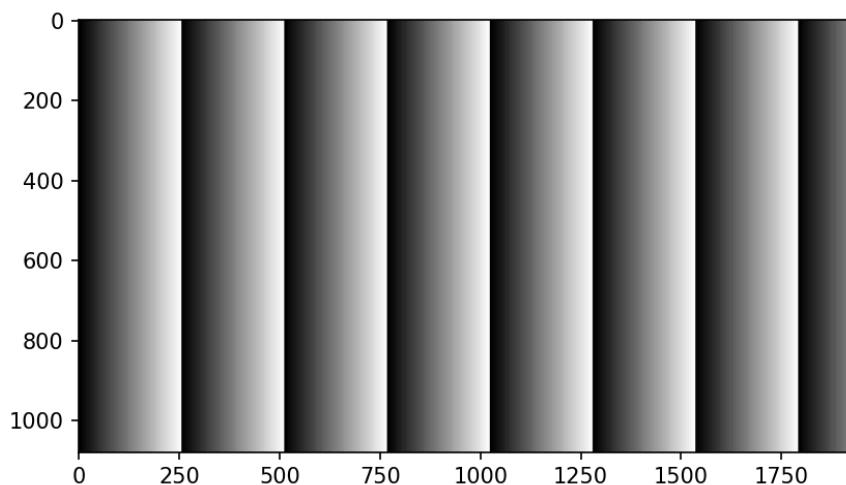
Device (opzionale) è il dispositivo su cui posizionare l'array creato. Predefinito: None, mentre *like* (opzionale) è Oggetto di riferimento per consentire la creazione di array che non sono array NumPy.

```
import numpy as np
import matplotlib.pyplot as plt
arr = np.arange(0, 1920, dtype=np.uint8)
arr = np.tile(arr, (1080, 1))
print(arr)
```

```
print(arr.shape)
print(arr.dtype)
print(arr.min(), arr.max())

plt.imshow(arr, cmap='gray')
plt.show()
```

In questo esempio, ci si aspetterebbe di vedere un gradiente. Tuttavia, poiché `uint8` va da 0 a 255, i valori si "riavvolgono" a 0, causando ripetizioni invece di un gradiente continuo. Questo può comunque essere utile per creare pattern regolari.



Ecco un esempio di creazione di una griglia 8x8 con barre verticali utilizzando `np.arange`:

```
import numpy as np
import matplotlib.pyplot as plt

# Creazione di una griglia 8x8 con barre verticali
bar_height = 8
bar_width = 8

# Crea un array di indici di colonne
columns = np.arange(bar_width)

# Alterna tra colonne bianche (255) e nere (0)
bars = (columns % 2) * 255
```

```
# Replica le barre verticali su tutte le righe
bars = np.tile(bars, (bar_height, 1))

plt.imshow(bars, cmap='gray')
plt.title('Barre verticali')
plt.show()
```

np.mgrid è uno strumento molto utile per creare mesh grid multidimensionali, che sono essenzialmente coordinate in uno spazio multidimensionale. Questo può essere particolarmente utile per la grafica, la visualizzazione e la creazione di pattern regolari o griglie di punti. Ecco un esempio per creare un pattern circolare:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(1920)
y = np.arange(1080)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2) / 50)

plt.imshow(Z, cmap='gray')
plt.title('Pattern Circolare')
plt.show()
```

La funzione **np.eye** di NumPy crea una matrice identità. Una matrice identità è una matrice quadrata in cui tutti gli elementi della diagonale principale sono 1 e tutti gli altri elementi sono 0.

```
np.eye(N, M=None, k=0, dtype=float, order='C')
```

La sintassi è **np.eye(N, M=None, k=0, dtype=float, order='C')**, dove **N** è il numero di righe della matrice. **M**, opzionale, è il numero di colonne della matrice e, se non viene specificato, assume lo stesso valore di **N**, creando così una matrice quadrata. Il parametro **k**, anch'esso opzionale, specifica l'indice della diagonale: la diagonale principale è **k=0**, le diagonali sopra la principale hanno **k>0** e quelle sotto la principale hanno **k<0**. **dtype** indica il tipo di dati degli elementi della matrice, che per default è **float**. **Order** specifica l'ordine degli elementi in memoria ('C' per C-style, 'F' per Fortran-style).

```
I = np.eye(3)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Aumentando o diminuendo il valore di k posso variare la posizione della diagonale:

```
I_shift = np.eye(3, 3, k=1)
```

```
[[0. 1. 0.]
 [0. 0. 1.]
 [0. 0. 0.]]
```

2.5.3 - NUMPY STACK

Oltre ai metodi di manipolazione che abbiamo visto, come lo slicing per inserire valori o la creazione di vettori da impilare, NumPy offre anche funzionalità per combinare e dividere matrici in modo efficiente.

Le funzioni di stacking di NumPy permettono di combinare insieme più matrici. Ad esempio, **vstack** combina le matrici verticalmente, mentre **hstack** le combina orizzontalmente. È possibile ottenere risultati simili con **concatenate** specificando l'asse: 0 per la combinazione lungo l'asse delle righe (verticale) e 1 per la combinazione lungo l'asse delle colonne (orizzontale).

```
import numpy as np

A = np.ones((2, 2))
B = np.zeros((2, 2))

# Impilare verticalmente
vstack_result = np.vstack((A, B))
print("Vertical Stack Result:")
print(vstack_result)

# Impilare orizzontalmente
hstack_result = np.hstack((A, B))
```

```
print("Horizontal Stack Result:")
print(hstack_result)
```

Vertical Stack Result:

```
[[1. 1.]
 [1. 1.]
 [0. 0.]
 [0. 0.]]
```

Horizontal Stack Result:

```
[[1. 1. 0. 0.]
 [1. 1. 0. 0.]]
```

La funzione **block** di NumPy è utilizzata per unire array in una matrice più grande secondo uno schema specificato. Puoi pensare a questa funzione come un modo per combinare più array più piccoli in un unico array più grande, organizzandoli in una griglia.

```
import numpy as np

A = np.ones((2, 2))
B = np.eye(2, 2)
C = np.zeros((2, 2))
D = np.diag((-3, -4))
```

avremo:

$$A = [[1, 1], \quad B = [[1, 0], \quad C = [[0, 0], \quad D = [[-3, 0],\\ [1, 1]] \quad [0, 1]] \quad [0, 0]] \quad [0, -4]]$$

```
result = np.block([[A, B], [C, D]])
```

```
[[ 1., 1., 1., 0.],
 [ 1., 1., 0., 1.],
 [ 0., 0., -3., 0.],
 [ 0., 0., 0., -4.]]
```

Oltre a combinare matrici, NumPy offre anche diversi modi per dividere array in sotto-array. Questi metodi sono utili per suddividere dati. La funzione **split** divide un array in più sotto-array come viste dell'array originale.

```
import numpy as np

A = np.arange(16).reshape(4, 4)
# Divide A in due sotto-array lungo l'asse 1 (orizzontale)
split_result = np.split(A, 2, axis=1)
print("Split Result:")
print(split_result)
```

*Split Result: [array([[0, 1], [4, 5], [8, 9], [12, 13]]),
 array([[2, 3], [6, 7], [10, 11], [14, 15]])]*

A volte può capitare di dover spacchettare i tre canali rgb di un'immagine o di dover fare l'operazione inversa. La funzione **dstack** di NumPy è utilizzata per impilare array lungo il terzo asse (profondità), mentre **dsplit** divide un array lungo il terzo asse. Quindi, **dsplit** può essere considerato come l'operazione inversa di **dstack**.

```
import numpy as np

# Creare tre matrici 2x2 per i canali R, G e B
R = np.array([[255, 0], [0, 255]])
G = np.array([[0, 255], [255, 0]])
B = np.array([[0, 0], [255, 255]])

# Impilare le tre matrici lungo il terzo asse per creare un'immagine
RGB = np.dstack((R, G, B))
print("Image RGB:")
print(RGB)

# Dividere l'immagine RGB lungo il terzo asse per ottenere i canali R,
# G e B
R_split, G_split, B_split = np.dsplt(RGB, 3)
print("\nChannel R:")
print(R_split)
print("\nChannel G:")
print(G_split)
print("\nChannel B:")
print(B_split)
```

Le funzioni **vsplit** e **hsplit** dividono un array in più sotto-array lungo l'asse verticale (righe) e l'asse orizzontale (colonne), rispettivamente.

Questo significa che puoi dividere un'immagine in parti uguali orizzontalmente o verticalmente.

```
import numpy as np
import matplotlib.pyplot as plt

# Creare una matrice rappresentante un'immagine RGB (ad esempio 4x4
pixel con 3 canali)
image = np.array([
    [[255, 0, 0], [255, 0, 0], [0, 255, 0], [0, 255, 0]],
    [[255, 0, 0], [255, 0, 0], [0, 255, 0], [0, 255, 0]],
    [[0, 0, 255], [0, 0, 255], [255, 255, 0], [255, 255, 0]],
    [[0, 0, 255], [0, 0, 255], [255, 255, 0], [255, 255, 0]]
], dtype=np.uint8)

# Dividere l'immagine in due parti uguali lungo l'asse verticale
top_half, bottom_half = np.vsplit(image, 2)

# Visualizzare le immagini divise
plt.subplot(1, 2, 1)
plt.imshow(top_half)
plt.title('Top Half')

plt.subplot(1, 2, 2)
plt.imshow(bottom_half)
plt.title('Bottom Half')

plt.show()
```

Queste funzioni di NumPy consentono di manipolare e visualizzare le immagini in modo molto efficiente, facilitando la creazione di pattern e la suddivisione di dati per analisi ulteriori.

2.6 - IL TEMPO DI ESECUZIONE

Dopo aver visto come creare e manipolare matrici in NumPy, diventa semplice intuire come eseguire operazioni aritmetiche su di esse. NumPy permette di eseguire operazioni come moltiplicazione, somma, divisione e sottrazione in modo molto intuitivo e veloce.

Come possiamo intuire, eseguire un'operazione su un'immagine grazie a NumPy è relativamente semplice. Un'immagine è una matrice cubica, ma nonostante questo nome altisonante, moltiplicare o sommare un

valore costante è un gioco da ragazzi. Basta scrivere `x * 1.2` o `x + 0.01` dove `x` è la nostra matrice.

Quando si lavora con il video, una delle principali sfide è mantenere il corretto frame rate. Per il video standard, questo significa eseguire tutte le operazioni necessarie entro 1/60 di secondo per mantenere un frame rate di 60 fps. Questa limitazione di tempo richiede che ogni calcolo di blending o operazione di contrasto sia estremamente efficiente perché ogni millisecondo in più potrebbe fare la differenza. Oltre a questo, c'è da dire che lavorare con il sistema `uint8` a volte può trarre in inganno.

2.6.1 - Considerazioni sulle Operazioni in `uint8`

Il formato `uint8` (unsigned integer a 8 bit) rappresenta i valori dei pixel come interi tra 0 e 255. Questo può creare delle insidie. Il problema più frequente lo abbiamo quando moltiplichiamo insieme due matrici `uint8`. Nelle formule che abbiamo visto, il bianco è normalizzato a 1. Questo significa che un qualunque colore moltiplicato per 1 rimane lo stesso, mentre un qualunque colore moltiplicato per 0 diventa nero. Ma se il bianco è 255, non ottengo lo stesso risultato.

Fin tanto che moltiplico la mia immagine per un valore costante non ho problemi, ma per capire quali sono le operazioni che possiamo usare dobbiamo introdurre il tempo.

2.6.2 - Misurare il Tempo di Esecuzione

Per mantenere il frame rate di 60fps, dobbiamo eseguire tutte le operazioni entro 1/60 di secondo. Per fare ciò, possiamo usare PyQt6 che fornisce un timer e permette di emettere un segnale che può essere usato in altre classi collegando una funzione.

Ecco un esempio di come creare un oggetto per stabilire il tempo:

```
class SynchObject(QObject):
    synch_SIGNAL = pyqtSignal()

    def __init__(self, fps=60, parent=None): # Set FPS to 60
        super().__init__(parent)
        self.fps = fps
        self.syncTimer = QTimer(self)
        self.syncTimer.timeout.connect(self.sync)
```

```
    self.syncTimer.start(1000 // fps)
    self._initialized = True

def sync(self):
    self.synch_SIGNAL.emit()
```

La classe **Synch** è molto semplice, è in pratica un orologio o un clock che viene creato una volta e che ogni 60esimo di secondo emette un segnale.

La classe **SynchObject** è molto semplice; è in pratica un orologio che ogni 60esimo di secondo emette un segnale. In PyQt esiste questo metodo dei segnali che permette di mandare dati da una porzione all'altra di codice. Se il tipo di dato: str, int, float non viene specificato di default è un valore boolean true.

Ho creato questo esempio veloce per chiare i segnali. La classe1 ha un pulsante, posso collegare il click all'emissione di un segnale che può essere intercettato dalla classe2 per far accadere qualcosa.

```
from PyQt6.QtCore import QObject, pyqtSignal
from PyQt6.QtWidgets import QPushButton, QLabel

class class1(QObject):
    segnaleEseguito = pyqtSignal()

    def __init__(self):
        super().__init__()

        self.button = QPushButton("Cliccami")
        self.button.clicked.connect(self.funzione_da_eseguire)
        self.button.show()

    def funzione_da_eseguire(self):
        print("Il pulsante è stato cliccato!")
        self.segnaleEseguito.emit()

class class2(QObject):

    def __init__(self):
        super().__init__()
        self.class1 = class1()
        self.class1.segnaleEseguito.connect(self.funzione_da_eseguire)
        self.lbl = QLabel("Aspetto il segnale")
        self.lbl.show()
```

```

def funzione_da_eseguire(self):
    self.lbl.setText("Il segnale è stato emesso")

if __name__ == '__main__':
    import sys
    from PyQt6.QtWidgets import QApplication

    app = QApplication(sys.argv)
    window1 = class1()
    window2 = class2()
    sys.exit(app.exec())

```

2.6.3- Sincronizzare output e input

Utilizzando il sistema dei segnali, possiamo creare una sincronia tra input e output. Tuttavia, è fondamentale che ogni operazione avvenga entro il tempo del clock del segnale; altrimenti, verrà utilizzata l'immagine precedente. Questo può causare la percezione di un rallentamento nel video.

Quando si lavora con video in tempo reale, ogni frame deve essere elaborato entro un determinato intervallo di tempo per mantenere un frame rate costante. Nel nostro caso, con un frame rate di 60 fps, ogni operazione deve essere completata entro 1/60 di secondo. Se l'operazione non è completata in tempo, il sistema utilizzerà il frame precedente, causando un ritardo percepibile nell'output video.

```

import numpy as np
from PyQt6.QtCore import QObject

class randomNoiseGenerator(QObject):

    def __init__(self, synch, parent=None):
        super().__init__(parent)
        self._frame = np.zeros((1080, 1920, 3), dtype=np.uint8)
        self.synch = synch
        self.synch.SIGNAL.connect(self.captureFrame)

    def captureFrame(self):
        self._frame = np.random.randint(0, 256, (1080, 1920, 3),
                                      dtype=np.uint8)

    def getFrame(self):
        return self._frame

```

Per visualizzare l'immagine animata possiamo usare una QLabel utilizzando lo stesso oggetto synch per aggiornare la visualizzazione dell'immagine:

```
import numpy as np
from PyQt6.QtCore import *
from PyQt6.QtGui import *
from PyQt6.QtWidgets import *

class TestNumpy(QObject):
    def __init__(self, synch, parent=None):
        super().__init__(parent)
        self._frame = np.zeros((1080, 1920, 3), dtype=np.uint8)
        self.synch = synch
        self.synch.synch_SIGNAL.connect(self.captureFrame)

    def captureFrame(self):
        self._frame = np.random.randint(0, 256, (1080, 1920, 3),
                                       dtype=np.uint8)

    def getFrame(self):
        return self._frame

class VideoApp(QWidget):
    def __init__(self, synch, test, parent=None):
        super().__init__(parent)
        self.lblViewer = QLabel()
        self.synch = synch
        self.test = test
        self.synch.synch_SIGNAL.connect(self.updateViewer)

        self.layout = QVBoxLayout()
        self.layout.addWidget(self.lblViewer)
        self.setLayout(self.layout) # Applica il layout alla finestra

    def updateViewer(self):
        frame = self.test.getFrame()
        qImage = QImage(frame.data, frame.shape[1], frame.shape[0],
                        QImage.Format.Format_RGB888)
        self.lblViewer.setPixmap(QPixmap.fromImage(qImage))

if __name__ == "__main__":
    import sys
    from cap2.cap2_5.synchObject import SynchObject
```

```
app = QApplication(sys.argv)
synch = SynchObject(60)
test = TestNumpy(synch)
video = VideoApp(synch, test)
video.show()
sys.exit(app.exec())
```

2.6.3- Impatto sul Frame Rate

Se le operazioni richieste sul frame non vengono completate entro il tempo del clock (1/60 di secondo), il sistema utilizzerà il frame precedente. Questo ritardo può causare una percezione di rallentamento del video, poiché i frame non vengono aggiornati in tempo reale.

Per evitare questo problema, è importante ottimizzare le operazioni che eseguiamo sui frame. Ad esempio, operazioni complesse come la trasformazione logaritmica o la regolazione del contrasto con funzioni non lineari possono richiedere più tempo rispetto a semplici operazioni di somma o moltiplicazione.

2.7 - MISURARE IL TEMPO

La sincronizzazione tra input e output è cruciale per mantenere un frame rate costante e garantire un'esperienza video fluida. Utilizzando il sistema dei segnali in PyQt6, possiamo assicurare che ogni operazione avvenga entro il tempo del clock, minimizzando il rischio di rallentamenti nel video. Ottimizzare le operazioni sulle matrici è essenziale per rispettare queste tempistiche e mantenere una qualità video elevata.

Ottimizzare la matematica in operazioni fra matrici non è semplice. C'è un corso del M.I.T. a riguardo, il 6.172 - Performance Engineering of Software Systems, tenuto dai professori Charles Leiserson e Julian Shun, che approfondisce questo argomento e lo si trova anche su youtube.

Ci sono almeno altri due aspetti da considerare oltre all'algoritmo stesso: il sistema operativo e Python. Sembra banale però se non si hanno dati alla mano non si riesce molto bene a capire come cambiare il codice per renderlo più veloce.

Un modo semplice per misurare il tempo di esecuzione in Python è usare la libreria **time**:

```
import time

def payload():
    pass

start = time.time()
payload() # eseguiamo un carico
print(f"Tempo di esecuzione: {time.time() - start:.6f} secondi")
```

La funzione **time.time()** restituisce il numero di secondi trascorsi dalla mezzanotte del 1° gennaio 1970 fino al momento in cui viene chiamata. Salvando il valore iniziale in una variabile, possiamo sottrarre il valore alla fine dell'operazione e stampare il tempo impiegato.

Tuttavia, misurare il tempo di esecuzione una sola volta potrebbe non essere indicativo. È meglio misurare un tempo medio ripetendo l'operazione più volte.

Per questo motivo, molti usano **timeit (pip install timeit)**, che fornisce un risultato più affidabile e scientifico.

```
from timeit import timeit

def payload():
    pass

timeIT = timeit(payload, number=10)
print(f"Tempo medio di esecuzione per 10 esecuzioni: {timeIT:.6f} secondi")
```

Un altro strumento molto utile è **cProfile**, che serve a misurare il tempo di esecuzione di diverse parti del codice. Questo processo è chiamato **profilazione deterministica**. Python, per come è scritto, monitora tutti gli eventi di chiamata, ritorno da una funzione ed eccezioni, misurando precisamente i tempi tra questi eventi e fornendo statistiche dettagliate sull'esecuzione del programma. Usare **cProfile** richiede un po' di risorse, però ci permette di individuare facilmente colli di bottiglia indesiderati

perché fornisce un report dettagliato delle chiamate di funzione. Ecco un esempio pratico:

```
import cProfile

def slow_function():
    total = 0
    for i in range(10000):
        total += sum(j for j in range(100))
    return total

def main():
    result = slow_function()
    print(result)

if __name__ == "__main__":
    cProfile.run('main()')

49500000
1020006 function calls in 0.112 seconds

Ordered by: standard name

      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
          1    0.000   0.000   0.112   0.112 <string>:1(<module>)
          1    0.003   0.003   0.112   0.112 profileTest.py:3(slow_function)
  1010000    0.050   0.000   0.050   0.000 profileTest.py:6(<genexpr>)
          1    0.000   0.000   0.112   0.112 profileTest.py:9(main)
          1    0.000   0.000   0.112   0.112 {built-in method builtins.exec}
          1    0.000   0.000   0.000   0.000 {built-in method builtins.print}
  10000    0.060   0.000   0.010   0.000 {built-in method builtins.sum}
          1    0.000   0.000   0.000   0.000  {method 'disable' of '_lsprof.Profiler'
objects}
```

Il report ci dice che il programma ha effettuato un totale di 1,020,006 chiamate di funzione e ha impiegato 0.112 secondi per eseguirsi. La maggior parte del tempo (0.060 secondi) è spesa nella funzione `sum`, chiamata 10,000 volte e c'è una funzione `<genexpr>`, che impiega 0.050 secondi ed è chiamata 1,010,000 volte, indicando un'operazione ripetitiva e costosa.

Questo è uno dei casi dove i large language Model come GPT, Gemini e Claude possono aiutarti a comprendere questi report. Nel nostro caso,

genexpr è l'espressione generatrice **j for j in range(100)** è la causa della lentezza; Il nostro modo di scrivere il codice può causare rallentamenti spesso inaspettati e ce ne accorgiamo solamente quando mettiamo tutto insieme. Grazie a questo check, possiamo ottimizzare la funzione **slow_function**:

```
import cProfile

def slow_function_optimized():
    total = 0
    for i in range(10000):
        total += sum(range(100))
    return total

def main_optimized():
    result = slow_function_optimized()
    print(result)

cProfile.run('main_optimized()')
```

```
49500000
10006 function calls in 0.005 seconds
```

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.005	0.005	<string>:1(<module>)
1	0.001	0.001	0.005	0.005	profileTest.py:18(slow_function_optimized)
1	0.000	0.000	0.005	0.005	profileTest.py:24(main_optimized)
1	0.000	0.000	0.005	0.005	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{built-in method builtins.print}
10000	0.003	0.000	0.003	0.000	{built-in method builtins.sum}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Abbiamo una differenza da 0.112 a 0.005 ottenendo lo stesso risultato.

Detto questo possiamo poi passare agli altri due aspetti che fanno parte della retorica dell'ottimizzazione.

Il primo aspetto riguarda il sistema operativo. Anche se ottimizziamo un'operazione per risparmiare 0.01 secondi, il sistema operativo potrebbe avere centinaia o migliaia di thread aperti per altre operazioni, perché la pubblicità è l'anima del commercio e il marketing ha bisogno di sapere chi sei per crearti un desiderio su misura. Tutto questo potrebbe interferire con la nostra ottimizzazione.

Il secondo aspetto è Python stesso. Python è un linguaggio interpretato, che aggiunge un livello di complessità esterno al codice che abbiamo scritto. Ogni istruzione deve essere interpretata e tradotta in codice macchina in tempo reale, il che introduce un overhead rispetto ai linguaggi compilati come C++. Per questo motivo, misurare il tempo di esecuzione è la prima cosa da fare.

2.7.1 - Complessità: Big “O”

In informatica, l'efficienza degli algoritmi si misura in "Big O", che descrive come il tempo di esecuzione o lo spazio utilizzato da un algoritmo cresce al crescere della dimensione dell'input. Ecco alcune delle notazioni Big O più comuni:

- **O(1)**: Tempo costante. L'algoritmo impiega lo stesso tempo indipendentemente dalla dimensione dell'input.
- **O(log n)**: Tempo logaritmico. L'algoritmo impiega meno tempo per ogni incremento dell'input.
- **O(n)**: Tempo lineare. Il tempo di esecuzione cresce in modo lineare con la dimensione dell'input.
- **O(n log n)**: Tempo log-lineare. Utilizzato spesso negli algoritmi di ordinamento efficienti.
- **O(n²)**: Tempo quadratico. Il tempo di esecuzione cresce proporzionalmente al quadrato della dimensione dell'input.

