

Rapport du projet Minigimp

Jules Fouchy et Antoine Libert

1 Fonctionnalités implémentées

Demandées :

Fonctionnalité	Fait	Pas fait	Commentaire
ADDLUM	x		
DIMLUM	x		
ADDCON	x		
DIMCON	x		
INVERT	x		
SEPIA	x		Utilise les deux fonctions ci-dessous, avec des paramètres bien choisis pour le changement de couleur.
GRAYSCALE	x		
COLORCHG	x		

Bonus :

Fonctionnalité	Fait	Pas fait	Commentaire
ADDBLACKS	x		Assombrit les parties sombres, n'affecte pas les claires
DIMBLACKS	x		Éclaircit les parties sombres, n'affecte pas les claires
ADDWHITES	x		Éclaircit les parties claires, n'affecte pas les sombres
DIMWHITES	x		Assombrit les parties claires, n'affecte pas les sombres
SAT	x		Saturation
MIRROR	x		Miroir vertical
VIG	x		Vignettage
BLUR	x		
BLURGRAD	x		Degradié de flou en forme d'ellipse
CONvCONV	x		Contraste fait avec un filtre de convolution
SHARPEN	x		
EDGES	x		
TEMPERATURE	x		Température de couleur
THRESH	x		Seuil, renvoie une image avec seulement deux couleurs
KMEANS	x		Utilise k-means pour représenter l'image avec seulement k couleurs

2 Chargement et sauvegarde des fichiers PPM, structure Image et LUT

2.1 Fichiers PPM

Nous avons essayé de faire les fonctions nous-même, elles marchaient presque. Ci-dessous on peut voir une image et ce à quoi elle ressemble après l'avoir ouverte et réenregistrée : le début marche bien, mais un décalage apparaît, et ça finit mal. Nous n'avons pas bien compris où était le hic, et avons fini par aller prendre le code sur internet.



2.2 Structure Image et LUT

Pour des questions de praticité, les LUTs sont stockées directement dans la structure Image. Cela permet de n'avoir qu'un paramètre à passer aux fonctions, au lieu de (Image *im , LUT *lut). Plus précisément le champs luts est un int[3][256], qui stocke les LUTs pour le rouge, le vert et le bleu. C'est plus pratique que d'avoir trois champs LUTRouge , LUTvert et LUTbleu car on peut faire une boucle sur c allant de 0 à 2 et appliquer les modifications à luts[c] ; on évite ainsi d'écrire trois fois la même ligne quand la fonction est la même pour les trois canaux de couleur (ce qui est presque toujours le cas).

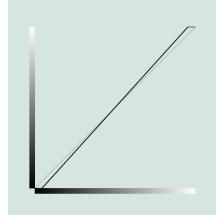
Première petite maladresse : nous avions commencé par faire des LUTs contenant des *unsigned char*, puisque c'est comme ça que sont stockés les pixels dans le tableau "pixels" de l'image ; mais cela causait de la perte d'information. En effet si par exemple on applique une fonction qui soustrait 10 à tous les pixels, alors tous ceux entre 0 et 10 se retrouvent écrasés à 0 à cause des *unsigned char* ; et si ensuite on applique une fonction qui éclaircirait les pixels sombres, tous les pixels entre 0 et 10 qui ont été écrasés à 0 vont être modifiés de la même manière.

Nous sommes donc passés à des LUTs contenant des *int*, et c'est seulement au moment d'appliquer la LUT finale à l'image que les valeurs négatives sont écrasées sur 0, et similairement pour 255.

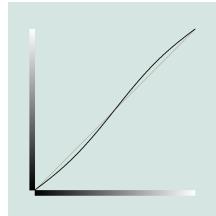
Pour la composition des LUTs, elle se fait au fur et à mesure : quand on crée une Image, on initialise ses LUTs avec l'identité (*i.e.* luts[c][k] = k), et chaque fonction comme *addLuminosity* modifie le champs luts (luts[c][k] += 10 par exemple). Tant qu'on ne fait pas d'historique, on se fiche de garder les LUTs séparées.

