

Extra Credit Facili

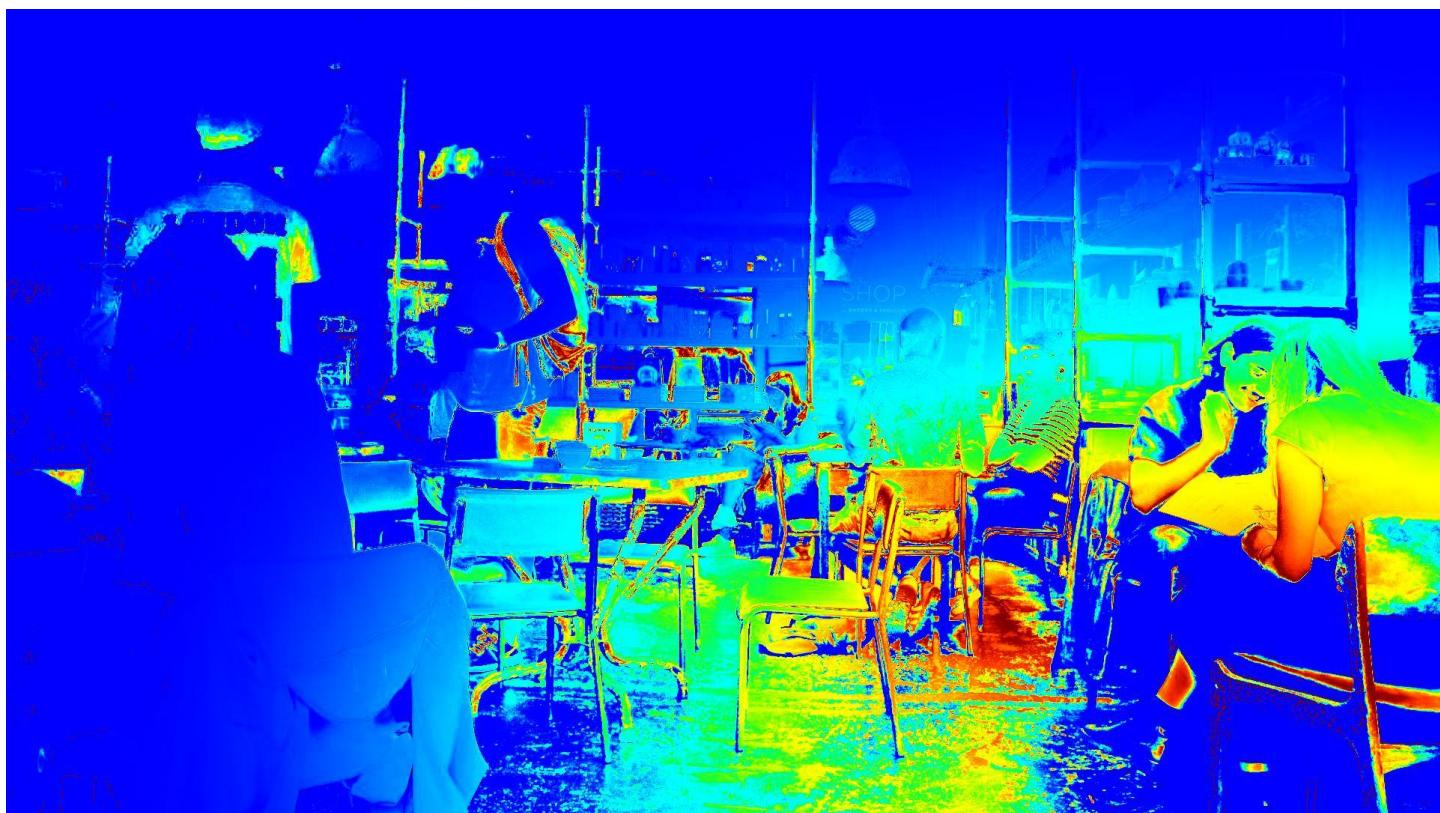
ShaderToy - Predator Thermal Vision.

Per iniziare con qualcosa di semplice, ho cercato un filtro che mostrasse l'immagine in input come se fosse vista attraverso una visione termica ispirata al film cult Predator. I credits per l'implementazione del filtro sono in fondo a questa sezione.

