
Rapport de projet Retouche d'images sur Android

ROMAINPC
L3 Informatique

Projet technologique

2019-2020



Table des matières

1	Introduction :	2
2	L'application :	3
2.1	Lancement :	3
2.2	Utilisation :	3
2.2.1	Aperçu :	3
2.2.2	Actions graphiques :	4
2.2.3	Actions code :	4
3	Effets :	5
3.1	Niveau de gris :	5
3.2	Sélection de couleur :	6
3.3	Teinte :	8
3.4	Extension linéaire de dynamique :	9
3.5	Égalisation d'histogramme :	12
3.6	Flou :	13
4	Structure du projet :	14
4.1	Classe MainActivity :	14
4.2	Classe Picture :	14
4.3	Classe Utils :	14
4.4	Classes Effects et RSEffects :	15
5	Performances :	15
5.1	Temps d'exécution :	15
5.1.1	Complexité générale :	15
5.1.2	Conversion HSV :	16
5.1.3	getPixels() :	16
5.1.4	Effets avec et sans renderscript :	16
5.2	Mémoire :	19
6	Remarques et améliorations :	20
6.1	Remarques sur le code :	20
6.2	Remarques sur les librairies Android :	21
6.3	Améliorations à court terme :	22
7	Conclusion :	23
8	Annexes :	23

1 Introduction :

Premier rendu noté, ce rapport présente mon travail de réalisation d'une application de retouche d'images exécutable sur un smartphone doté du système d'exploitation Android.

Réalisé dans le cadre du Projet Technologique lors du 1er semestre de ma Licence 3 Informatique à l'Université de Bordeaux, les objectifs de ce rendu individuel étaient les suivants :

- Réaliser une interface pour afficher et modifier une image.
- Pouvoir passer l'image en niveaux de gris.
- Pouvoir teindre l'image
- Pouvoir ne garder qu'un partie de sa coloration.
- Utiliser différentes méthodes d'augmentation du contraste en manipulant des histogrammes.
- Accélérer les précédents effets au moyen de la technologie RenderScript
- Convoyer les images pour appliquer certains effets.

Lien GitHub du projet : <https://github.com/ROMAINPC/ProjetL3-Bitmap>

Sauf contre indication, les aperçus et les données d'exécutions (temps et mémoire) proviennent de l'exécution sur le Nomu S30 (Voir caractéristiques en 5.1.4).

2 L'application :

Version minimale Android requise : Lollipop (5.0) (API level 21)

2.1 Lancement :

Le projet n'a pas encore été conçu pour être disponible sous un format APK.

Le dépôt git est conçu pour être importé dans Android Studio.

Après avoir cloné ce premier rendez vous sur Android Studio puis **File > New > Import Project** et sélectionnez le dossier créé par le clone.

2.2 Utilisation :

2.2.1 Aperçu :

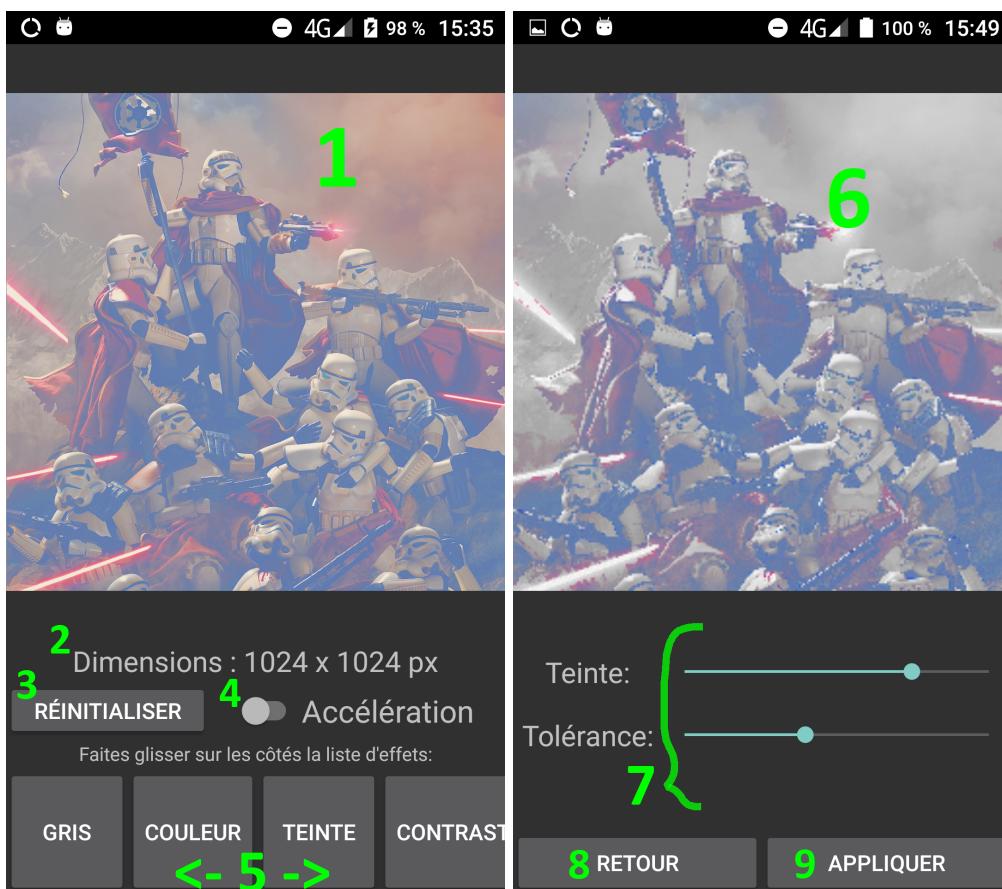


Figure 1 - Accueil.

