

RAPPORT FINAL

PROJET TECHNOLOGIQUE ANDROID

Sarah CHOUROUQ

Tom DEPORTE

Vincent DUFAU

Firas MESSAOUD

Avril 2020

1 Introduction

L'application permet de réaliser de nombreuses modifications sur une image. L'ensemble de ces modifications peuvent être utilisées aussi bien sur une image en niveaux de gris que sur une image en couleur (l'effet n'est cependant pas garanti, une colorisation d'une image en niveaux de gris ne transformera pas l'image).

Pour ce projet, nous avons utilisé des librairies spécifiques :

- GraphView : cette librairie nous est utile afin d'obtenir la représentation de l'histogramme de l'image sous forme d'un graphique. Cette librairie est disponible [ici](#).
- ColorPickerView : cette librairie nous permet de faire un choix de couleur au travers d'un disque contenant l'ensemble des couleurs. Elle est disponible [à cet endroit](#).

Le smartphone utilisé lors du développement et sur lequel les tests ont été effectués est un Samsung A50 qui a comme numéro de modèle **SM-A505FN**. La version d'Android utilisée tout au long du projet est la version **9**.

2 Présentation de l'application

2.1 Liste des fonctionnalités implémentées

Les différentes fonctions et algorithmes implémentés dans l'application sont :

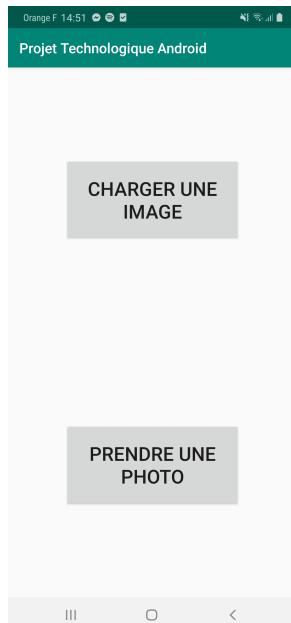
- Le choix de l'image ou la prise de photo
- Le passage en niveaux de gris
- L'inversion des couleurs
- La colorisation
- La conservation
- La modification de la luminosité
- La modification du contraste
- L'extension du dynamique
- L'égalisation d'histogramme
- Le filtre moyenneur
- Le filtre Gaussien
- Le filtre de Sobel
- Le filtre Laplacien
- L'effet croquis / crayonné
- L'effet cartoon
- L'effet de neige
- L'incrustation d'objets
- La visualisation de l'histogramme sous forme de graphique
- L'algorithme RGBToHSV recodé
- L'algorithme HSVToRGB recodé
- Le zoom et le scroll
- La possibilité de revenir en arrière ou en avant
- La réinitialisation de l'image
- La sauvegarde de l'image

2.2 Interface de l'application

L'application est divisé en 3 parties : le menu pour choisir l'image à transformer, la partie principale dans laquelle il est possible de visualiser l'image choisie au préalable et d'y appliquer des effets, et une dernière partie permettant d'afficher l'histogramme de l'image sous la forme d'un graphe.

2.2.1 Le menu de chargement de l'image

Cette vue propose à l'utilisateur de cliquer sur deux boutons différents.



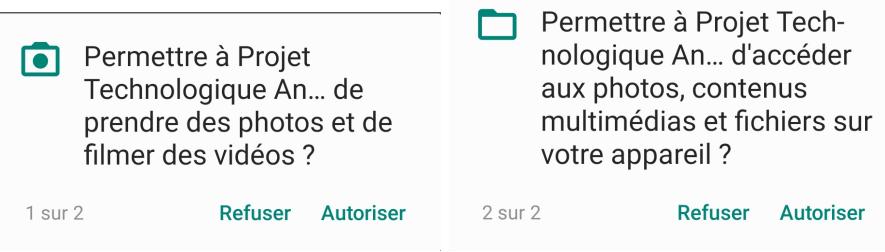
Si le premier bouton est cliqué, l'utilisateur va être redirigé vers sa galerie et va pouvoir y choisir n'importe quelle image présente.

Si le second bouton est cliqué, l'appareil photo du téléphone va s'activer et l'utilisateur va pouvoir prendre une photo qui sera utilisée ensuite par l'application.

Cette seconde option nécessite néanmoins des permissions : l'utilisateur doit permettre à l'application d'accéder à la caméra et aux fichiers internes du téléphone.

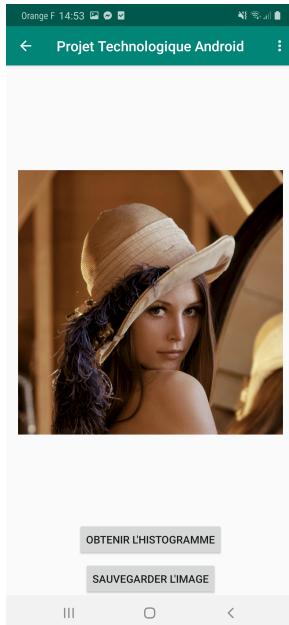
Sans ces permissions, l'utilisateur ne pourra pas prendre de photo avec l'application et sauvegarder les images qu'il aura modifié.

Permissions pour la caméra et la sauvegarde refusées



2.2.2 La vue principale de l'application

L'écran est divisé en deux "parties" différentes, l'image et des boutons permettant de faire des actions. L'image prend la majorité de l'écran et il y a au maximum 4 boutons disponibles (2 au minimum).



Le bouton "obtenir l'histogramme" permet de changer de vue afin de visualiser le graphique représentant l'histogramme de l'image.

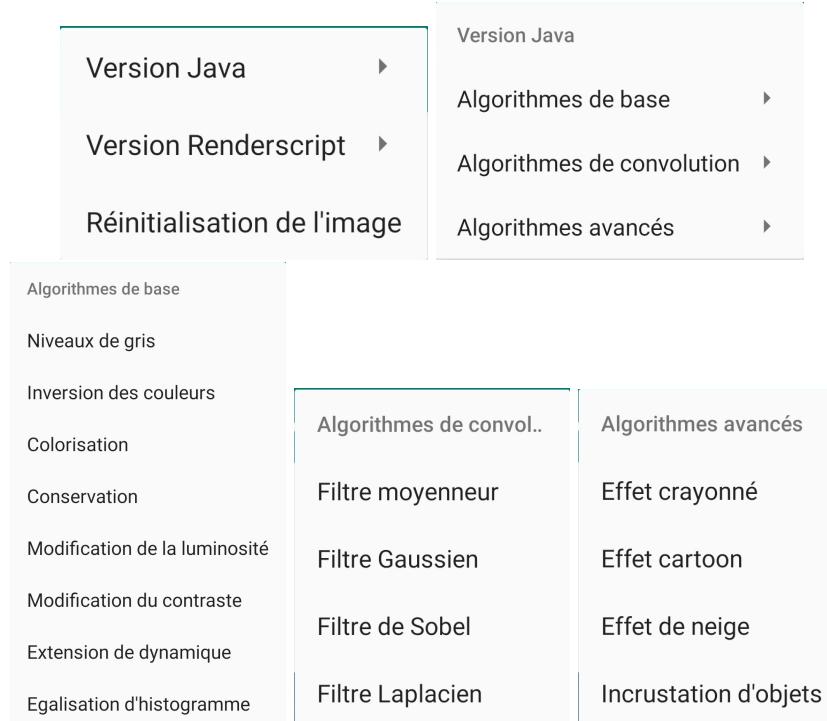
Le bouton "sauvegarder l'image" permet d'effectuer la sauvegarde de l'image. Si les permissions n'ont pas été accordées au préalable, elles sont demandées à l'utilisateur comme expliqué précédemment.

Les deux autres boutons ne sont pas toujours visibles. Il s'agit de boutons permettant de revenir en arrière ou en avant par rapport aux manipulations qu'a réalisé l'utilisateur. Si aucune manipulation n'a été effectuée ou que le bouton revenir en arrière a permis de retourner à l'état initial de l'image, il n'est alors plus possible de revenir en arrière et le bouton n'est plus disponible

et visible. Pour ce qui du bouton retour en avant, il n'est disponible et visible que lorsque l'utilisateur retourne en arrière. A chaque nouvelle manipulation, ce bouton disparaît, même si l'utilisateur était au préalable revenu en arrière.