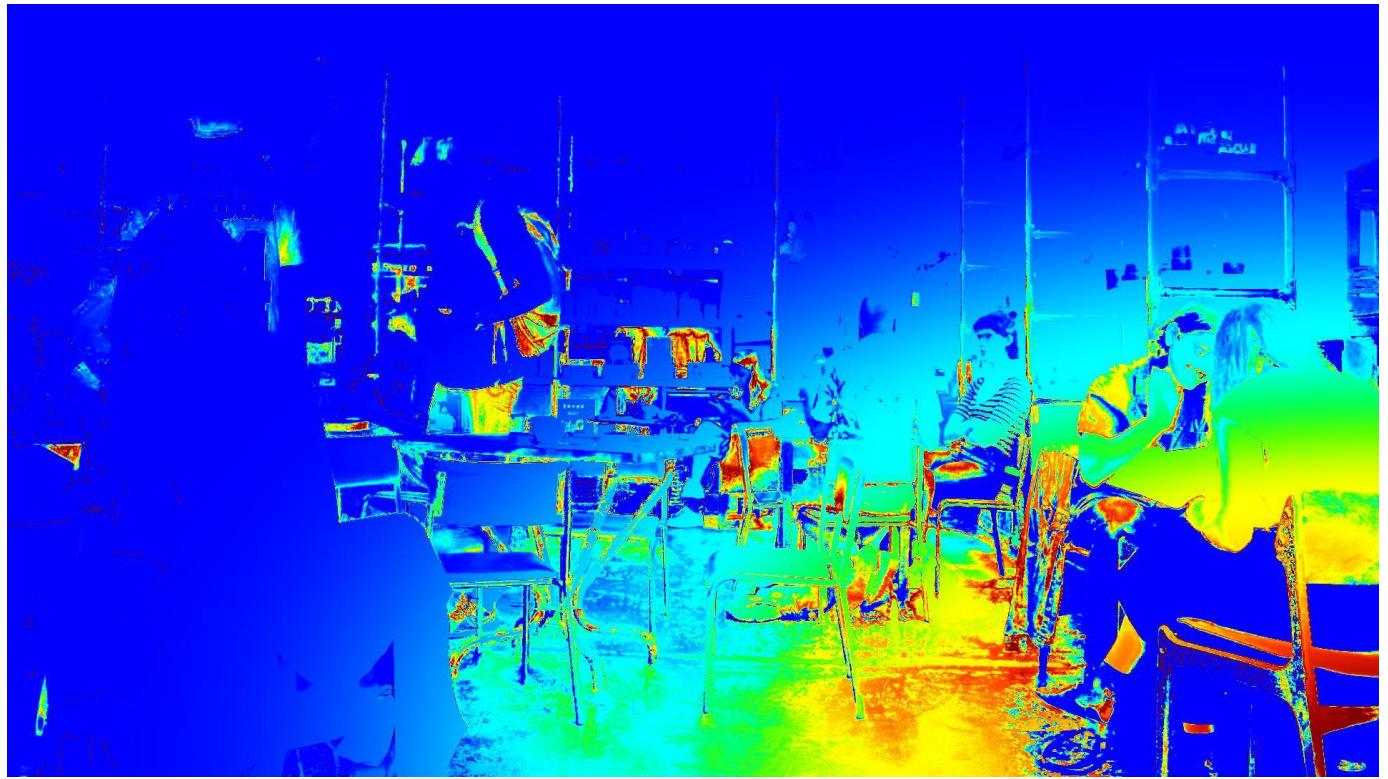
In breve, il filtro “preleva” da ogni pixel dell’immagine i valori red e green, e utilizza diversi successivi smoothstep (nel calcolo è stata usata la funzione built-in della libreria yocto), per modificare l’RGB ed ottenere l’effetto termico. Per calcolare i valori di “heat” viene presa in considerazione anche la grandezza dell’immagine in input (nel nostro caso adattato ad un’immagine statica, su ShaderToy in base alla risoluzione della Webcam), oltre che alla posizione del singolo pixel preso in considerazione.

Ho effettuato cinque test con aumento successivo di exposure e contrast, per sperimentare il comportamento della visione termica all’aumentare della luminosità dell’immagine. Il filtro è “attivabile” da linea di comando con il comando --predator.