Ad esempio, otteniamo una complessità $O(n^2)$ ogni volta che annidiamo due for loop insieme. Se, per esempio, Python o un altro linguaggio aggiunge i suoi for per interpretare il codice, la complessità aumenta. Quindi, cercare il modo migliore di scrivere l'algoritmo è fondamentale.

La metto al secondo posto perché l'efficienza viene determinata per n numeri e sul caso peggiore. A volte capita di avere un range di valori in cui il caso peggiore non arriverà mai e a parità di "O", una funzione O(n^2) risulta più veloce di una O(log n)!

```
from timeit import timeit

def op1():
    for x in range(1, 1920):
        for y in range(1, 1080):
            pass # Fai qualcosa qui, ad esempio: z = x * y

# Misura il tempo di esecuzione della funzione `op1` con 10 esecuzioni
execution_time = timeit(op1, number=10)
print(f"Tempo di esecuzione per 10 esecuzioni di op1: {execution_time:.6f} secondi")
```

Tempo di esecuzione per 10 esecuzioni di op1: 0.192328 secondi

Che significa che ogni op1 usa 0.019 secondi, troppo elevato per un'animazione fluida che richiede meno di 0.016 secondi per frame (60 fps).

2.7.2 - OverHead

Quando scriviamo codice in Python, dobbiamo considerare che ogni operazione introduce un certo overhead. Ad esempio, un'operazione di moltiplicazione di due matrici con complessità O(n^2) sarà più lenta in Python rispetto a un linguaggio compilato, a causa del tempo aggiuntivo necessario per l'interpretazione del codice.

L'overhead è un concetto importante da comprendere. In Python, quando creiamo una variabile, il linguaggio deve gestire diversi aspetti in background, come l'allocazione della memoria e il type inference (determinazione del tipo di dato). Questo processo richiede tempo e risorse aggiuntive rispetto a linguaggi compilati o a basso livello.

Python utilizza un sistema di gestione della memoria più complesso rispetto alla semplice allocazione di registri. Gli oggetti in Python hanno un

overhead aggiuntivo per memorizzare informazioni come il tipo di dato e il conteggio dei riferimenti.

Questo comporta dei costi in termini di tempo perché, se ad esempio considero il mio numero 0-255 a 8 bit, in un solo registro a 64 bit posso ad esempio mettere 8 pixel!

NumPy è implementato in C e quando creiamo una struttura dati tipo una matrice in NumPy, potrebbe ad esempio impacchettare 8 pixel in un singolo registro, rendendo la matrice 8 volte più piccola.

```
import numpy as np

def op2():
    w = 1920
    h = 1080
    x = np.arange(1, w)
    y = np.arange(1, h)
    return np.sum(np.outer(x, y))

execution_time = timeit(op2, number=10)
print(f"Tempo di esecuzione per 10 esecuzioni di op2: {execution_time:.6f} secondi")
```

Tempo di esecuzione per 10 esecuzioni di op2: 0.040971 secondi!

L'altra buona notizia è che openCv usa numpy per rappresentare le immagini e entrambi possono fare operazioni in modo molto, molto veloce e alla fine di questo capitolo capiremo come.

2.7.3 - Numpy lightSpeed

Ottimizzare il codice non è solo una questione di ridurre il numero di operazioni, ma anche di scegliere le operazioni giuste per l'hardware e il linguaggio usato. Per esempio, le operazioni su matrici NumPy sono

ottimizzate per eseguire calcoli in parallelo utilizzando librerie ad alte prestazioni scritte in C.

Scegliere il modo giusto di scrivere il codice, come utilizzare funzioni vettorializzate di NumPy anziché loop Python, può fare una grande differenza nelle prestazioni. Misurare il tempo di esecuzione e profilare il codice ci permette di capire dove sono i colli di bottiglia e di ottimizzare le parti più lente per migliorare il framerate del video.

La vectorization in NumPy è un concetto fondamentale che permette di eseguire operazioni su interi array in modo efficiente, senza l'uso di cicli espliciti. Questo giro di paroloni significa sostanzialmente che, per attraversare una matrice e fare un qualunque tipo di operazione, il metodo classico è annidare due for loop:

```
from timeit import timeit
import numpy as np

# Creazione delle matrici casuali
A = np.random.rand(1920, 1080)
B = np.random.rand(1920, 1080)

# Funzione per la moltiplicazione tradizionale con for loop
def traditional_multiplication(A, B):
    result = np.zeros(A.shape)
    for i in range(A.shape[0]):
        for j in range(A.shape[1]):
            result[i, j] = A[i, j] * B[i, j]
    return result

# Misurazione del tempo di esecuzione per la moltiplicazione vettorizzata
execution_time_vectorized = timeit(lambda: A * B, number=10)
print(f"Tempo di esecuzione per 10 esecuzioni della
      moltiplicazione vettorizzata:
      {execution_time_vectorized:.6f} secondi")

# Misurazione del tempo di esecuzione per la moltiplicazione tradizionale
execution_time_traditional = timeit(lambda: traditional_multiplication(A, B),
                                       number=10)
print(f"Tempo di esecuzione per 10 esecuzioni della
      moltiplicazione tradizionale:
      {execution_time_traditional:.6f} secondi")
```

Una qualunque operazione matematica tra matrici in NumPy è vettorizzata, ovvero l'operazione è il processo di conversione di operazioni scalari (che operano su singoli elementi) in operazioni vettoriali (che operano su interi array).

Sono istruzioni speciali dette SIMD (Single Instruction, Multiple Data) messe a disposizione dai compilatori C per ottimizzare alcuni calcoli. Questo significa sostanzialmente che NumPy tratta le matrici non usando un ciclo for loop annidato, ma considerando la matrice come un'entità unica.

Tempo di esecuzione per 10 esecuzioni della moltiplicazione vettorizzata: 0.038088 secondi
Tempo di esecuzione per 10 esecuzioni della moltiplicazione tradizionale: 3.906912 secondi

In questo rientrano le operazioni aritmetiche come +, -, *, /, *, alcune funzioni matematiche: np.sin(), np.exp(), np.log(), le operazioni logiche >, <, ==, ! e le funzioni di riduzione: np.sum(), np.mean(), np.max().

3.0 - Matematica del video

Come abbiamo visto nei capitoli precedenti, possiamo utilizzare NumPy per creare delle barre, un rumore o un gradiente. Il passo successivo è creare dei generatori di segnale che sono utili innanzitutto perché ci permettono di testare il mixer senza avere una scheda di acquisizione o una telecamera USB a disposizione.

Quello che dobbiamo fare adesso è creare una classe base da usare per i vari input. La nostra classe base ideale deve avere delle funzioni per ottenere il frame catturato, una funzione per catturare il frame e per aggiornare il framerate.

```
import time

import numpy as np
from PyQt6.QtCore import QObject, QSize

class BaseClass(QObject):

    def __init__(self, synchObject, resolution=QSize(1920, 1080)):
        super().__init__()
        """
        Initializes the base class with a synchronization object and a resolution.
        Connects the synchronization signal to the capture_frame function.
        """
        # init object
        self.synch_Object = synchObject
        self.resolution = resolution
        self._frame = np.zeros((resolution.height(), resolution.width(), 3),
                              dtype=np.uint8)
        # init variables
        self.start_time = time.time()
        self.frame_count = 0
        self.total_time = 0
        self.fps = 0
        self.last_update_time = time.time()
        # init connections
        self.initConnections()

    def initConnections(self):
        """
        Initializes the connections between the synchronization object
        and the captureFrame function.
        """
        self.synch_Object.synch_SIGNAL.connect(self.captureFrame)

    def captureFrame(self):
        """
        This class is a base class for all frame processors.
        It captures a frame and updates the FPS value.
        """


```

```

:rtype:
"""
self.updateFps()

def getFrame(self):
    """
    Returns the current frame.
    :return:
    """
    if self._frame is None:
        return self.returnBlackFrame()
    return self._frame

def updateFps(self):
    """
    Updates the FPS value.
    :return:
    """
    self.frame_count += 1
    current_time = time.time()
    elapsed_time = current_time - self.last_update_time
    if elapsed_time >= 1.0: # Update FPS every second
        self.fps = self.frame_count / elapsed_time
        self.frame_count = 0
        self.last_update_time = current_time

def returnBlackFrame(self):
    """
    Returns a black frame.
    """
    return np.zeros((self.resolution.height(), self.resolution.width(), 3),
                   dtype=np.uint8)

```

Il calcolo del frame rate (FPS, frames per second) è essenziale per garantire che le operazioni video siano eseguite in modo fluido e coerente. Nel contesto delle applicazioni video, il frame rate rappresenta il numero di immagini (frame) visualizzate al secondo. Per mantenere un frame rate stabile, è necessario monitorare e aggiornare costantemente il tempo impiegato per elaborare e visualizzare ogni frame.

Nel codice fornito, utilizziamo diverse variabili e funzioni per monitorare il frame rate e garantire che l'elaborazione dei frame avvenga entro il tempo desiderato. Ecco una spiegazione dettagliata di come funziona il calcolo del frame rate:

```
self.start_time = time.time() # Memorizza l'orario di inizio
self.frame_count = 0          # Conta il numero di frame elaborati
self.total_time = 0           # Totale del tempo trascorso
self.fps = 0                  # Frame per secondo attuali
self.last_update_time = time.time() # Ultimo aggiornamento del tempo
```

Ogni volta che viene aggiornato il frame rate, la funzione **updateFps** esegue una serie di operazioni. Innanzitutto, incrementa il contatore dei frame di uno, tenendo traccia del numero di frame elaborati dall'ultimo aggiornamento degli FPS. Successivamente, ottiene il tempo corrente e calcola il tempo trascorso dall'ultimo aggiornamento degli FPS sottraendo il tempo dell'ultimo aggiornamento dal tempo corrente. Se è trascorso almeno un secondo dall'ultimo aggiornamento, la funzione calcola il frame rate dividendo il numero di frame elaborati per il tempo trascorso, fornendo così il numero di frame al secondo. Una volta calcolato il frame rate, il contatore dei frame viene resettato a zero per iniziare un nuovo conteggio per il prossimo secondo e il tempo dell'ultimo aggiornamento viene aggiornato con il tempo corrente, preparandosi per il prossimo ciclo di aggiornamento.

Ho aggiunto anche una funzione `return blackFrame`, in questo modo nel caso in cui ci sia un problema di acquisizione o di altra natura restituisce un frame nero della risoluzione specificata al posto di un `None` e previene i crash del programma.

Proviamo ad esempio a creare un generatore di colore.

```
class ColorGenerator(BaseClass):
    _color = QColor(250, 0, 0) # Colore iniziale: rosso

    def __init__(self, synchObject, resolution=QSize(1920, 1080)):
        super().__init__(synchObject, resolution)
        self.setColor(self._color) # Inizializza l'immagine con il colore specificato

    def setColor(self, color: QColor):
        """
        Imposta il colore dell'immagine generata.
        """
        self._color = color
        self._frame[:, :] = [color.blue(), color.green(), color.red()]

    def captureFrame(self):
```

```
"""
    Sovrascrive la funzione captureFrame della classe base, mantenendo la funzionalità
originale.
"""

super().captureFrame()

def getFrame(self):
    """
        Sovrascrive la funzione getFrame della classe base, mantenendo la funzionalità originale.
    """
    return super().getFrame()
```

Come vedete, ereditando la maggior parte del codice dalla classe base, non dobbiamo riscriverlo e tutte le classi che creiamo utilizzano gli stessi meccanismi.

In PyQt, il modo più semplice per visualizzare un'immagine è in una QLabel. Qt per le immagini utilizza due classi, QImage e QPixmap, che purtroppo hanno un loro costo nell'economia del programma, ma come vedremo ci sono dei modi per ottimizzare l'operazione.

Creiamo quindi un altro file per testare il generatore di colore che chiameremo colorGenerator_TEST.py

Gli diamo quindi la struttura classica che rende l'init un po più leggibile al volo inserendo prima l'inizializzazione dei widget che verranno usati per costruire l'interfaccia, poi degli oggetti esterni che saranno usati, quindi delle variabili e per finire gli altri gli altri init.

La prima cosa che facciamo è legare l'oggetto **synch** a una funzione che chiama il **getFrame** dell'input che restituisce una matrice numpy.

L'ordine dei canali RGB in opencv è BGR, per fortuna Qt offre per le immagini a 8 bit un profilo colore compatibile, **Format_BGR888**, che ci evita di dover convertire l'immagine ogni sessantesimo di secondo.

```
import time

import numpy as np
from PyQt6.QtCore import *
from PyQt6.QtGui import *
from PyQt6.QtWidgets import *
```

```

from cap2.cap2_6.synchObject import SynchObject
from cap3.cap3_1.colorGenerator import ColorGenerator

class VideoApp(QApplication):
    def __init__(self, argv):
        super().__init__(argv)
        # init internal widget
        self.widget = QWidget()
        self.viewer = QLabel()
        self.fpsLabel = QLabel("FPS: 0.00") # Initialize FPS label
        self.displayLabel = QLabel()

        # init external object
        self.synchObject = SynchObject(60)
        self.input1 = ColorGenerator(self.synchObject)
        # init layout
        self.initLayout()
        # init connections
        self.initConnections()
        # init geometry
        self.initGeometry()
        # init a single shot timer to stop the app after 10 seconds
        QTimer.singleShot(10000, self.stopApp)

    def initLayout(self):
        mainLayout = QVBoxLayout()
        mainLayout.addWidget(self.viewer)
        mainLayout.addWidget(self.fpsLabel)
        mainLayout.addWidget(self.displayLabel)
        self.widget.setLayout(mainLayout)

    def initConnections(self):
        self.synchObject.synch_SIGNAL.connect(self.displayFrame)

    def initGeometry(self):
        self.viewer.setFixedSize(1920, 1080)
        self.widget.show()

    def displayFrame(self):
        frame = self.input1.getFrame()
        if frame is not None and frame.size != 0:
            start_time = time.time()
            image = QImage(frame.data, frame.shape[1], frame.shape[0],
                           QImage.Format.Format_BGR888)
            self.viewer.setPixmap(QPixmap.fromImage(image))
            display_time = time.time() - start_time
            self.displayLabel.setText(f"Frame displayed in {display_time:.6f} seconds")
            self.fpsLabel.setText(f"FPS: {self.input1.fps:.2f}") # Update FPS label

    def stopApp(self):
        print(f"Media FPS: {self.input1.fps:.2f}")
        self.exit()

```

```

def mainApp():
    app = VideoApp(sys.argv)
    app.exec()

if __name__ == "__main__":
    import sys
    import cProfile
    import pstats
    import io

    pr = cProfile.Profile()
    pr.enable()
    mainApp()
    pr.disable()
    s = io.StringIO()
    sortby = 'cumulative'
    ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
    ps.print_stats()
    print(s.getvalue())

```

Media FPS: 62.44

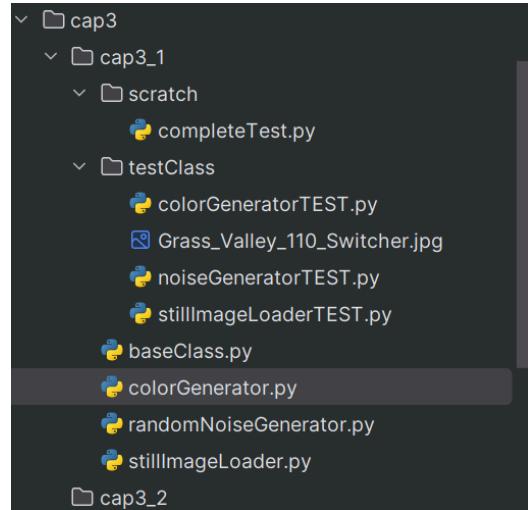
14642 function calls (14565 primitive calls) in 10.064 seconds

Ordered by: cumulative time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	10.064	10.064	C:\pythonCode\openPyVisionBook\openPyVisionBook\cap3\cap3_1\testClass\colorGeneratorTEST.py:62(mainApp)
1	8.026	8.026	10.001	10.001	{built-in method exec}
622	0.005	0.000	1.975	0.003	C:\.....\synchObject.py:15(sync)
622	0.013	0.000	1.970	0.003	{method 'emit' of 'PyQt6.QtCore.pyqtBoundSignal' objects}
622	0.041	0.000	1.950	0.003	
C:\.....estClass\colorGeneratorTEST.py:47(displayFrame)					
622	1.272	0.002	1.272	0.002	{built-in method setPixmap}
622	0.620	0.001	0.620	0.001	{built-in method fromImage}
1	0.011	0.011	0.062	0.062	
C:\.....cap3\cap3_1\testClass\colorGeneratorTEST.py:13(__init__)					
1	0.000	0.000	0.043	0.043	
C:\.....(initGeometry)					
1	0.043	0.043	0.043	0.043	{built-in method show}
1244	0.014	0.000	0.014	0.000	{built-in method setText}

3.1 Organizzare i file

Abbiamo creato quindi una struttura base e possiamo organizzare il codice in modo da non avere un unico gigantesco file. Continueremo a fare esempi certo, però ci saranno alcuni codici che ci porteremo avanti e che implementeremo di volta in volta. Possiamo salvare ad esempio la classe Base in un file `baseClass.py`, il `synchObject` in un file `.py` e così come i vari input, quindi avremo il nostro `colorGenerator.py` e potrete trovare i file nella cartella 4.9.



Oltre al color generator possiamo creare altri due input, un `randomNoiseGenerator` e uno `still image Loader`.

Ci sono vari modi per generare un rumore e vi stupirà sapere che esistono alcuni metodi abbastanza famosi, come il rumore di Perlin, che gli ha fatto vincere un oscar per gli effetti speciali.

Il metodo generalmente più veloce e intuitivo è quello di usare `numpy` come:

```
self._frame = np.random.randint(0, 255, (self.resolution.height(), self.resolution.width(), 3), dtype=np.uint8)
```

quindi, molto semplicemente la nostra classe diventerà:

```
import numpy as np
from PyQt6.QtCore import *

from cap3.cap3_1.baseClass import BaseClass
```

```

class RandomNoiseGenerator(BaseClass):

    def __init__(self, synchObject, resolution=QSize(1920, 1080)):
        super().__init__(synchObject, resolution)
        self._frame = np.zeros((resolution.height(), resolution.width(),
                               3), dtype=np.uint8)

    def captureFrame(self):
        """
        Sovrascrive la funzione captureFrame della classe base,
        mantenendo la funzionalità originale.
        """
        super().captureFrame()
        self._frame = np.random.randint(0, 255, (self.resolution.height(),
                                                self.resolution.width(), 3), dtype=np.uint8)

    def getFrame(self):
        """
        Sovrascrive la funzione getFrame della classe base,
        mantenendo la funzionalità originale.
        """
        return super().getFrame()

```

Possiamo scrivere un test per la classe in coda al codice, oppure creare una cartella con tutti i file, uno per ogni test, usando il codice che vi mostrato nella classe VideoApp.

Proviamo ora a creare un input abbastanza semplice da realizzare, ma molto utile. Un caricatore per immagini statiche.

```

import cv2
import numpy as np
from PyQt6.QtCore import *
from PyQt6.QtGui import *

from cap3.cap3_1.baseClass import BaseClass

class StillImageLoader(BaseClass):
    _color = QColor(250, 0, 0) # Colore iniziale: rosso

    def __init__(self, imagePath, synchObject, resolution=QSize(1920, 1080)):
        super().__init__(synchObject, resolution)
        self.loadImage(imagePath)

    def loadImage(self, imagePath):

```

```

"""
Carica l'immagine e fa il resize a 1920x1080 se necessario.
"""
try:
    image = cv2.imread(imagePath)
    # se le dimensioni dell'immagine sono diverse da quelle
    # specificate, ridimensiona l'immagine
    if image.shape[:2] != (self.resolution.height(), self.resolution.width()):
        image = cv2.resize(image,
                           (self.resolution.width(), self.resolution.height()))

    self._frame = image
except Exception as e:
    print(f"Error loading image: {e}")
    self._frame = np.zeros((self.resolution.height(),
                           self.resolution.width(), 3), dtype=np.uint8)

def captureFrame(self):
    """
    Sovrascrive la funzione captureFrame della classe base,
    mantenendo la funzionalità originale.
    """
    super().captureFrame()

def getFrame(self):
    """
    Sovrascrive la funzione getFrame della classe base,
    mantenendo la funzionalità originale.
    """
    return super().getFrame()

```

In questo modo se il caricamento dell'immagine fallisce, viene creata un'immagine nera e se invece l'immagine viene caricata, ma non è delle dimensioni che ci si aspetta viene ridimensionata. Come abbiamo fatto con il color Generator, dovremo testare la classe scrivendo una video App apposita, che vi lascio come esercizio.

3.2 - Invert

Ora che sappiamo usare NumPy e abbiamo lo scheletro per generare immagini, possiamo aggiungere degli effetti per migliorarle. Per semplificare i calcoli e per introdurre il problema successivo, ci concentriamo sulla velocità di esecuzione.

Il primo effetto che vedremo è l'inversione dei colori (invert). Per invertire un'immagine posso fare $255 - \text{pixel_value}$, un'operazione molto veloce per immagini a 8 bit. Tuttavia, ogni operazione aggiunge un piccolo delay e alla fine della catena questo delay si fa sentire.

Possiamo fare un test per vedere quale metodo è più veloce a parità di risultati.

```
def invert255(_image):
    return 255 - _image

def invertBitwise(_image):
    return np.bitwise_not(_image)

def invertCV2(_image):
    return cv2.bitwise_not(_image)

if __name__ == "__main__":
    image = np.random.randint(0, 256, (1920, 1080, 3), dtype=np.uint8)
    inv255 = timeit.timeit(lambda: invert255(image), number=100)
    invBitwise = timeit.timeit(lambda: invertBitwise(image), number=100)
    invCV2 = timeit.timeit(lambda: invertCV2(image), number=100)
    print(f"255 - image inversion duration: {inv255:.6f} seconds")
    print(f"Bitwise Np inversion duration: {invBitwise:.6f} seconds")
    print(f"OpenCV Bitwise inversion duration: {invCV2:.6f} seconds")
```

```
255 - image inversion duration: 0.226272 seconds
Bitwise inversion duration: 0.218525 seconds
OpenCV inversion duration: 0.120458 seconds
```

Dai risultati possiamo vedere che tutte le operazioni sono molto veloci ($0.22/100 = 0.0022$ secondi per operazione). Numpy con l'operazione bitwise è leggermente più veloce dell'operazione aritmetica, ma OpenCV è il più veloce di tutti con un tempo di 0.00120 secondi per operazione, il che significa che è quasi istantaneo.

3.3 – AutoScreen

È utile invertire un'immagine? Ci sono alcuni effetti, come ad esempio lo screen, usati per creare effetti di luce. L'effetto Screen è particolarmente utile per creare effetti di luce come quelli usati nei raggi laser, nei bagliori e nelle esplosioni. Con il metodo Screen, i valori dei pixel nei due layer vengono invertiti, moltiplicati e poi nuovamente invertiti. Il risultato è l'opposto di Multiply: ovunque uno dei due layer fosse più scuro del bianco, il composito risultante sarà più luminoso.

La formula matematica per l'effetto Screen è la seguente:

$$f(a,b) = 1 - (1-a) \cdot (1-b)$$

dove a è il valore del layer base e b è il valore del layer superiore.

Steve Wright descrive l'operazione Screen come un metodo elegante per combinare immagini che emettono luce con un'immagine di sfondo, senza usare un matte. Questo è utile quando si desidera combinare la luce da un'immagine con un'altra, come per un flare di lente, il raggio di un'arma energetica, o il bagliore attorno a una lampadina, un fuoco o un'esplosione. L'importante è che l'elemento di luce non blocchi la luce dallo sfondo.