Figure 2 - Paramètres d'effet.

2.2.2 Actions graphiques :

La figure 1 montre ce que l'application affiche au démarrage. Voici l'action des boutons :

- 1 Image affichée, les dimensions seront limitées à 3072px (avec format préservé).
- 2 Dimensions chargées (peuvent donc être inférieures au fichier original).
- 3 Bouton de réinitialisation de l'image.
- 4 Si ce Switch est activé, le rendu sera beaucoup plus rapide (technologie RenderScript). Sur certains effets uniquement.
- 5 Faites glisser cette liste d'effets latéralement, cliquez sur un effet pour passer sur la figure 2.

La figure 2 permet de visualiser un effet, de le paramétrier et de le confirmer :

- 6 Aperçu, l'image est pixelisée, c'est normal cette aperçu a des dimensions limitée à 384px.
- 7 Sliders de paramétrage, glissez les pour modifier l'effet, différent d'un effet à l'autre (il peut aussi ne pas y avoir de réglage).
- 8 Retourne à l'accueil, l'image ne subie aucune modification.
- 9 Applique l'effet sur l'image (n'est pas toujours instantané) et retourne à l'accueil, l'image affichée est modifiée.

2.2.3 Actions code :

Comme dit précédemment cette application n'est pas encore vraiment déployable, en effet vous souhaitez peut-être réaliser les actions suivantes : Pour changer l'image à modifier rendez vous dans

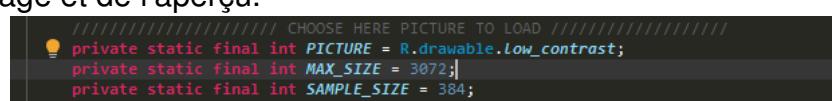
app/src/main/java/fr/romainpc/bitmapproject/activities/MainActivity.java

Et changez la première variable globale par une autre valeur de **R.drawable**.

Les images disponibles se situent dans :

app/src/main/res/drawable-nodpi

Les 2 variables en dessous permettent de changer les dimensions limites de l'image et de l'aperçu.



Il est également possible dans la méthode **applyEffect()** de modifier le type d'histogramme utilisé par les effets de contraste. Via l'énumération **Picture.Histogram**.

3 Effets :

3.1 Niveau de gris :

L'effet du bouton **GRIS** crée une image en noir et blanc, plus exactement en niveau de gris, le principe est de pondérer les canaux de rouge de vert et de bleu de chaque pixel et de les égaliser.

Il est possible de régler le poids de chaque canal pour la moyenne. Par défaut les réglages utilisent les poids recommandés par la Commission Internationale de l'Éclairage pour les écrans numériques.

A savoir : $Gris = 0.3 \times Rouge + 0.11 \times Vert + 0.59 \times Bleu$



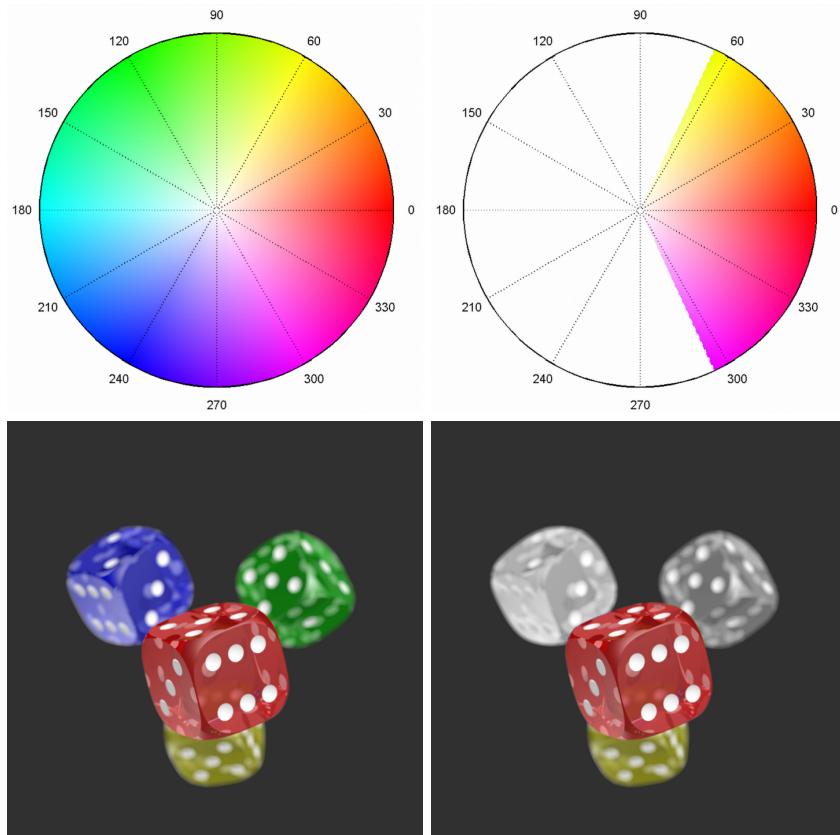
On notera que si la somme des trois valeurs n'est pas égale à 1, alors il y a soit une perte de luminosité soit une saturation des couleurs, ce qui bien utilisé peut donner certains effets :



Cet effet peut-être accéléré avec RenderScript.

