Relazione progetto picBloom per Multimedia e Laboratorio

Alessandro Lauria W82000198

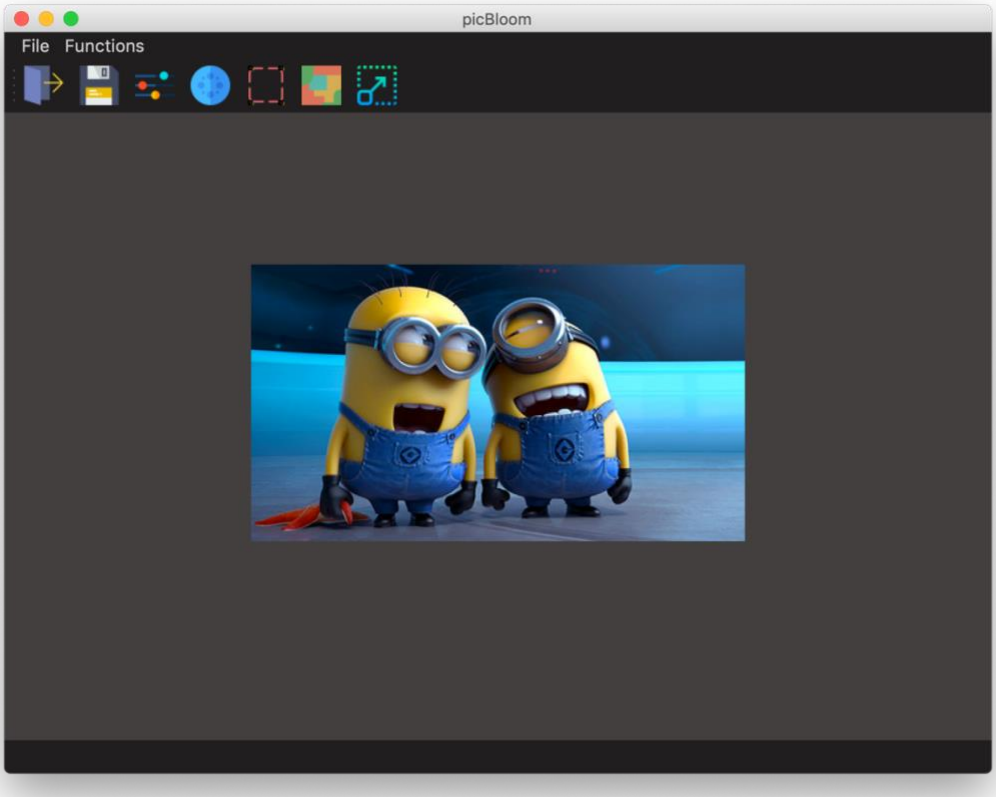
Prof. Filippo Stanco
Prof. Dario Allegra

INTRODUZIONE

Scopo del progetto è ricreare una versione minimale di Photoshop che permetta all'utente di applicare alcune operazioni di manipolazione dell'immagini in maniera semplice e veloce. Le tecnologie scelte per la realizzazione del suddetto sono state:

- Il linguaggio Python
- Il framework PyQt5 per la creazione di applicativi desktop crossplatform
- Svariate librerie Python, come Pillow ed OpenCV, sviluppate per la gestione delle immagini

Di seguito viene mostrata l'interfaccia grafica:



MODIFICA DEL COLORE

Le prime tre operazioni sviluppate sono state:

- Cambio della luminosità
- Cambio del contrasto
- Cambio della saturazione

La **luminosità** è stata implementata cambiando spazio di colore da RGB ad HSV aggiungendo (o sottraendo) un valore costante al Value di ogni pixel dell'immagine. Inizialmente si era provato ad accedere ai singoli pixel usando un doppio ciclo for e per ogni canale incrementare il valore del pixel, ma la soluzione è risultata poco efficiente. Sfruttando gli array numpy e dovendo lavorare su di un solo canale (Value) si è riusciti ad ottimizzare notevolmente l'operazione.

$$value = value + index$$

Risultato:

Il **contrasto** è stato ottenuto assegnando ad ogni pixel un nuovo valore, calcolato secondo la seguente formula:

$$factor = \frac{259 \times (index + 255)}{255 \times (259 - index)}$$

$$red = factor \times (red - 128) + 128$$

$$green = factor \times (green - 128) + 128$$

$$blue = factor \times (blue - 128) + 128$$

Si era provato dapprima ad implementare il contrasto andando ad aggiungere un valore ai pixel sopra una certa soglia e a sottrarre lo stesso a quelli sotto, ma si è rivelata essere una strategia fallimentare andando ad ottenere risultati non coerenti.