Un punto chiave dell'effetto Screen è che funziona meglio quando il layer superiore è su uno sfondo nero puro. Se alcuni dei pixel non sono neri, finiranno per contaminare l'immagine di sfondo nel risultato finale. Per ottenere un risultato ottimale, è fondamentale che i pixel circostanti l'elemento di luce rimangano neri puri.

In pratica, l'effetto Screen si comporta come una doppia esposizione: i valori di luminosità si avvicinano a 1.0 senza mai superarlo, e il nero su qualsiasi immagine non cambia l'immagine sottostante. Questo permette di combinare le luci senza saturare completamente l'immagine.

Se faccio lo screen della stessa immagine, aumento la luminosità senza saturare i bianchi e mantenendo i neri, un po' come succede con gamma.

La formula per lo screen è $1 - (1-a) \cdot (1-b)$, il problema è che le immagini sono uint8 quindi posso scrivere $255(255-a)*(255-b)$ e ho un errore che non viene segnalato né da numpy né da opencv. Utilizzando il

formato uint8 ho massimo 255 bit ma se moltiplico 255^2 non ottengo 65025, ottengo 255 perchè ho un overflow.

Questo è un problema molto ricorrente che dovremo gestire. Per risolverlo posso fare in due modi. Il primo è normalizzare le matrici in modo da non avere overflow, il secondo è trasformare le immagini invertite in tipo uint16 o uint32.

```
import cv2
import numpy as np
import timeit

def screenNormalized(image):
    # normalizza l'immagine tra 0 e 1
    normalized_image = image / 255.0
    # applica l'effetto screen
    screen = 1 - (1 - normalized_image) * (1 - normalized_image)
    return (screen * 255).astype(np.uint8)

def screenNumpy(image):
    inv1 = cv2.bitwise_not(image).astype(np.uint16)
    mult = (inv1 * inv1) // 255
    return cv2.bitwise_not(mult.astype(np.uint8))

def screenOpenCV(image):
    inv1 = cv2.bitwise_not(image)
    mult = cv2.multiply(inv1, inv1, scale=1.0 / 255.0)
    return cv2.bitwise_not(mult).astype(np.uint8)

# Genera un'immagine di test
image = np.full((1080, 1920, 3), 127, dtype=np.uint8)
# visualizza l'immagine
screenNormalizedImage = screenNormalized(image)
screenNumpyImage = screenNumpy(image)
screenOpenCVImage = screenOpenCV(image)
cv2.imshow("Frame", image)
cv2.imshow("ScreenNormalized", screenNormalizedImage)
cv2.imshow("ScreenNumpy", screenNumpyImage)
cv2.imshow("ScreenOpenCV", screenOpenCVImage)

# Esegue il test
screenNormalizedTest = timeit.timeit(lambda: screenNormalized(image),
                                      , number=100)
screenNumpyTest = timeit.timeit(lambda: screenNumpy(image), number=100)
```

```
screenOpenCVTest = timeit.timeit(lambda: screenOpenCV(image), number=100)

print(f"Normalized screen duration: {screenNormalizedTest:.6f}
      seconds - matrix check: {screenNormalizedImage[0, 0]}")
print(f"Numpy screen duration: {screenNumpyTest:.6f}
      seconds - matrix check: {screenNumpyImage[0, 0]}")
print(f"OpenCV screen duration: {screenOpenCVTest:.6f}
      seconds - matrix check: {screenOpenCVImage[0, 0]}")

cv2.waitKey(0)
```

Quello che mi aspetto è che i valori da 127 arrivi a 191 perchè:

$$\begin{aligned}255-127 &= 128 \\128/255 &= 0,50 \\1 - (1-0,5)*(1-0,5) &= 0,75 \\0,75 * 255 &= 191,25\end{aligned}$$

arrotondato a 191, il risultato è:

*Normalized screen duration: 6.908173 seconds - matrix check: [190 190 190]
Numpy screen duration: 1.415529 seconds - matrix check: [191 191 191]
OpenCV screen duration: 0.526234 seconds - matrix check: [191 191 191]*

Se provo a fare la stessa cosa con un'immagine casuale tipo:

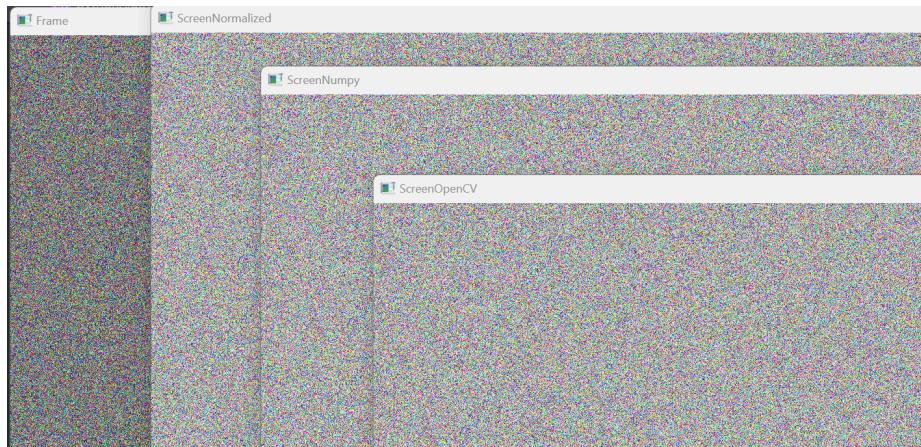
image = np.random.randint(0, 256, (1080, 1920, 3), dtype=np.uint8)

Ottengo i seguenti risultati:

Normalized screen duration: 7.201791 seconds - matrix check: [249 140 137]

Numpy screen duration: 1.467107 seconds - matrix check: [250 141 138]

OpenCV screen duration: 0.538933 seconds - matrix check: [250 140 138]



3.4 - Split RGB

Ci sono occasioni in cui è utile avere i canali colore separati, sia per creare maschere che per applicare effetti. Sia NumPy che OpenCV offrono diversi metodi per separare i canali colore:

```
import cv2
import numpy as np
import timeit

# Carica un'immagine di test con quattro canali
# (incluso il canale alfa)
image = np.random.randint(0, 256, (1920, 1080, 4), dtype=np.uint8)

def numpy_index_4ch():
    b = image[:, :, 0]
    g = image[:, :, 1]
    r = image[:, :, 2]
    a = image[:, :, 3]
    return b, g, r, a

def numpy_split_4ch():
    b, g, r, a = np.split(image, 4, axis=2)
    return b.squeeze(), g.squeeze(), r.squeeze(), a.squeeze()
```

```

def list_comprehension_4ch():
    b, g, r, a = [image[:, :, i] for i in range(4)]
    return b, g, r, a

def numpy_dsplit_4ch():
    b, g, r, a = np.dsplit(image, 4)
    return b.squeeze(), g.squeeze(), r.squeeze(), a.squeeze()

def numpy_moveaxis_4ch():
    b, g, r, a = np.moveaxis(image, -1, 0)
    return b, g, r, a

def small_opencv_split():
    b, g, r, a = cv2.split(image)
    return b, g, r, a

# Test delle prestazioni
methods = [numpy_index_4ch, numpy_split_4ch,
           list_comprehension_4ch, numpy_dsplit_4ch, numpy_moveaxis_4ch,
           small_opencv_split]
for method in methods:
    time = timeit.timeit(method, number=10000)
    print(f"{method.__name__}: {time:.6f} seconds")

```

```

numpy_index_4ch: 0.005418 seconds
numpy_split_4ch: 0.057531 seconds
list_comprehension_4ch: 0.007197 seconds
numpy_dsplit_4ch: 0.059373 seconds
numpy_moveaxis_4ch: 0.021403 seconds
small_opencv_split: 34.294690 seconds

```

Il tempo di esecuzione va diviso per 10mila, però c'è chiaramente un vincitore.

Sono abbastanza sorpreso della comprehension list, non mi aspettavo fosse così veloce e un po 'deluso da cv2.split perchè era il mio metodo favorito, nel capitolo successivo però capiremo come fa ad essere così veloce. A questo punto conviene fare un test per vedere chi è più veloce a mettere insieme un'immagine a 4 canali.

```

import cv2
import numpy as np
import timeit

# Crea quattro matrici separate per simulare i canali B, G, R, A
b = np.random.randint(0, 256, (1080, 1920), dtype=np.uint8)
g = np.random.randint(0, 256, (1080, 1920), dtype=np.uint8)
r = np.random.randint(0, 256, (1080, 1920), dtype=np.uint8)
a = np.random.randint(0, 256, (1080, 1920), dtype=np.uint8)

def numpy_stack():
    return np.stack((b, g, r, a), axis=-1)

def numpy_dstack():
    return np.dstack((b, g, r, a))

def numpy_concatenate():
    return np.concatenate((b[..., np.newaxis], g[..., np.newaxis],
                          r[..., np.newaxis], a[..., np.newaxis]), axis=2)

def list_to_array():
    return np.array([b, g, r, a]).transpose(1, 2, 0)

def opencv_merge():
    return cv2.merge([b, g, r, a])

def manual_assignment():
    img = np.empty((1080, 1920, 4), dtype=np.uint8)
    img[:, :, 0] = b
    img[:, :, 1] = g
    img[:, :, 2] = r
    img[:, :, 3] = a
    return img

# Test delle prestazioni
methods = [numpy_stack, numpy_dstack, numpy_concatenate, list_to_array,
opencv_merge, manual_assignment]
for method in methods:
    time = timeit.timeit(method, number=1000)
    print(f"{method.__name__}: {time:.6f} seconds")

```

numpy_stack: 3.311415 seconds
numpy_dstack: 3.297586 seconds
numpy_concatenate: 3.262816 seconds
list_to_array: 1.288917 seconds
opencv_merge: 1.613998 seconds
manual_assignment: 3.479700 seconds

I valori qui vanno divisi per 1000. In questo caso invece list to array e opencv sono i più veloci. Da quello che emerge mettere insieme un'immagine partendo dai singoli canali colore è molto più lento che non separarla.

3.5 - Gamma

Come visto nel capitolo 3, l'operazione di gamma è $x^{1/\text{gamma}}$ computazionalmente costosa in termini di complessità Big O. Tuttavia, esiste un metodo ottimizzato con OpenCV chiamato Look Up Table (LUT). In questa sezione, implementeremo il calcolo della correzione gamma utilizzando sia OpenCV che NumPy, per comprendere meglio le differenze di prestazioni.

```
import cv2
import numpy as np
import timeit
import matplotlib.pyplot as plt

# Funzione per applicare la correzione gamma usando LUT
def apply_gamma_lut(image, gamma):
    inv_gamma = 1.0 / gamma
    table = np.array([(i / 255.0) ** inv_gamma * 255
                      for i in range(256)]).astype(np.uint8)
    return cv2.LUT(image, table)

# Funzione per applicare la correzione gamma usando np.power
def apply_gamma_numpy(image, gamma):
    inv_gamma = 1.0 / gamma
    image = image / 255.0
    image = np.power(image, inv_gamma)
    return np.uint8(image * 255)

# Funzione per applicare la correzione gamma usando skimage
def apply_gamma_cv2(image, gamma):
    inv_gamma = 1.0 / gamma
    image = image / 255.0
    image = cv2.pow(image, inv_gamma)
    return np.uint8(image * 255)
```

```

# Crea un'immagine di test con quattro canali (incluso il canale alfa)
image = np.random.randint(0, 256, (1920, 1080, 4), dtype=np.uint8)

# Valore di gamma
gamma_value = 0.2

# Metodi per applicare la correzione gamma
methods = {
    'gamma_lut': apply_gamma_lut,
    'gamma_cv2': apply_gamma_cv2,
    'gamma_numpy': apply_gamma_numpy,
}

# Test delle prestazioni
for method_name, method in methods.items():
    time = timeit.timeit(lambda: method(image, gamma_value), number=1000)
    print(f"{method_name}: {time:.6f} seconds")

# Verifica che tutti i metodi producano lo stesso risultato
results = [method(image, gamma_value) for method_name, method in
methods.items()]
for i in range(1, len(results)):
    if not np.array_equal(results[0], results[i]):
        print(f"Il metodo {list(methods.keys())[i]} produce un risultato diverso")

print("Verifica completata.")

# Visualizzazione delle immagini
fig, axes = plt.subplots(1, 4, figsize=(20, 10))

# Immagine originale
axes[0].imshow(cv2.cvtColor(image, cv2.COLOR_BGRA2RGBA))
axes[0].set_title('Original')

# Risultati dei metodi
for ax, (method_name, result) in zip(axes[1:], methods.items()):
    ax.imshow(cv2.cvtColor(result[list(methods.keys()).index(method_name)], cv2.COLOR_BGRA2RGBA))
    ax.set_title(method_name)

# Disattiva gli assi
for ax in axes:
    ax.axis('off')

plt.show()

```

gamma_lut: 1.166029 seconds
gamma_cv2: 48.619653 seconds

```
gamma_numpy: 153.835296 seconds
```

I valori qui vanno divisi per 1000. Il risultato può sembrare scontato, ma non lo è affatto. Le LUT sono estremamente efficienti per le operazioni di mappatura dei valori dei pixel, poiché consentono di sostituire ogni valore di pixel con un valore pre-calcolato in una singola operazione di lookup. Questo riduce drasticamente il tempo di calcolo rispetto all'applicazione di una funzione di potenza su ogni pixel individualmente.

3.5.1 – Approfondimento sulle LUT

Come fa ad essere così veloce? OpenCV e chi ha studiato questo metodo hanno fatto una specie di hack mettendo insieme due operazioni molto veloci. La prima è la generazione di una tabella di lookup. Per esempio, avere un gamma di 2.2 genera una lista di valori pre-calcolati:

```
[ 0 20 28 33 38 42 46 49 52 55 58 61 63 65 68 70 72 74  
 76 78 80 81 83 85 87 88 90 91 93 94 96 97 99 100 102 103  
104 106 107 108 109 111 112 113 114 115 117 118 119 120 121 122 123 124  
125 126 128 129 130 131 132 133 134 135 136 136 137 138 139 140 141 142  
143 144 145 146 147 147 148 149 150 151 152 153 153 154 155 156 157 158  
158 159 160 161 162 162 163 164 165 165 166 167 168 168 169 170 171 171  
172 173 174 174 175 176 176 177 178 178 179 180 181 181 182 183 183 184  
185 185 186 187 187 188 189 189 190 190 191 192 192 193 194 194 195 196  
196 197 197 198 199 199 200 200 201 202 202 203 203 204 205 205 206 206  
207 208 208 209 209 210 210 211 212 212 213 213 214 214 215 216 216 217  
217 218 218 219 219 220 220 221 222 222 223 223 224 224 225 225 226 226  
227 227 228 228 229 229 230 230 231 231 232 232 233 233 234 234 235 235  
236 236 237 237 238 238 239 239 240 240 241 241 242 242 243 243 244 244  
245 245 246 246 247 247 248 248 249 249 249 250 250 251 251 252 252 253  
253 254 254 255]
```

A questo punto, per sostituire i valori si utilizza un ciclo for annidato il cui carico di lavoro consiste nell'usare il valore del pixel come indice. Ad esempio, se il valore è 190, farà `lut[190] = 223`, velocizzando notevolmente i calcoli come abbiamo visto, rendendoli fino a 48 volte più veloci.

3.5.2 - Implementazione di un Generatore di Rumore con Correzione Gamma

Per dimostrare l'efficienza delle LUT nella correzione gamma, implementiamo un generatore di rumore casuale che permette di applicare la gamma nei tre modi diversi, con un controllo per cambiare il valore del gamma da 0.1 a 3.0.

```
import time
import cv2
import numpy as np
from skimage import exposure
from PyQt6.QtCore import *
from PyQt6.QtWidgets import *
from PyQt6.QtGui import *

class SynchObject(QObject):
    synch_SIGNAL = pyqtSignal()

    def __init__(self, fps=60, parent=None): # Set FPS to 60
        super().__init__(parent)
        self.fps = fps
        self.syncTimer = QTimer(self)
        self.syncTimer.timeout.connect(self.sync)
        self.syncTimer.start(1000 // fps)
        self._initialized = True

    def sync(self):
        self.synch_SIGNAL.emit()

class BaseClass(QObject):
    def __init__(self, synchObject, resolution=QSize(1920, 1080)):
        super().__init__()
        self.synch_Object = synchObject
        self.resolution = resolution
        self._frame = np.zeros((resolution.height(), resolution.width(),
                               3), dtype=np.uint8)
        self.synch_Object.synch_SIGNAL.connect(self.captureFrame)
        self.start_time = time.time()
        self.frame_count = 0
        self.total_time = 0
        self.fps = 0
        self.last_update_time = time.time()
```

```
self.gamma_value = 2.2 # Default gamma value
self.gamma_method = self.apply_gamma_lut

def captureFrame(self):
    self.updateFps()
    self._frame = np.random.randint(0, 256, (self.resolution.height(),
                                              self.resolution.width(), 3), dtype=np.uint8)
    self._frame = self.gamma_method(self._frame, self.gamma_value)

def getFrame(self):
    return self._frame

def updateFps(self):
    self.frame_count += 1
    current_time = time.time()
    elapsed_time = current_time - self.last_update_time
    if elapsed_time >= 1.0: # Update FPS every second
        self.fps = self.frame_count / elapsed_time
        self.frame_count = 0
        self.last_update_time = current_time

def apply_gamma_lut(self, image, gamma=2.2):
    inv_gamma = 1.0 / gamma
    table = np.array([(i / 255.0) ** inv_gamma * 255
                      for i in range(256)]).astype(np.uint8)
    return cv2.LUT(image, table)

def apply_gamma_numpy(self, image, gamma=2.2):
    inv_gamma = 1.0 / gamma
    image = image / 255.0
    image = np.power(image, inv_gamma)
    return np.uint8(image * 255)

def apply_gamma_cv2(self, image, gamma=2.2):
    inv_gamma = 1.0 / gamma
    image = image / 255.0
    image = cv2.pow(image, inv_gamma)
    return np.uint8(image * 255)

def set_gamma_method(self, method_name):
    if method_name == "lut":
        self.gamma_method = self.apply_gamma_lut
    elif method_name == "numpy":
        self.gamma_method = self.apply_gamma_numpy
    elif method_name == "cv2":
        self.gamma_method = self.apply_gamma_cv2

def set_gamma_value(self, value):
    self.gamma_value = value / 100 # Convert from slider value to gamma
value
```

```

class VideoApp(QApplication):
    def __init__(self, argv):
        super().__init__(argv)
        self.synchObject = SynchObject(60) # Set FPS to 60
        self.input1 = BaseClass(self.synchObject)
        self.widget = QWidget()
        self.mainLayout = QVBoxLayout()
        self.viewer = QLabel()
        self.fpsLabel = QLabel()
        self.displayLabel = QLabel()
        self.mainLayout.addWidget(self.viewer)
        self.mainLayout.addWidget(self.fpsLabel)
        self.mainLayout.addWidget(self.displayLabel)
        self.widget.setLayout(self.mainLayout)
        self.widget.show()
        self.viewer.setFixedSize(1920, 1080)

        self.create_controls()

        self.uiTimer = QTimer(self)
        self.uiTimer.timeout.connect(self.display_frame)
        self.uiTimer.start(1000 // 30) # Update UI at 30 FPS

    def create_controls(self):
        control_layout = QHBoxLayout()

        # Create gamma value slider
        self.gamma_slider = QSlider(Qt.Orientation.Horizontal)
        self.gamma_slider.setRange(10, 300) # Range from 0.1 to 3.0 (scaled
by 100)
        self.gamma_slider.setValue(220) # Default value (2.2 scaled by 100)
        self.gamma_slider.valueChanged.connect(self.update_gamma_value)
        control_layout.addWidget(QLabel("Gamma:"))
        control_layout.addWidget(self.gamma_slider)

        # Create buttons for gamma methods
        lut_button = QPushButton("LUT")
        lut_button.clicked.connect(lambda:
            self.input1.set_gamma_method("lut"))
        numpy_button = QPushButton("NumPy")
        numpy_button.clicked.connect(lambda:
            self.input1.set_gamma_method("numpy"))
        cv2_button = QPushButton("OpenCV")
        cv2_button.clicked.connect(lambda:
            self.input1.set_gamma_method("cv2"))

        control_layout.addWidget(lut_button)
        control_layout.addWidget(numpy_button)

```

```
control_layout.addWidget(cv2_button)

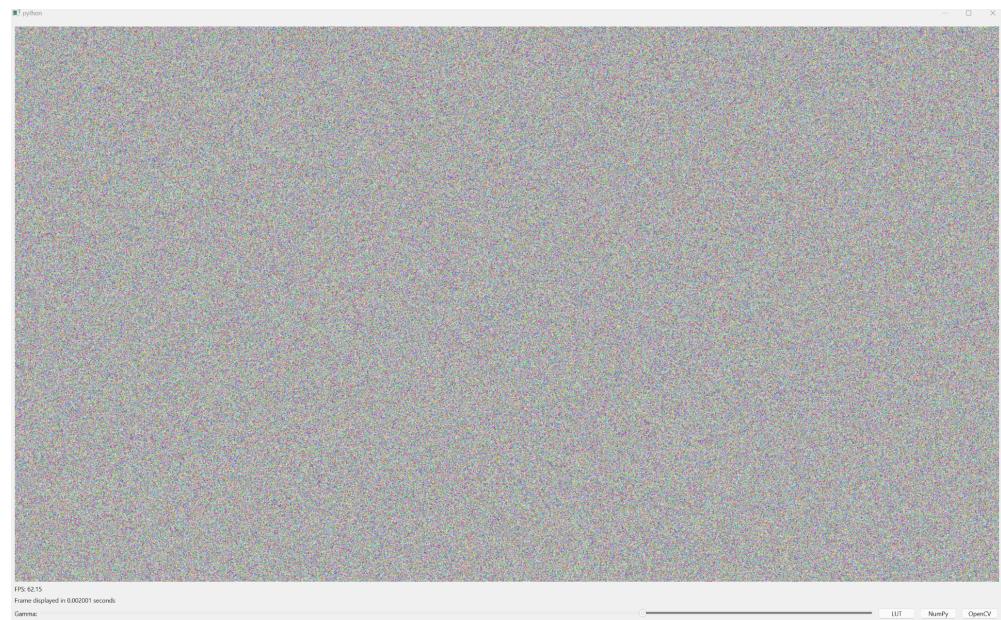
self.setLayout(control_layout)

def update_gamma_value(self):
    gamma_value = self.gamma_slider.value()
    self.input1.set_gamma_value(gamma_value)

def display_frame(self):
    frame = self.input1.getFrame()
    if frame is not None and frame.size != 0:
        start_time = time.time()
        image = QImage(frame.data, frame.shape[1], frame.shape[0],
                       QImage.Format.Format_BGR888)
        self.viewer.setPixmap(QPixmap.fromImage(image))
        display_time = time.time() - start_time
        self.displayLabel.setText(f"Frame displayed in
                                    {display_time:.6f} seconds")
        self.fpsLabel.setText(f"FPS: {self.input1.fps:.2f}")

def stop_app(self):
    print(f"Media FPS: {self.input1.fps:.2f}")
    self.exit()
```

Questa implementazione permette di cambiare il valore gamma e il metodo di correzione gamma in tempo reale. Utilizzando i pulsanti e la slider, è possibile vedere come ciascun metodo influenza il frame rate e la qualità dell'immagine.



```
# Example usage of the ColorGenerator class
if __name__ == "__main__":
    import sys

    def main():
        app = VideoApp(sys.argv)
        app.exec()

    if __name__ == '__main__':
        import cProfile
        import pstats
        import io

        pr = cProfile.Profile()
        pr.enable()
        main()
        pr.disable()
        s = io.StringIO()
        sortby = 'cumulative'
        ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
        ps.print_stats()
```

```
print(s.getvalue())
```

3.6 -CONTRAST

Non esiste una definizione unica di contrasto, ma possiamo generalizzare dicendo che rappresenta la differenza tra il valore massimo e minimo di un'immagine: minore è la differenza fra ombre e luci, maggiore è il dettaglio. Ci sono diverse formule per calcolare il valore del contrasto di un'immagine. Una delle più famose è:

$$C = \frac{L_1 + L_2}{L_1 - L_2}$$

Questa formula calcola un valore unico basato sulla differenza di luminanza tra due superfici, ed è utile per immagini semplici con due aree principali di luminanza.