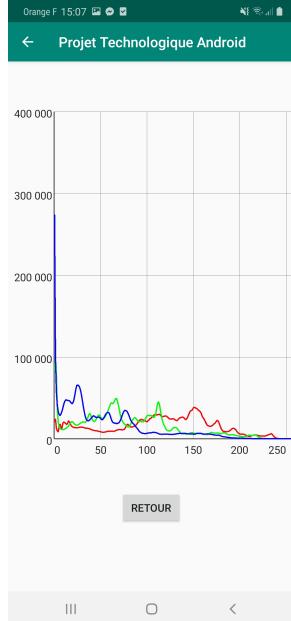
Pour ce qui est des fonctions implémentées, elles sont disponibles grâce à un menu en haut à droite de l'application. Ce menu est divisé en sous-menus qui permet de diviser les parties Java et Renderscript de l'application. Chacun de ces sous-menus sont eux-mêmes divisés en sous-menus permettant de diviser les phases d'évolution du projet.



2.2.3 Le graphique de l'histogramme

Lorsque l'utilisateur clique sur le bouton "obtenir l'histogramme" de la vue principale, il est redirigé vers une nouvelle vue qui permet uniquement d'observer l'histogramme de l'image avec toutes les modifications faites.

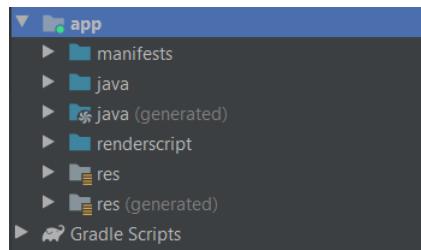
Ce graphique est réalisé à l'aide de la librairie GraphView comme précisé dans l'introduction.



Dans le cas d'une image en couleurs, 3 courbes seront visibles (une par canaux rouge, vert et bleu). Dans le cas d'une image en niveaux de gris, ces 3 courbes se superposent et seule une courbe bleue est visible. Le bouton "retour" permet de revenir à la vue principale de l'application.

3 Organisation du code

L'architecture générale du code de l'application est la suivante :

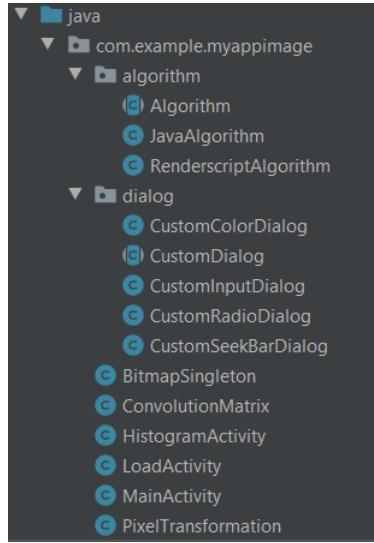


Le dossier java contient tout les fichiers .java, que ce soit les algorithmes, les activités ou les appels aux fonctions renderscript.

Le dossier renderscript contient tout les fichiers .rs qui correspondent aux versions renderscript des algorithmes de traitement d'image.

Le dossier res contient tout ce qui a un rapport au graphisme de l'application comme les layouts, le menu, les strings ou encore les images utilisées dans l'application (boutons retour arrière et avant, incrustation d'objets).

Le dossier java est le plus important en terme de contenu et possède l'architecture suivante :



3.1 Le package Algorithm

Ce package contient tout ce qui a rapport aux algorithmes, c'est-à-dire aux différents traitements d'image disponibles dans l'application.

On peut y retrouver 3 classes : Algorithm, JavaAlgorithm et RenderscriptAlgorithm.

La classe Algorithm est une classe abstraite dont hérite les deux autres classes. Elle permet notamment aux deux autres classes d'avoir un comportement similaire et de réduire la duplication de code entre ces deux classes. Cette classe va permettre aux deux autres classes de posséder les mêmes méthodes (à l'exception de l'incrustation d'objets).

La classe JavaAlgorithm regroupe tout les traitements d'image fait en code Java et la classe RenderscriptAlgorithm regroupe tout les traitements d'image fait en Renderscript.

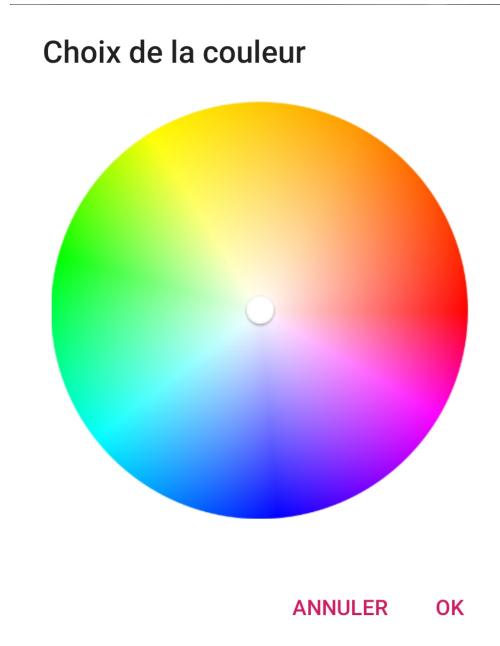
3.2 Le package Dialog

Ce package va permettre de créer plus facilement des popups. Au lieu de créer un nouveau Dialog et d'écrire de nombreuses lignes de codes similaires à chaque fois que c'est nécessaire dans l'application, ce package permet de simplifier grandement l'utilisation des Dialog et de rendre le code plus propre.

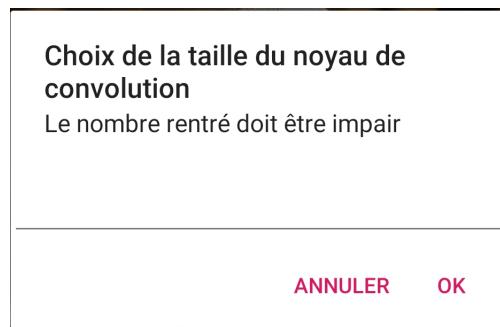
La classe CustomDialog est une classe abstraite. Toutes les autres classes héritent de CustomDialog ce qui permet pour chaque type de Dialog différents d'avoir le même comportement et la même instanciation.

Pour l'ensemble des traitements d'image implémentés dans l'application, 4 popups totalement différents ont été nécessaire. Chacun de ces popups sont liés à une classe spécifique de ce package.

Ainsi, la classe CustomColorDialog permet de créer à l'aide de la librairie ColorPickerView un popup permettant de choisir une couleur qui sera par la suite renvoyée et utilisée pour les traitements.



La classe CustomInputDialog laisse à l'utilisateur la possibilité de rentrer une valeur qui aura une conséquence sur les algorithmes.



La classe CustomRadioDialog donne à l'utilisateur le choix entre différentes propositions. Ces propositions sont retransmises sous la forme d'une liste avec

un bouton lié à chaque item de la liste. Il ne peut choisir qu'une seule proposition.



Le dernier type de Dialog existant dans l'application est le CustomSeekBarDialog. Ici, l'utilisateur va pouvoir choisir une valeur dans un certain intervalle en bougeant un point sur une droite représentant l'ensemble des valeurs possibles.



3.3 Les autres classes de l'application

Le dossier java est également composé de 6 autres classes. Trois d'entre elles sont des Activity et chacune de ces Activity correspond à une vue différente de l'application.

Ainsi, la classe LoadActivity correspond à la première vue de l'application, celle avec les 2 boutons pour choisir une image. Cette classe est responsable de la gestion du choix de l'utilisateur par rapport aux boutons et de la création de la Bitmap liée à l'image choisie ou prise à l'aide de la caméra.

La classe MainActivity correspond à la vue principale de l'application où l'on retrouve l'image, le menu comprenant les traitements et des boutons décrits auparavant. Elle gère l'appel aux différents boutons et surtout elle gère le menu permettant d'utiliser les algorithmes afin de transformer l'image.

La classe HistogramActivity est la dernière Activity de l'application et elle permet de visualiser l'histogramme de l'image.

Ensuite vient la classe BitmapSingleton, elle permet d'avoir un singleton représentant la bitmap de l'image utilisée dans l'application. Cette classe est

utile afin de pouvoir utiliser la bitmap dans différentes classes et notamment dans les différentes activités de l'application.