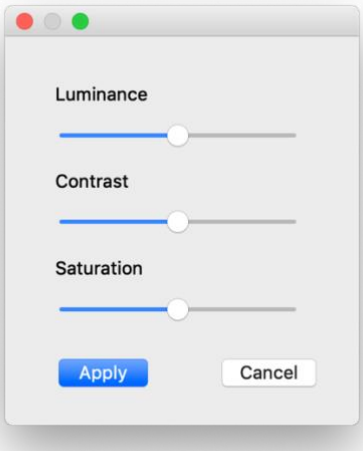
Risultato:

Anche per l'implementazione della **saturazione** si è passati dallo spazio di colore RGB ad HSV andando a incrementare (o decrementare) il valore della saturazione secondo un dato indice. I problemi iniziali sono sorti non avendo ben chiaro il range di azione della saturazione per la libreria Pillow, della quale inoltre la conversione ad HSV, anche senza modifiche, portava a una visualizzazione ambigua dell'immagine. Sfruttando numpy e openCV si è stati in grado di fare una corretta conversione e assegnamento del nuovo valore.

$$saturation = saturation + index$$

Risultato:

Per completare è stato permesso all'utente di applicare le differenti modifiche in simultanea, senza ricominciare dall'immagine di partenza una volta cambiato il tipo di filtro applicato.



FILTRI DI SFOCATURA

Per applicare la sfocatura sono state scelte diverse tecniche riportanti ognuna sia un effetto diverso sull'immagine, sia la rimozione di un diverso tipo di rumore. In particolare, sono stati implementati:

- Filtro di media aritmetica
- Filtro di media geometrica
- Filtro di media armonica
- Filtro mediano

Si è trattato di selezionare un kernel di dimensione k x k da far scorrere lungo tutta l'immagine e applicare le formule dei rispettivi filtri per ottenere il valore da assegnare al pixel al centro della finestra.

Media Aritmetica: $f(x,y) = \frac{1}{mn} \sum_{(s,t) \in S_{xy}} g(s,t)$

Media Geometrica: $f(x,y) = \left[\prod_{(s,t) \in S_{xy}} g(s,t) \right]^{\frac{1}{mn}}$

Media Armonica: $f(x,y) = \frac{mn}{\sum_{(s,t) \in S_{xy}} \frac{1}{g(s,t)}}$

Mediano: $f(x,y) = median_{(s,t) \in S_{xy}}(g(s,t))$

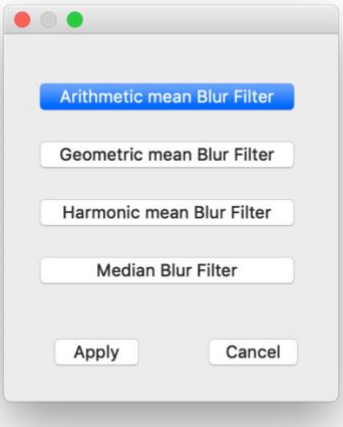
Di seguito i vari risultati:

Originale: Aritmetica:

Geometrica:

Armonica:

Mediano:



EDGE DETECTION

Gli algoritmi scelti per l'edge detection sono:

- Canny
- DroG

Inizialmente è stato sviluppato il **DroG**, potendolo successivamente sfruttare in Canny. Dapprima è stato applicato un filtro di smooting gaussiano. Tramite la Fast Fourier Transform

si è passati dal dominio spaziale al dominio delle frequenze, dove è stata applicata la convoluzione due volte con i kernel SobelX e SobelY:

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Utilizzando l'anti-trasformata si è tornati al dominio spaziale e sommando i valori assoluti delle matrici ottenute si è raggiunto il seguente risultato:

Realizzando la convoluzione tramite doppio ciclo for l'operazione risultava parecchio lenta. Sfruttando la fast fourier transform invece le cose sono diventate molto più veloci.

Sono stati riscontrati iniziali problemi nella visualizzazione del risultato in quanto non si era considerato che venissero ritornati per i bordi sia i valori negativi che quelli positivi. Facendo il valore assoluto della matrice risultante il problema è stato risolto.

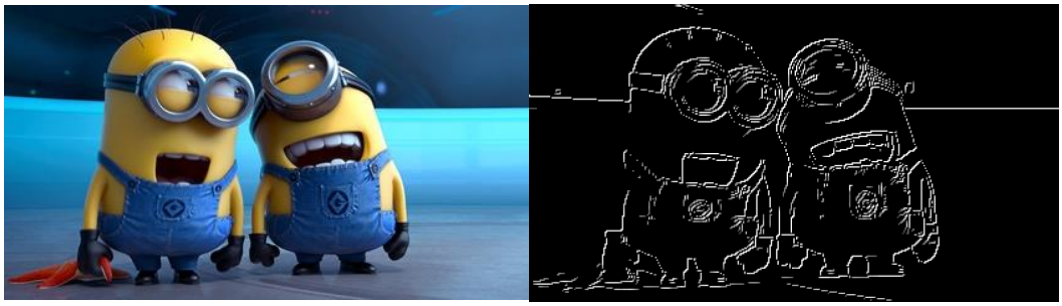
L'implementazione del filtro **Canny** è avvenuta seguendo i quattro step principali della procedura. Inizialmente è stato applicato un filtro gaussiano per ridurre il rumore.