Il contrasto di Michelson, invece, utilizza il livello massimo e il livello minimo campionati dall'immagine, ed è definito come:

$$C = \frac{L_{max} + L_{min}}{L_{max} - L_{min}}$$

Il contrasto di Weber è utilizzato principalmente per oggetti isolati su uno sfondo uniforme, come un isolotto in mezzo al mare, ed è calcolato come:

$$C = \frac{L_{target} - L_{background}}{L_{background}}$$

Un altro approccio è il contrasto RMS, che utilizza i valori medi per creare un rapporto di contrasto:

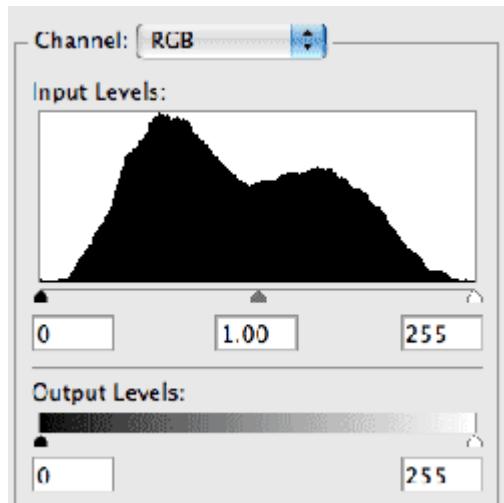
$$C = \sqrt{\frac{1}{n} \sum_{i=1}^n (L_i - \bar{L})^2}$$

Che sarebbe un po come fare una media di tutti i valori medi, mentre il sistema CIE definisce il contrasto in termini di differenza percepita di luminanza:

$$C = \frac{\Delta L}{L_{background}}$$

Nel capitolo 3 abbiamo introdotto due formule basate sul grafico di IO. La prima, fornita da Ron Brinkman, è $y = (x - 0.33) * 3$, che mostra un metodo manuale di contrasto simile a quello di Photoshop e abbiamo visto come in realtà teoricamente il metodo ideale per il contrasto non perde pixel ma agisce sui valori creando curve, ad esempio:

$$C = \frac{1}{1 + \exp(-10 * (x - 0.5))}$$



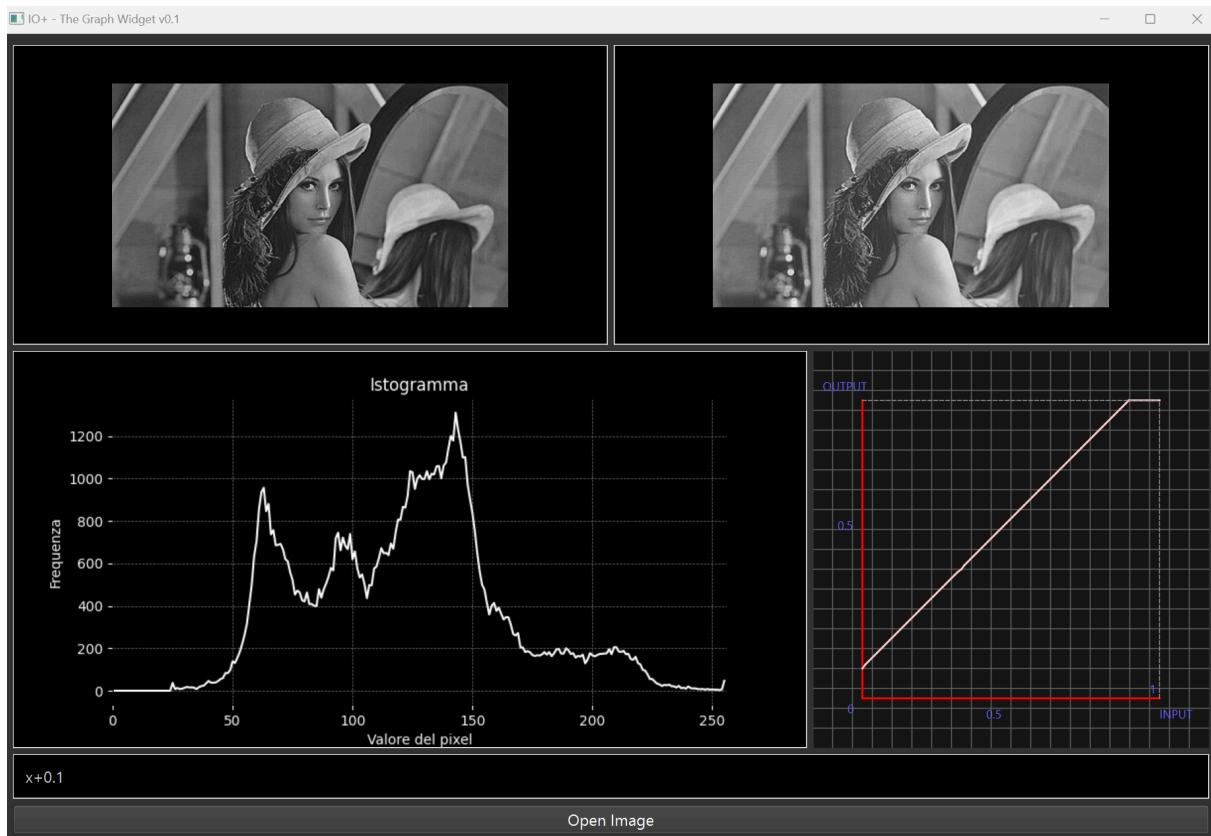
Le formule di Brinkman e la nostra formula sigmoide rappresentano due approcci diversi per aumentare il contrasto:

- **Brinkman:** Una manipolazione lineare e manuale, simile all'uso di livelli in Photoshop, utile per un controllo preciso e rapido del contrasto.
- **Formula sigmoide:** Un approccio non lineare che crea una curva di contrasto più naturale, ideale per immagini in cui è importante preservare i dettagli in tutte le aree di luminosità.

Entrambi gli approcci possono essere utili a seconda del tipo di immagine e dell'effetto desiderato. Questi metodi non sono esclusivi; in molti casi, combinare diverse tecniche può produrre i risultati migliori.

Ho scritto un'app in Python che utilizza una delle immagini di test più famose: Lenna. Questa app permette di applicare diverse formule di

contrasto e vedere il risultato in tempo reale. Potete scaricare l'immagine di test da [Wikipedia](#).



```
import sys
import numpy as np
import cv2
from PyQt6.QtWidgets import *
from PyQt6.QtGui import *
from PyQt6.QtCore import *
import matplotlib.pyplot as plt
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas

class ImageProcessor:
    @staticmethod
    def apply_expression(img, expression):
        x = img / 255.0 # Normalizzare l'immagine
        y = eval(expression, {"x": x, "np": np}) # Eval con contesto sicuro
```

```

y = np.clip(y, 0, 1)
return np.uint8(y * 255)

@staticmethod
def compute_histogram(img):
    hist, bins = np.histogram(img.flatten(), 256, [0, 256])
    fig, ax = plt.subplots(figsize=(8, 4))

    # Impostare uno sfondo scuro e una griglia
    fig.patch.set_facecolor('black')
    ax.set_facecolor('black')
    ax.grid(color='gray', linestyle='--', linewidth=0.5)

    ax.plot(hist, color='white')
    ax.set_xlim([0, 256])
    ax.set_xlabel("Valore del pixel", color='white')
    ax.set_ylabel("Frequenza", color='white')
    ax.set_title("Istogramma", color='white')

    ax.tick_params(colors='white')

    canvas = FigureCanvas(fig)
    canvas.draw()
    width, height = fig.get_size_inches() * fig.get_dpi()
    image = np.frombuffer(canvas.tostring_rgb(),
                          dtype='uint8').reshape(int(height), int(width), 3)
    plt.close(fig)

    return QImage(image.data, image.shape[1], image.shape[0],
QImage.Format_RGB888)
}

class GraphWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('IO+ - The Graph Widget v0.1')

        # Init widgets
        self.open_button = QPushButton('Open Image', self)
        self.input_box = QLineEdit(self)
        self.input_box.setPlaceholderText("Formula HERE: (es. x*1.2, x+0.4, (x-0.33)*3) where x is the image")
        self.graph_widget = GraphDrawingWidget()
        self.image_label = QLabel(self)
        self.hist_label = QLabel(self)
        self.orig_image_label = QLabel(self)

        self.image_path = None
        self.orig_img = None

```

```

    self.init_ui()
    self.init_connections()
    self.init_geometry()
    self.init_style()

def init_ui(self):
    main_layout = QVBoxLayout()
    upper_layout = QHBoxLayout()
    upper_layout.addWidget(self.orig_image_label)
    upper_layout.addWidget(self.image_label)
    lower_layout = QHBoxLayout()
    lower_layout.addWidget(self.hist_label)
    lower_layout.addWidget(self.graph_widget)
    main_layout.addLayout(upper_layout)
    main_layout.addLayout(lower_layout)
    main_layout.addWidget(self.input_box)
    main_layout.addWidget(self.open_button)
    self.setLayout(main_layout)

def init_connections(self):
    self.open_button.clicked.connect(self.on_open_image)
    self.input_box.textChanged.connect(self.on_update_graph)

def init_geometry(self):
    self.setGeometry(100, 100, 1200, 800)

def init_style(self):
    # Stile delle QLabel e QPushButton
    style = """
    QLabel {
        background-color: black;
        border: 1px solid #FFFFFF;
        min-height: 300px;
    }
    QPushButton {
        font-size: 16px;
    }
    QLineEdit {
        background-color: black;
        color: rgb(200, 200, 200);
        placeholder-text-color: rgb(250, 100, 100);
        selection-color: white;
        selection-background-color: red;
        border: 1px solid #FFFFFF;
        padding: 10px;
        font-size: 16px;
    }
"""
    self.setStyleSheet(style)

```

```

def on_open_image(self):
    options = QFileDialog.Option.ReadOnly
    file_name, _ = QFileDialog.getOpenFileName(self, 'Apri immagine', '',
'Image Files (*.png *.jpg *.bmp)', options=options)
    if file_name:
        self.image_path = file_name
        self.orig_img = cv2.imread(self.image_path, cv2.IMREAD_GRAYSCALE)
        if self.orig_img is None:
            raise ValueError("Immagine non valida")
        size = self.image_label.size()
        self.display_image(self.orig_img, self.orig_image_label)
        self.update_image_and_histogram(self.input_box.text())

def on_update_graph(self, text):
    if self.orig_img is not None:
        self.update_image_and_histogram(text)
    self.graph_widget.update_curve(text)

def update_image_and_histogram(self, text):
    try:
        if self.orig_img is None:
            raise ValueError("Immagine non valida")

        img = ImageProcessor.apply_expression(self.orig_img, text)
        hist_img = ImageProcessor.compute_histogram(img)

        self.hist_label.setPixmap(QPixmap.fromImage(hist_img))
        self.display_image(img, self.image_label)

    except Exception as e:
        print(f"Errore nell'aggiornamento dell'immagine e dell'istogramma: {e}")

    def display_image(self, img, label):
        q_img = QImage(img.data, img.shape[1], img.shape[0], img.strides[0],
QImage.Format.Format_Grayscale8)
        label.setPixmap(QPixmap.fromImage(q_img))
        # centra l'immagine
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)

class GraphDrawingWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.setFixedSize(400, 400)

        # Define colors
        self.gridColor = QColor(100, 100, 100)
        self.axisColor = QColor(255, 0, 0)
        self.lineColor = QColor(250, 200, 200)

```

```

self.dotLineColor = QColor(155, 155, 155)
self.textColor = QColor(100, 100, 255)
self.backColor = QColor(20, 20, 20)

self.expression = 'x'
self.curve = np.linspace(0, 1, 100)
self.update_curve(self.expression)

def update_curve(self, expression):
    self.expression = expression
    x = np.linspace(0, 1, 100)
    try:
        y = eval(self.expression, {"x": x, "np": np})
        self.curve = np.clip(y, 0, 1)
    except Exception as e:
        self.curve = x # If there's an error, revert to the identity
curve
    self.update()

def paintEvent(self, event):
    painter = QPainter(self)
    painter.setRenderHint(QPainter.RenderHint.Antialiasing)

    # Background
    painter.fillRect(self.rect(), self.backColor)

    # Draw the grid
    painter.setPen(QPen(self.gridColor, 1, Qt.PenStyle.SolidLine))
    for x in range(0, self.width(), 20):
        painter.drawLine(x, 0, x, self.height())
    for y in range(0, self.height(), 20):
        painter.drawLine(0, y, self.width(), y)

    # Draw the axes
    painter.setPen(QPen(self.axisColor, 2, Qt.PenStyle.SolidLine))
    painter.drawLine(50, self.height() - 50, self.width() - 50,
self.height() - 50) # X axis
    painter.drawLine(50, self.height() - 50, 50, 50) # Y axis

    # Draw labels and ticks
    painter.setPen(QPen(self.textColor, 6))
    painter.setFont(painter.font())
    painter.drawText(self.width() - 50, self.height() - 30, 'INPUT')
    painter.drawText(10, 40, 'OUTPUT')
    painter.drawText(35, self.height() - 35, '0')
    painter.drawText(self.width() - 60, self.height() - 55, '1')
    painter.drawText(25, (self.height() - 70) // 2 + 15, '0.5')
    painter.drawText((self.width() - 50) // 2, self.height() - 30, '0.5')

    # Draw the curve

```

```

    painter.setPen(QPen(self.lineColor, 2, Qt.PenStyle.SolidLine))
    for i in range(1, len(self.curve)):
        start_x = 50 + (self.width() - 100) * (i - 1) / (len(self.curve) - 1)
        end_x = 50 + (self.width() - 100) * i / (len(self.curve) - 1)
        start_y = self.height() - 50 - (self.height() - 100) * self.curve[i - 1]
        end_y = self.height() - 50 - (self.height() - 100) * self.curve[i]
        painter.drawLine(int(start_x), int(start_y), int(end_x), int(end_y))

    # Draw dashed lines
    pen = QPen(self.dotLineColor, 1, Qt.PenStyle.DashLine)
    painter.setPen(pen)
    painter.drawLine(50, 50, self.width() - 50, 50) # Line from (0,1) to (1,1)
    painter.drawLine(self.width() - 50, self.height() - 50, self.width() - 50, 50) # Line from (1,0) to (1,1)

painter.end()

def set_palette(app):
    app.setStyle("Fusion")
    dark_palette = QPalette()
    dark_palette.setColor(QPalette.ColorRole.Window, QColor(53, 53, 53))
    dark_palette.setColor(QPalette.ColorRole.WindowText, Qt.GlobalColor.white)
    dark_palette.setColor(QPalette.ColorRole.Base, QColor(42, 42, 42))
    dark_palette.setColor(QPalette.ColorRole.AlternateBase, QColor(66, 66, 66))
    dark_palette.setColor(QPalette.ColorRole.ToolTipBase, Qt.GlobalColor.white)
    dark_palette.setColor(QPalette.ColorRole.ToolTipText, Qt.GlobalColor.white)
    dark_palette.setColor(QPalette.ColorRole.Text, Qt.GlobalColor.white)
    dark_palette.setColor(QPalette.ColorRole.Button, QColor(53, 53, 53))
    dark_palette.setColor(QPalette.ColorRole.ButtonText, Qt.GlobalColor.white)
    dark_palette.setColor(QPalette.ColorRole.BrightText, Qt.GlobalColor.red)
    dark_palette.setColor(QPalette.ColorRole.Link, QColor(42, 130, 218))
    dark_palette.setColor(QPalette.ColorRole.Highlight, QColor(42, 130, 218))
    dark_palette.setColor(QPalette.ColorRole.HighlightedText, Qt.GlobalColor.white)
    dark_palette.setColor(QPalette.ColorGroup.Disabled, QPalette.ColorRole.WindowText, QColor(127, 127, 127))
    dark_palette.setColor(QPalette.ColorGroup.Disabled, QPalette.ColorRole.Text, QColor(127, 127, 127))
    dark_palette.setColor(QPalette.ColorGroup.Disabled, QPalette.ColorRole.ButtonText, QColor(127, 127, 127))
    dark_palette.setColor(QPalette.ColorGroup.Disabled, QPalette.ColorRole.Highlight, QColor(80, 80, 80))
    dark_palette.setColor(QPalette.ColorGroup.Disabled, QPalette.ColorRole.HighlightedText, QColor(127, 127, 127))
    app.setPalette(dark_palette)

def main():
    app = QApplication(sys.argv)
    set_palette(app)
    widget = GraphWidget()
    widget.show()
    sys.exit(app.exec())

if __name__ == '__main__':
    main()

```

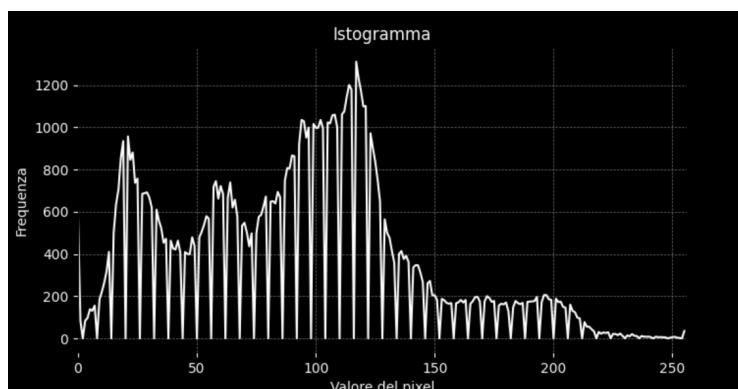
Quando nella casella di testo scriviamo espressioni come $x + 0.1$ o $x * 0.1$, succede quello che abbiamo descritto nel capitolo precedente. Qui, in più, possiamo visualizzare l'immagine e il suo istogramma. Un istogramma rappresenta la frequenza di ciascun valore di luminosità (o colore) presente nell'immagine, permettendo di visualizzare rapidamente la quantità di pixel con diverse intensità.

Per semplificare il calcolo, stiamo utilizzando un'immagine in bianco e nero. In questo modo, abbiamo un solo istogramma che rappresenta, sull'asse x, i valori di luminosità da 0 a 255 e, sull'asse y, la frequenza di ciascun tono di grigio.

In questo modo, possiamo vedere come sommando piccole quantità come $x+0.01x + 0.01x+0.01$, l'istogramma tenda a spostarsi verso destra, mentre moltiplicando per quantità piccole come 0.1 o 0.2, l'istogramma si ribilanci a sinistra. Ron Brinkman suggeriva $(x+0.33)\times 3$. Potete provare anche voi $(x-0.15)\times 1.3$, mentre questo sarà il risultato dell'altra nostra espressione.

Quello che può capitare, soprattutto con immagini a 8 bit, è che l'istogramma diventi a "pettine" che può indicare che ci sono problemi con i livelli di quantizzazione dell'immagine, spesso dovuti a operazioni di manipolazione che riducono la gamma dei valori disponibili, causando un raggruppamento dei pixel in pochi livelli di luminosità.

L'immagine che vediamo appare corretta, ma l'istogramma ci mostra questo problema che è spesso un segno che l'immagine ha perso delle informazioni e potrebbe avere una qualità visiva inferiore.



Questo non significa che non dobbiamo usare queste tecniche, ma che dobbiamo usarle con cautela.

3.7 -CONTRAST MORE

Abbiamo esplorato diverse tecniche per migliorare il contrasto delle immagini, partendo dalle basi teoriche fino a implementazioni pratiche con librerie come NumPy e OpenCV. Dopo aver discusso le formule di contrasto, abbiamo introdotto metodi automatici che possono essere applicati per migliorare rapidamente la qualità delle immagini.

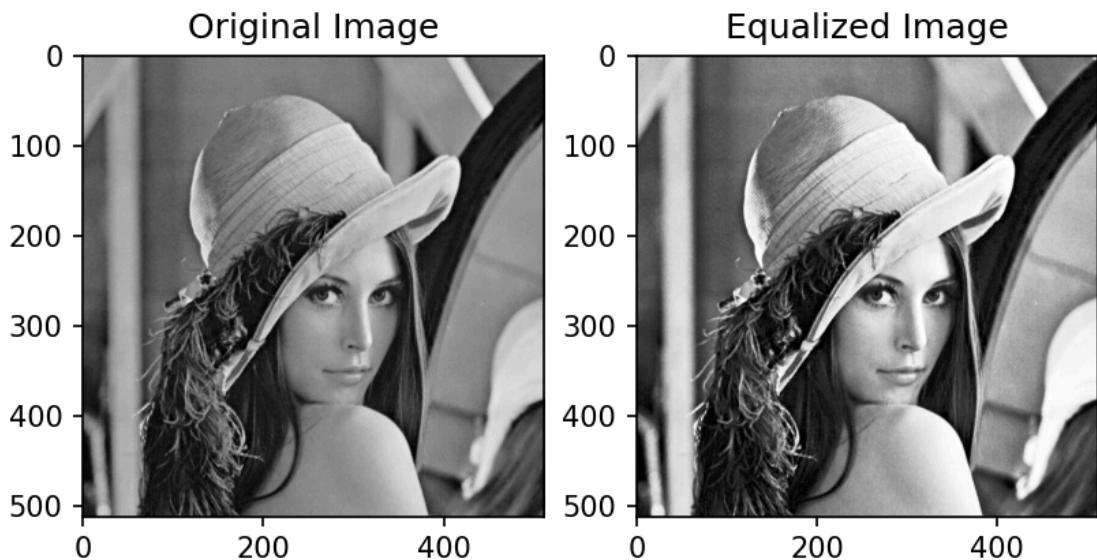
Un metodo efficace per migliorare il contrasto è l'equalizzazione dell'istogramma, che redistribuisce i valori di luminanza in modo uniforme su tutta la gamma disponibile. Questo metodo è particolarmente utile per immagini con contrasto ridotto, poiché enfatizza le differenze di luminanza esistenti.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Carica l'immagine in scala di grigi
img = cv2.imread(r'\Lenna_(test_image).png', 0)

# Equalizza l'istogramma
equ = cv2.equalizeHist(img)

# Visualizza l'immagine originale e quella equalizzata
plt.subplot(121), plt.imshow(img, cmap='gray'), plt.title('Original Image')
plt.subplot(122), plt.imshow(equ, cmap='gray'), plt.title('Equalized Image')
plt.show()
```



Il CLAHE, [contrast limited adaptive histogram equalization](#), è un'evoluzione dell'equalizzazione dell'istogramma è il CLAHE, che divide l'immagine in piccoli blocchi (tile) e equalizza ciascuno separatamente. Questo approccio limita l'amplificazione del rumore e migliora il contrasto locale, rendendolo utile per immagini con variazioni di contrasto significative.

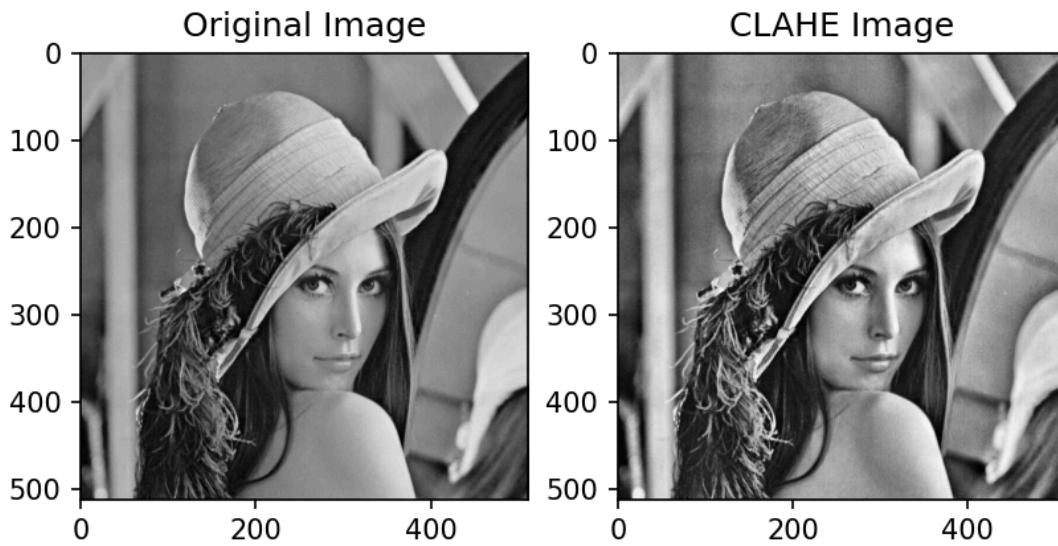
```
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Carica l'immagine in scala di grigi
img = cv2.imread(r'\Lenna_(test_image).png', 0)

clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))

# Applica CLAHE
clahe_img = clahe.apply(img)

# Visualizza l'immagine originale e quella con CLAHE
plt.subplot(121), plt.imshow(img, cmap='gray'), plt.title('Original Image')
plt.subplot(122), plt.imshow(clahe_img, cmap='gray'), plt.title('CLAHE Image')
plt.show()
```



Abbiamo visto come diverse tecniche di contrasto possano essere applicate sia manualmente che automaticamente. L'equalizzazione dell'istogramma e il CLAHE sono strumenti potenti per migliorare il contrasto delle immagini, rendendo dettagli più visibili e migliorando la qualità visiva generale, anche se, quando applicate direttamente ai canali RGB, possono alterare i colori originali, causando un'alterazione indesiderata dell'immagine.

Ho inserito come bonus del capitolo una versione migliorata del grafico di IO che include anche alcune tecniche di contrasto automatico.

3.8 -CONTRAST SPEED

Ovviamente, in soldoni vogliamo capire se le nostre operazioni di contrasto ci garantiscono di mantenere il frame rate e quale delle operazioni ci permette di ottenere il risultato che ci serve.

Possiamo provare a creare un metodo per il contrasto lineare simile a quello proposto da Brinkman che in alcuni casi può essere una soluzione rapida ed efficace, un metodo simile a quello che abbiamo creato usando la

sigmoide usando una gamma, visto che l'operazione come abbiamo visto è molto rapida e implementare anche i due metodi per il contrasto automatico usando histogram e clahe.