3 Transformations utilisant une LUT

Deuxième petite maladresse : nous avons commencé par utiliser des fonctions affines, par exemple pour le contraste :



Le gros problème avec ces fonctions est qu'aux extrémités on ne peut pas éviter de déborder de $[0, 255]$, et donc de perdre du détail. Pour le contraste nous sommes donc passés à une courbe en "s", bien connue des photographes pour, justement, ajouter du contraste :



La première manière qui nous soit venue à l'esprit pour faire cette courbe a été d'ajouter à l'identité une fonction sinusoïdale valant $-\lambda$ en 0, 0 en 128, et λ en 255 (où λ est le paramètre donné à la fonction *contrast*).

Pour la luminosité nous avons néanmoins gardé une fonction affine ($x \mapsto x + \lambda$) par souci de simplicité et parce que l'effet était suffisamment bien ainsi.

De même l'inversion, qui est par définition $x \mapsto 255 - x$, est affine.

Étant lancés sur cette idée de penser les effets en terme de courbes, nous avons enchaîné avec des fonctions n'affectant que les parties sombres ou les parties claires, grâce à des exponentielles décroissantes : $x \mapsto x \pm xe^{-\lambda(\frac{x}{255})^\alpha}$ pour éclaircir ou assombrir les parties sombres, et symétriquement $x \mapsto x \pm (255 - x)e^{-\lambda(\frac{255-x}{255})^\alpha}$ pour les parties claires.

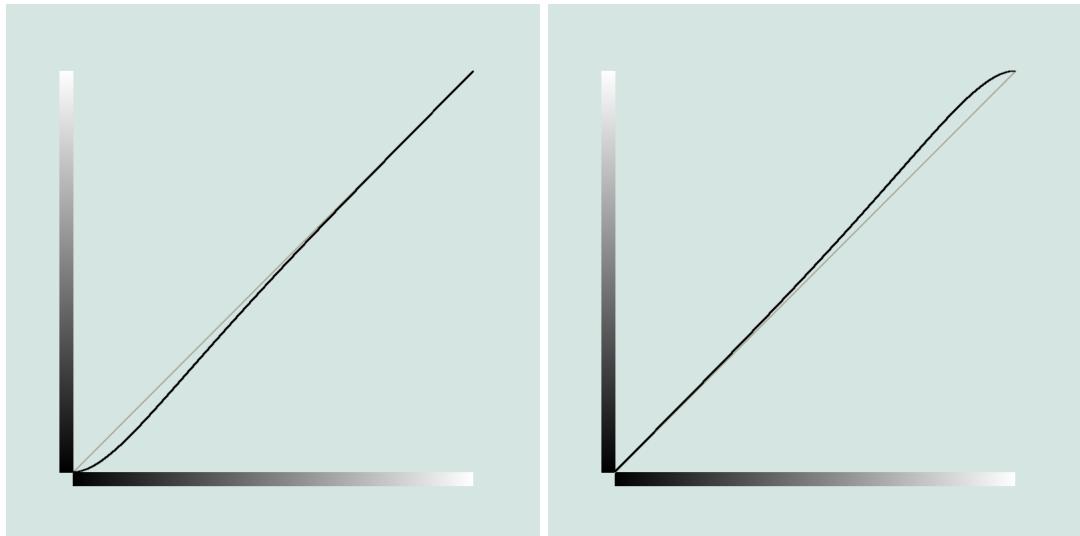




Image originale , *addBlacks* , *addWhites*

addContrast , *addBlacks&addWhites*



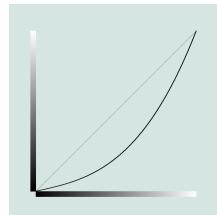
4 L'effet sépia

Pour faire l'effet sépia on passe d'abord l'image en noir et blanc, puis on augmente le rouge et diminue un peu le vert et encore un peu plus le bleu.

Là aussi nous avions commencé par des transformations linéaires sur les canaux de couleur, mais on perdait le blanc pur qui devenait un rouge/marron plus ou moins foncé et l'image se retrouvait décontrastée.



Nous sommes donc passés à un polynôme de degré 3 : $x \mapsto (a(\frac{x}{255})^3 + b\frac{x}{255}) \times \frac{255}{a+b}$.



Même si un degré 2 aurait suffit pour interpoler trois points ((0,0) , (255,255) et un point qui aurait indiqué à quel point 128 était diminué), nous avons préféré prendre un polynôme impair afin que le comportement hors de [0,255] ne soit pas trop absurde (les valeurs négatives restent négatives).



5 Les autres transformations

Puisque le reste des transformations dépend soit de la position du pixel dans l'image, soit des trois canaux de couleur simultanément pour chaque canal, on ne peut pas utiliser de LUT (une LUT de 256^3 de long perdrait de son intérêt). On modifie donc directement l'image.