La classe PixelTransformation est une classe statique, sans constructeur. Toutes ses méthodes sont statiques. Elle regroupe les méthodes permettant de transformer un pixel en un autre grâce à un calcul où les transformations de modèles de couleur. On va donc pouvoir retrouver dans cette classe la méthode permettant de transformer en pixel en niveaux de gris et les méthodes permettant de passer un pixel de rgb à hsv ou de hsv à rgb.

La classe ConvolutionMatrix est la classe responsable des traitements de convolutions. Elle va effectuer les calculs de convolution afin de retourner les valeurs transformées afin de changer l'image selon le masque de convolution lié au traitement choisi par l'utilisateur.

4 Traitements disponibles

A l'exception de l'incrustation d'objets, tout les traitements ont été réalisés à la fois en Java et en Renderscript.

Les temps peuvent parfois paraître long l'utilisation du profiler en est la cause. Les traitements sont exécutés plus rapidement sans le profiler.

4.1 Le passage en niveaux de gris

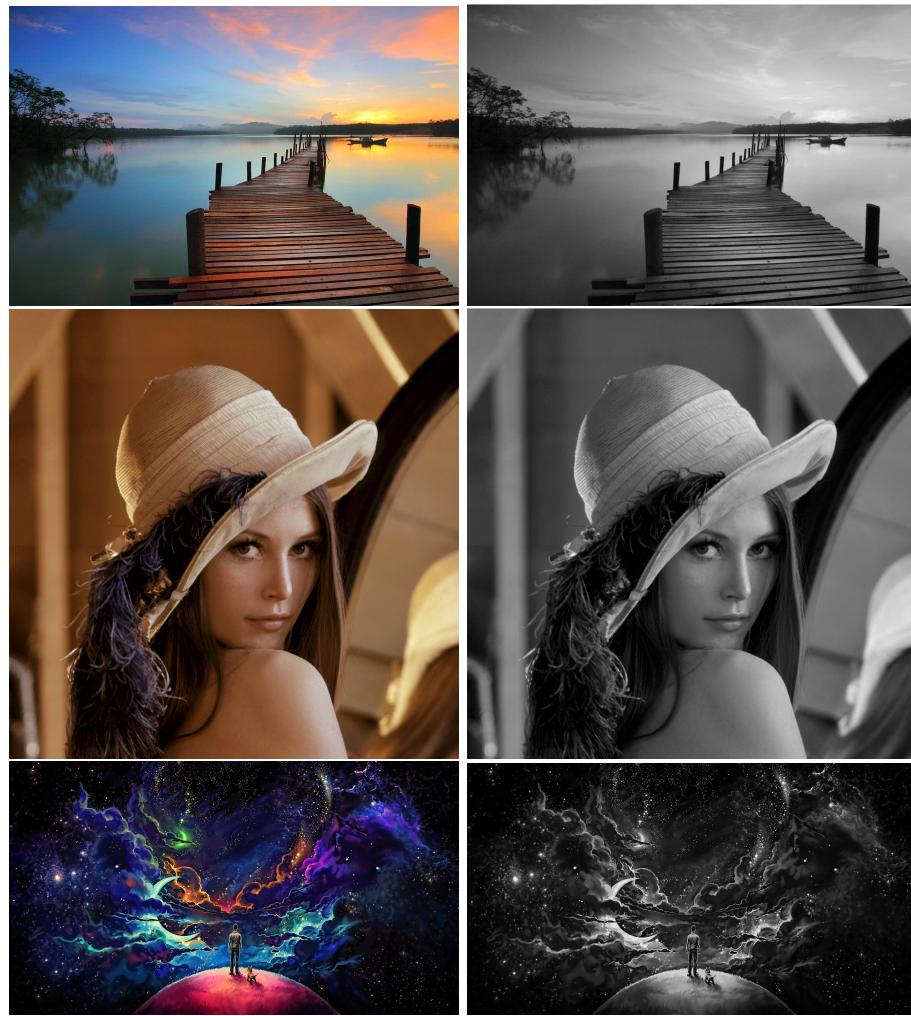
4.1.1 Présentation

Le passage en niveaux de gris est un algorithme simple visant à transformer l'image dans une version entièrement en niveaux de gris.

Deux versions en Java sont disponibles mais une seule est viable.

Pour cet algorithme, les tests ont été réalisés sur des images en couleurs.

4.1.2 Résultats obtenus



4.1.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Première version Java (getPixel / setPixel)	0.4 s	8.5 s	20 s
Deuxième version Java (getPixels / setPixels)	0.1 s	0.17 s	0.25 s
Version Renderscript	0.04 s	0.08 s	0.2 s

4.1.4 Conclusion

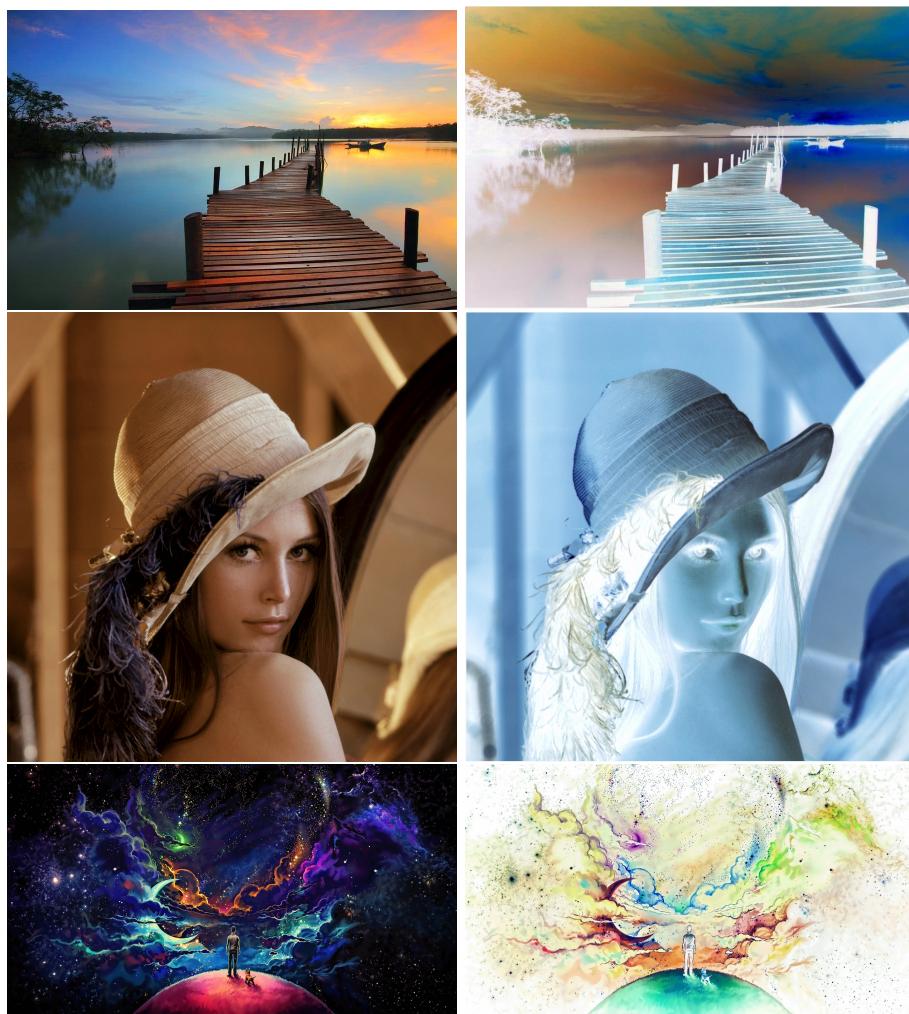
La première version n'est pas utilisable. Les autres versions ont des temps d'exécutions qui sont satisfaisants. Pour ce traitement, l'utilisation du Renderscript ou de la seconde version de l'algorithme en Java semble être équivalente en terme de temps d'exécution.

4.2 L'inversion des couleurs

4.2.1 Présentation

Ce traitement consiste à inverser les couleurs de l'image. Pour cela, il suffit d'inverser chaque composante de chaque pixel.