```
import cv2
import numpy as np
import timeit

# Metodo 1: Modifica Lineare del Contrasto
def adjust_contrast_linear(image, alpha=1.5, beta=0):
    return cv2.convertScaleAbs(image, alpha=alpha, beta=beta)

# Metodo 2: Regolazione Gamma
def adjust_contrast_gamma(image, gamma=1.0):
    inv_gamma = 1.0 / gamma
    table = np.array([(i / 255.0) ** inv_gamma * 255 for i in range(256)]).astype(np.uint8)
    return cv2.LUT(image, table)

def clahe(image, clip_limit=2.0, tile_grid_size=(8, 8)):
    channels = cv2.split(image)
    clahe = cv2.createCLAHE(clipLimit=clip_limit, tileSize=tile_grid_size)
    clahe_channels = [clahe.apply(channel) for channel in channels]
    return cv2.merge(clahe_channels)

def histogram_equalization(image):
    channels = cv2.split(image)
    eq_channels = [cv2.equalizeHist(channel) for channel in channels]
    return cv2.merge(eq_channels)

def add_text_to_image(image, text):
    font = cv2.FONT_HERSHEY_SIMPLEX
    font_scale = 1
    color = (255, 255, 255)
    thickness = 2
    position = (10, 50)
    return cv2.putText(image, text, position, font, font_scale, color,
thickness, cv2.LINE_AA)

# Genera un'immagine di test
image =
cv2.imread(r"\openPyVisionBook\openPyVisionBook\cap3\cap3_6\lena_std.tif")
```

```

# Crea immagini con testo
original_with_text = add_text_to_image(image.copy(), "Original")
linear_with_text = add_text_to_image(adjust_contrast_linear(image, alpha=1.5,
beta=20).copy(), "Linear Contrast")
gamma_with_text = add_text_to_image(adjust_contrast_gamma(image,
gamma=1.2).copy(), "Gamma Adjustment")
clahe_with_text = add_text_to_image(clahe(image).copy(), "CLAHE")
hist_eq_with_text = add_text_to_image(histogram_equalization(image).copy(),
"Histogram Equalization")

# Crea un mosaico di immagini su due righe
first_row = np.hstack((original_with_text, linear_with_text,
gamma_with_text))
second_row = np.hstack((original_with_text, clahe_with_text,
hist_eq_with_text))
big_image = np.vstack((first_row, second_row))

# Visualizza i risultati
cv2.imshow("Contrast Adjustment", big_image)
linear_test = timeit.timeit(lambda: adjust_contrast_linear(image, alpha=1.5,
beta=20), number=1000)
gamma_test = timeit.timeit(lambda: adjust_contrast_gamma(image, gamma=1.2),
number=1000)
clahe_test = timeit.timeit(lambda: clahe(image), number=1000)
hist_eq_test = timeit.timeit(lambda: histogram_equalization(image),
number=1000)

print(f"Linear Contrast Adjustment: {linear_test:.2f} seconds for 1000
iterations = {linear_test / 1000:.4f} ms per iteration")
print(f"Gamma Adjustment: {gamma_test:.2f} seconds for 1000 iterations =
{gamma_test / 1000:.4f} ms per iteration")
print(f"CLAHE: {clahe_test:.2f} seconds for 1000 iterations = {clahe_test / 1000:.4f} ms per iteration")
print(f"Histogram Equalization: {hist_eq_test:.2f} seconds for 1000
iterations = {hist_eq_test / 1000:.4f} ms per iteration")
cv2.waitKey(0)
cv2.destroyAllWindows()

```

Linear Contrast Adjustment: 0.09 seconds for 1000 iterations = 0.0001 ms per iteration

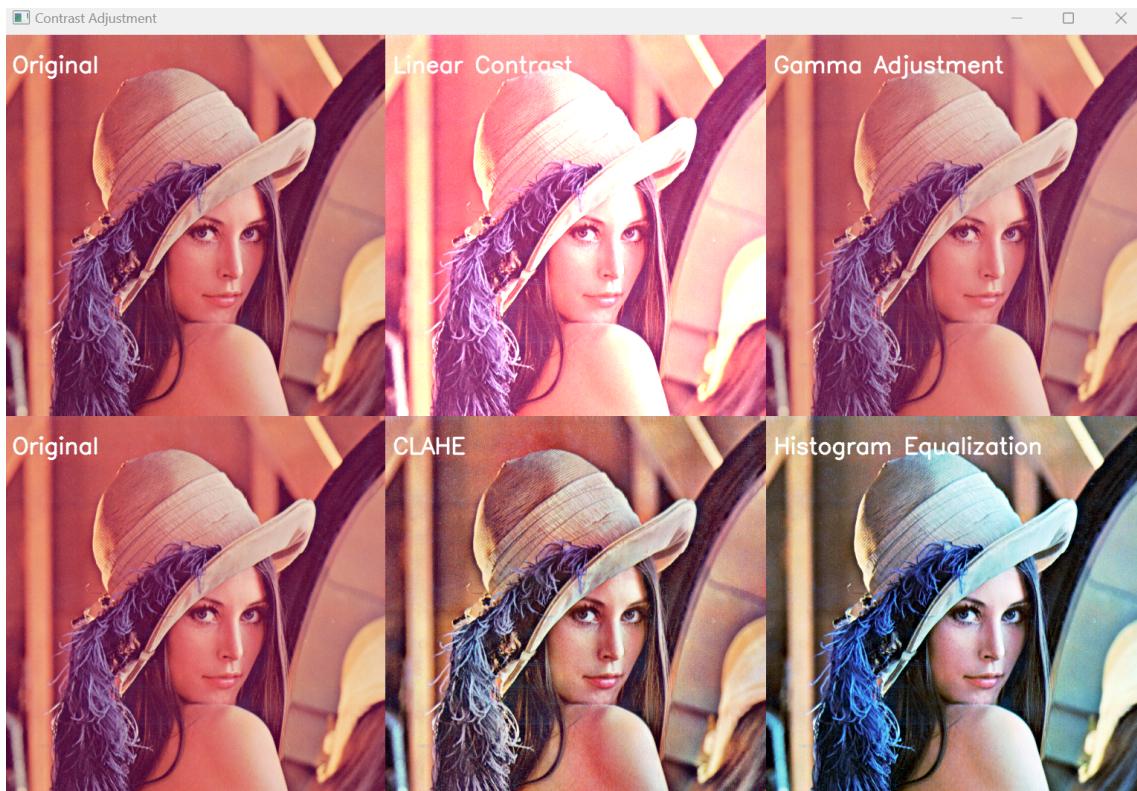
Gamma Adjustment: 0.13 seconds for 1000 iterations = 0.0001 ms per iteration

CLAHE: 0.88 seconds for 1000 iterations = 0.0009 ms per iteration

Histogram Equalization: 0.73 seconds for 1000 iterations = 0.0007 ms per iteration

La buona notizia è che sono tutte operazioni velocissime, quella un po meno buona che però non è poi così cattiva è che Clahe e histogram usati in questo modo tendono a far virare i colori e osservando i colori dell'immagine originale ci si accorge come in effetti l'immagine ha una

dominante rosso/magenta, però chiaramente potrebbe non essere l'operazione che vorremmo ottenere.



Per affrontare questo problema, possiamo applicare queste tecniche nel dominio YUV, che separa la luminanza (Y) dalle informazioni cromatiche (U e V). Questo approccio permette di migliorare il contrasto senza influire sui colori originali dell'immagine.

Il metodo `clahe_yuv` applica il CLAHE (Contrast Limited Adaptive Histogram Equalization) solo al canale Y (luminanza) dell'immagine convertita in YUV. Il CLAHE viene utilizzato per migliorare il contrasto locale senza amplificare eccessivamente il rumore, limitando il contrasto massimo per evitare artefatti visivi.

1. **Conversione a YUV:** L'immagine RGB viene convertita nello spazio colore YUV.

-
2. **CLAHE sul canale Y:** Il CLAHE viene applicato solo al canale Y, che contiene le informazioni di luminanza.
 3. **Riconversione a RGB:** L'immagine modificata viene riconvertita nello spazio colore RGB.

Questo metodo mantiene i colori più naturali rispetto all'applicazione del CLAHE direttamente sui canali RGB, poiché l'operazione di contrasto non influisce sui canali U e V, che contengono le informazioni cromatiche.

Analogamente, il metodo `histogram_equalization_yuv` applica l'equalizzazione dell'istogramma al canale Y di un'immagine YUV. Questo processo redistribuisce i valori di luminanza in modo uniforme su tutta la gamma disponibile, migliorando il contrasto globale dell'immagine.

1. **Conversione a YUV:** Come nel metodo CLAHE, l'immagine RGB viene prima convertita in YUV.
2. **Equalizzazione sul canale Y:** L'equalizzazione dell'istogramma viene applicata solo al canale Y, migliorando il contrasto senza influire sui colori.
3. **Riconversione a RGB:** L'immagine modificata viene quindi riconvertita nello spazio colore RGB.

```
import cv2
import numpy as np
import timeit

# Metodo 1: Modifica Lineare del Contrasto
def adjust_contrast_linear(image, alpha=1.5, beta=0):
    return cv2.convertScaleAbs(image, alpha=alpha, beta=beta)

# Metodo 2: Regolazione Gamma
def adjust_contrast_gamma(image, gamma=1.0):
    inv_gamma = 1.0 / gamma
    table = np.array([(i / 255.0) ** inv_gamma * 255
                      for i in range(256)]).astype(np.uint8)
    return cv2.LUT(image, table)

def clahe(image, clip_limit=2.0, tile_grid_size=(8, 8)):
    channels = cv2.split(image)
    clahe = cv2.createCLAHE(clipLimit=clip_limit,
                           tileGridSize=tile_grid_size)
```

```

clahe_channels = [clahe.apply(channel) for channel in channels]
return cv2.merge(clahe_channels)

def histogram_equalization(image):
    channels = cv2.split(image)
    eq_channels = [cv2.equalizeHist(channel) for channel in channels]
    return cv2.merge(eq_channels)

def clahe_yuv(image, clip_limit=2.0, tile_grid_size=(8, 8)):
    yuv_img = cv2.cvtColor(image, cv2.COLOR_BGR2YUV)
    clahe = cv2.createCLAHE(clipLimit=clip_limit,
                           tileGridSize=tile_grid_size)
    yuv_img[:, :, 0] = clahe.apply(yuv_img[:, :, 0])
    return cv2.cvtColor(yuv_img, cv2.COLOR_YUV2BGR)

def histogram_equalization_yuv(image):
    yuv_img = cv2.cvtColor(image, cv2.COLOR_BGR2YUV)
    yuv_img[:, :, 0] = cv2.equalizeHist(yuv_img[:, :, 0])
    return cv2.cvtColor(yuv_img, cv2.COLOR_YUV2BGR)

def add_text_to_image(image, text):
    font = cv2.FONT_HERSHEY_SIMPLEX
    font_scale = 1
    color = (255, 255, 255)
    thickness = 2
    position = (10, 50)
    return cv2.putText(image, text, position, font, font_scale, color,
thickness, cv2.LINE_AA)

# Genera un'immagine di test
image =
cv2.imread(r"\openPyVisionBook\openPyVisionBook\cap3\cap3_6\lena_std.tif")

# Crea immagini con testo
original_with_text = add_text_to_image(image.copy(), "Original")
linear_with_text = add_text_to_image(adjust_contrast_linear(image, alpha=1.5,
beta=20).copy(), "Linear Contrast")
gamma_with_text = add_text_to_image(adjust_contrast_gamma(image,
gamma=1.2).copy(), "Gamma Adjustment")
clahe_with_text = add_text_to_image(clahe(image).copy(), "CLAHE")
hist_eq_with_text = add_text_to_image(histogram_equalization(image).copy(),
"Histogram Equalization")
clahe_yuv_with_text = add_text_to_image(clahe_yuv(image).copy(), "CLAHE YUV")
hist_eq_yuv_with_text =
add_text_to_image(histogram_equalization_yuv(image).copy(), "Histogram YUV")

```

```

# Crea un mosaico di immagini su due righe
first_row = np.hstack((original_with_text, linear_with_text, clahe_with_text,
hist_eq_with_text))
second_row = np.hstack((original_with_text, gamma_with_text,
clahe_yuv_with_text, hist_eq_yuv_with_text))
big_image = np.vstack((first_row, second_row))

# Visualizza i risultati
cv2.imshow("Contrast Adjustment", big_image)

# Test delle prestazioni
linear_test = timeit.timeit(lambda: adjust_contrast_linear(image, alpha=1.5,
beta=20), number=1000)
gamma_test = timeit.timeit(lambda: adjust_contrast_gamma(image, gamma=1.2),
number=1000)
clahe_test = timeit.timeit(lambda: clahe(image), number=1000)
hist_eq_test = timeit.timeit(lambda: histogram_equalization(image),
number=1000)
clahe_yuv_test = timeit.timeit(lambda: clahe_yuv(image), number=1000)
hist_eq_yuv_test = timeit.timeit(lambda: histogram_equalization_yuv(image),
number=1000)

cv2.waitKey(0)
cv2.destroyAllWindows()

```

Linear Contrast Adjustment: 0.09 seconds for 1000 iterations = 0.0001 ms per iteration

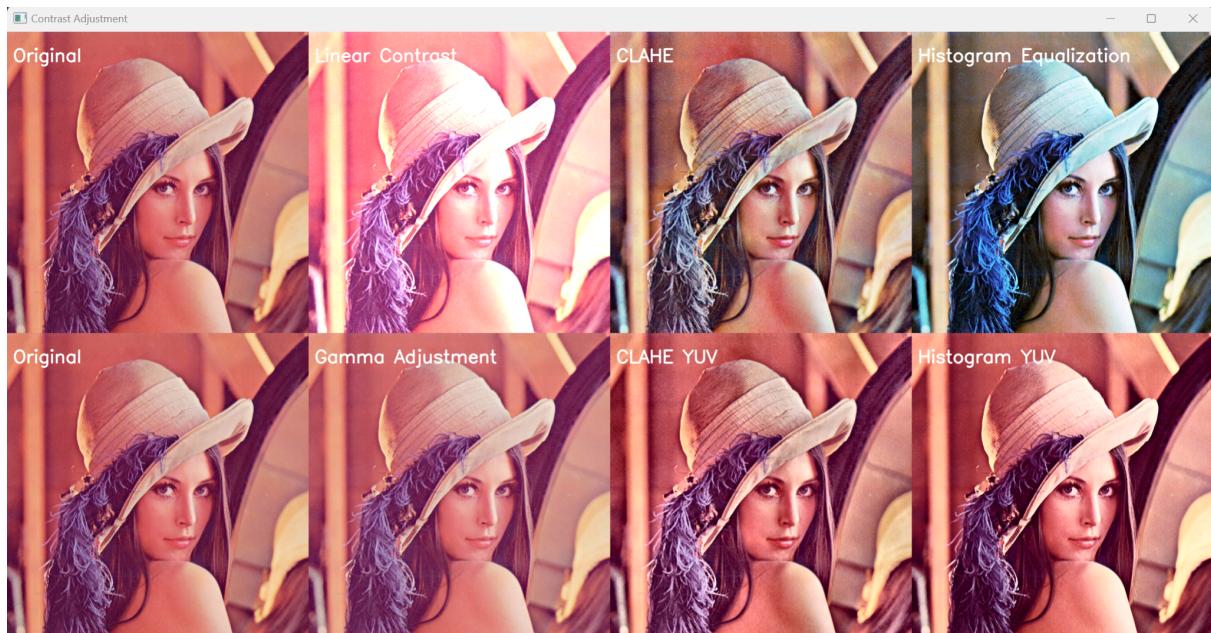
Gamma Adjustment: 0.11 seconds for 1000 iterations = 0.0001 ms per iteration

CLAHE: 0.93 seconds for 1000 iterations = 0.0009 ms per iteration

Histogram Equalization: 0.56 seconds for 1000 iterations = 0.0006 ms per iteration

CLAHE YUV: 0.56 seconds for 1000 iterations = 0.0006 ms per iteration

Histogram YUV: 0.37 seconds for 1000 iterations = 0.0004 ms per iteration



3.9 -BASE CLASS EXTENDED

Il passo successivo è aggiungere queste operazioni fondamentali che abbiamo visto alla nostra classe base.

Per il momento creiamo delle variabili che possono essere modificate per attivare o meno un certo effetto:

```
clip_limit = 2.0
tile_grid_size = (8, 8)
gamma = 1.0
isFrameInverted = False
isFrameAutoScreen = False
isFrameCLAHE = False
isFrameHistogramEqualization = False
isFrameCLAHEYUV = False
isFrameHistogramEqualizationYUV = False
```

A questo punto possiamo aggiungere i vari metodi in coda al codice:

```
@staticmethod
def invertFrame(image):
    """
    Inverts the frame colors.
    """
    return cv2.bitwise_not(image)

@staticmethod
def autoScreenFrame(image):
    """
    Automatically creates a screen frame.
    """
    inv1 = cv2.bitwise_not(image)
    mult = cv2.multiply(inv1, inv1, scale=1.0 / 255.0)
    return cv2.bitwise_not(mult).astype(np.uint8)

@staticmethod
def getRGBChannels(frame):
    """
    Returns the RGB channels of a frame.
    """
    return cv2.split(frame)

@staticmethod
def setRGBChannels(channels):
    """
    Sets the RGB channels of a frame.
    """
    return cv2.merge(channels)

@staticmethod
def applyGammaByLut(image, gamma):
    inv_gamma = 1.0 / gamma
    table = np.array([(i / 255.0) ** inv_gamma * 255
                      for i in range(256)]).astype(np.uint8)
    return cv2.LUT(image, table)

@staticmethod
def applyCLAHE(image, clip_limit=2.0, tile_grid_size=(8, 8)):
    """
    Applies the Contrast Limited Adaptive Histogram Equalization (CLAHE) to
    the image.
    """
    clahe = cv2.createCLAHE(clipLimit=clip_limit, tileSize=tile_grid_size)
    return clahe.apply(image)

@staticmethod
def applyHistogramEqualization(image):
    """
    Applies the Histogram Equalization to the image.
    """
```

```

"""
    return cv2.equalizeHist(image)

@staticmethod
def applyCLAHEYUV(image, clip_limit=2.0, tile_grid_size=(8, 8)):
    """
        Applies the Contrast Limited Adaptive Histogram Equalization (CLAHE) to
        the Y channel of the YUV image.
    """
    yuv_img = cv2.cvtColor(image, cv2.COLOR_BGR2YUV)
    clahe = cv2.createCLAHE(clipLimit=clip_limit, tileGridSize=tile_grid_size)
    yuv_img[:, :, 0] = clahe.apply(yuv_img[:, :, 0])
    return cv2.cvtColor(yuv_img, cv2.COLOR_YUV2BGR)

@staticmethod
def applyHistogramEqualizationYUV(image):
    """
        Applies the Histogram Equalization to the Y channel of the YUV image.
    """
    yuv_img = cv2.cvtColor(image, cv2.COLOR_BGR2YUV)
    yuv_img[:, :, 0] = cv2.equalizeHist(yuv_img[:, :, 0])
    return cv2.cvtColor(yuv_img, cv2.COLOR_YUV2BGR)

```

Ora quello che rimane da fare è stabilire nel getframe se è stata attivata una delle funzioni, di restituire il frame modificato.

```

def getFrame(self):
    if self.isFrameInverted:
        self._frame = self.invertFrame(self._frame)
    if self.isFrameAutoScreen:
        self._frame = self.autoScreenFrame(self._frame)
    if self.gamma != 1.0:
        self._frame = self.applyGammaByLut(self._frame, self.gamma)
    if self.isFrameCLAHE:
        self._frame = self.applyCLAHE(self._frame)
    if self.isFrameHistogramEqualization:
        self._frame = self.applyHistogramEqualization(self._frame)
    if self.isFrameCLAHEYUV:
        self._frame = self.applyCLAHEYUV(self._frame)
    if self.isFrameHistogramEqualizationYUV:
        self._frame =
            self.applyHistogramEqualizationYUV(self._frame)
    return self._frame

```

L'ordine degli effetti è stato scelto considerando che:

-
1. Gli effetti che modificano i colori (come l'inversione o lo screen) devono essere applicati prima delle correzioni di contrasto.
 2. Gli effetti di correzione del contrasto (CLAHE, equalizzazione dell'istogramma) devono essere applicati per ultimi, per ottimizzare la qualità visiva.

Le operazioni avvengono nella funzione `getFrame` perchè di solito non viene modificata nella classe figlia.

Questa architettura ti permette di mantenere la classe base pulita e focalizzata, mentre le classi figlie possono concentrarsi sul compito specifico di generare il frame, sapendo che gli effetti verranno applicati correttamente al momento del rendering.

4.0 - Mix Video



Il modo più semplice per mixare due immagini insieme è quello di sommarle. Suona folle, ma questo è matematicamente quello che si faceva anche in pellicola e il risultato un tempo si chiamava “double exposure”. Se il soggetto viene ripreso su uno sfondo nero, sommandolo a un'altra immagine si crea un effetto fantasma. Georges Méliès ha creato centinaia di film usando questa tecnica.