3.2 Sélection de couleur :

L'effet du bouton **COULEUR** conserve tel quel les pixels d'une teinte sélectionnée (valeur H du système HSV). Les autres pixels sont grisés en annulant leur saturation (S de HSV). La teinte se représente par une valeur en degré (sur 360°), l'effet propose donc d'ajuster la teinte à conserver ainsi qu'un angle de tolérance :



Cet effet peut-être accéléré avec RenderScript.

3.3 Teinte :

Deux boutons permettent de changer la teinte de l'image, **TEINTE 1** applique la teinte sélectionnée à tous les pixels tandis que **TEINTE 2** augmente l'angle de teinte avec l'angle sélectionné. (On remarquera que glisser le bouton au maximum ne change rien puisque l'on décale de 360 °).



Ces effets peuvent-être accélérés avec RenderScript.

3.4 Extension linéaire de dynamique :

L'Extension linéaire de dynamique accessible avec le bouton **CONTRASTE** 1 est une méthode d'augmentation du contraste en "élargissant" l'histogramme de l'image.

Cette application offre 3 types d'histogrammes :

- L'histogramme sur le niveau de gris naturel (voir formule en 3.1).
- L'histogramme sur la luminance (V de HSV ou B de HSB).
- Le triple histogramme sur les canaux RGB.

On remarquera que la luminance dans l'espace de couleur HSB correspond à un niveau de gris à saturation nulle (ici sur GIMP) :



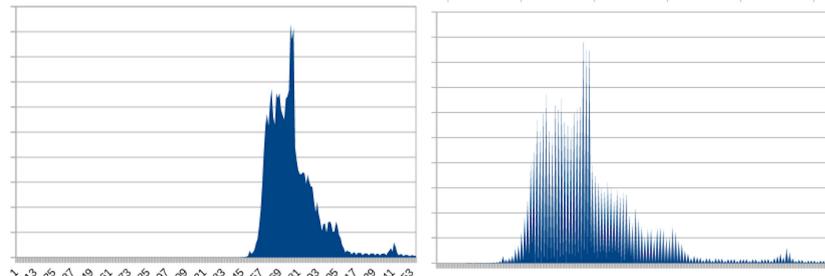
On peut supposer que c'est une raison du manque de différences entre le contraste de l'image sur l'histogramme des gris et sur l'histogramme de la luminance après avoir grisé l'image. Les histogrammes confirment cette impression.

C'est pourquoi il est recommandé d'utiliser l'histogramme sur la luminance, et de griser l'image au préalable pour obtenir une image en noir et blanc.

NB : Les aperçus des histogrammes qui vont suivre proviennent d'un tableau dans lequel a été copié les valeurs écrites dans la console. (voir la classe Utils).

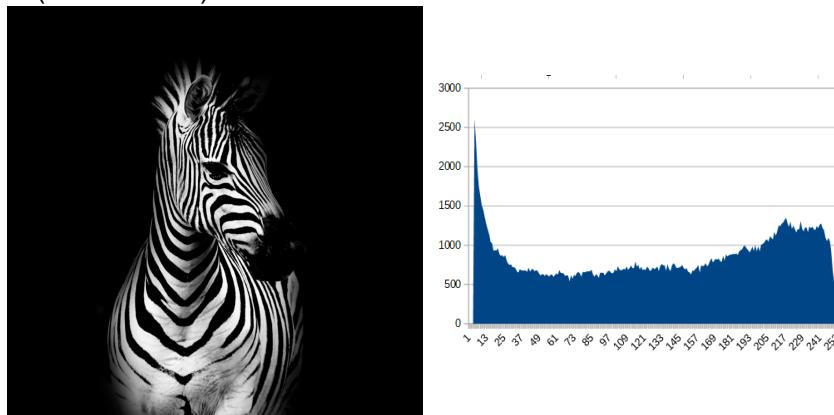


En appliquant ce contraste (sur V), on obtient ce nouvel histogramme (même échelle qu'à gauche) :



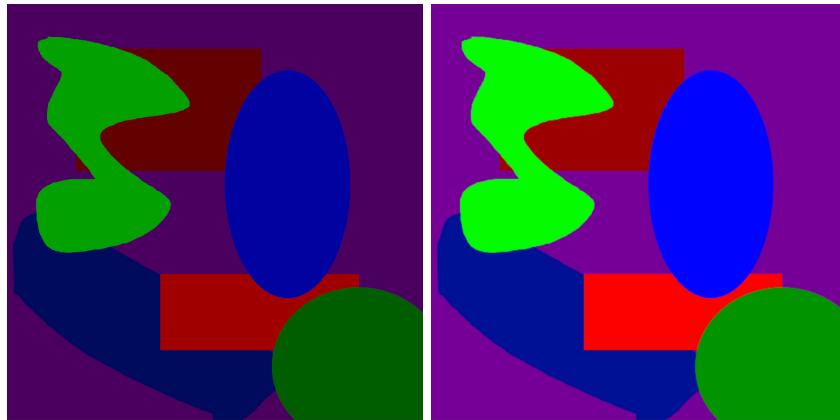
On visualise bien l'extension de l'histogramme, cependant on voit aussi la limite de cette méthode, en effet cette image ne possède aucun pixel à faible ou moyenne luminance il est donc possible de grandement écarter l'histogramme (on notera également que dans ce cas l'image s'assombrie à cause du décalage de la majorité des pixels à gauche).