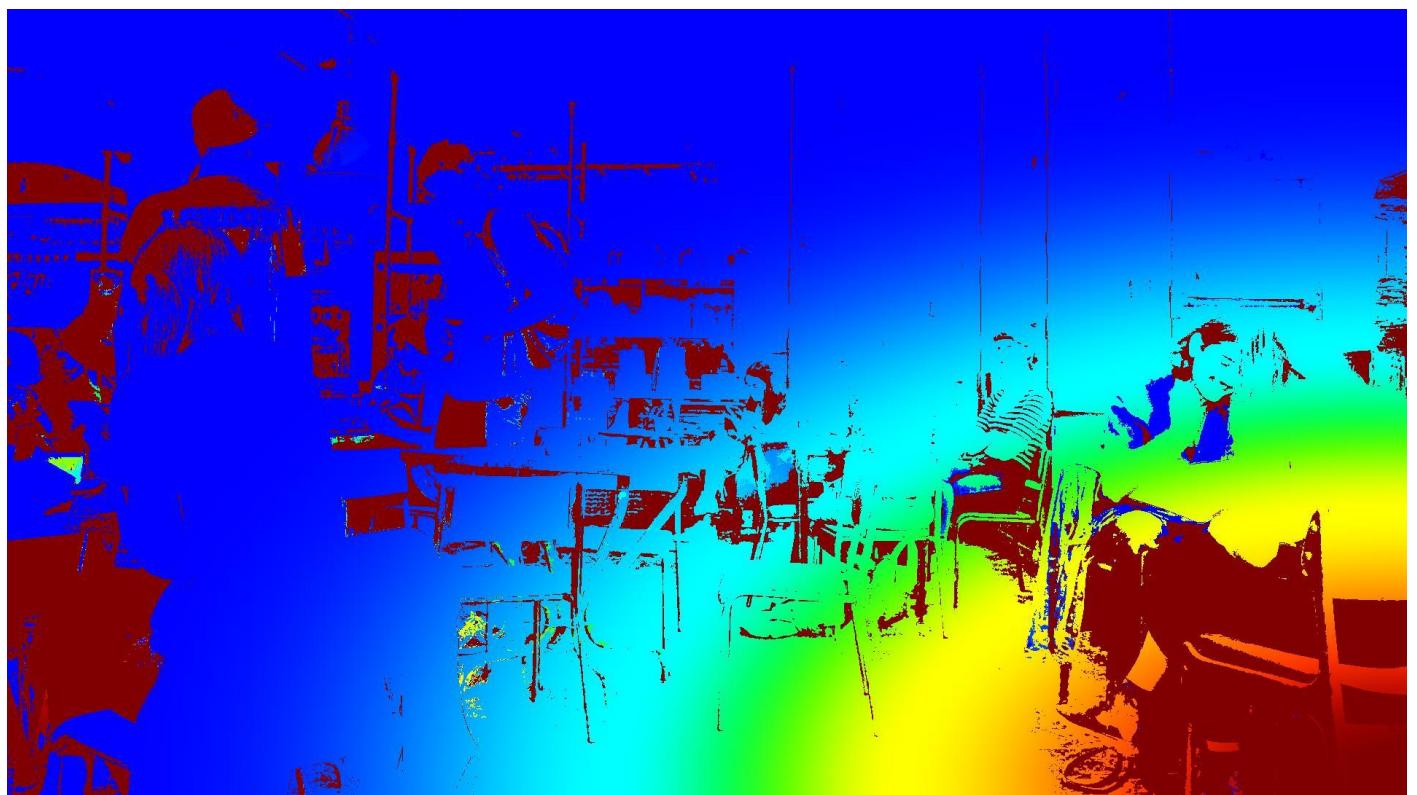
Riporto di seguito tre esempi significativi, tutti applicati sull’immagine fornita *toa_heftiba_people*:



In questa prima applicazione, l’immagine viene filtrata senza alcuna modifica precedente, passata quindi solo attraverso il filtro Predator Thermal Vision.



Aumentando l'exposure possiamo notare come la visione termica riesca a definire sempre meno dettagli dell'immagine fornita in input, come nel risultato seguente



Applicando un forte contrasto oltre ad un forte exposure notiamo come la definizione dell'immagine venga meno, risultando in un'immagine molto più "blurry"

CREDITS

L'algoritmo per l'applicazione del filtro Predator Thermal Vision è stato implementato adattando il codice consultabile alla seguente pagina di Shader Toy: <https://www.shadertoy.com/view/MdGSWD>. Credits all'utente **maldicion069** per l'implementazione. Non sono state usate librerie esterne.