```
out_image = clamp(input1 + input2)
```

Questo approccio è una trasposizione diretta del metodo analogico utilizzato in pellicola, dove le esposizioni multiple venivano sommate per creare un'unica immagine finale. In termini pratici, se il soggetto viene ripreso su uno sfondo nero e poi sommato a un'altra immagine, si ottiene un effetto fantasma. La semplicità di questa operazione rende facile comprendere come le immagini possono essere combinate, ma introduce anche limitazioni come il clamping dei valori.

Nei programmi di grafica, le operazioni matematiche sui pixel sono spesso denominate “Blending Modes”. In Photoshop, l'operazione “Add” è conosciuta come “Color Dodge”. Altri modi di fusione includono “Screen” e “Lighten”, che permettono di ottenere effetti simili con diverse operazioni

matematiche che servono a contenere i fenomeni di clamping. Potete approfondire la matematica dietro ai blending modes su pegttop.net, dove Jens Gruschel mantiene un blog che descrive la matematica usata dai motori grafici (<https://www.pegttop.net/delphi/articles/blendmodes/>).

4.1 - MixBus

Abbiamo introdotto la doppia esposizione usata da Méliès a cavallo fra la fine dell'800 e l'inizio del 900. La sua tecnica era quella di riprendere uno sfondo dove, ad esempio, c'era una parte nera e quindi, tornando indietro con la pellicola, la parte completamente nera continuava ad essere vergine, e poteva essere nuovamente esposta. Questo trucco, conosciuto come double exposure, è stato usato molto più ampiamente fino all'avvento degli effetti speciali digitali.

In digitale, la tecnica si basa sulla gestione della trasparenza di un'immagine, dicendo che il nero è completamente trasparente e il bianco è completamente opaco, mentre il grigio medio (0.5) è semitrasparente. La formula matematica per ottenere questo, come ci ricorda Szelinski della Washington University (<http://szeliski.org/Book/>), è:

$$I_{out} = \alpha_1 * I_1 + (1 - \alpha_1) * I_2$$

dove α_1 e α_2 sono due valori da 0 a 1 che indicano la trasparenza delle immagini. Nel nostro caso, quando facciamo un mix, vogliamo che una delle due immagini diventi sempre più trasparente e l'altra sempre più opaca. Quindi, avranno un α comune e la formula diventa:

$$I_{out} = \alpha * I_1 + (1 - \alpha) * I_2$$

Il nostro α ovviamente è variabile, quindi tirando su la leva del mix lo portiamo da 0 a 1 e tirandola giù facciamo il viceversa. Nei mixer c'è anche un pulsante per fare il mix automatico, quindi la logica dell'operazione si complica.

Per ottenere un mix senza preoccuparsi che venga usato lo slider o il pulsante, si può assumere che ci sia sempre un'immagine live chiamata

"program" e che la variabile "fade" possa andare in una sola direzione da 0 a 1, in questo modo quando facciamo il mix, con la slider o con il pulsante, spostiamo sempre la variabile da 0 a 1, cambiando l'opacità della "preview".

Quello che dobbiamo fare è creare un MixBus, ovvero un canale video che contiene sempre due immagini: una sempre completamente trasparente (la preview) e una sempre completamente opaca (il program). Se muoviamo lo slider, aumentiamo l'opacità della preview e diminuiamo quella del program. Quando arriviamo a 1, facciamo il cut, invertiamo i segnali e azzeriamo la variabile fade.

Per fare la somma pesata delle due immagini in OpenCV si può usare:

```
cv2.addWeighted(input1, alpha1, input2, alpha2, gamma)
```

Posso quindi creare un file mixBus.py

```
class mixBus(QObject):
    _fade = 0
    FadeTime = 100

    def __init__(self, input1, input2, parent=None):
        super().__init__(parent)
        self.previewInput = input1
        self.programInput = input2
        self.autoMix_timer = QTimer(self)
        self.autoMix_timer.timeout.connect(self._fadeTo)

    def getMixed(self):
        prw_frame = self.previewInput.getFrame()
        prg_frame = self.programInput.getFrame()
        if self._fade == 0:
            return prw_frame, prg_frame
        else:
            return prw_frame, cv2.addWeighted(prw_frame, self._fade, prg_frame, 1 - self._fade, 0)

    def cut(self):
        self._fade = 0
        self.previewInput, self.programInput = self.programInput, self.previewInput

    def autoMix(self):
        self.autoMix_timer.start(self.fadeTime)

    def _fadeTo(self):
```

```

        self._fade += 0.1
        if self._fade > 1:
            self.autoMix_timer.stop()
            self.cut()

def fadeTo(self, value: int):
    if value == 0:
        self._fade = 0
    else:
        self._fade = value/100

```

Con questa classe possiamo, una volta passati i riferimenti di due input, fare il mix automatico e manuale, e il cut. Per farlo, creiamo un widget con due label, una per il preview e una per il program, inseriamo un pulsante per il cut, uno per l'autoMix e una slider per il mix manuale. Usiamo come input1 un generatore di colore e come input2 un generatore di rumore.

```

class testMixBus(QWidget):
    def __init__(self, syncObject, parent=None):
        super().__init__(parent)
        self.syncObject = syncObject
        self.input1 = ColorGenerator(self.syncObject)
        self.input2 = RandomNoiseGenerator(self.syncObject)
        self.mixBus = mixBus(self.input1, self.input2)
        self.lblPreview = QLabel()
        self.lblProgram = QLabel()
        self.btnCut = QPushButton("CUT")
        self.btnAutoMix = QPushButton("AutoMix")
        self.sldFade = QSlider()
        self.sldFade.setOrientation(Qt.Orientation.Horizontal)
        self.sldFade.setRange(0, 100)
        self.initUI()
        self.setGeometry()
        self.initConnections()

    def initUI(self):
        mainLayout = QVBoxLayout()
        viewerLayout = QHBoxLayout()
        viewerLayout.addWidget(self.lblPreview)
        viewerLayout.addWidget(self.lblProgram)
        spacer = QSpacerItem(20, 40,
                            QSizePolicy.Policy.Minimum, QSizePolicy.Policy.Expanding)
        buttonLayout = QHBoxLayout()
        buttonLayout.addItem(spacer)
        buttonLayout.addWidget(self.btnCut)

```

```
buttonLayout.addWidget(self.btnAutoMix)
buttonLayout.addWidget(self.sldFade)
mainLayout.addLayout(viewerLayout)
mainLayout.addLayout(buttonLayout)
self.setLayout(mainLayout)

def initGeometry(self):
    self.lblPreview.setFixedSize(640, 360)
    self.lblProgram.setFixedSize(640, 360)

def initConnections(self):
    self.syncObject.synch_SIGNAL.connect(self.updateFrame)
    self.btnCut.clicked.connect(self.cut)
    self.btnAutoMix.clicked.connect(self.autoMix)
    self.sldFade.valueChanged.connect(self.setFade)

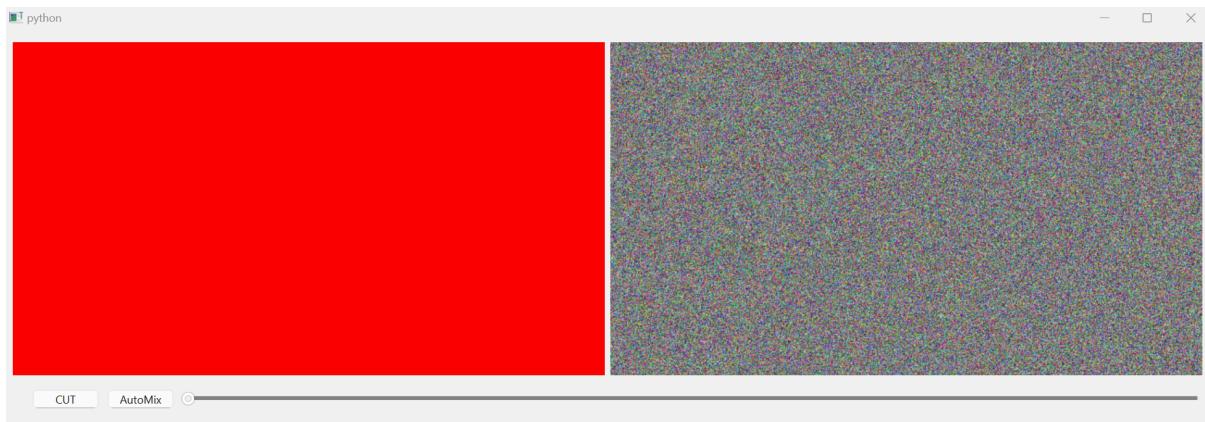
def updateFrame(self):
    prw_frame, prg_frame = self.mixBus.getMixed()
    prw_frame = cv2.resize(prw_frame, (640, 360))
    prg_frame = cv2.resize(prg_frame, (640, 360))
    prw_image = QImage(prw_frame.data, prw_frame.shape[1], prw_frame.shape[0],
QImage.Format.Format_BGR888)
    prg_image = QImage(prg_frame.data, prg_frame.shape[1], prg_frame.shape[0],
QImage.Format.Format_BGR888)
    self.lblPreview.setPixmap(QPixmap.fromImage(prw_image))
    self.lblProgram.setPixmap(QPixmap.fromImage(prg_image))

def cut(self):
    self.mixBus.cut()

def autoMix(self):
    self.mixBus.autoMix()

def setFade(self):
    self.mixBus.setFade(self.sldFade.value())

if __name__ == '__main__':
    import sys
    app = QApplication(sys.argv)
    syncObject = SyncObject()
    test = testMixBus(syncObject)
    test.show()
    sys.exit(app.exec())
```



Con questa implementazione, possiamo visualizzare in tempo reale come funzionano i diversi metodi di fusione delle immagini, utilizzando sia il mix manuale che automatico.

5.2 - Wipe Left To Right

Nel capitolo precedente, abbiamo visto come utilizzare NumPy per fare lo slicing di una matrice. Possiamo considerare una transizione "wipe" in modo simile a uno slicing. Durante la transizione, l'immagine risultante è composta da una parte dell'immagine di anteprima (preview) e una parte dell'immagine in programma (program).

Quando la variabile `_fade` è a 0, vediamo completamente l'immagine di programma. Quando `_fade` è a 0.5, vediamo metà schermo con l'immagine di anteprima e l'altra metà con l'immagine di programma. Infine, quando `_fade` raggiunge 1, vediamo solo l'immagine di anteprima. Come per la transizione "mix", riportiamo la variabile `_fade` a 0, fermiamo il timer e eseguiamo il "cut".

Sperimentalmente, mi sono accorto che, nonostante l'operazione venga fatta correttamente e in tempo, in alcuni punti l'animazione risulta andare a scatti. A mio avviso, il problema è dovuto alla creazione al volo di una `wipePosition`. Per risolvere questo fenomeno, ho calcolato nell'init una lista di posizioni.

La transizione "wipe" utilizza una lista di posizioni pre-calcolate per garantire un movimento fluido e consistente. Ad ogni passo del timer, aggiorniamo la posizione del "wipe" utilizzando uno slicing di NumPy per combinare le due immagini in base alla posizione corrente.

Ecco come viene implementato il "wipe" da sinistra a destra:

- **Inizializzazione delle posizioni del wipe:** Viene creata una lista di posizioni utilizzando `np.linspace`, che divide la larghezza dell'immagine (1920 pixel) in un numero di passi determinato dal tempo del wipe (`_wipeTime`).
- **Aggiornamento della posizione:** Ad ogni intervallo del timer, la posizione del wipe viene aggiornata incrementando un contatore (`_wipe`). La posizione corrente viene poi utilizzata per combinare le immagini di anteprima e programma.
- **Combina le immagini:** Utilizzando lo slicing di NumPy, combiniamo le due immagini in base alla posizione corrente del wipe.

Questa implementazione garantisce che il movimento del wipe sia fluido e che l'immagine risultante sia una combinazione dinamica delle due immagini, basata sulla posizione corrente del wipe.

Ecco un esempio di codice che mostra come implementare la transizione "wipe":

```
def wipeLeftToRight(self, preview_frame, program_frame):  
    wipe_position = int(self._wipe_position_list[self._wipe])  
    wipe_frame = program_frame.copy()  
    wipe_frame[:, :wipe_position] = preview_frame[:, :wipe_position]  
    return wipe_frame
```

Il codice completo diventa:

```
import time  
from enum import Enum  
  
import cv2  
import numpy as np  
from PyQt6.QtCore import *  
from PyQt6.QtGui import QImage, QPixmap
```

```

from PyQt6.QtWidgets import *

from cap5.cap5_4.colorGenerator import ColorGenerator
from cap5.cap5_4.randomNoiseGenerator import RandomNoiseGenerator
from cap5.cap5_4.synchObject import SynchObject


class MIX_TYPE(Enum):
    FADE = 0
    WIPE_LEFT_TO_RIGHT = 1
    WIPE_RIGHT_TO_LEFT = 2
    WIPE_TOP_TO_BOTTOM = 3
    WIPE_BOTTOM_TO_TOP = 4


class mixBus5_6(QObject):
    _fade = 0
    _fadeTime = 100
    _wipe = 0
    _wipeTime = 90
    effectType = MIX_TYPE.WIPE_LEFT_TO_RIGHT

    def __init__(self, input1, input2, parent=None):
        super().__init__(parent)
        self.previewInput = input1
        self.programInput = input2
        self.autoMix_timer = QTimer(self)
        self.autoMix_timer.timeout.connect(self._fader)
        self._init_wipe_positions()

    def _init_wipe_positions(self):
        _wipe_step = max(1, self._wipeTime)
        self._wipe_position_list = np.linspace(0, 1920, _wipe_step)

    def getMixed(self):
        prw_frame = self.previewInput.getFrame()
        prg_frame = self.programInput.getFrame()
        if self._fade == 0:
            return prw_frame, prg_frame
        else:
            if self.effectType == MIX_TYPE.FADE:
                return prw_frame, cv2.addWeighted(prw_frame, self._fade,
                                                prg_frame, 1 - self._fade, 0)
            elif self.effectType == MIX_TYPE.WIPE_LEFT_TO_RIGHT:
                return prw_frame, self.wipeLeftToRight(prw_frame, prg_frame)

    def setFade(self, value: int):
        if value == 0:
            self._fade = 0
        else:

```

```

        self._fade = value / 100

def cut(self):
    self._fade = 0
    self._wipe = 0
    self.previewInput, self.programInput = self.programInput, self.previewInput

def autoMix(self):
    self.autoMix_timer.start(1000//60)

def _fader(self):
    if self.effectType == MIX_TYPE.FADE:
        self._fade += 0.01
        if self._fade > 1:
            self.autoMix_timer.stop()
            self.cut()
    elif self.effectType == MIX_TYPE.WIPE_LEFT_TO_RIGHT:
        self._wipe += 1
        self._fade += 0.01
        if self._wipe > len(self._wipe_position_list)-1:
            self.autoMix_timer.stop()
            self.cut()

def wipeLeftToRight(self, preview_frame, program_frame):
    wipe_position = int(self._wipe_position_list[self._wipe])
    wipe_frame = program_frame.copy()
    wipe_frame[:, :wipe_position] = preview_frame[:, :wipe_position]
    return wipe_frame

```

4.3 Wipe Right To Left

Per invertire il wipe in modo che vada da destra a sinistra, possiamo creare una lista di posizioni con `np.linspace` che vanno da 1920 a 0. Questo ci permette di avere una serie di posizioni decrescenti, iniziando dal lato destro dell'immagine e procedendo verso il lato sinistro.

Durante il wipe, combiniamo le immagini di anteprima e programma utilizzando lo slicing di NumPy. Invece di usare:

```
wipe_frame[:, :wipe_position] = preview_frame[:, :wipe_position]
```

usiamo:

```
wipe_frame[:, wipe_position:] = preview_frame[:, wipe_position:]
```

Questo perché vogliamo che l'immagine di anteprima (preview) appaia gradualmente a partire dal lato destro. Con questa logica, i pixel a destra della posizione corrente (**wipe_position**) vengono sostituiti dai pixel dell'immagine di anteprima, creando un effetto di wipe inverso.

```
class MIX_TYPE(Enum):
    FADE = 0
    WIPE_LEFT_TO_RIGHT = 1
    WIPE_RIGHT_TO_LEFT = 2
    WIPE_TOP_TO_BOTTOM = 3
    WIPE_BOTTOM_TO_TOP = 4

class MixBus4_3(QObject):
    _fade = 0
    _fadeTime = 100
    _wipe = 0
    _wipeTime = 90
    effectType = MIX_TYPE.WIPE_LEFT_TO_RIGHT

    def __init__(self, input1, input2, parent=None):
        super().__init__(parent)
        self.previewInput = input1
        self.programInput = input2
        self.autoMix_timer = QTimer(self)
        self.autoMix_timer.timeout.connect(self._fader)
        self._init_wipe_positions()

    def _init_wipe_positions(self):
        _wipe_step = max(1, self._wipeTime)
        self._wipe_position_leftToRight_list = np.linspace(0, 1920, _wipe_step)
        self._wipe_position_rightToLeft_list = np.linspace(1920, 0, _wipe_step)
        self._wipe_position_topToBottom_list = np.linspace(0, 1080, _wipe_step)
        self._wipe_position_bottomToTop_list = np.linspace(1080, 0, _wipe_step)

    def getMixed(self):
        prw_frame = self.previewInput.getFrame()
        prg_frame = self.programInput.getFrame()
        if self._fade == 0:
            return prw_frame, prg_frame
        else:
            if self.effectType == MIX_TYPE.FADE:
                return prw_frame, cv2.addWeighted(prw_frame, self._fade,
                                                prg_frame, 1 - self._fade, 0)
            elif self.effectType == MIX_TYPE.WIPE_LEFT_TO_RIGHT:
                return prw_frame, self.wipeLeftToRight(prw_frame, prg_frame)
            elif self.effectType == MIX_TYPE.WIPE_RIGHT_TO_LEFT:
                return prw_frame, self.wipeRightToLeft(prw_frame, prg_frame)
```

```
def setFade(self, value: int):
    if value == 0:
        self._fade = 0
    else:
        self._fade = value / 100
        self._wipe = value

def cut(self):
    self._fade = 0
    self._wipe = 0
    self.previewInput, self.programInput = self.programInput, self.previewInput

def autoMix(self):
    self.autoMix_timer.start(1000 // 60)

def _fader(self):
    if self.effectType == MIX_TYPE.FADE:
        self._fade += 0.01
        if self._fade > 1:
            self.autoMix_timer.stop()
            self.cut()
    elif self.effectType in [MIX_TYPE.WIPE_LEFT_TO_RIGHT,
                           MIX_TYPE.WIPE_RIGHT_TO_LEFT]:
        self._wipe += 1
        self._fade += 0.01
        if self._wipe > len(self._wipe_position_leftToRight_list) - 1:
            self.autoMix_timer.stop()
            self.cut()

def wipeLeftToRight(self, preview_frame, program_frame):
    wipe_position = int(self._wipe_position_leftToRight_list[self._wipe])
    wipe_frame = program_frame.copy()
    wipe_frame[:, :wipe_position] = preview_frame[:, :wipe_position]
    return wipe_frame

def wipeRightToLeft(self, preview_frame, program_frame):
    wipe_position = int(self._wipe_position_rightToLeft_list[self._wipe])
    wipe_frame = program_frame.copy()
    wipe_frame[:, wipe_position:] = preview_frame[:, wipe_position:]
    return wipe_frame
```

4.4 Wipe Up And Down

Per implementare il wipe in verticale, sia dall'alto verso il basso che dal basso verso l'alto, utilizziamo una logica simile a quella del wipe orizzontale, ma con le righe dell'immagine anziché le colonne.

Per il wipe dall'alto verso il basso, creiamo una lista di posizioni con `np.linspace` che va da 0 a 1080 (l'altezza dell'immagine) e come visto ne creiamo anche uno per andare da 1080 a 0. Ad ogni passo del timer, utilizziamo lo slicing di NumPy per combinare le due immagini in base alla posizione corrente del wipe.

Durante il wipe, utilizziamo lo slicing per copiare le righe della preview sopra le righe corrispondenti del programma:

```
def wipeTopToBottom(self, preview_frame, program_frame):
    wipe_position = int(self._wipe_position_topToBottom_list[self._wipe])
    wipe_frame = program_frame.copy()
    wipe_frame[:wipe_position, :] = preview_frame[:wipe_position, :]
    return wipe_frame
```

Mentre per il wipe dal basso verso l'alto, utilizziamo lo slicing di NumPy per sostituire le righe sotto la posizione corrente (`wipe_position`) con le righe corrispondenti dell'immagine di anteprima, per il wipe dall'alto verso il basso, utilizziamo lo slicing di NumPy per sostituire le righe sopra la posizione corrente (`wipe_position`) con le righe corrispondenti dell'immagine di anteprima.

Questo approccio garantisce che il movimento del wipe sia fluido e che l'immagine risultante sia una combinazione dinamica delle due immagini, basata sulla posizione corrente del wipe.