En revanche sur une image possédant déjà des valeurs extrêmes (blanc et noir), il est impossible d'étendre l'histogramme, c'est le cas avec l'image suivante (1024*1024) :

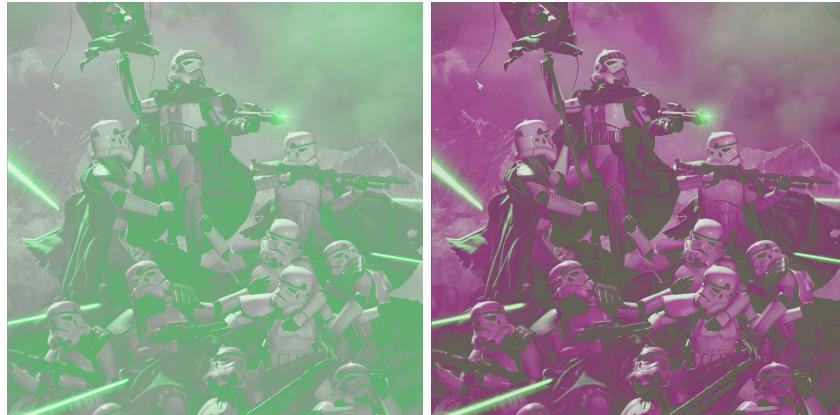


(Les pixels noirs ont été retirés de l'histogramme car trop nombreux)

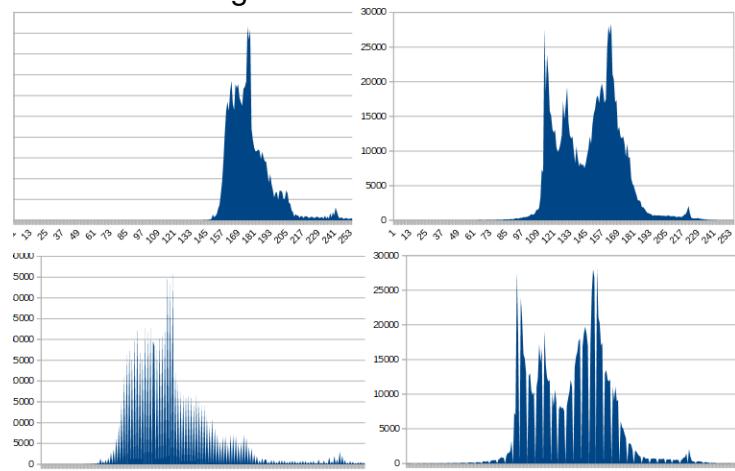
Quand aux histogrammes RGB, ils sont à utiliser avec précaution, en effet beaucoup d'images possèdent toujours au moins un pixel avec une valeur extrême pour chaque canal de l'espace RGB. Ce qui ne donne aucun effet sur les images en question. Le contraste marche un peu mieux sur l'image suivante faite à la main avec peu de luminances différentes (ici l'histogramme est décalé à droite) :



Cependant cette image possède une large gamme de couleurs, sur une image avec une forte dominance de couleur l'extension des histogrammes concernés change radicalement la couleur de l'image :



Sur les histogrammes de Vert et de Bleu on voit bien la perte du vert, quand on rouge qui n'a pas changé, il se mélange plus fortement au bleu pour donner la couleur magenta :