Di seguito sono stati calcolati magnitudo e gradiente, passando nel dominio delle frequenze, sfruttando il DroG.

La fase successiva è stata quella del Non Maximun Suppression, dove sono stati classificati i punti di edge deboli ed edge forti andando a considerare l'andamento del gradiente per ogni punto, valutando quattro angoli quali 0°, 45°, 90° e i restanti.

In fine è stato fatto dell'edge linking per legare gli edge riconosciuti come tali andando a considerare, per ogni pixel facente parte degli edge deboli, gli otto connessi e vedendo se fossero vicini ad un edge forte.

Risultato:

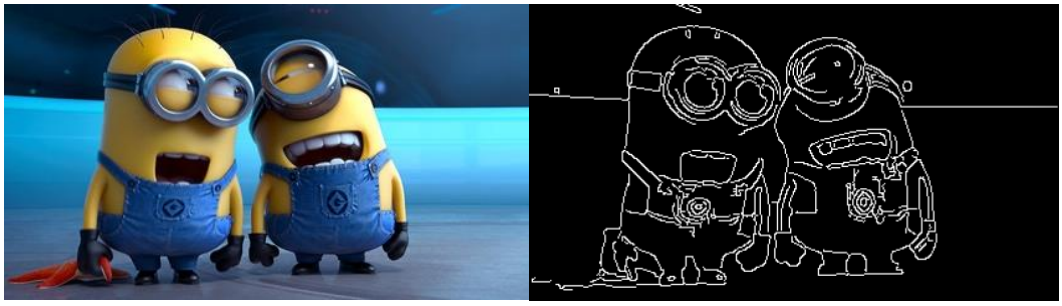


Risulta evidente che l'algoritmo, così implementato, riscontra qualche problema nel riconoscimento di alcuni edge che nella teoria dovrebbero essere facilmente individuabili (come ad esempio la parte sinistra della testa del minion nell'immagine di prova).

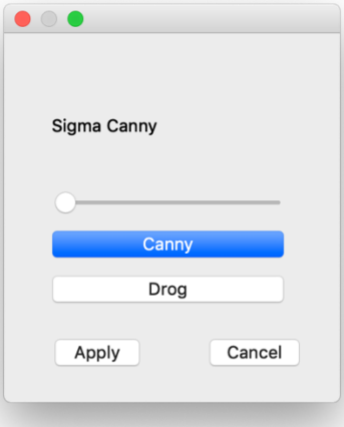
È stata cercata dunque una soluzione migliore per rendere il software più preciso. Come prima cosa si è provato ad aumentare la dimensione dei kernel SobelX e SobelY ipotizzando una possibile problematica nell'applicazione del DroG.

Il test non ha avuto risultati soddisfacenti, ottenendo un'immagine quasi uguale a quella precedente.

Si è proceduto, quindi, applicando uno smooting gaussiano ai kernel sobelX e sobelY prima di avviare la convoluzione all'immagine. La suddetta scelta ha permesso di ottenere un risultato decisamente migliore che viene mostrato di seguito:



È stata inoltre data la possibilità all'utente di scegliere il sigma per canny con uno slider apposito.



STATISTICAL REGION MERGING

Utilizzando l'union find l'immagine è stata suddivisa in regioni di 4-connessi che sono state successivamente legate in base al seguente predicato:

```
def predicate(self, r1, r2):
    g = 256.0
    logdelta = self._logdelta

    w = self._sizes
    out = self._data

    d2 = (out[r1] - out[r2]) ** 2

    log_r1 = min(g, w[r1]) * numpy.log(1.0 + w[r1])
    log_r2 = min(g, w[r2]) * numpy.log(1.0 + w[r2])

    q = self._q
    dev1 = g ** 2 / (2.0 * q * w[r1]) * (log_r1 + logdelta)
    dev2 = g ** 2 / (2.0 * q * w[r2]) * (log_r2 + logdelta)
    dev = dev1 + dev2

    return (d2 < dev).all()
```

Le regioni vengono ordinate e successivamente per ogni coppia viene valutato se si tratti di regioni distinte che rispettano il predicato. In caso di risposta affermativa viene fatto il merge delle due regioni, altrimenti vengono mantenute separate.