4.2.2 Résultats obtenus



4.2.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java	0.04 s	0.18 s	0.18 s
Version Renderscript	0.03 s	0.05 s	0.17 s

4.2.4 Conclusion

Comme pour le passage en niveaux de gris, l'utilisation de Java ou de Renderscript n'est pas ici une question importante. Les deux permettent d'avoir des temps d'exécution similaire. Le Renderscript n'a pas l'air beaucoup plus performant que Java pour ce traitement,

4.3 La colorisation

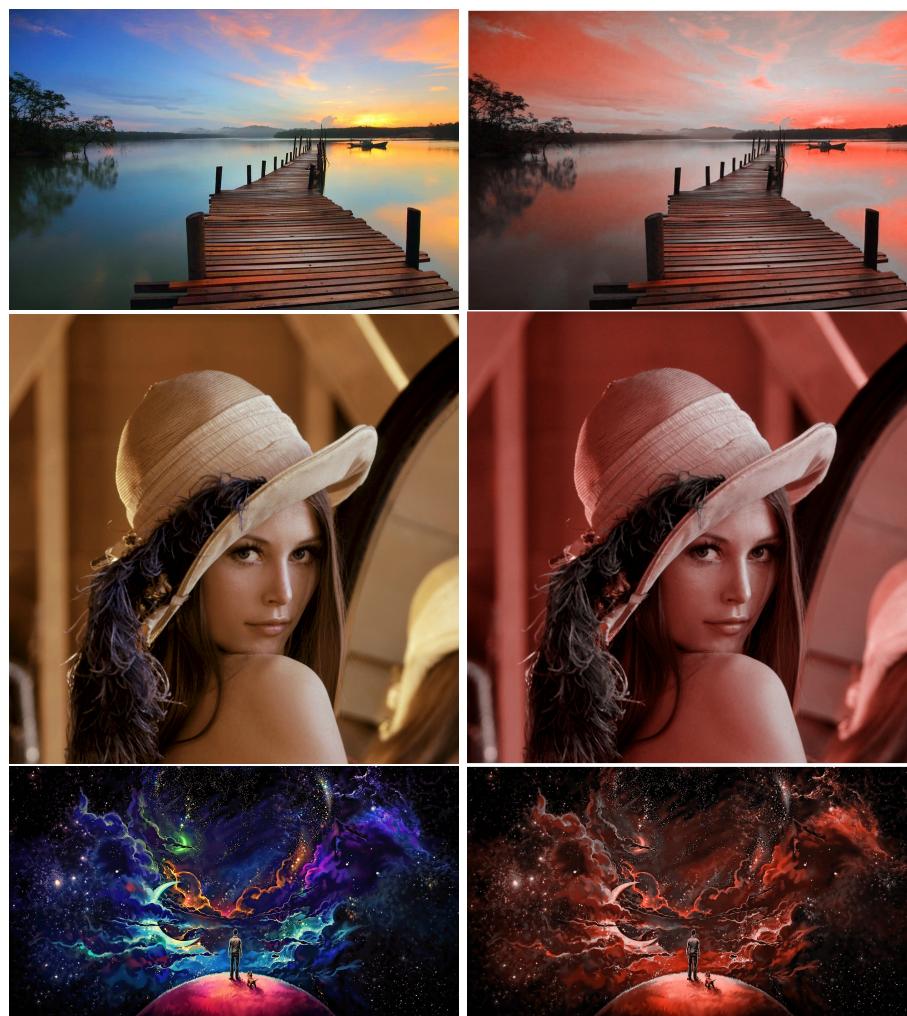
4.3.1 Présentation

La colorisation consiste à transformer la teinte de chaque pixel de l'image pour correspondre à la teinte choisie par l'utilisateur.

Au niveau du code, il est également possible de réaliser ce traitement avec les fonctions implémentées par Android (RGBToHSV et HSVToColor).

Cette fonctionnalité produira un effet seulement sur les images qui sont déjà en couleurs.

4.3.2 Résultats obtenus



4.3.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java (fonctions Android)	6.6 s	66 s	153 s
Version Java (fonctions recodées)	5 s	53 s	110 s
Version Renderscript	0.09 s	0.1 s	0.2 s

4.3.4 Conclusion

Les fonctions implémentées dans Android Studio ne sont vraiment pas utilisables car bien trop longues. Ainsi, les fonctions recodées offrent des performances légèrement supérieures même si elles ne sont pas optimales. Le Renderscript, quant à lui, permet d'avoir des temps d'exécution très bas, toujours inférieurs à 1 seconde. De plus, plus l'image est grande, plus le temps d'exécution est long pour les versions Java contrairement à la version Renderscript.

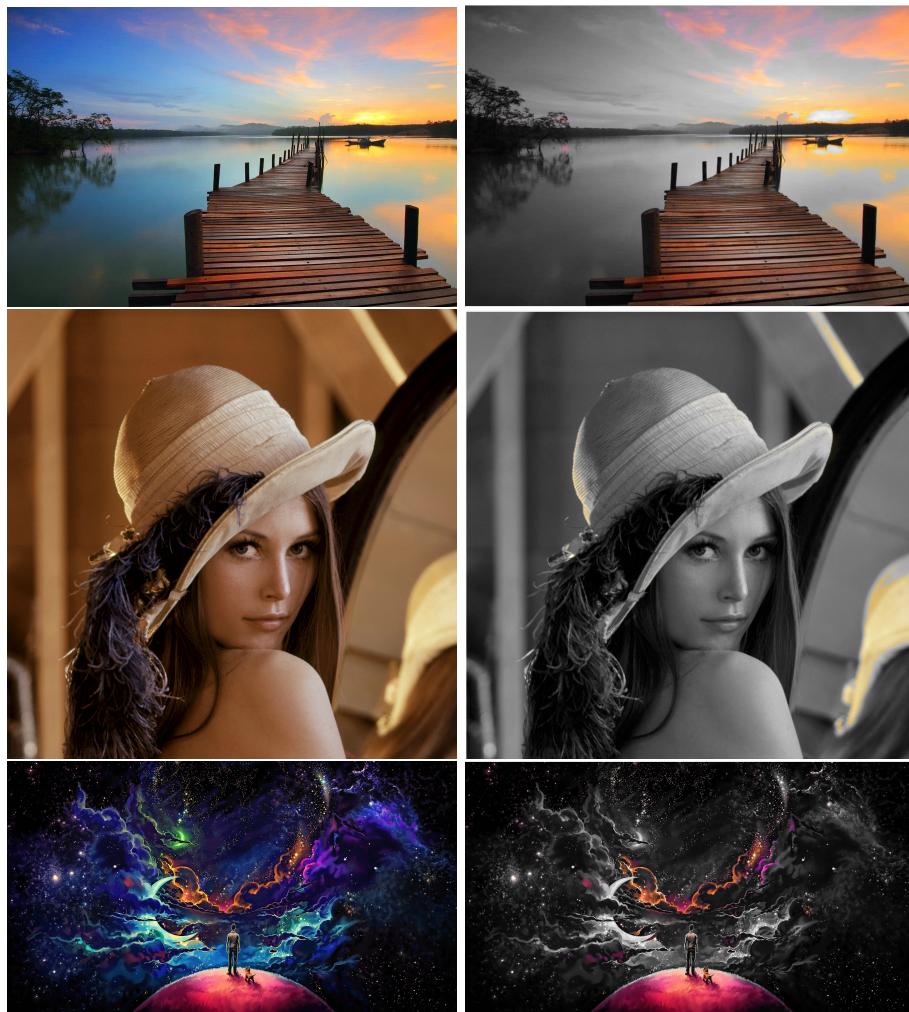
4.4 La conservation

4.4.1 Présentation

La conservation est un traitement qui va permettre de mettre tout les pixels de l'image en niveaux de gris qui ne font pas partie d'un certain intervalle choisi par l'utilisateur. Il doit sélectionner deux couleurs et la forme de l'intervalle (intérieur ou extérieur des couleurs sélectionnées).

Comme pour la colorisation, il est possible de réaliser ce traitement avec les fonctions implémentées par Android et cette fonctionnalité produira un effet seulement sur les images qui sont déjà en couleurs.

4.4.2 Résultats obtenus



4.4.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java (fonctions Android)	5 s	45 s	100 s
Version Java (fonctions recodées)	2.5 s	27 s	40 s
Version Renderscript	0.03 s	0.09 s	0.2 s

4.4.4 Conclusion

De la même manière que pour la colorisation, les fonctions implémentées par Android Studio offrent les moins bonnes performances et l'utilisation du Renderscript semble être la réponse idéale à ce traitement d'image, avec des temps avoisinant le dixième de seconde.