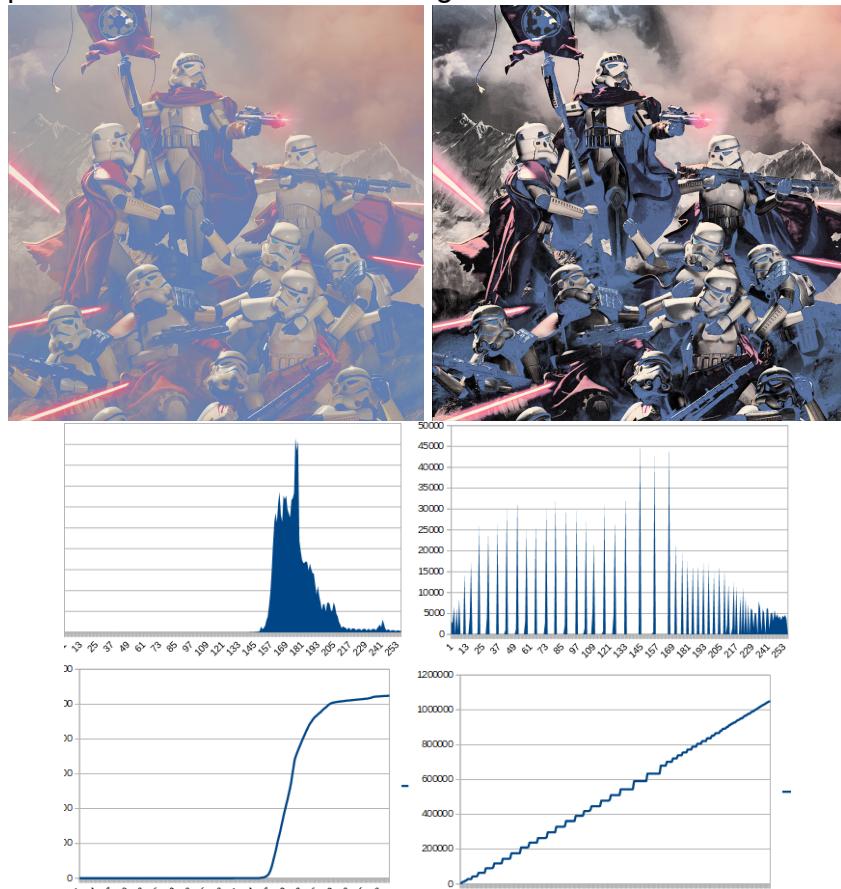


3.5 Égalisation d'histogramme :

L'égalisation d'histogramme du bouton **CONTRASTE 2** tend aussi à élargir l'histogramme, ou plutôt à "l'étaler" uniformément. Cela revient surtout à linéariser la courbe de l'histogramme cumulé.

De même que pour l'extension linéaire (3.4) l'histogramme sur le niveau de gris n'est pas vraiment utile. Le contraste marchera aussi moins bien sur une image déjà très uniforme dans sa luminance.

On peut voir un résultat avec l'image ci-dessous :



On peut voir clairement l'étalement sur toute la plage de l'histogramme et la linéarisation de la courbe des valeurs cumulées, de plus on notera que comme souvent l'égalisation tend à discréteriser les valeurs intermédiaires alors qu'il existe plusieurs pixels pour chaque valeur aux extrémités.

Voici également l'effet appliqué sur les canaux RGB :

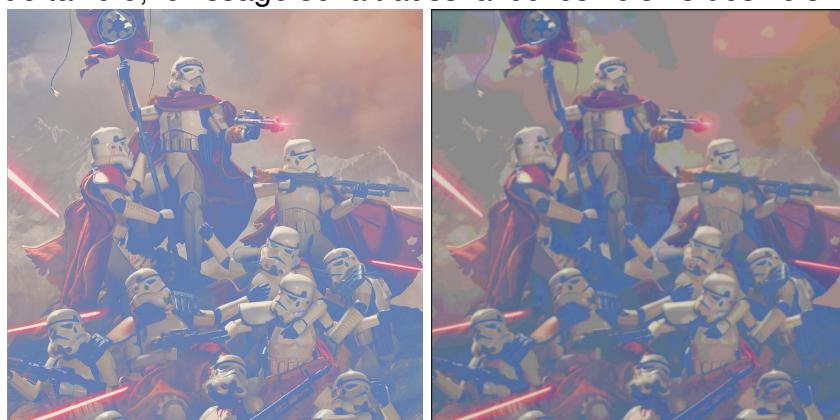


Pour cet effet aussi il faut se méfier des images à forte dominance de couleur, ici malgré une forte saturation le résultat est plutôt beau.

3.6 Flou :

Le flou via le bouton **FLOU** utilise la méthode de convolution pour lisser les pixels de l'image en les moyennant avec les pixels alentours.

Trois positions de l'intensité sont disponibles, à gauche l'effet ne fait rien, au milieu les pixels sont lissés en les moyennant avec ceux adjacents (diagonales comprises), on parle d'un "noyau" de taille 3. A droite c'est un noyau de taille 5, le lissage se fait aussi avec les voisins des voisins.



On notera l'apparition d'une bordure noire de 2 pixels, ces pixels ne sont pas lissés. La bordure est d'épaisseur égale à la taille du noyau arrondi à l'inférieur.

La convolution est très gourmande mais n'est malheureusement pas encore disponible en RenderScript. Il faut donc éviter de l'utiliser avec un noyau trop grand. (voir 5.1.1)

4 Structure du projet :

Cette section présente la structure du code du projet, réalisé en Java (et en C) il est possible de générer sa JavaDoc.

4.1 Classe MainActivity :

Classe principale du programme, c'est une Activity Android qui s'occupe donc de l'affichage graphique des composants mais qui réalise aussi les actions demandées lorsque l'utilisateur interagi avec ces composants.

Lors de la création de l'activité l'image sélectionnée est chargée sous la forme d'une instance de **Picture**(4.2). De même que l'image de petite taille qui sert d'aperçu. Une instance de RenderScript est également générée.

4.2 Classe Picture :

Cette classe encapsule principalement une instance de **Bitmap** mais en simplifie la manipulation, en effet **Bitmap** fournie par Android permet de générer et modifier une série de données en 2 dimensions, bref une image.

Les constructeurs disponibles permettent de créer une image à partir d'une autre image ou d'un fichier. Il est également possible de définir des dimensions requises (le format restera préservé).

Il est possible de réinitialiser l'image à sa valeur d'origine avec **reset()** ou de le faire en rechargeant le fichier avec **reload()**. Il est également possible de sauvegarder l'image (entraîne un fort surcoût mémoire !) avec **quickSave()** et de charger la dernière sauvegarde avec **quickLoad()**.

La classe offre aussi une méthode pour calculer et retourner l'histogramme d'une image.

4.3 Classe Utils :

Contient des méthodes statiques utiles pour factoriser le code, notamment une méthode version de la méthode **calculateInSampleSize()**, conseillée pour charger efficacement les images.