È stato permesso all'utente di scegliere il valore q che caratterizza la distanza tra le regioni, in modo che più alto sarà questo valore più regioni si andranno a creare. Di seguito il risultato:

TRASFORMAZIONI

Le trasformazioni dell'immagine implementate sono state:

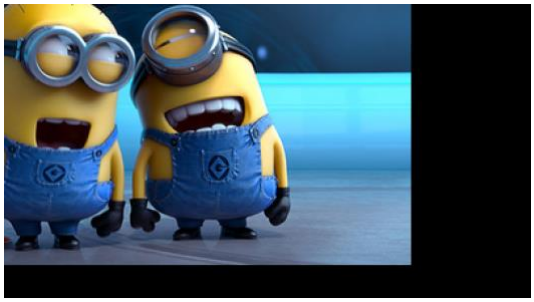
- Traslazione lungo x e y
- Scaling
- Rotazione

La **traslazione** è stata implementata facendo uno shift della matrice numpy contenete l'immagine lungo le direzioni x ed y.

Dato che lo shift andava a rimettere i pixel che uscivano dalla finestra nella zona nera che si andava a creare, è stato necessario fare un ulteriore processamento dell'immagine andando a settare a zero i pixel indesiderati.

Questo comunque non intaccando le prestazioni dell'operazione che si è mantenuta fluida.

Risultato:



Per quanto riguarda le operazioni di rotazione e scaling è stata data la possibilità all'utente di decidere, tramite un checkbox, se applicarle modificando la dimensione della finestra oppure mantenendola.

Lo **scaling** è stato ottenuto creando una nuova immagine vuota di dimensioni date da un certo index. Per mantenere l'aspect ratio è stata sviluppata la seguente formula che permette di ottenere un secondo indice da sommare all'altezza:

$$aspect\ ratio = \frac{width}{height}$$

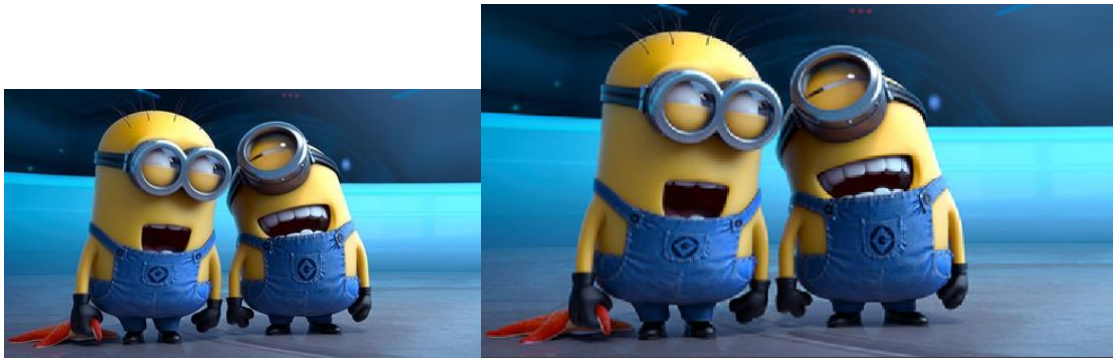
$$index2 = \frac{index1 + width - (aspect\ ratio \times height)}{aspect\ ratio}$$

Viene successivamente scalata l'immagine originale e inserita punto a punto in quella nuova, facendo distinzione tra ingrandimento e rimpicciolimento e conseguenti diverse procedure.

Risultato:

$$Resize = False$$

$$Resize = True$$



La **rotazione** è stata implementata ruotando le coordinate 2D dell'immagine e creando una nuova immagine che contiene la vecchia con le nuove coordinate.

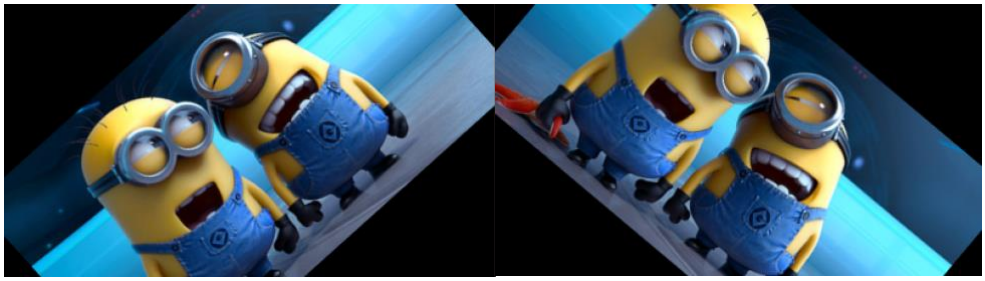
Le principali difficoltà sono state riscontrate nel settare il centro dell'immagine come punto di rotazione, per questo è stata creata la seguente funzione:

```
def rotate_coords(self, x, y, theta, ox, oy):

    s, c = np.sin(theta), np.cos(theta)
    x, y = np.asarray(x) - ox, np.asarray(y) - oy
    return x * c - y * s + ox, x * s + y * c + oy
```

Risultato:

$$Resize = False$$



$$Resize = True$$