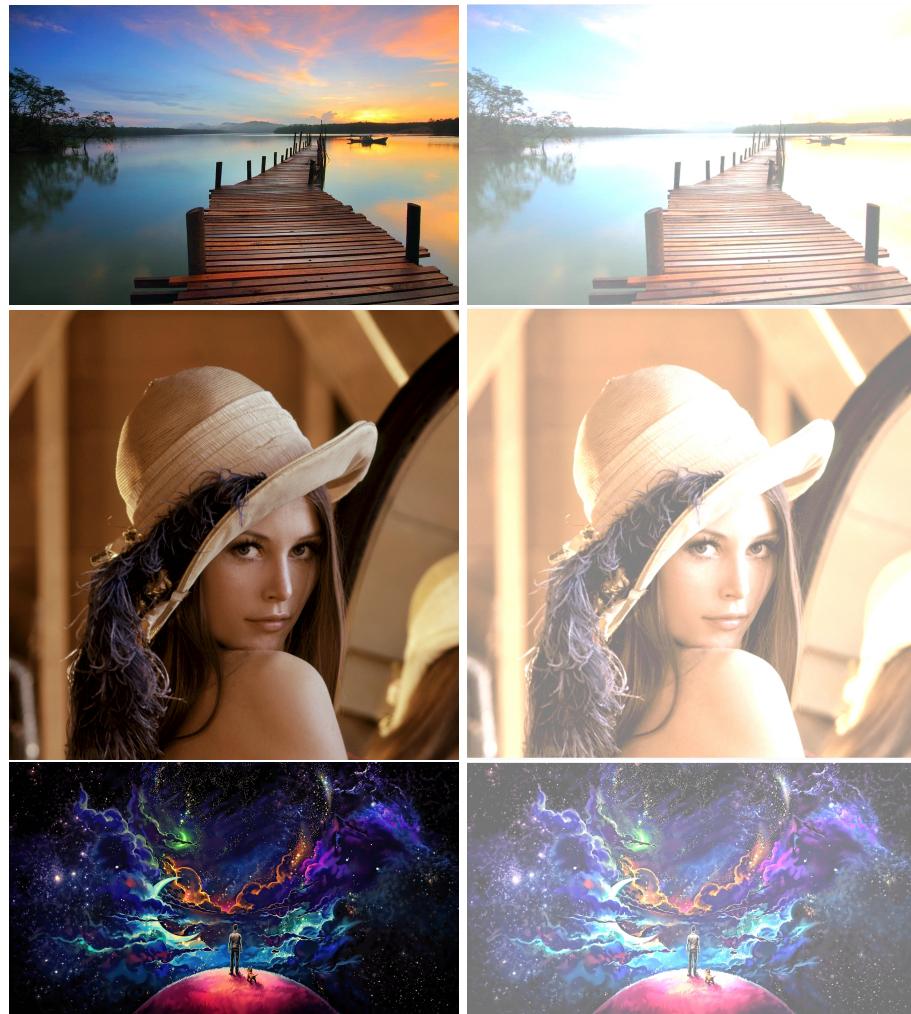
4.5 La modification de la luminosité

4.5.1 Présentation

Cette fonctionnalité permet de transformer l'image en changeant sa luminosité selon un facteur paramétrable par l'utilisateur grâce à une SeekBar.

Il est possible d'augmenter mais aussi de diminuer la luminosité.

4.5.2 Résultats obtenus



4.5.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java	0.03 s	0.13 s	0.22 s
Version Renderscript	0.03 s	0.05 s	0.17 s

4.5.4 Conclusion

Le Renderscript n'apporte pas un gain de temps très important comme pour l'inversion des couleurs. Les deux versions sont totalement utilisables.

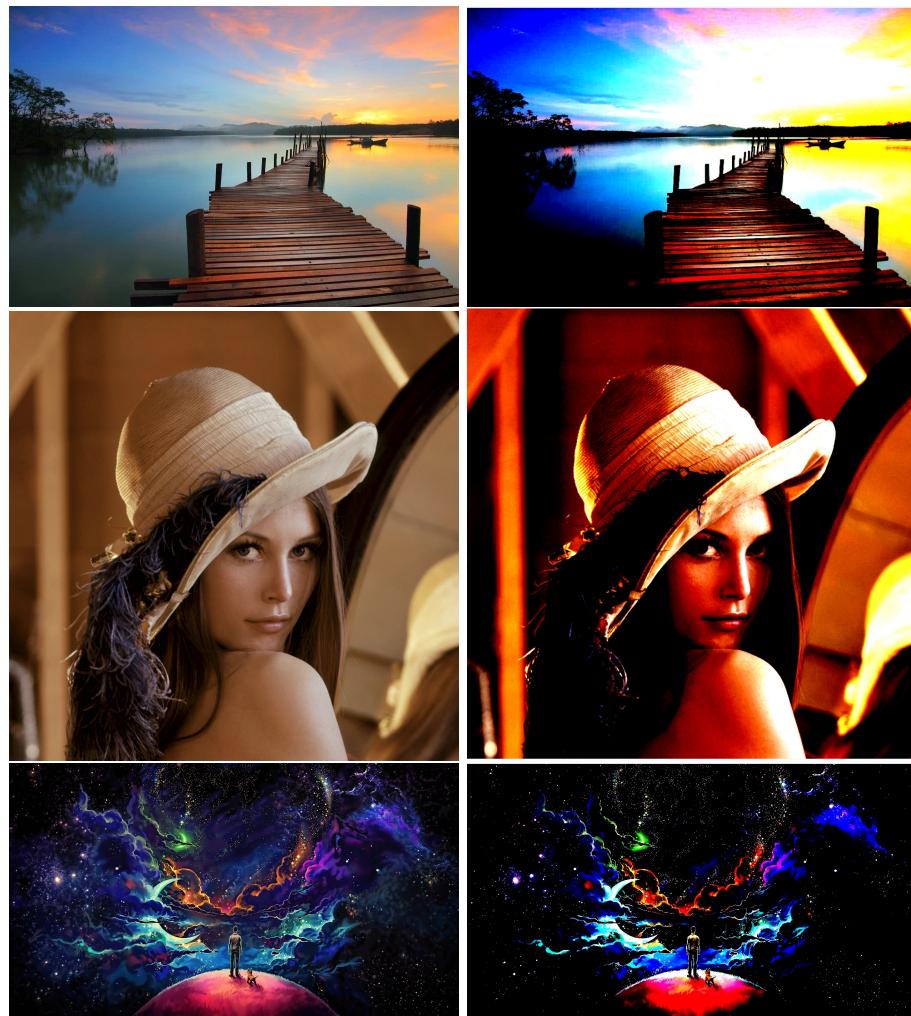
4.6 La modification du contraste

4.6.1 Présentation

Cette fonctionnalité permet de transformer l'image en changeant son contraste selon un facteur paramétrable par l'utilisateur grâce à une SeekBar.

Il est possible d'augmenter (> 1) mais aussi de diminuer le contraste ([0-1]).

4.6.2 Résultats obtenus



4.6.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java	0.05 s	0.26 s	0.32 s
Version Renderscript	0.02 s	0.1 s	0.19 s

4.6.4 Conclusion

Le traitement en Java semble être un tout petit moins efficace que celui en Renderscript mais ça reste négligeable, les deux algorithmes prennent moins d'une seconde à s'exécuter.

4.7 L'extension de dynamique

4.7.1 Présentation

L'extension de dynamique va transformer la valeur des pixels dans le cas où l'intervalle de valeurs concernant le V (value) du HSV n'est pas maximal. Cela aura pour effet d'améliorer le contraste de l'image. Ce traitement peut être efficace en particulier sur les images en niveaux de gris. Dans le cas des images en couleur, l'intervalle est en général maximal.

4.7.2 Résultats obtenus

Pour les images utilisées jusqu'ici afin de réaliser les tests, il n'y a pas de changements visuel au terme de l'exécution de cet algorithme. Ainsi, les temps pour ces images apparaîtront dans le tableau des résultats mais pas la visualisation des résultats.