Mais fournie également les méthodes de conversion RGB \iff HSV.
(Voir 5.1.2)

4.4 Classes Effects et RSEffects :

Contiennent tout les effets, applicables sur des **Picture** ou directement sur des **Bitmap** pour la classe **Effects**.

5 Performances :

5.1 Temps d'exécution :

Toutes les mesures dans cette section ont été réalisée sur le profileur d'Android Studio.



Les temps d'exécution ont été mesurés dans le détail des Thread, **sans compter** les 200ms avant et après le temps de calcul. Ces blocs constants de 200ms environ viennent du reste du programme.



Dans le cas où un Thread lié au tas interrompt le thread de calcul, la mesure est ré-effectuée.

5.1.1 Complexité générale :

Tous les algorithmes étant basés sur un parcours d'image, on peut calculer la complexité de chaque effet.

On notera dans certains cas le besoin d'un parcours d'histogramme, ces derniers étant actuellement de taille 256, une complexité de $O(256)$ est négligeable.

Pour le reste des algorithmes, un parcours de tous les pixels de l'image est effectué, que ce soit pour appliquer un effet ou aussi générer un histogramme. Ce qui amène la complexité des effets à $O(C \times n \times m)$ soit une

complexité quadratique $O(n \times m)$ où n et m sont les dimensions de l'image.

Les deux algorithmes de contraste ne sont pas de complexité cubique du fait de l'utilisation d'une Look Up Table (LUT) intermédiaire ou d'un tableau des valeurs cumulés.

Cas de la convolution :

Pour les algorithmes de convolution c'est un peu plus compliqué, chaque pixel parcourt le noyau une fois, la complexité d'un algorithme convolutif est donc de l'ordre de $O(n \times m \times k^2)$ où k correspond à la largeur du noyau.

5.1.2 Conversion HSV :

La librairie **Color** offre des méthodes de conversion d'une couleur de l'espace RGB vers l'espace HSV et inversement. Cependant pour des raisons pas facilement identifiables avec le profileur ces méthodes sont très inefficaces. D'où l'intérêt de les implémenter soi-même. Ainsi pour appliquer l'effet de teinte sur une image carrée de 3072px (donc en passant de RGB à HSV puis RGB) il faut **10,01 secondes**, contre **53.06 secondes** pour les méthodes de la librairie **Color**.

5.1.3 getPixels() :

Il peut être intéressant de comparer l'efficacité entre les méthodes **getPixels()** et **getPixel()** (idem set).

La méthode **grayLevelOld()** parcourt l'image et accède aux pixels un par un avec les méthodes **getPixel()** et **setPixel()**.

Tandis que **grayLevel()** parcourt un tableau où sont préalablement chargés les pixels avec **getPixels()** et le réinsère dans l'image avec **setPixels()**.

Toujours sur l'image en 3072px de côté, **grayLevelOld()** applique un niveau de gris en **60 secondes** contre **2.21 secondes** pour sa version tableau intermédiaire.

Il est évident que l'ensemble de l'application utilise donc les méthodes **getPixels()** et **setPixels()**.

5.1.4 Effets avec et sans renderscript :

Le tableau suivant montre les temps d'exécutions des différents effets. Les temps sont exprimés en secondes (ces temps peuvent varier d'une

seconde d'une exécution à l'autre).

Les effets sont testés sur des images carrées de 3072 pixels (9,4Mpx), 1024 pixels (1Mpx) et 4608px (21,2Mpx).

Les tests sont effectués sur l'appareil suivant :

- Modèle : Nomu S30
- Android : Marshmallow (6.0) (API level 23).
- Affichage : 1080 x 1920 px , 480dpi (xxhdpi)
- Processeur : 8 Noyaux (arm64, Cortex-A53 (1GHz et 2GHz))
- RAM : 3883 Mo (disponible 1954)

et sur l'émulateur suivant :

- Modèle AVD : Nexus 5X
- Android : Marshmallow (7.1.1) (API level 25).
- Affichage : 1080 x 1920 px , 420dpi (xhdpi ?)
- Processeur : 4 Coeurs (limités par le PC hôte, i7-6700K)
- RAM : 1536 Mo alloués.

	Nomu S30		Nexus 5X	
Effets	1024px	3072px	1024px	3072px
Niveau de gris	0,2	2,2	0,2	0,4
Garder couleur	1,4	11,21	0,2	1,6
Teinte	1,2	10,21	0,2	1,8
Décalage de Teinte	1,2	10,61	0,2	1,6
Contraste par extension	2,4	18,42	0,4	3,0
Contraste par égalisation	2,2	18,01	0,4	3,2
Contraste par extension RGB	0,6	4,4	0,2	16,5
Contraste par égalisation RGB	0,6	4,6	0,2	18,9
Contraste par égalisation RGB	3,6	32,02	0,4	3,22

Version RenderScript :

	Nomu S30		Nexus 5X	
Effets	3072px	4608px	3072px	4608px
Niveau de gris	0,2	0,2	0,1	0,2
Garder couleur	0,6	1,21	0,6	1,21
Teinte	0,4	0,6	0,4	0,6
Décalage de Teinte	0,2	0,6	0,4	0,6

On remarquera une utilisation du processeur de 50% environ pour les mesures sans accélérations contre 80% avec RenderScript.

RenderScript est suffisamment efficace pour rendre négligeable le temps de calcul sur ces effets pour des images de tailles standards.