```
class MIX_TYPE(Enum):
    FADE = 0
    WIPE_LEFT_TO_RIGHT = 1
    WIPE_RIGHT_TO_LEFT = 2
    WIPE_TOP_TO_BOTTOM = 3
    WIPE_BOTTOM_TO_TOP = 4

class MixBus4_4(QObject):
```

```

_fade = 0
_fadeTime = 100
_wipe = 0
_wipeTime = 90
effectType = MIX_TYPE.FADE

def __init__(self, input1, input2, parent=None):
    super().__init__(parent)
    self.previewInput = input1
    self.programInput = input2
    self.autoMix_timer = QTimer(self)
    self.autoMix_timer.timeout.connect(self._fader)
    self._init_wipe_positions()

def _init_wipe_positions(self):
    _wipe_step = max(1, self._wipeTime)
    self._wipe_position_leftToRight_list = np.linspace(0, 1920,
_wipe_step)
    self._wipe_position_rightToLeft_list = np.linspace(1920, 0,
_wipe_step)
    self._wipe_position_topToBottom_list = np.linspace(0, 1080,
_wipe_step)
    self._wipe_position_bottomToTop_list = np.linspace(1080, 0,
_wipe_step)

def getMixed(self):
    prw_frame = self.previewInput.getFrame()
    prg_frame = self.programInput.getFrame()
    if self._fade == 0:
        return prw_frame, prg_frame
    else:
        if self.effectType == MIX_TYPE.FADE:
            return prw_frame, cv2.addWeighted(prw_frame, self._fade,
                                              prg_frame, 1 - self._fade, 0)
        elif self.effectType == MIX_TYPE.WIPE_LEFT_TO_RIGHT:
            return prw_frame, self.wipeLeftToRight(prw_frame, prg_frame)
        elif self.effectType == MIX_TYPE.WIPE_RIGHT_TO_LEFT:
            return prw_frame, self.wipeRightToLeft(prw_frame, prg_frame)
        elif self.effectType == MIX_TYPE.WIPE_TOP_TO_BOTTOM:
            return prw_frame, self.wipeTopToBottom(prw_frame, prg_frame)
        elif self.effectType == MIX_TYPE.WIPE_BOTTOM_TO_TOP:
            return prw_frame, self.wipeBottomToTop(prw_frame, prg_frame)

def setFade(self, value: int):
    if value == 0:
        self._fade = 0
    else:
        self._fade = value / 100
        self._wipe = value

```

```

def cut(self):
    self._fade = 0
    self._wipe = 0
    self.previewInput, self.programInput = self.programInput,
self.previewInput

def autoMix(self):
    self.autoMix_timer.start(1000 // 60)

def _fader(self):
    if self.effectType == MIX_TYPE.FADE:
        self._fade += 0.01
        if self._fade > 1:
            self.autoMix_timer.stop()
            self.cut()
    elif self.effectType in [MIX_TYPE.WIPE_LEFT_TO_RIGHT,
                            MIX_TYPE.WIPE_RIGHT_TO_LEFT,
                            MIX_TYPE.WIPE_TOP_TO_BOTTOM,
                            MIX_TYPE.WIPE_BOTTOM_TO_TOP]:
        self._wipe += 1
        self._fade += 0.01
        if self._wipe > len(self._wipe_position_leftToRight_list) - 1:
            self.autoMix_timer.stop()
            self.cut()

def wipeLeftToRight(self, preview_frame, program_frame):
    wipe_position =
        int(self._wipe_position_leftToRight_list[self._wipe])
    wipe_frame = program_frame.copy()
    wipe_frame[:, :wipe_position] = preview_frame[:, :wipe_position]
    return wipe_frame

def wipeRightToLeft(self, preview_frame, program_frame):
    wipe_position =
        int(self._wipe_position_rightToLeft_list[self._wipe])
    wipe_frame = program_frame.copy()
    wipe_frame[:, wipe_position:] = preview_frame[:, wipe_position:]
    return wipe_frame

def wipeTopToBottom(self, preview_frame, program_frame):
    wipe_position =
        int(self._wipe_position_topToBottom_list[self._wipe])
    wipe_frame = program_frame.copy()
    wipe_frame[:wipe_position, :] = preview_frame[:wipe_position, :]
    return wipe_frame

def wipeBottomToTop(self, preview_frame, program_frame):
    wipe_position =
        int(self._wipe_position_bottomToTop_list[self._wipe])
    wipe_frame = program_frame.copy()

```

```
wipe_frame[wipe_position:, :] = preview_frame[wipe_position:, :]
return wipe_frame
```

4.5 Stinger

Uno stinger è una sequenza di immagini usata al posto della transizione wipe. Solitamente, è un'animazione che introduce un logo a pieno schermo e, mentre è a pieno schermo, viene eseguito il cut tra preview e program. Questo sistema è molto utilizzato per introdurre video, i replay nello sport, l'apertura e la chiusura della fascia pubblicitaria, e così via.

Queste sequenze di immagini hanno un canale in più chiamato alpha o matte. Questo quarto canale è una mappa in bianco e nero usata per determinare la trasparenza dell'immagine. Dove i valori sono zero (nero), l'immagine è completamente trasparente, mentre dove i valori sono 255 (bianco), l'immagine è completamente opaca. Nei valori intermedi, l'immagine diventa semitrasparente.

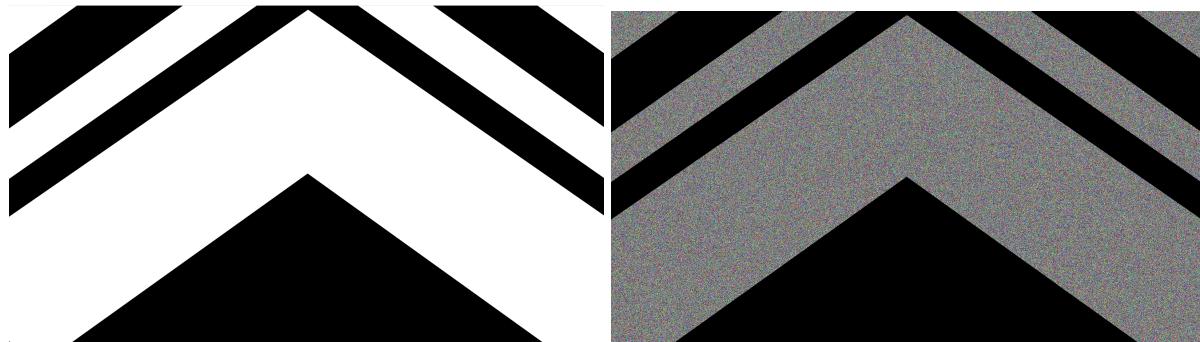
Per compositare insieme due immagini, le due matrici devono essere identiche in dimensioni e tipo, quindi devono essere due matrici da [1920x1080, 3]. L'operazione di compositing può essere descritta dalla seguente formula:

$$\text{output} = \text{program} \times (1 - \alpha) + \text{stinger} \times \alpha$$

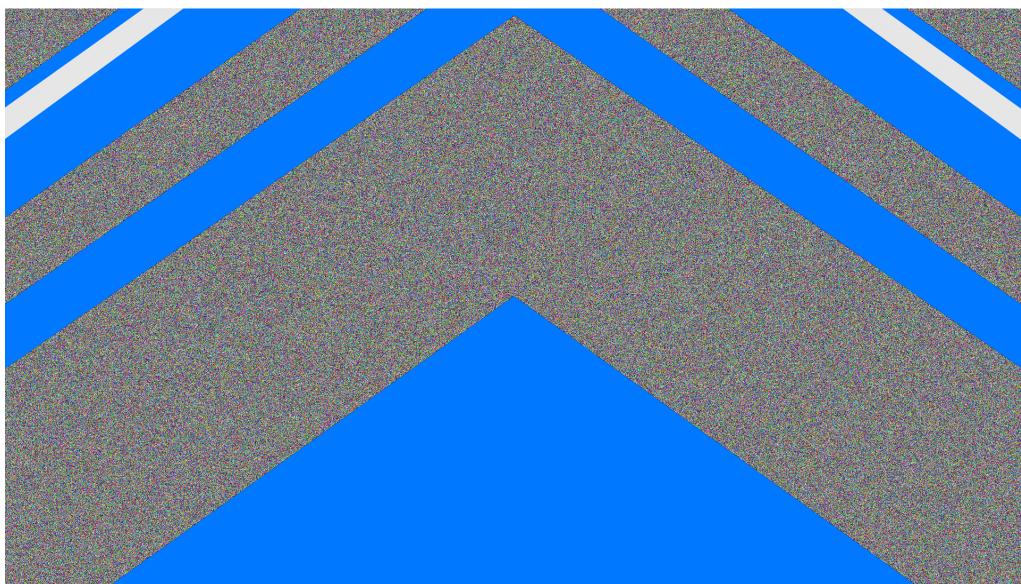
Prendiamo un'immagine dello stinger :



Questa immagine come detto ha 4 canali, i primi 3 b, g, r sono i canali colore, il quarto invece è un'immagine, riportata accanto, in bianco e nero che identifica in quali punti l'immagine è trasparente, nero o zero, o completamente opaca 255 o bianco.



Per mixarlo insieme a un'altra immagine la procedura è questa se invertiamo il canale alfa, otteniamo il suo negativo che moltiplicato sulla nostra immagine di program, apparirà nera nei punti dove lo stinger non è nero, e lo stesso lo stinger apparirà nero nei punti dove in base al suo alpha deve essere completamente trasparente. Sommando insieme le due immagini ottengo l'immagine composita:



Questa tecnica è la trasposizione matematica di quello che si faceva in pellicola. Il nero rappresentava la parte vergine della pellicola, quella non ancora esposta. Con una Stampatrice ottica, si creavano delle maschere su dei fogli di acetato, il Matte, lasciando la parte che per noi è nera trasparente e dipingendo di nero la parte che per noi è bianco. In questo modo veniva fatta una copia della pellicola girata dal vivo con questi buchi nell'immagine. Il processo veniva ripetuto usando un'altra immagine usando il negativo del matte e quindi veniva fatta una proiezione contemporanea delle due pellicole sovrapposte e veniva copiata su una nuova pellicola.

Potete approfondire molto sulle tecniche usate partendo da wikipedia (https://en.wikipedia.org/wiki/Optical_printer) .

Per ricreare questa tecnica e farla in tempo reale ho fatto vari esperimenti per cercare il metodo migliore per fare questa operazione. La moltiplicazione fra matrici è un'operazione dispendiosa; Mi sono accorto però che molte delle operazioni richieste possono essere fatte prima per poi usare i risultati già calcolati. Esaminiamo la lista delle operazioni necessarie:

1. Splittare i canali dell'immagine stinger:
 $b,g,r,a = \text{SPLIT}(stinger)$
2. Creare una matrice alpha replicata su tre canali:
 $\alpha = \text{MERGE}(a,a,a)$
3. Calcolare l'overlay moltiplicando i canali b, g, r per alpha:
 $overlay = \text{MERGE}(b,g,r) \times \alpha$
4. Invertire alpha:
 $iA = 1 - a$
 $invert_alpha = \text{MERGE}(iA,iA,iA)$
5. Calcolare il background:
 $background = program \times invert_alpha$
6. Sommare background e overlay:
 $output = background + overlay$

Ogni operazione introduce un piccolo delay. Per ridurre questo problema, possiamo precalcolare alcune operazioni quando carichiamo lo stinger. Possiamo creare una classe speciale che pre-elabora le immagini dello stinger e fornisce i risultati pre-calcolati tramite funzioni come

getInvAlpha(index) o **getPremultiplyStinger(index)**. In questo modo, riduciamo il calcolo in diretta a due sole operazioni.

```
import os
import time
import cv2
import numpy as np
from PyQt6.QtCore import QObject

class StingerInputLoader(QObject):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.stingerPath = ''
        self.stingerImages = [] # original list of images with alpha channel
        self.stingerRGBImages = [] # list of images without alpha channel
        self.stingerAlphaImages = [] # list of alpha channel images
        self.stingerInvAlphaImages = [] # list of inverted alpha channel images
        self.stingerPreMultipliedImages = [] # list of pre-multiplied images

    def setPath(self, path):
        self.stingerPath = path
        self.loadStingerFrames(path)
        self._findAlphaInvertAndMerge(self.stingerImages)

    def getStingerFrame(self, index):
        return self.stingerImages[index]

    def getStingerAlphaFrame(self, index):
        return self.stingerAlphaImages[index]

    def getStingerInvAlphaFrame(self, index):
        return self.stingerInvAlphaImages[index]

    def getStingerPremultipliedFrame(self, index):
        return self.stingerPreMultipliedImages[index]

    def _findAlphaInvertAndMerge(self, imageList):
        for image in imageList:
            b, g, r, a = cv2.split(image)
            alpha = cv2.merge((a, a, a))
            invA = cv2.bitwise_not(a)
            invAlpha = cv2.merge((invA, invA, invA))
            bgr = cv2.merge((b, g, r))
            self.stingerAlphaImages.append(alpha)
            self.stingerInvAlphaImages.append(invAlpha)
            self.stingerRGBImages.append(bgr)
            self.stingerPreMultipliedImages.append(cv2.multiply(alpha, bgr))
```

```

def loadStingerFrames(self, path):
    for filename in os.listdir(path):
        if filename.endswith('.png'):
            image_path = os.path.join(path, filename)
            image = cv2.imread(image_path, cv2.IMREAD_UNCHANGED)
            self.stingerImages.append(image)

if __name__ == '__main__':
    path = r'\openPyVisionBook\openPyVisionBook\cap5\cap5_5\stingerTest'
    stingerInputLoader = StingerInputLoader()

    timeStart = time.time()
    stingerInputLoader.setPath(path)
    print(f"Time to load stinger frames: {time.time() - timeStart} seconds")

```

Time to load stinger frames: 2.4963250160217285 seconds

Questo risultato è una bella notizia, nel senso che in totale ci farà risparmiare 2.5 secondi, però implementata in questo modo, bloccherà l'interfaccia utente fino a fine caricamento. Quello che possiamo fare in questo caso è creare un thread separato usando QThread di PyQt che ci permette di caricare le immagini senza bloccare l'interfaccia.

```

import os
import time
import cv2
import numpy as np
from PyQt6.QtCore import *
from PyQt6.QtGui import *
from PyQt6.QtWidgets import *

class StingerLoaderThread(QThread):
    stingerReady = pyqtSignal()
    progressUpdated = pyqtSignal()

    def __init__(self, _path, parent=None):
        super().__init__(parent)
        self.path = _path
        self.stingerImages = []
        self.stingerRGBImages = []
        self.stingerAlphaImages = []
        self.stingerInvAlphaImages = []
        self.stingerPreMultipliedImages = []

    def run(self):
        self.loadStingerFrames(self.path)

```

```

self._findAlphaInvertAndMerge(self.stingerImages)
self._setPremultipliedFrame(self.stingerRGBImages)
self.stingerReady.emit()

def loadStingerFrames(self, _path):
    for filename in os.listdir(_path):
        if filename.endswith('.png'):
            image_path = os.path.join(_path, filename)
            image = cv2.imread(image_path, cv2.IMREAD_UNCHANGED)
            self.stingerImages.append(image)
            self.progressUpdated.emit()

def _findAlphaInvertAndMerge(self, imageList):
    for image in imageList:
        b, g, r, a = cv2.split(image)
        a = a / 255.0
        alpha = cv2.merge((a, a, a))
        invAlpha = cv2.merge((1 - a, 1 - a, 1 - a))
        self.stingerAlphaImages.append(alpha)
        self.stingerInvAlphaImages.append(invAlpha)
        self.stingerRGBImages.append(cv2.merge((b, g, r)))
        self.progressUpdated.emit()

def _setPremultipliedFrame(self, imageList):
    for image, alpha in zip(imageList, self.stingerAlphaImages):
        premultiplied = cv2.multiply(image.astype(np.float32),
                                     alpha, dtype=cv2.CV_8U)
        self.stingerPreMultipliedImages.append(premultiplied)
        self.progressUpdated.emit()

class StingerDisplay(QWidget):
    def __init__(self, loaderThread, parent=None):
        super().__init__(parent)
        self.loaderThread = loaderThread
        self.progressBar = QProgressBar(self)
        self.lbl = QLabel("Loading Stinger Frames...", self)
        self.timeLabel = QLabel("Time: 0.0s", self)
        self.timer = QTimer(self)
        self.startTime = time.time()
        self.initUI()
        self.initConnections()
        self.loaderThread.start()
        self.timer.start(100)

    def initUI(self):
        mainLayout = QVBoxLayout()
        mainLayout.addWidget(self.lbl)
        mainLayout.addWidget(self.progressBar)
        mainLayout.addWidget(self.timeLabel)
        self.setLayout(mainLayout)

```

```

    self.setWindowTitle('Stinger Loader Progress')
    self.progressBar.setRange(0, 100)

def initConnections(self):
    self.loaderThread.progressUpdated.connect(self.updateProgressBar)
    self.loaderThread.stingerReady.connect(self.onStingerReady)
    self.timer.timeout.connect(self.animateProgressBar)

@pyqtSlot()
def updateProgressBar(self):
    pass # La progress bar viene animata dal timer

@pyqtSlot()
def animateProgressBar(self):
    value = (self.progressBar.value() + 1) % 101
    self.progressBar.setValue(value)
    elapsed_time = time.time() - self.startTime
    self.timeLabel.setText(f"Time: {elapsed_time:.1f}s")

@pyqtSlot()
def onStingerReady(self):
    self.timer.stop()
    self.progressBar.setValue(100)
    elapsed_time = time.time() - self.startTime
    self.timeLabel.setText(f"Completed in: {elapsed_time:.1f}s")
    print("All done!")

if __name__ == '__main__':
    path = r'openPyVisionBook\openPyVisionBook\cap5\cap5_5\stingerTest'
    stingerLoaderThread = StingerLoaderThread(path)
    app = QApplication([])
    stingerDisplay = StingerDisplay(stingerLoaderThread)
    stingerDisplay.show()

    app.exec()

```

Il thread chiama varie funzioni tramite start il cui scopo è quello di riempire queste liste:

```

    self.stingerImages = []
    self.stingerRGBImages = []
    self.stingerAlphaImages = []
    self.stingerInvAlphaImages = []
    self.stingerPreMultipliedImages = []

```

Mentre scrivevo questo codice mi sono imbattuto in varie problematiche. Innanzitutto abbiamo visto che ci sono operazioni più veloci di quelle che ho usato che abbiamo misurato nel capitolo precedente. Il problema è che non sempre viene creato uno spazio continuo in memoria e questo in termini pratici si traduce in una fase di caricamento rapidissimo, solo che poi c'era un rallentamento significativo nella fase di moltiplicazione fra matrici.

Su github nella cartella del paragrafo ho lasciato i codici che ha usato in questa fase di test. Il risultato è che, nonostante, controllivamente e prese singolarmente, queste operazioni siano molto veloci, la maggiore velocità è dovuta alla mancata ottimizzazione della memoria che poi porta a un rallentamento in fase di esecuzione.

```
Time to load stinger frames: 2.374817371368408 seconds
Time to load stinger frames2: 2.9349265098571777 seconds
Time to load stinger frames3: 2.1785390377044678 seconds
Execution time of stingerFunction3: 4.307186499936506 seconds
Execution time of stingerFunctionFast: 4.3145863998215646 seconds
Execution time of stingerFunction3: 19.326996800024062 seconds
```

Il primo metodo usava cv2.split e cv.merge e mi aspettavo che usando altri metodi, che presi singolarmente risultavano più veloci come l'indexing o le liste di comprensione; Per qualche motivo lo stesso calcolo diventava inaspettatamente più lento, nonostante all'apparenza le immagini usate risultassero identiche.

Ho trascorso un paio di giorni a tentare di capire cosa c'era dietro le quinte e alla fine ho scoperto che i tre metodi restituivano di fatto 3 array identici, ma non rappresentati come continui in memoria e questo durante la moltiplicazione crea un rallentamento molto significativo.

4.6 Stinger More

Ora, abbiamo una classe stinger che ci permette tramite un indice di accedere a tutte le immagini precalcolate. Quello che dobbiamo fare

praticamente è prendere il frame moltiplicarlo per l'inverso dell'alpha e quindi sommarlo allo stinger già premoltiplicato.

E' forse sempre bene ricordarsi che se moltiplico insieme due uint8, ho sempre un risultato inatteso perchè 255 non mi da quello che in senso stretto mi fa vedere la formula, ovvero una moltiplicazione per 1. Quindi alpha e inverted alfa sono già stati normalizzati nel thread dello stinger.

```
def onStingerReady(self):
    self.frames = self.loaderThread.stingerPreMultipliedImages
    self.invMasks = self.loaderThread.stingerInvAlphaImages
    self.timer.start(1000 // 60) # Update every 16.67ms (60 FPS)

def updateFrame(self):
    program = np.random.randint(0, 256, (1080, 1920, 3), dtype=np.uint8)
    if self.frames:
        timeStart = time.time()
        stinger_frame = self.frames[self.currentIndex]
        inv_mask = self.invMasks[self.currentIndex]
        program_masked = cv2.multiply(program, inv_mask, dtype=cv2.CV_8U)
        result = cv2.add(stinger_frame, program_masked)
        height, width, channel = result.shape
        qImg = QImage(result.data, width, height, QImage.Format.Format_BGR888)
        self.label.setPixmap(QPixmap.fromImage(qImg))
        self.label.setScaledContents(True)
        self.label.resize(width, height)
        self.currentIndex = (self.currentIndex + 1) % len(self.frames)
        self.lblTime.setText(f"Time: {time.time() - timeStart:.6f}")
```

Quello che praticamente devo andare a fare è caricare l'immagine premoltiplicata e l'alpha invertito all'indice corrente, quindi moltiplicare insieme il program con l'alpha invertito e poi sommarli insieme.

Purtroppo l'operazione supera il tempo che ci serve, ci mette circa 0.22 secondi, mentre noi abbiamo bisogno che venga fatta entro gli 0.016.

Se ho un canale alpha in bianco e nero dove i valori sono solo bianchi o solo neri, posso usare bitwise_and che esegue l'operazione nell'ordine degli 0.0022 secondi. Il problema è che se l'immagine ha i bordi sfumati la parte semitrasparente viene tagliata.

La maggior parte di un canale alpha però è completamente trasparente solo nel caso peggiore. Nella maggior parte dei casi le parti

diverse da 0 e 1 sono quasi sempre la maggioranza. Questo è quello che fa l'opzione `dtype=cv2.CV_8U` dice che sto moltiplicando per comodità una maschera a un'immagine e questa immagine ha uno spazio colore srgb a 8 bit e che mi porta ad avere un tempo medio di esecuzione di 0.008468.

```
class StingerDisplay(QWidget):
    def __init__(self, loaderThread, parent=None):
        super().__init__(parent)
        # init widgets
        self.loaderThread = loaderThread
        self.lblViewer = QLabel(self)
        self.lblTime = QLabel("Loading Stinger Frames...")
        self.lblMediaTime = QLabel("Time: 0.0s")
        self.progressBar = QProgressBar(self)
        self.vwr_timer = QTimer(self)
        self.prgBar_timer = QTimer(self)

        # init variables
        self.startTime = time.time()
        self.currentIndex = 0
        self.totalTimeSpent = 0
        self.mediaTimeSpend = 0
        self.mediaIndex = 0
        self.frames = []
        self.invMasks = []

        # init UI
        self.initUI()
        self.setGeometry()
        self.setStyle()
        self.initConnections()

    def initUI(self):
        mainLayout = QVBoxLayout()
        mainLayout.addWidget(self.lblViewer)
        timeLayout = QHBoxLayout()
        timeLayout.addWidget(self.lblTime)
        timeLayout.addWidget(self.lblMediaTime)
        timeLayout.addWidget(self.progressBar)
        mainLayout.addLayout(timeLayout)
        self.setLayout(mainLayout)

    def initConnections(self):
        self.loaderThread.stingerReady.connect(self.onStingerReady)
        self.loaderThread.progressUpdated.connect(self.updateProgressBar)
        self.vwr_timer.timeout.connect(self.updateFrame)
        self.prgBar_timer.timeout.connect(self.animateProgressBar)
        self.prgBar_timer.start(100)
```

```

def initStyle(self):
    lblStyle = ("QLabel {"
                "background-color: #000000;"
                "color: #00FF00;"
                "border: 1px solid #00FF00;"
                "border-radius: 5px;}")
    self.lblViewer.setStyleSheet(lblStyle)

def initGeometry(self):
    self.setGeometry(10, 50, 1920, 1080)
    self.progressBar.setRange(0, 100)

@pyqtSlot(int)
def updateProgressBar(self, value):
    self.progressBar.setValue(value)

@pyqtSlot()
def animateProgressBar(self):
    if self.progressBar.value() < 100:
        value = (self.progressBar.value() + 1) % 101
        self.progressBar.setValue(value)
    elapsed_time = time.time() - self.startTime
    self.lblMediaTime.setText(f"Time: {elapsed_time:.1f}s")