4.7.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java	2.5 s	25 s	71 s
Version Renderscript	0.1 s	0.1 s	0.2 s

4.7.4 Conclusion

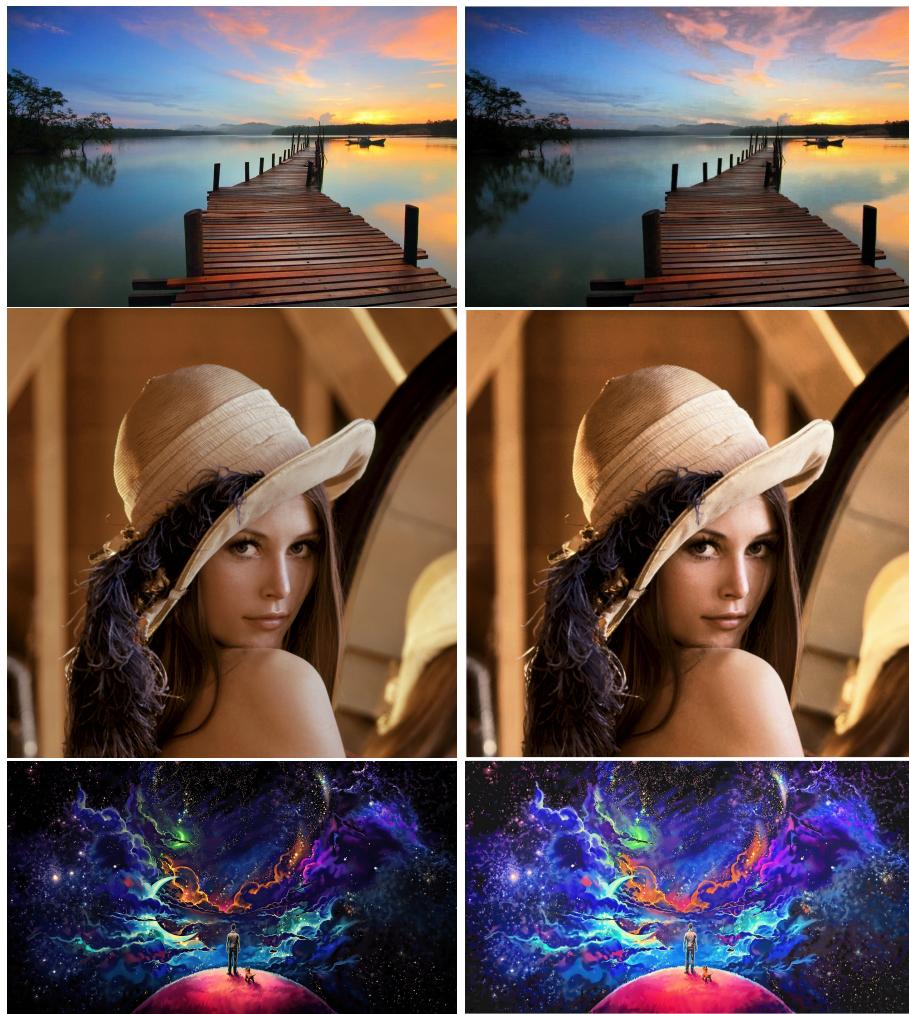
L'algorithme en Renderscript est beaucoup plus efficace avec des résultats quasi-instantané contrairement à celui codé en Java qui prend beaucoup plus de temps à être exécuté.

4.8 L'égalisation d'histogramme

4.8.1 Présentation

L'égalisation d'histogramme permet tout comme l'extension de dynamique d'améliorer le contraste de l'image. Les résultats de ce traitement sont généralement bien plus visibles notamment sur des images en couleurs.

4.8.2 Résultats obtenus



4.8.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java	8 s	94 s	200 s
Version Renderscript	0.15 s	0.16 s	0.36 s

4.8.4 Conclusion

La version Java ne peut pas être utilisée, elle amène à des temps d'attente bien trop important (ici 200 secondes même si le profiler était actif, ce qui alonge grandement ce temps) alors que le Renderscript reste très efficace même pour de grandes images avec des temps inférieurs à la seconde.

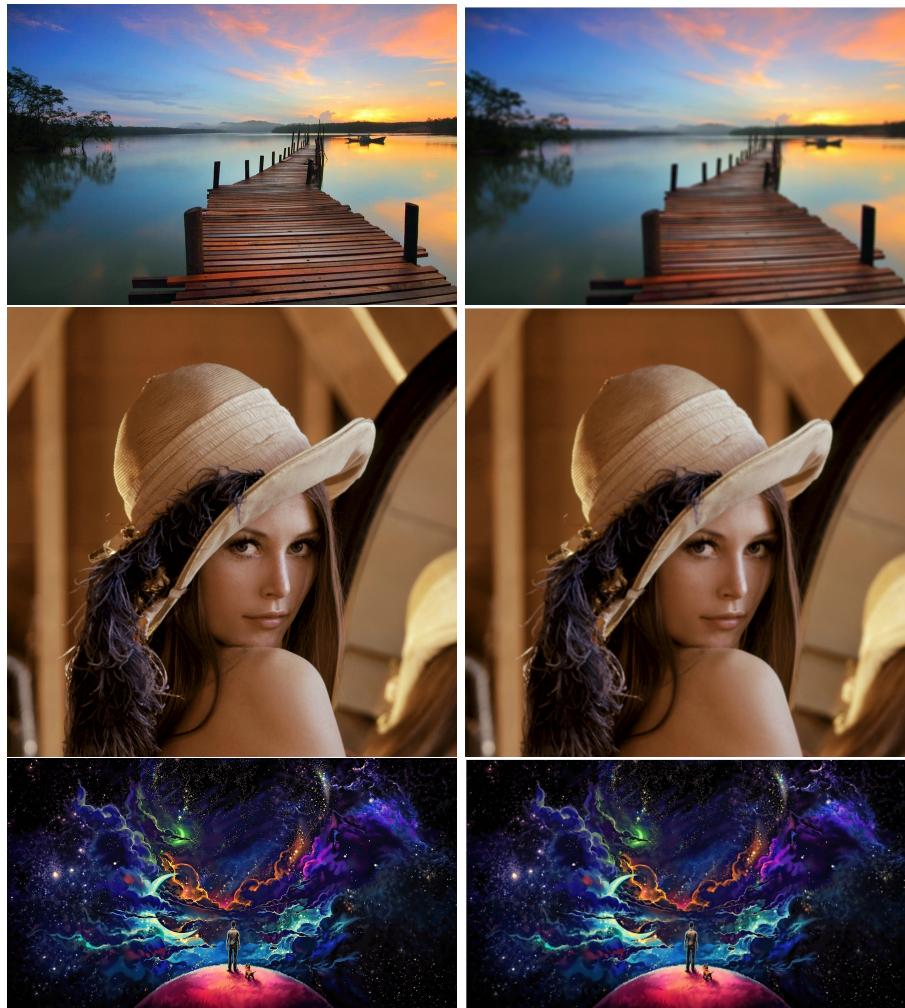
4.9 Le filtre moyenneur

4.9.1 Présentation

Le filtre moyenneur est un traitement de convolution permettant de flouter l'image. Chaque pixel va se transformer selon la moyenne des pixels l'entourant et selon la taille du noyau utilisé. Plus la taille du noyau est grande, plus l'effet sera observable mais le temps d'exécution sera plus long. L'utilisateur a le choix de la taille de ce noyau et il est contraint de rentrer une valeur impaire.

4.9.2 Résultats obtenus

Les résultats affichés sont ceux obtenus avec un filtre de taille 7 x 7.



4.9.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java 3 x 3	0.4 s	6.4 s	13.5 s
Version Java 5 x 5	2 s	15 s	40 s
Version Java 7 x 7	3.5 s	35 s	83 s
Version Renderscript 3 x 3	0.2 s	0.2 s	0.4 s
Version Renderscript 5 x 5	0.2 s	0.3 s	0.8 s
Version Renderscript 7 x 7	0.2 s	0.6 s	1.1 s

4.9.4 Conclusion

On peut observer que pour la version Java, le temps d'exécution dépend fortement de la taille du noyau utilisé et de la taille de l'image. Plus ils sont grands, plus le temps pour exécuter le traitement l'est aussi. Ca semble également vrai pour le Renderscript mais dans des proportions bien moins importantes. L'utilisation du Renderscript pour ce traitement est une nouvele fois très recommandée.