Gaussian Blur.

L'effetto di Gaussian Blur è stato implementato seguendo un algoritmo trovato sempre su ShaderToy, credits in fondo alla sezione. Questo filtro è utilizzato principalmente per noise reduction, attraverso la "convolution" tra l'immagine ed una funzione Gaussiana, dipendente in particolare dalla costruzione di un kernel ed un fattore sigma di "ampiezza". Questa implementazione in particolare non è sicuramente delle più veloci, e la sua esecuzione varia di molto al variare della dimensione del kernel (di un fattore molto minore invece al variare del valore *sigma*). Dopo diversi tests effettuati ho verificato che questa implementazione non è ragionevolmente scalabile, nonostante funzioni bene per test locali come quelli di questo homework:

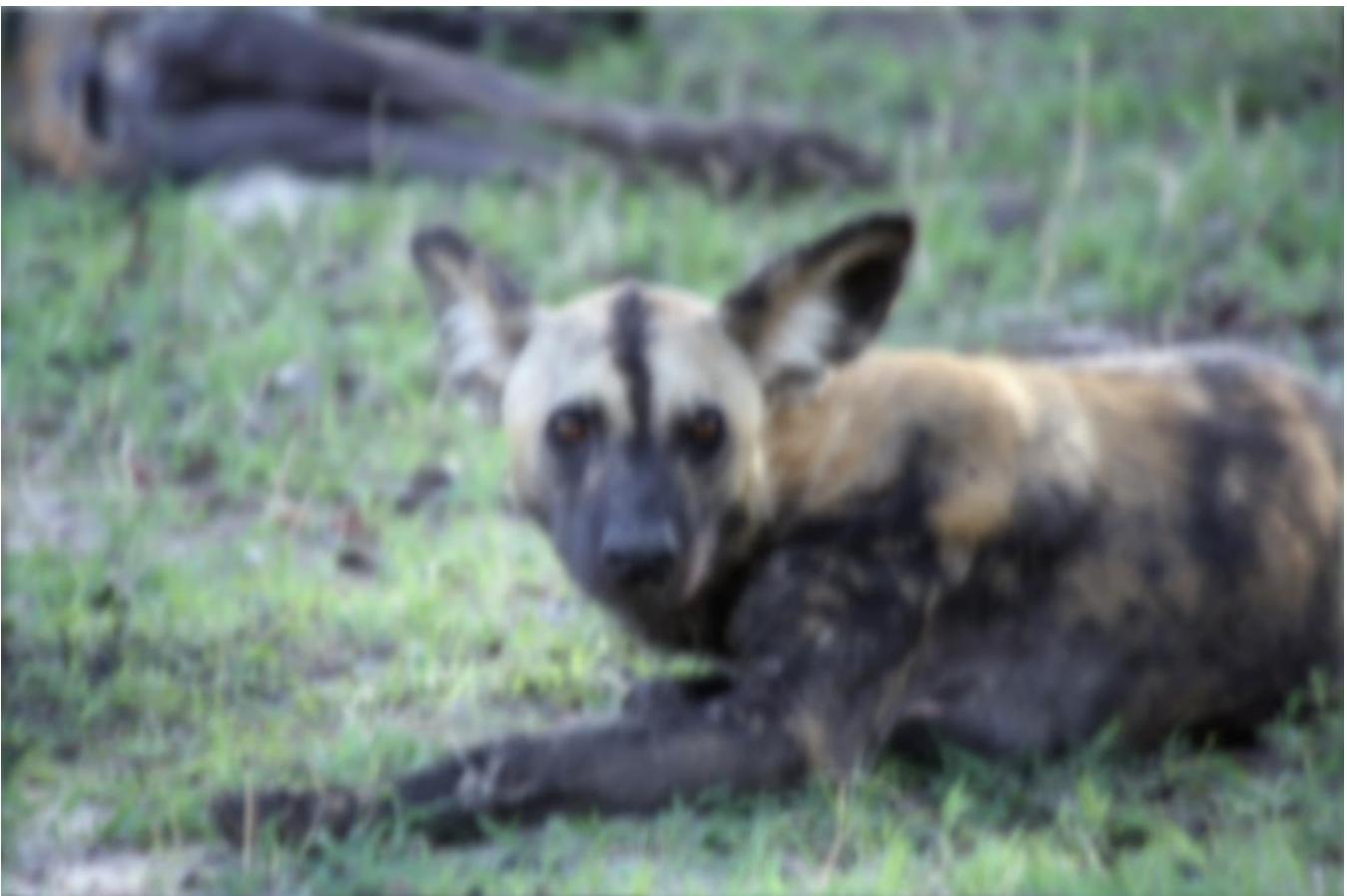
- La dimensione dell'immagine impatta significativamente le prestazioni, viene usata infatti come immagine di esempio un'immagine JPEG di un licaone, di dimensioni 1030x687
- Il valore mSize (da cui deriva anche la grandezza del kernel) impatta in maniera molto più significativa del valore sigma: per un valore mSize di 11, vengono facilmente generate in tempo ragionevole tre immagini con valori sigma 0.5, 7 e 15. Per un valore mSize di 13 invece, vengono generate in tempo ragionevole solo le immagini con fattore sigma 0.5 e 7. Il valore sigma 15 è stato infatti stoppato