Avec l'image de 4608, le bouton réinitialiser met pratiquement 1 seconde.

Les valeurs de l'émulateur sont trop aberrantes pour être considérées.

En revanche pour le véritable appareil, il est intéressant de voir que de 1024px à 3072px les temps sont environ multipliés par 9. En effet il y a 9 fois plus de pixels dans la grande image que dans la petite.

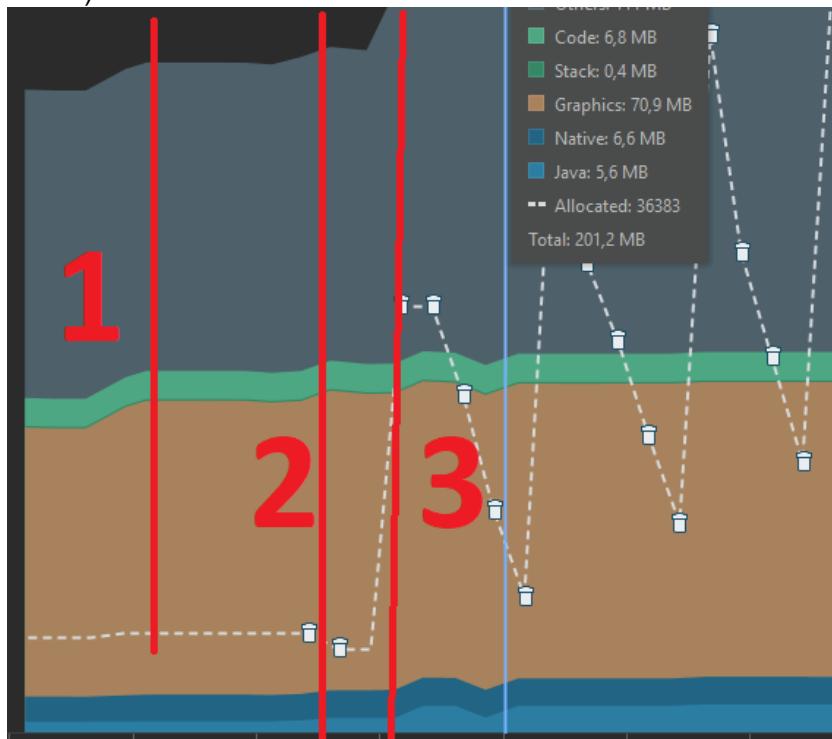
Résultat particulier : même avec 3 histogrammes les contrastes sont plus rapides en mode RGB. On peut en déduire que la conversion RGB ⇔ HSV est la méthode là plus inefficace de cette application puisqu'elle ralentie tous les autres effets.

Ce tableau ne le montre pas mais les conversions HSV et RGB ont un léger impact sur le temps d'exécution selon que les effets sont appliqués

sur des couleurs ou des niveaux de gris différents.

5.2 Mémoire :

Voici le graphe des données mémoires (Grande image de 3072px pour bien la situer) :



En 1 on peut voir dans la partie graphique la Bitmap chargée, en effet pour un image de 3072 de côté, c'est 9 437 184pxs, donc le quadruple en octets (canaux RGB et alpha). Ce qui donne théoriquement une image de 38Mo, en pratique c'est beaucoup plus (66 Mo).

On remarque également un espace mémoire de même dimension dans la partie du haut grise claire. Ce qui peut correspond à la copie de l'originale sauvegardée sous forme de tableau dans la classe **Picture**. C'est aussi peut-être une duplication de la partie orange (voir ci dessous).

Le premier décroché du graphique montre le chargement de l'aperçu en dimensions 384.

En 2 l'utilisateur a cliqué sur un bouton d'effet. Un petit décroché supplémentaire indique le chargement du tableau de **quickSave()** de l'aperçu. Cette version dupliquée de l'aperçu sera conservée jusqu'à la fin du programme.

En 3 l'utilisateur applique l'effet désiré, on remarque une petite montée de la partie bleue, les allocations mémoires utilisées pour les calculs. Ce n'est pas montré ici mais en utilisant l'accélération, la partie bleue foncée ("Natives") peut atteindre 36Mo, ce qui doit correspondre aux allocations RenderScript.

Enfin plus étonnant, la partie grise claire double de volume, c'est peut-être seulement à ce moment là qu'est réellement alloué le tableau qui stocke la version d'origine. Comme l'effet vient d'être appliqué l'image diffère enfin de l'originale. (D'autant que **quickSave()** n'est jamais appelée sur l'image principale).

6 Remarques et améliorations :

6.1 Remarques sur le code :

Du fait du comportement particulier de la librairie Bitmap d'un appareil à l'autre en fonction de sa densité de pixels, les images ont été placées dans de dossier **drawable-nodpi** ce qui évite tout sur-dimensionnement et garantit le contrôle total des dimensions de l'image. (voir aussi 2.2.3)

La génération des histogrammes est coûteuse, bien que la complexité générale ne soit pas affectée, ce sont des parcours supplémentaires de l'image qui mis bout à bout peuvent nuire à l'expérience utilisateur. La génération systématique de l'histogramme est donc très discutable, elle a été conservée car dans cette application l'image principale ne subit un effet que lorsque celui ci est confirmé, et l'histogramme doit être recalculé.