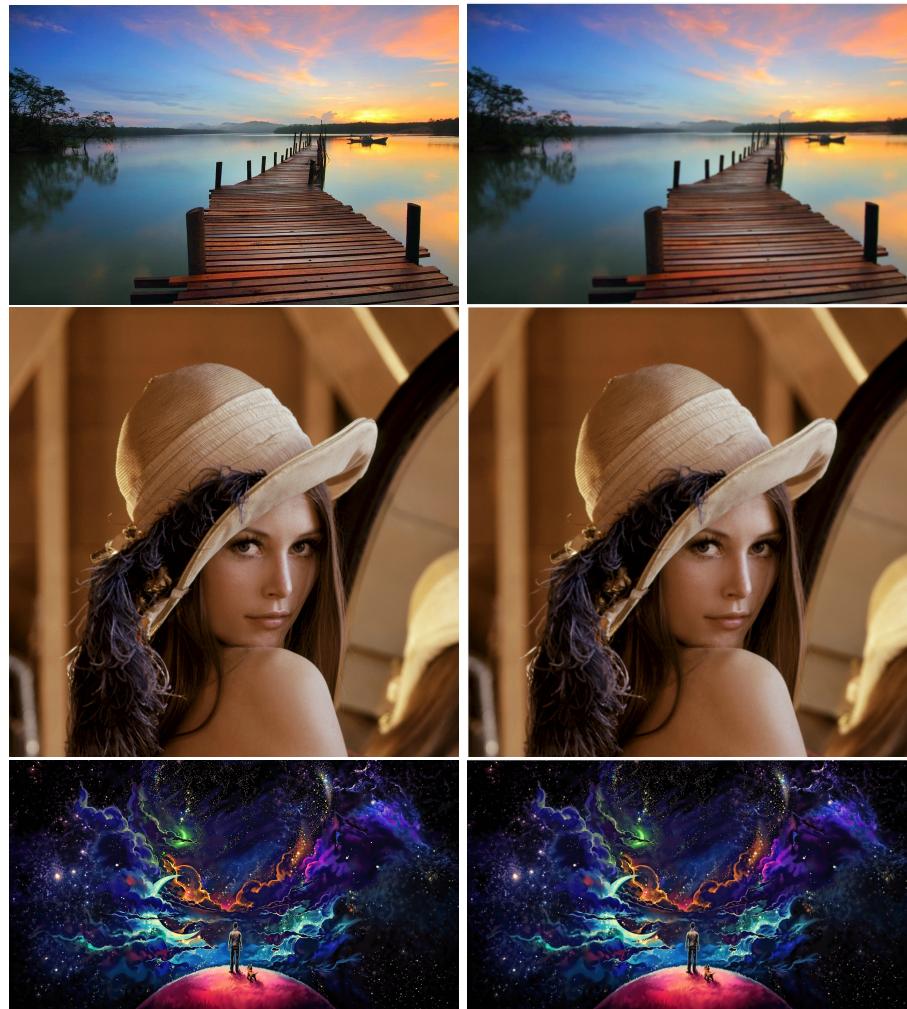
Les résultats ne sont pas ici très visibles car les images ont une trop grande résolution par rapport au noyau utilisé, avec des images plus petites, les résultats sont bien mieux visibles.

4.10 Le filtre Gaussien

4.10.1 Présentation

Comme pour le filtre moyenneur, le filtre Gaussien va flouter l'image. L'effet est toutefois différent car les valeurs du noyau sont différentes. Le filtre moyenneur fait une réelle moyenne des pixels voisins alors que le filtre de Gauss va mettre l'accent sur le pixel en lui-même, puis sur les pixels autour de lui verticalement et horizontalement et enfin les pixels autour de lui en diagonale.

4.10.2 Résultats obtenus



4.10.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java 3 x 3	0.3 s	5.5 s	13 s
Version Java 5 x 5	2 s	14 s	38 s
Version Renderscript 3 x 3	0.2 s	0.2 s	0.4 s
Version Renderscript 5 x 5	0.2 s	0.3 s	0.8 s

4.10.4 Conclusion

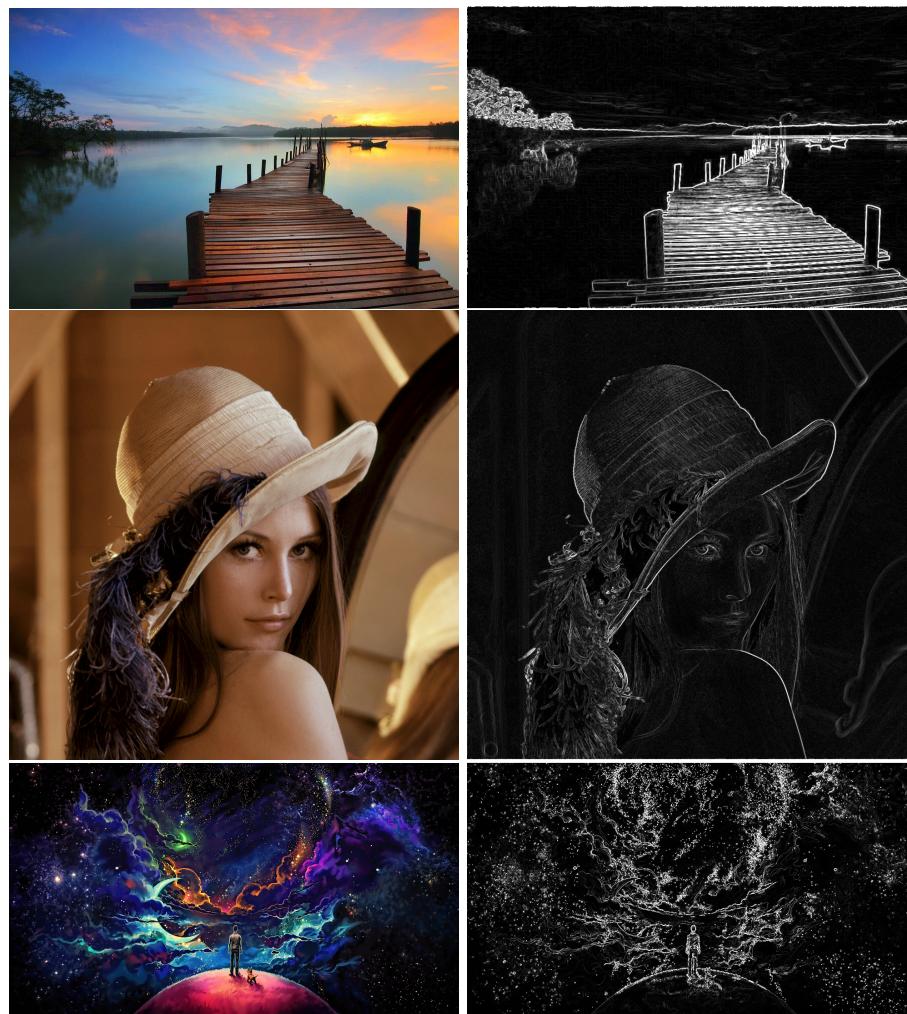
La méthode utilisée pour cet algorithme est exactement la même que celle utilisée pour le filtre moyenneur, la seule différence résidant dans les valeurs du noyau. Les résultats en terme de performances sont donc totalement équivalents et le constat est le même.

4.11 Le filtre de Sobel

4.11.1 Présentation

Le filtre de Sobel permet de détecter les contours d'une image. On applique deux filtres de convolution sur l'image au préalable transformée en niveaux de gris permettant d'obtenir les gradients (dérivées horizontale et verticale de chaque point) des pixels et de faire apparaître les contours présents dans l'image.