Decido quindi di generare in output per dimostrazione dimensioni più ragionevoli, con mSize = 13 (definito all'interno della funzione di applicazione del Gaussian Blur) e con sigma = 1, 7, 11 passabili da linea di comando con comando --gaussian. Riporto subito sotto i risultati dell'applicazione del filtro, l'immagine su cui applicheremo il Gaussian Blur per questo esempio è la seguente: di dimensioni 1030x687.





Applicando il Gaussian Blur con fattore sigma = 1 per ottenere una sfocatura minima, La sfocatura è poco visibile ma è presente, ci servirà per visualizzare ancora meglio la differenza che il fattore sigma comporta nell'applicazione del filtro Gaussian Blur.



Qui utilizziamo il fattore sigma = 7 per la sfocatura:

La differenza è ora molto più palese. Riporto in ultimo l'esempio di sfocatura con fattore sigma = 11 per mostrare come questa piccola differenza (di solo 4) sia visibile ma non troppo significativa, aumentando però in maniera importante il fattore di tempo necessario per generare l'immagine in output. La maggior "quantità" di blur è più visibile ai bordi delle macchie nere del licaone.



CREDITS

L'algoritmo per l'applicazione del Gaussian Blur è stato implementato adattando il codice consultabile alla seguente pagina di Shader Toy: <https://www.shadertoy.com/view/XdfGDH>.

Credits all'utente **mrharicot** per l'implementazione. Non sono state usate librerie esterne.

EXTRA CREDIT DIFFICILE(?)

Cross-Hatching.

Leggendo le specifiche di un Local Laplacian Filter, ed in generale degli algoritmi di Edge Detection, ho trovato interessante diverse applicazioni molto pratiche di questo filtro. Tra le diverse alternative, quella che mi è più piaciuta (sia a livello estetico che come concept) è l'applicazione di alcuni aspetti di Edge Detection per creare un effetto Crosshatching, motivo per cui ho deciso di implementare questo filtro piuttosto che la sola Edge Detection/LoG/DoD. Come per le altre sezioni, i credits sono alla fine della spiegazione.

Questo filtro si rifà ad una vera e propria tecnica artistica chiamata, appunto, Hatching. Traducendo da wikipedia: <l'Hatching è una tecnica artistica usata per creare shading effects attraverso la rappresentazione di linee parallele molto vicine tra loro. Quando queste linee parallele sono poste ad un certo angolo con altri set di linee parallele chiamiamo questa tecnica cross-hatching.>

Gli aspetti di Edge Detection utilizzati nell'algoritmo riguardano il calcolo di alcuni parametri di posizione per comparare il pixel preso in considerazione con un altro pixel di coordinate differenti, ricavato applicando la funzione built-in di eval_image ad un punto uv. Più nello specifico, questo filtro segue questi step:

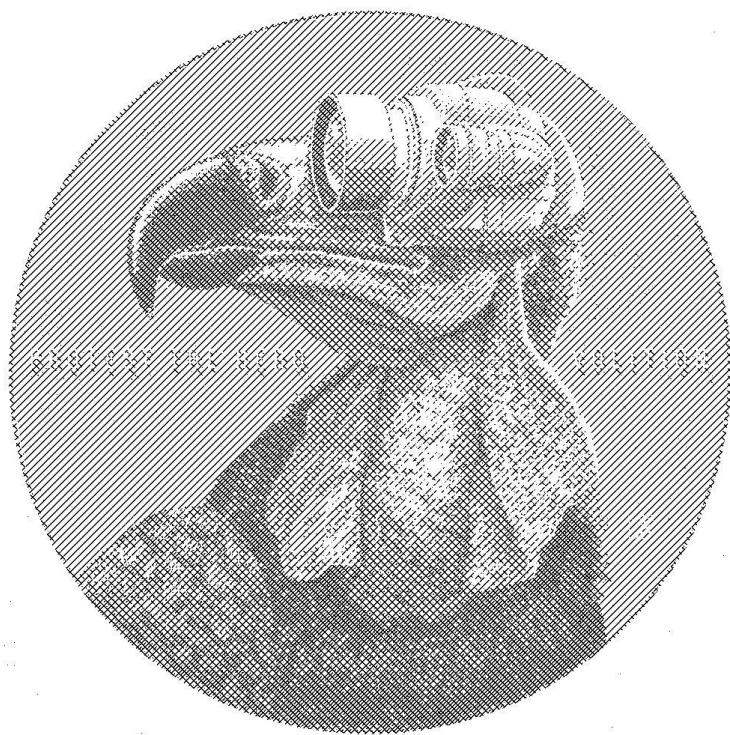
- 1) Calcoliamo un punto uv utilizzando le coordinate del pixel traslate attraverso questa trasformazione: $\{x / \text{larghezza}, (y / \text{larghezza}) / \text{ratio}\}$, dove ratio è il rapporto tra la larghezza e l'altezza dell'immagine
- 2) Trovato il colore del pixel nel nuovo punto attraverso eval_image(immagine, uv), calcoliamo la sua luminosità come somma pesata dei canali RGB. Questa somma è definita come *luma*
- 3) La luminosità verrà comparata ai diversi valori di hatch_brightness definiti (quattro in tutto). In base alla soglia, il nuovo colore del pixel viene calcolato attraverso una funzione di modulo che dipende dalle coordinate del pixel e dalla densità scelta per l'hatching. Le soglie sono definite in maniera differente per la versione grey-scale e la versione colorized del filtro
- 4) Se si utilizza il grey-scaling il nuovo colore sarà proprio la soglia hatch_brightness definita, altrimenti verrà eseguito il prodotto tra i canali rgb trovati nel passo 2) e la soglia definita

Molti parametri che riguardano l'effetto di Cross-hatching sono definiti in *grade_params*. In particolare, quelli che impattano di più la "resa visiva" del filtro sono i quattro valori hatch_n ed i valori di density e width. Data la maggior difficoltà di questo filtro ho effettuato diversi test per provare l'efficienza e la resa visiva. Dal punto di vista dell'efficienza il filtro termina sempre in tempi ottimi, ed è quindi ragionevolmente scalabile anche per immagini di maggiori dimensioni. Di seguito mostro alcuni test effettuati con diversi valori di density e width del cross-hatching. L'immagine di base è la seguente, di grandezza 1920x1080:



È stata scelta appositamente per il colore molto vicino al bianco intorno al cerchio centrale, in quanto questo filtro rende meglio su immagini poco "confusionarie", con dettagli concentrati in spazi ragionevoli dell'immagine. Nonostante la cover art sia dettagliata in sé, le sue forme si prestano particolarmente bene all'algoritmo di cross-hatching.

DENSITY 10.0f -- WIDTH 1.0f



Questo primo risultato è prodotto in grey-scaling, bypassando quindi il colore dell'immagine originale. Overall l'immagine risulta molto ben definita e distinguibile. Nel secondo risultato mostrato sotto è stato invece tenuto conto del colore. Il nuovo colore viene calcolato come descritto nell'elenco numerato riportato sopra, step 2) - 3).



Come visibile, la maggior chiarezza dei colori fa “perdere definizione” all’immagine, che risulta molto meno visibile rispetto alla versione in grey-scaling. Subito sotto riporto in serie alcuni output generati testando diversi valori di densità ed ampiezza delle linee parallele usate per il filtro.