5.1 Noir&Blanc et Saturation

Pour le noir et blanc, on calcule la valeur du pixel en niveau de gris, qui est une combinaison linéaire des trois canaux de couleur avec des coefficients pris sur internet, et on assigne ce même niveau de gris aux trois canaux de couleur.

Pour la saturation, on calcule aussi le pixel en niveau de gris, et on prend un barycentre du pixel couleur et du pixel gris.



Image originale , saturation augmentée , saturation diminuée

5.2 Effet miroir

Pas grand-chose à dire, on copie le pixel (i,j) sur le pixel $(i, \text{width}-1-j)$ (ou inversement).

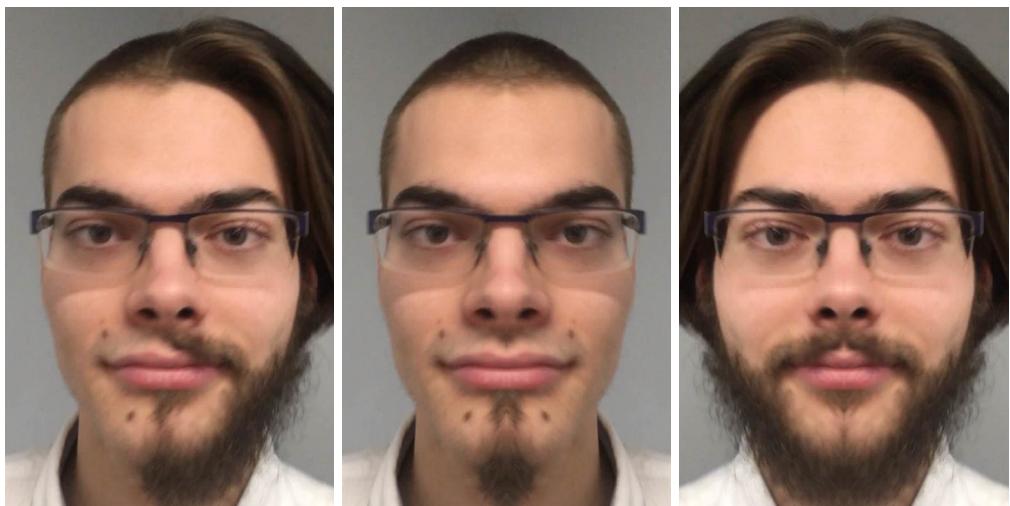


Image originale , miroir gauche , miroir droit

5.3 Vignettage

On assombrit chaque pixel proportionnellement à sa distance au centre. Pour obtenir un dégradé en ellipse plutôt qu'en cercle, on utilise la distance euclidienne mais en dilatant l'un des axes : $D = \sqrt{(x - x_c)^2 + (\frac{y - y_c}{\lambda})^2}$ où l'on prend $\lambda = \frac{\text{largeur}}{\text{hauteur}}$ pour que l'ellipse ait le même aspect que l'image.



Image originale , vignettage centré au milieu de l'image , vignettage centré sur la tête

5.4 Filtres de convolution

On ne peut pas faire la convolution en place, on crée donc une nouvelle image, et à la fin on modifie le pointeur pour qu'il pointe vers la nouvelle image.

Après avoir implémenté le produit de convolution, il suffit d'aller chercher les bons coefficients sur internet pour obtenir les différents effets : flou, contraste, sharpening, détection des contours.

Petite critique : nous nous sommes contentés d'un flou en moyenne, et on peut voir des artefacts carrés apparaître quand le flou est fort, comme dans les feuilles ci-dessous. Un flou gaussien aurait été le bienvenu.

Ensuite pour faire le flou en dégradé elliptique, on utilise exactement la même technique que pour le vignettage.



5.5 Température de couleur

On convertit une température (en Kelvin) en une couleur RGB grâce aux courbes que des gens très serviables ont mesurées expérimentalement (<http://www.tannerhelland.com/4435/convert-temperature-rgb-algorithm-code/>). Ensuite, pour appliquer la couleur à l'image, on prend un barycentre entre chaque pixel et cette couleur.



Image originale , image refroidie , image réchauffée

5.6 Effet seuil

Tous les pixels dont le niveau de gris est au-dessous du seuil sont convertis dans une couleur, et ceux au-dessus dans une autre.