@pyqtSlot()
def onStingerReady(self):
    self.frames = self.loaderThread.stingerPreMultipliedImages
    self.invMasks = self.loaderThread.stingerInvAlphaImages
    self.vwr_timer.start(1000 // 60) # Update every 16.67ms (60 FPS)
    self.prgBar_timer.stop()

def updateFrame(self):
    program = np.random.randint(0, 256, (1080, 1920, 3), dtype=np.uint8)
    if self.frames:
        timeStart = time.time()
        stinger_frame = self.frames[self.currentIndex]
        inv_mask = self.invMasks[self.currentIndex]
        program_masked = cv2.multiply(program, inv_mask, dtype=cv2.CV_8U)
        result = cv2.add(stinger_frame, program_masked)
        height, width, channel = result.shape
        qImg = QImage(result.data, width, height, QImage.Format.Format_BGR888)
        self.lblViewer.setPixmap(QPixmap.fromImage(qImg))
        self.lblViewer.setScaledContents(True)
        self.lblViewer.resize(width, height)
        self.currentIndex = (self.currentIndex + 1) % len(self.frames)
        self.lblTime.setText(f"Time: {time.time() - timeStart:.6f}")
        self.updateMediaTime(timeStart)

def updateMediaTime(self, timeStart):
    endTime = time.time() - timeStart

```

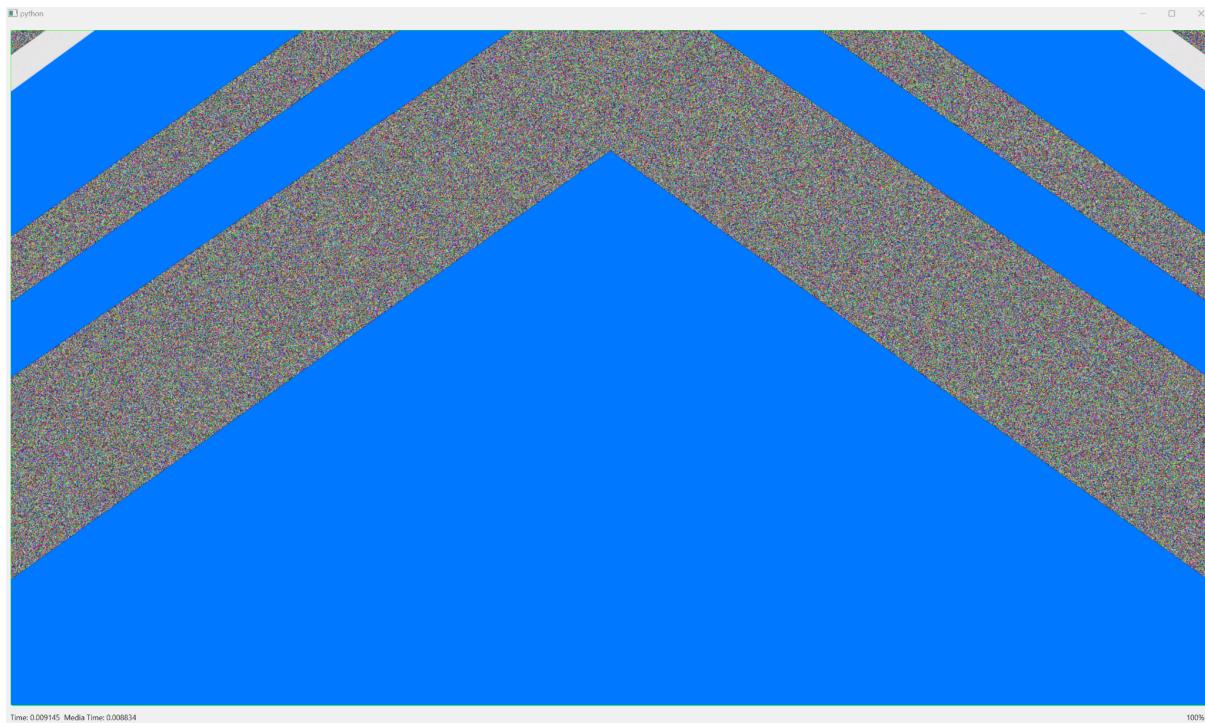
```
self.totalTimeSpent += endTime
self.mediaIndex += 1
self.mediaTimeSpend = self.totalTimeSpent / self.mediaIndex
self.lblTime.setText(f"Time: {endTime:.6f}")
self.lblMediaTime.setText(f"Media Time: {self.mediaTimeSpend:.6f}")

def closeEvent(self, event):
    print(f"Media Time: {self.mediaTimeSpend:.6f}")
    self.vwr_timer.stop()
    event.accept()

if __name__ == '__main__':
    app = QApplication([])

    path = r'/cap5/cap5_5/stingerTest'
    loaderThread = StingerLoaderThread(path)
    stingerDisplay = StingerDisplay(loaderThread)
    stingerDisplay.show()
    loaderThread.start()

    app.exec()
```



*Stinger frames loaded in 1.264967 seconds
Alpha, Inverted Alpha and RGB frames found in 4.267617 seconds
Premultiplied frames found in 1.454782 seconds
Total frames: 136
Media Time: 0.008719*

Il risultato ci da un buon tempo medio, ma la domanda successiva è possiamo fare meglio?

Per rimanere sotto i 10 ms per frame (0.010 secondi), le uniche ottimizzazioni ulteriori che potremmo considerare, se mai ne avessi bisogno, potrebbero essere:

1. Utilizzo di GPU con CUDA o OpenCL per le operazioni di moltiplicazione e addizione.
2. Ottimizzazione della conversione da NumPy array a QImage, se quella parte dovesse diventare un collo di bottiglia.

La GPU è un qualcosa di interessante e che è sempre più discusso, sfortunatamente non ho trovato una grande letteratura al momento con python per riuscire a sfruttare la scheda grafica però si può ottimizzare la trasformazione da array numpy a QImage in modo da poterlo visualizzare nell'interfaccia con un costo minore.

Ci sono vari metodi per farlo e come al solito il modo migliore è quello di farli competere l'uno contro l'altro per vedere chi vince ed eventualmente cercare di mettere insieme il meglio di tutti.

Un primo metodo che poi può diventare comune anche agli altri è quello di rendere gli array numpy continui in memoria perchè velocizza i calcoli.

```
result = np.ascontiguousarray(result)
qImg = QImage(result.data, width, height, result.strides[0],
              QImage.Format.Format_BGR888)
```

Evita la copia dei dati: Invece di creare una nuova QImage ad ogni frame, puoi riutilizzare la stessa QImage e aggiornare i suoi dati:

```
class StingerDisplay(QWidget):
    def __init__(self, loaderThread, parent=None):
        # ... (codice esistente) ...
        self.qImg = QImage(1920, 1080, QImage.Format.Format_BGR888)

    def updateFrame(self):
        # ... (codice esistente) ...
        result = cv2.add(stinger_frame, program_masked)
        self.qImg.setData(result.data)
        self.label.setPixmap(QPixmap.fromImage(self.qImg))
```

Queste prime due indicazioni possono diventare la base per quelle che saranno invece le nostre tre classi ottimizzate.

4.7 STINGER OPTIMIZATION

Il primo metodo consiste nell'utilizzare un puntatore per i dati. Usando `sip.voidptr`, si può creare un puntatore ai dati dell'array NumPy senza copiare i dati.

```
import sip

# ... (nel metodo updateFrame)
result = cv2.add(stinger_frame, program_masked)
ptr = sip.voidptr(result.data)
qImg = QImage(ptr, width, height,
QImage.Format.Format_BGR888)
```

Il miglioramento è piccolo ma c'è e ci porta ad avere un Media Time di 0.008084 ovvero il risparmio medio è di 0,000635.

Un altro sistema che ho trovato è quello di usare QOpenGLWidget. Se stai facendo molto rendering di immagini, potresti considerare di passare a `QOpenGLWidget` invece di `QLabel`. Questo può offrire prestazioni significativamente migliori, specialmente per immagini di grandi dimensioni.

```
from PyQt6.QtOpenGLWidgets import QOpenGLWidget
from PyQt6.QtGui import QSurfaceFormat

class GLImageWidget(QOpenGLWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.image = QImage()

    def setImage(self, image):
        self.image = image
        self.update()

    def paintGL(self):
        if not self.image.isNull():
            painter = QPainter(self)
            painter.drawImage(self.rect(), self.image)
            painter.end()

class StingerDisplay(QWidget):
    def __init__(self, loaderThread, parent=None):
```

```
# ... (codice esistente) ...
self.glWidget = GLImageWidget(self)
self.layout.addWidget(self.glWidget)

def updateFrame(self):
    # ... (codice esistente) ...
    result = cv2.add(stinger_frame, program_masked)
    qImg = QImage(result.data, width, height,
QImage.Format.Format_BGR888)
    self.glWidget.setImage(qImg)
```

Con questa ottimizzazione ottengo un Media Time di 0.007786 che rispetto all'inizio sono 0,000936 secondi in meno.

Un altro metodo molto promettente è quello di creare una sorta di memoria condivisa usando QSharedMemory. Per qualche motivo siccome sto usando un QThread potrebbe in qualche modo essere d'aiuto.

```
from PyQt6.QtCore import QSharedMemory

class StingerDisplay(QWidget):
    def __init__(self, loaderThread, parent=None):
        # ... (codice esistente) ...
        self.shared_memory = QSharedMemory('ImageData')
        self.shared_memory.create(1920 * 1080 * 3)

    def updateFrame(self):
        # ... (codice esistente) ...
        result = cv2.add(stinger_frame, program_masked)
        self.shared_memory.lock()
        memcpy(int(self.shared_memory.data()), result.ctypes.data,
result.nbytes)
        self.shared_memory.unlock()
        qImg = QImage(self.shared_memory.data(), width, height,
QImage.Format.Format_BGR888)
        self.label.setPixmap(QPixmap.fromImage(qImg))
```

Sfortunatamente questo metodo non è molto di aiuto perchè dai dati sperimentali ottengo un Media Time di 0.008373 che è comunque migliore della mia ipotesi di partenza con un miglioramento di 0,000346.

Mettere insieme il meglio di questi tre metodi è quello che ho fatto nel codice che segue che mi da un tempo medio di 0.007310 che ha un risparmio medio di 0,001409 secondi.

```

import os
import time
from PyQt6.QtCore import *
from PyQt6.QtGui import *
from PyQt6.QtWidgets import *
from PyQt6.QtOpenGLWidgets import QOpenGLWidget
import cv2
import numpy as np

class StingerLoaderThread(QThread):
    stingerReady = pyqtSignal()
    progressUpdated = pyqtSignal(int) # Signal to update progress with a percentage value
    somethingDone = pyqtSignal(str, str)

    def __init__(self, _path, parent=None):
        super().__init__(parent)
        self.path = _path
        self.stingerImages = []
        self.stingerRGBImages = []
        self.stingerAlphaImages = []
        self.stingerInvAlphaImages = []
        self.stingerPremultipliedImages = []

    def run(self):
        self.loadStingerFrames(self.path)
        self._findAlphaInvertAndMerge(self.stingerImages)
        self._setPremultipliedFrame(self.stingerRGBImages)
        self.stingerReady.emit()

    def loadStingerFrames(self, _path):
        startTime = time.time()
        files = [f for f in os.listdir(_path) if f.endswith('.png')]
        total_files = len(files)
        for idx, filename in enumerate(files):
            image_path = os.path.join(_path, filename)
            image = cv2.imread(image_path, cv2.IMREAD_UNCHANGED)
            self.stingerImages.append(image)
            self.progressUpdated.emit(int((idx + 1) / total_files * 100))
        print(f"Stinger frames loaded in {time.time() - startTime:.6f} seconds")
        returnString = f"Stinger frames loaded in {time.time() - startTime:.6f} seconds"
        endTime = time.time() - startTime
        self.somethingDone.emit(returnString, f"{endTime:.6f}")

    def _findAlphaInvertAndMerge(self, imageList):
        timeStart = time.time()
        total_images = len(imageList)
        for idx, image in enumerate(imageList):
            b, g, r, a = cv2.split(image)
            a = a / 255.0

```

```

alpha = cv2.merge((a, a, a))
invAlpha = cv2.merge((1 - a, 1 - a, 1 - a))
self.stingerAlphaImages.append(alpha)
self.stingerInvAlphaImages.append(invAlpha)
self.stingerRGBImages.append(cv2.merge((b, g, r)))
self.progressUpdated.emit(int((idx + 1) / total_images * 100))
returnString = f"Alpha, Inverted Alpha and RGB frames found in
                           {time.time() - timeStart:.6f} seconds"
endTime = time.time() - timeStart
self.somethingDone.emit(returnString, f"{endTime:.6f}")

def _setPremultipliedFrame(self, imageList):
    timeStart = time.time()
    total_images = len(imageList)
    for idx, (image, alpha) in enumerate(zip(imageList, self.stingerAlphaImages)):
        premultiplied = cv2.multiply(image.astype(np.float32),
                                      alpha, dtype=cv2.CV_8U)
        self.stingerPreMultipliedImages.append(premultiplied)
        self.progressUpdated.emit(int((idx + 1) / total_images * 100))
    returnString = f"Premultiplied frames found in
                           {time.time() - timeStart:.6f} seconds"
    endTime = time.time() - timeStart
    self.somethingDone.emit(returnString, f"{endTime:.6f}")

    print(f"Total frames: {len(self.stingerPreMultipliedImages)}")

class GLImageWidget(QOpenGLWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.image = QImage()

    def setImage(self, image):
        self.image = image
        self.update()

    def paintGL(self):
        if not self.image.isNull():
            painter = QPainter(self)
            painter.drawImage(self.rect(), self.image)
            painter.end()

class StingerDisplay(QWidget):
    floatCount = 0

    def __init__(self, _loaderThread, parent=None):
        super().__init__(parent)
        # init widgets
        self.loaderThread = _loaderThread

```

```

self.glWidget = GLImageWidget(self)
self.lblTime = QLabel("Time: 0.0")
self.lblMediaTime = QLabel("Media Time: 0.0")
self.prgBar = QProgressBar(self)
self.timer = QTimer(self)
self.prgBar_timer = QTimer(self)
# init variables
self.startTime = time.time()
self.currentIndex = 0
self.totalTimeSpent = 0
self.mediaTimeSpend = 0
self.mediaIndex = 0
self.frames = []
self.invMasks = []
self.qImg = QImage(1920, 1080, QImage.Format.Format_BGR888)
# init UI
self.initUI()
self.initGeometry()
self.initStyle()
self.initConnections()

def initUI(self):
    mainLayout = QVBoxLayout()
    mainLayout.addWidget(self.glWidget)
    timeLayout = QHBoxLayout()
    timeLayout.addWidget(self.lblTime)
    timeLayout.addWidget(self.lblMediaTime)
    timeLayout.addWidget(self.prgBar)
    mainLayout.addLayout(timeLayout)
    self.setLayout(mainLayout)

def initStyle(self):
    lblStyle = ("QLabel {"
                "background-color: #000000;"
                "color: #00FF00;"
                "border: 1px solid #00FF00;"
                "border-radius: 5px;}")
    self.glWidget.setStyleSheet(lblStyle)
    self.prgBar.setMaximum(100)

def initGeometry(self):
    self.setGeometry(0, 0, 1920, 1080)
    self.setWindowTitle('Stinger Display')
    self.glWidget.setFixedSize(1920, 1080)

def initConnections(self):
    self.loaderThread.stingerReady.connect(self.onStingerReady)
    self.loaderThread.progressUpdated.connect(self.updateProgressBar)
    self.loaderThread.somethingDone.connect(self.updateProgressBarText)
    self.timer.timeout.connect(self.updateFrame)

```

```

    self.prgBar_timer.timeout.connect(self.animateProgressBar)
    self.prgBar_timer.start(100)

@pyqtSlot(int)
def updateProgressBar(self, value):
    self.prgBar.setValue(value)

@pyqtSlot()
def animateProgressBar(self):
    if self.prgBar.value() < 100:
        value = (self.prgBar.value() + 1) % 101
        self.prgBar.setValue(value)
    elapsed_time = time.time() - self.startTime
    self.lblMediaTime.setText(f"Time: {elapsed_time:.1f}s")

@pyqtSlot(str, str)
def updateProgressBarText(self, returnString, timeString):
    print(f"{returnString} in {timeString} seconds")
    self.floatCount += float(timeString)
    if self.prgBar.value() >= 100:
        self.prgBar.setValue(100)
        self.prgBar_timer.stop()

@pyqtSlot()
def onStingerReady(self):
    self.frames = self.loaderThread.stingerPreMultipliedImages
    self.invMasks = self.loaderThread.stingerInvAlphaImages
    self.timer.start(1000 // 60) # Update every 16.67ms (60 FPS)
    self.prgBar_timer.stop()

def updateFrame(self):
    program = np.random.randint(0, 256, (1080, 1920, 3), dtype=np.uint8)
    if self.frames:
        timeStart = time.time()
        stinger_frame = self.frames[self.currentIndex]
        inv_mask = self.invMasks[self.currentIndex]
        program_masked = cv2.multiply(program, inv_mask, dtype=cv2.CV_8U)
        result = cv2.add(stinger_frame, program_masked)
        result_contiguous = np.ascontiguousarray(result)
        height, width, channel = result_contiguous.shape
        self.qImg = QImage(result_contiguous.data, width, height,
                           QImage.Format.Format_BGR888)
        self.glWidget.setImage(self.qImg)
        self.currentIndex = (self.currentIndex + 1) % len(self.frames)
        self.updateMediaTime(timeStart)

def updateMediaTime(self, timeStart):
    endTime = time.time() - timeStart
    self.totalTimeSpent += endTime
    self.mediaIndex += 1

```

```

self.mediaTimeSpend = self.totalTimeSpent / self.mediaIndex
self.lblTime.setText(f"Time: {endTime:.6f}")
self.lblMediaTime.setText(f"Media Time: {self.mediaTimeSpend:.6f}")

def closeEvent(self, event):
    print(f"Media Time: {self.mediaTimeSpend:.6f}")
    self.timer.stop()
    event.accept()

if __name__ == '__main__':
    app = QApplication([])
    path = r'openPyVisionBook\openPyVisionBook\cap5\cap5_5\stingerTest'
    loaderThread = StingerLoaderThread(path)
    stingerDisplay = StingerDisplay(loaderThread)
    stingerDisplay.show()
    loaderThread.start()
    app.exec()

```

4.8 STINGER IN THE MIXBUS

Ovviamente il nostro obiettivo a questo punto è inserire lo stinger nel mixbus in modo da averlo insieme agli effetti. Dobbiamo scrivere un po di codice:

```

def stinger(self, preview_frame, program_frame):
    stinger_frame = self.stinger_frames[self._wipe]
    inv_mask = self.stinger_invMasks[self._wipe]

    if self._wipe < len(self.stinger_frames) // 2:
        program_masked = cv2.multiply(program_frame, inv_mask,
                                       dtype=cv2.CV_8U)
        result = cv2.add(stinger_frame, program_masked)
        return np.ascontiguousarray(result)
    else:
        preview_masked = cv2.multiply(preview_frame, inv_mask,
                                       dtype=cv2.CV_8U)
        result = cv2.add(stinger_frame, preview_masked)
        return np.ascontiguousarray(result)

```

Quello che succede è che wipe sarà l'indice della lista di sting. Quando la classe stinger loader avrà finito tutte le operazioni il codice inizializza le due liste. Parte il timer con automix e fino a che siamo prima della metà della lista l'operazione verrà fatta con il program. Di solito verso la metà lo sting copre completamente lo schermo e in questo punto cambiamo il

program con il preview.

```
def _fader(self):
    if self.effectType == MIX_TYPE.FADE:
        self._fade += 0.01
        if self._fade > 1:
            self.autoMix_timer.stop()
            self.cut()
    elif self.effectType in
        [MIX_TYPE.WIPE_LEFT_TO_RIGHT, MIX_TYPE.WIPE_RIGHT_TO_LEFT,
         MIX_TYPE.WIPE_TOP_TO_BOTTOM, MIX_TYPE.WIPE_BOTTOM_TO_TOP]:
        self._wipe += 1
        self._fade += 0.01
        if self._wipe > len(self._wipe_position_leftToRight_list) - 1:
            self.autoMix_timer.stop()
            self.cut()
    elif self.effectType == MIX_TYPE.WIPE_STINGER:
        self._wipe += 1
        self._fade += 0.01
        if self._wipe > len(self.stinger_frames) - 1:
            self.autoMix_timer.stop()
            self.cut()
```

Abbiamo visto anche che al posto delle label di Qt, è molto più conveniente creare un viewer che in parte abbiamo già scritto nell'esempio precedente.

```
from PyQt6.QtGui import QImage, QPainter
from PyQt6.QtOpenGLWidgets import QOpenGLWidget

class OpenGLViewer(QOpenGLWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.image = QImage()

    def setImage(self, image):
        self.image = image
        self.update()

    def paintGL(self):
        if not self.image.isNull():
            painter = QPainter(self)
            painter.drawImage(self.rect(), self.image)
            painter.end()
```

Ora non ci rimane che creare la classe `testMixBus5_8` per testare il nuovo `mixBus`, quindi possiamo aggiungere alla lista di effetti anche lo stinger, quindi utilizzare `openGLViewer` al posto delle label e possiamo anche usare l'esempio usato nel capitolo `5_5` ricordate? Il widget con la progress bar che indica quanto manca alla fine del caricamento per dare un timing all'utente.

```
class testMixBus4_8(QWidget):
    def __init__(self, syncObject, loaderThread, parent=None):
        super().__init__(parent)
        self.syncObject = syncObject
        self.input1 = ColorGenerator(self.syncObject)
        self.input2 = RandomNoiseGenerator(self.syncObject)
        self.mixBus = MixBus5_8(self.input1, self.input2, loaderThread)
        self.previewViewer = OpenGLViewer()
        self.programViewer = OpenGLViewer()
        self.btnCut = QPushButton("CUT")
        self.btnAutoMix = QPushButton("AutoMix")
        self.sldFade = QSlider()
        self.cmbEffect = QComboBox()
        self.cmbEffect.addItem("Fade", "Wipe Left to Right", "Wipe Right to Left",
                               "Wipe Top to Bottom", "Wipe Bottom to Top", "Stinger")
        self.sldFade.setOrientation(Qt.Orientation.Horizontal)
        self.sldFade.setRange(0, 100)
        self.initUI()
        self.setGeometry()
        self.initConnections()

    def initUI(self):
        mainLayout = QVBoxLayout()
        viewerLayout = QHBoxLayout()
        viewerLayout.addWidget(self.previewViewer)
        viewerLayout.addWidget(self.programViewer)
        spacer = QSpacerItem(20, 40, QSizePolicy.Policy.Minimum,
QSizePolicy.Policy.Expanding)
        buttonLayout = QHBoxLayout()
        buttonLayout.addItem(spacer)
        buttonLayout.addWidget(self.btnCut)
        buttonLayout.addWidget(self.btnAutoMix)
        buttonLayout.addWidget(self.sldFade)
        buttonLayout.addWidget(self.cmbEffect)
```

```

mainLayout.addLayout(viewerLayout)
mainLayout.addLayout(buttonLayout)
self.setLayout(mainLayout)

def initGeometry(self):
    self.previewViewer.setFixedSize(640, 360)
    self.programViewer.setFixedSize(640, 360)

def initConnections(self):
    self.syncObject.synch_SIGNAL.connect(self.updateFrame)
    self.btnCut.clicked.connect(self.cut)
    self.btnAutoMix.clicked.connect(self.autoMix)
    self.sldFade.valueChanged.connect(self.setFade)
    self.cmbEffect.currentIndexChanged.connect(self.setEffect)

def updateFrame(self):
    prw_frame, prg_frame = self.mixBus.getMixed()
    prw_frame = cv2.resize(prw_frame, (640, 360))
    prg_frame = cv2.resize(prg_frame, (640, 360))
        prw_image = QImage(prw_frame.data, prw_frame.shape[1], prw_frame.shape[0],
QImage.Format.Format_BGR888)
        prg_image = QImage(prg_frame.data, prg_frame.shape[1], prg_frame.shape[0],
QImage.Format.Format_BGR888)
    self.previewViewer.setImage(prw_image)
    self.programViewer.setImage(prg_image)

def cut(self):
    self.mixBus.cut()

def autoMix(self):
    self.mixBus.autoMix()

def setFade(self):
    self.mixBus.setFade(self.sldFade.value())

def setEffect(self):
    effect = self.cmbEffect.currentText()
    if effect == "Fade":
        self.mixBus.effectType = MIX_TYPE.FADE
    elif effect == "Wipe Left to Right":
        self.mixBus.effectType = MIX_TYPE.WIPE_LEFT_TO_RIGHT
    elif effect == "Wipe Right to Left":
        self.mixBus.effectType = MIX_TYPE.WIPE_RIGHT_TO_LEFT
    elif effect == "Wipe Top to Bottom":
        self.mixBus.effectType = MIX_TYPE.WIPE_TOP_TO_BOTTOM
    elif effect == "Wipe Bottom to Top":
        self.mixBus.effectType = MIX_TYPE.WIPE_BOTTOM_TO_TOP
    elif effect == "Stinger":
        self.mixBus.effectType = MIX_TYPE.WIPE_STINGER

```

```
if __name__ == '__main__':
    import sys
    app = QApplication(sys.argv)
    path = r'\\openPyVisionBook\\openPyVisionBook\\cap5\\cap5_5_StingerIdea\\stingerTest'
    loaderThread = StingerLoaderThread(path)
    synchObject = SynchObject()
    stingerDisplay = StingerDisplay(loaderThread)
    stingerDisplay.show()
    test = testMixBus5_8(synchObject, loaderThread)
    test.show()
    sys.exit(app.exec())
```