Nei diversi test che ho effettuato, risalta in particolare il fatto che diminuendo la densità ed aumentando l’ampiezza, possiamo aumentare il livello di dettaglio, dando più o meno l’effetto di cross-hatching voluto. Manipolando correttamente questi valori è quindi possibile effettuare un fine-tuning molto accurato. Nelle pagine seguenti riporto diversi esempi di density e width. In casi “estremi” il grey-scaling scurisce molto l’immagine, ma fa parte del funzionamento del filtro a tutti gli effetti, motivo per cui come specificato è importante il fine-tuning di questi due valori per l’effetto desiderato.

DENSITY = 5.0 -- WIDTH 2.0



Nel grey-scaling in particolare è molto visibile la struttura con linee parallele. Il livello di dettaglio rimane comunque alto, anche più delle prime immagini mostrate



È meno notabile l'effetto cross-hatching dati i colori, ma è comunque visibile come sia molto più chiara dell'immagine originale

DENSITY = 2.0 -- WIDTH 2.0

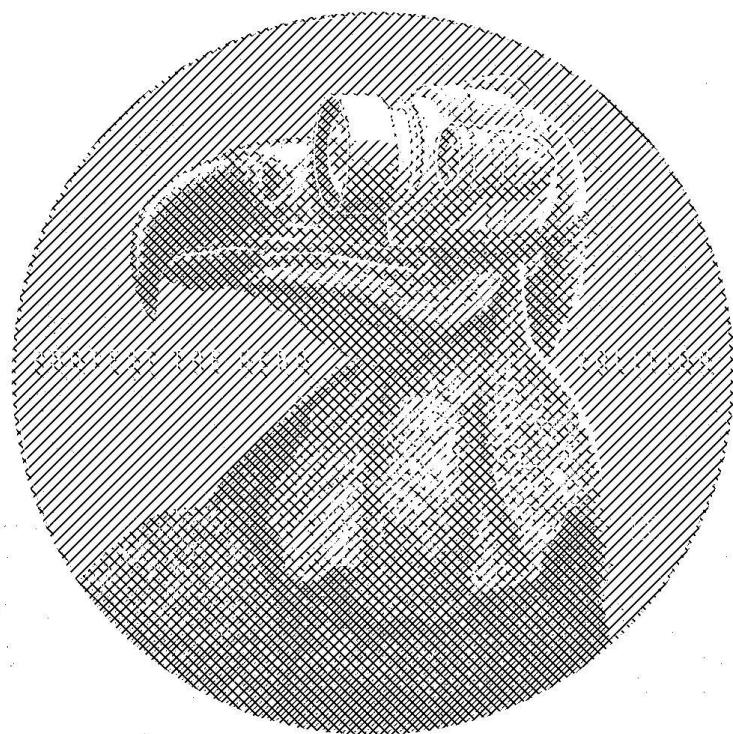


Come detto prima, “esagerando” con alcuni parametri (in questo caso il density) è possibile ottenere effetti simili. Non è necessariamente sbagliato vista l’implementazione, ma dipende dall’effetto che vogliamo ottenere.



Come visibile, una minore densità consente anche di mantenere meglio il colore originale dell’immagine che altrimenti verrebbe “disperso”

DENSITY 15.0f -- WIDTH 2



In questo caso l'effetto cross-hatching è molto più visibile



DENSITY 15.0f -- WIDTH 0.5f



La minore larghezza delle rette parallele comporta un tratto molto più leggero, rendendo l'immagine meno visibile



Quanto detto sopra è ancora più visibile nell'utilizzo dei colori

CREDITS

Per l'implementazione è stato adattato il codice dalla seguente pagina ShaderToy, con alcune variazioni: <https://www.shadertoy.com/view/MdX3Dr>, credits all'utente **doomedbunnies** per l'implementazione. In particolare, non sono stati inclusi la funzione lookup e l'implementazione del Simple Sobel, sia per problemi di implementazione, sia per mia preferenza visiva del risultato finale. È stata inoltre effettuata un'implementazione manuale della funzione *mod* usata nella pagina ShaderToy (modOpenImplementation nel codice) in quanto presente in OpenCV (in ogni caso è una semplice implementazione $x - y * \text{floor}(x/y)$).

L'immagine usata è la copertina dell'album Volition (2013), dalla band progressive-metalcore/mathcore americana Protest The Hero, credits a Jeff Jordan per l'illustrazione.