Si le seuil n'est pas donné en paramètre, on calcule la valeur médiane des niveaux de gris dans l'image et on s'en sert comme seuil, de sorte qu'il y aura autant de pixels au-dessus qu'au-dessous.

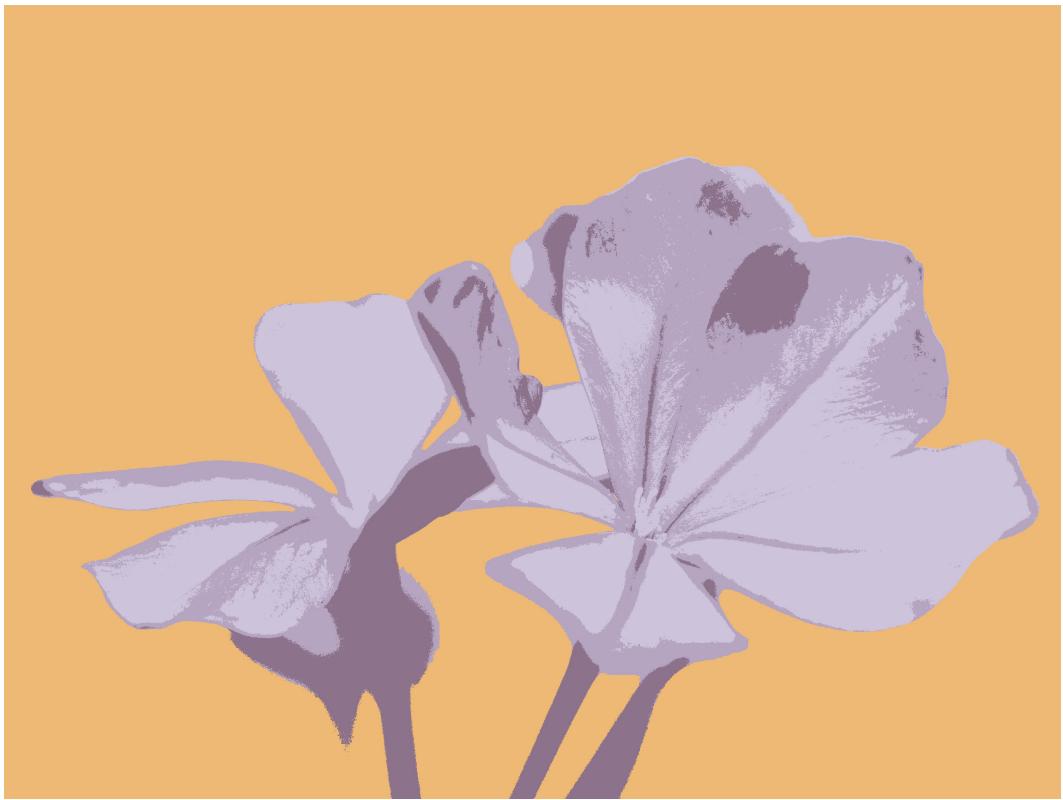


Image originale , image noir et blanc avec un seuil de 100 , image jaune et bleu avec un seuil fixé à la médiane

5.7 k-means

Une implémentation de l'algorithme de k-means dans sa forme la plus simple : on veut répartir les pixels en k groupes de couleur : on commence par choisir k couleurs au hasard pour être les centre de nos groupes, puis on détermine pour chaque pixel de quel centre il est le plus proche, et en même temps on calcule la moyenne des pixels pour chaque groupe. À la fin de l'itération ces moyennes deviennent les nouveaux centres des groupes, et on recommence à calculer à quel groupe appartient chaque pixel, jusqu'à ce que la moyenne et le centre soient très proche. À ce moment

on n'a plus qu'à changer la couleur de chaque pixel pour la couleur du groupe auquel il appartient.



6 Histogramme

Nous avons commencé par un histogramme en bâton et en noir et blanc. Ensuite pour afficher les histogrammes des trois canaux sur une même image, nous nous sommes inspirés de l'histogramme de Lightroom : l'histogramme du rouge n'affecte que le canal rouge en mettant 0 ou 255, et idem pour les deux autres canaux : le résultat est que quand les histogrammes rouge et vert se superposent on obtient du jaune, et ainsi de suite.

Notre deuxième amélioration a été de relier les sommets, afin d'avoir une courbe continue, plus esthétique.