Les fonctions de génération et d'utilisation des histogrammes travaillent avec une liste de tableaux, cette solution permet d'utiliser des méthodes d'histogramme avec 1 ou 3 tableaux, mais en effet cela affecte la lisibilité du code.

Plusieurs types d'histogrammes existent, certains sont sur 3 canaux mais tous sont de taille 256. On pourrait le reprocher pour l'histogramme sur la luminance. Cependant comme de toute manière la couleur des pixels de l'écran est encodée sur 24 bits il n'y a théoriquement pas de perte d'information de RGB vers HSV.

Bien que très pratique la classe Picture a cependant un coup mémoire non négligeable, du fait de la sauvegarde permanente de l'image originale. La sauvegarde rapide double cette quantité en mémoire et doit donc être utilisée avec parcimonie. (Ici seulement pour l'aperçu).

On notera l'utilisation d'un tableau cumulé de **long** pour l'égalisation d'histogramme. Les résolutions d'images actuelles tournant dans l'ordre des millions de pixels (9.4 Mpx pour 3072px de côté), la multiplication par 255 dans la formule de l'algorithme fait sortir les valeurs des limites du **int**.

Du fait du manque de documentation de la librairie RenderScript et de la non reconnaissance des .rs par les outils classiques de formatage, les scripts RenderScript ne sont pas les plus propres du projet et sont souvent une traduction brute du code java en code C. Ils sont heureusement assez courts.

6.2 Remarques sur les librairies Android :

La librairie **Bitmap** bien que fonctionnelle n'est pas évidente d'utilisation, c'est en réalité une librairie pour charger et surtout afficher des images. Les surcouches de dimensionnement et d'interpolation nous éloigne de la simple représentation d'une image par une structure de pixels fixe en 2 dimensions.

Je n'ai pas eu le temps de vérifier le comportement de l'interpolation lors de l agrandissement de petites images, que ce soit au chargement de l'image ou à son affichage dans un **ImageView**.

Les méthodes **getPixels()** et **setPixels()** travaillent avec un tableau 1 dimension. On pourrait le reprocher à la librairie **Bitmap**. Un tableau 2D (même s'il faudrait mettre 2 for à tout les algorithmes) serait plus intuitif et permettrait en théorie de ne pas stocker de manière contiguë autant de pixels en mémoire. (Bien que je ne connaisse pas le fonctionnement de la mémoire en Java, étant gérée automatiquement (et tant mieux)).

La technologie RenderScript souffre d'un manque étonnant de documentation et de posts sur les forums, on notera également qu'une partie des fonctionnalités nécessitent Android 7.0 minimum. Bien que je n'ai pas pu approfondir leur utilité, c'est étonnant pour cette librairie qui commence relativement à dater. 58% des appareils en circulation début 2020

peuvent donc faire tourner ces fonctionnalités. Je n'ai donc pas essayé de les utiliser ayant une version inférieure.

6.3 Améliorations à court terme :

La version minimale de l'API pour cette application est la 21, (90% des appareils). Je n'ai pas réussi à l'abaisser au niveau 19 (96%) pour pouvoir tester aussi sur un ancien appareil, le problème semble toutefois venir d'un réglage d'Android Studio.

Il faudra supprimer la bordure noire pour la convolution. Mais également implémenter son renderscript, la convolution étant très gourmande. Une traduction des méthodes java en C devrait fonctionner une fois trouvé le moyen de passer des tableaux. Le noyau en lui même ne sera pas parallélisé.

Il pourra être utile d'ajouter des composants pour choisir le type d'histogramme à utiliser directement dans les réglages d'effets. De même il deviendra urgent de pouvoir choisir une image dans la galerie depuis l'application, mais aussi de pouvoir sauvegarder l'image retouchée.

L'affichage des histogrammes directement sur l'application est aussi à faire.

Il faudra essayer d'optimiser les conversions RGB \iff HSV, voir les mesures en 5.1.4

7 Conclusion :

Travailler sur des images est très intéressant, du fait du résultat immédiat et très visuel. Appréciant l’algorithmie avec un peu de mathématiques concrètes, je suis satisfait d’avoir fait ce projet. Cependant l’implémentation nous rattrape très vite, ce projet s’est avéré finalement très chronophage sur la partie Android. (les librairies et l’IDE Android confirment leur réputation d’usine à gaz).

Par ailleurs je n’ai finalement pas autant avancé ce projet que prévu, de part son statut de dernier rendu du semestre. J’ai donc préféré concentrer mes efforts sur une compréhension approfondie des librairies Android et sur l’implémentation d’une structure robuste afin de faciliter l’ajout des fonctionnalités à venir.

8 Annexes :

Représentation de la couleur :

Système de couleur RGB :

https://fr.wikipedia.org/wiki/Rouge_vert_bleu

Système de couleur HSV ou HSB :

https://fr.wikipedia.org/wiki/Teinte_Saturation_Valeur

Documentation Android :

Activity :

<https://developer.android.com/reference/android/app/Activity>

Bitmap :

<https://developer.android.com/reference/android/graphics/Bitmap> <https://developer.android.com/topic/performance/graphics>

Gestion des dimensions des Bitmap :

<https://developer.android.com/topic/performance/graphics/load-bitmap>

RenderScript :

<https://developer.android.com/guide/topics/renderscript/compute>

Utilisations des API :

<https://developer.android.com/about/dashboards>