4.11.2 Résultats obtenus



4.11.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java	0.5 s	8.5 s	22.5 s
Version Renderscript	0.2 s	0.3 s	0.6 s

4.11.4 Conclusion

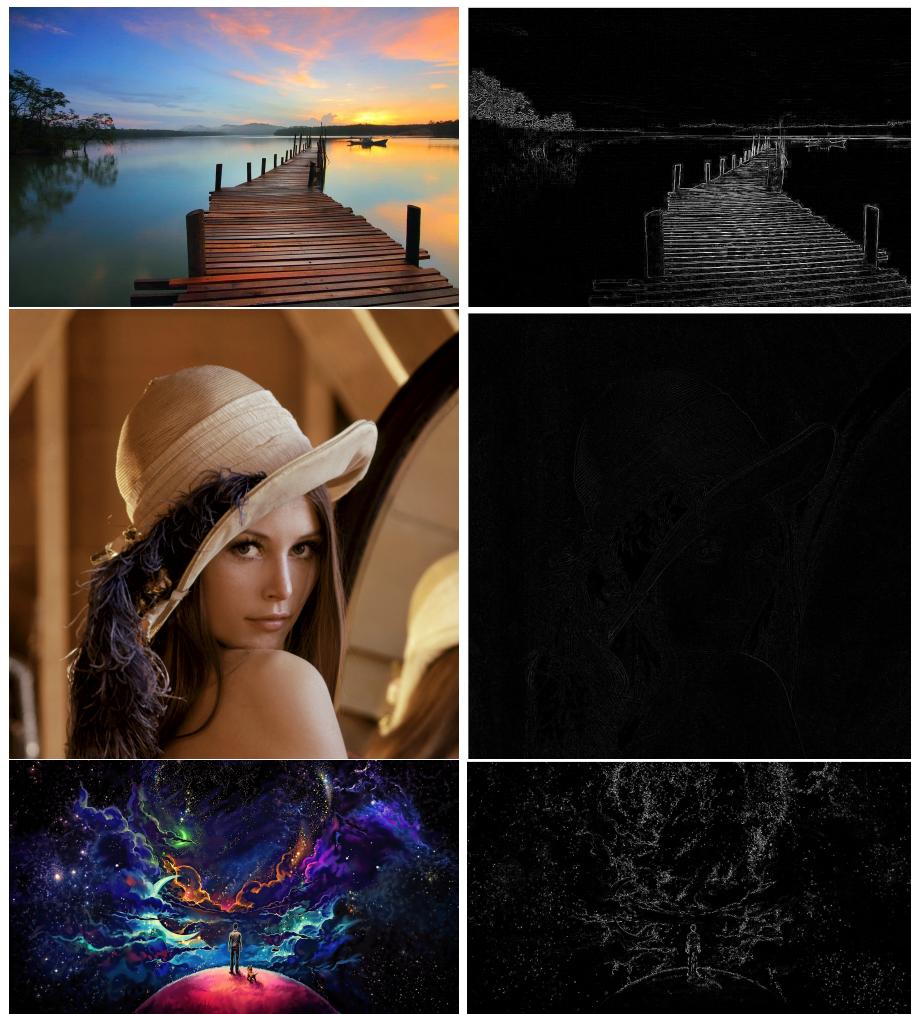
Le Renderscript fournit toujours des résultats bien plus satisfaisant que Java en terme de performances pour ce traitement.

4.12 Le filtre Laplacien

4.12.1 Présentation

Tout comme le filtre de Sobel, le filtre Laplacien permet de détecter les contours. Cependant, ils sont bien moins marqués que pour le filtre de Sobel. Contrairement au filtre de Sobel, un seul noyau est utilisé pour le filtre Laplacien.

4.12.2 Résultats obtenus



4.12.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java	0.2 s	6.7 s	16.5 s
Version Renderscript	0.1 s	0.27 s	0.6 s

4.12.4 Conclusion

Cet algorithme étant construit de la même façon que le filtre de Sobel, la conclusion est la même que pour ce dernier (voir résultat pour Lena). Le Renderscript est supérieur à Java pour l'exécution de ce traitement.

4.13 L'effet croquis/ crayonné

4.13.1 Présentation

Ce traitement n'ayant pas été vu en cours, la présentation sera un peu plus détaillée.

Ce traitement permet d'obtenir un dessin qu'une personne aurait été capable de réaliser à partir d'une image. Les traits utilisés donnent vraiment l'impression qu'ils ont été fait au crayon à papier.

Afin d'obtenir un tel effet, plusieurs transformations sont nécessaires. Il faut dans un premier temps une copie de l'image originelle en niveaux de gris. Puis, il faut créer une nouvelle copie de la copie qui sera inversée (inversion des couleurs) puis floutée (filtre de Gauss 5x5).

Une fois en possession de ces deux images, il faut appliquer une formule spéciale à partir des pixels de chaque images afin de créer les nouveaux pixels. Cette formule appelée "color dodge" est de la forme suivante :

Si

$$pixelInvert == 255$$

Alors

$$newPixel = pixelInvert$$

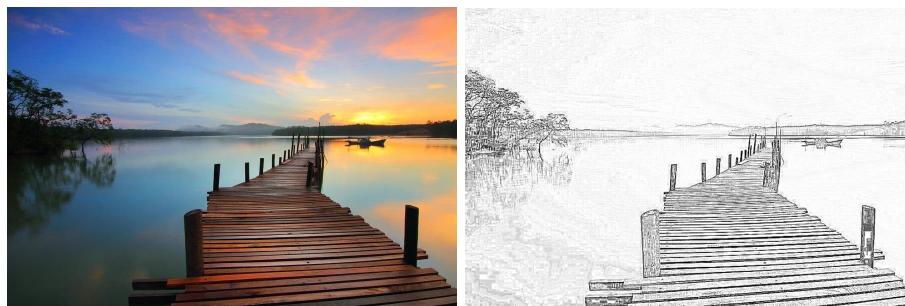
Sinon

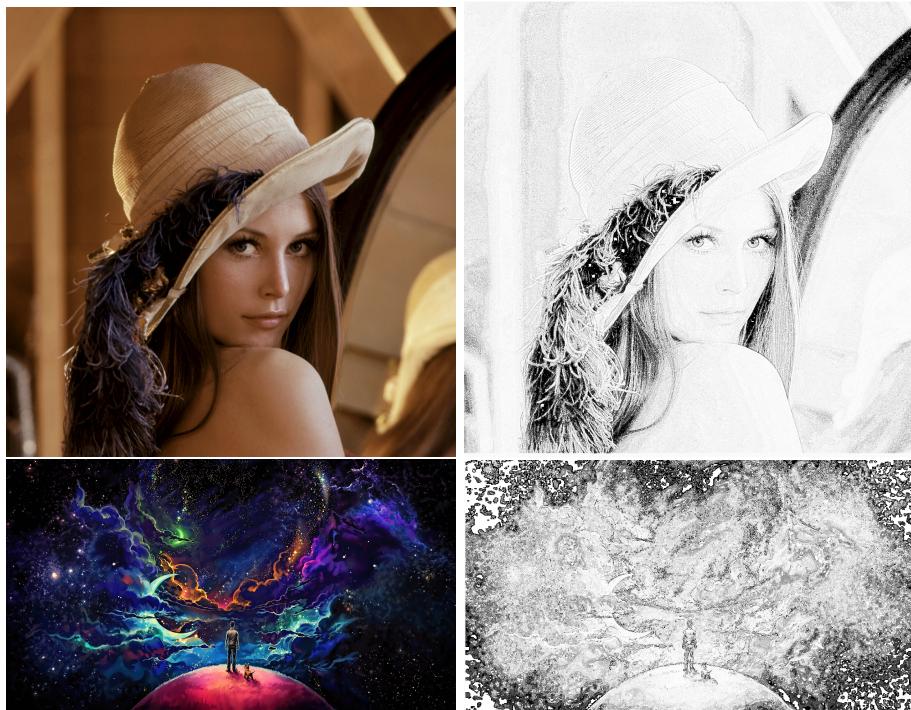
$$newPixel = Math.min(255, ((pixelCopy << 8)/(255 - pixelInvert)))$$

Cette formule permet d'éclaircir la couche inférieure en fonction de la valeur de la couche supérieure : plus la couche supérieure est lumineuse, plus sa couleur affecte la couche inférieure. Le mélange de n'importe quelle couleur avec le blanc donne du blanc. Le mélange avec le noir ne modifie pas l'image.

De plus, pour l'expérience utilisateur, trois choix de niveaux de détails sont proposés afin d'avoir un croquis plus ou moins détaillé.

4.13.2 Résultats obtenus





4.13.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java	1 s	14 s	44 s
Version Renderscript	0.2 s	0.6 s	1.2 s

4.13.4 Conclusion

Ce traitement fait appel à d'autres traitements comme le passage en niveaux de gris, l'inversion des couleurs et le floutage à l'aide du filtre de Gauss avant même de réaliser les opérations propres à cet algorithme. De ce fait, le temps d'exécution dépend fortement en parti du temps des effets énoncés précédemment et donc le Renderscript est beaucoup plus efficace que le Java notamment à cause du floutage qui prend beaucoup plus de temps en Java qu'en Renderscript.

4.14 L'effet cartoon

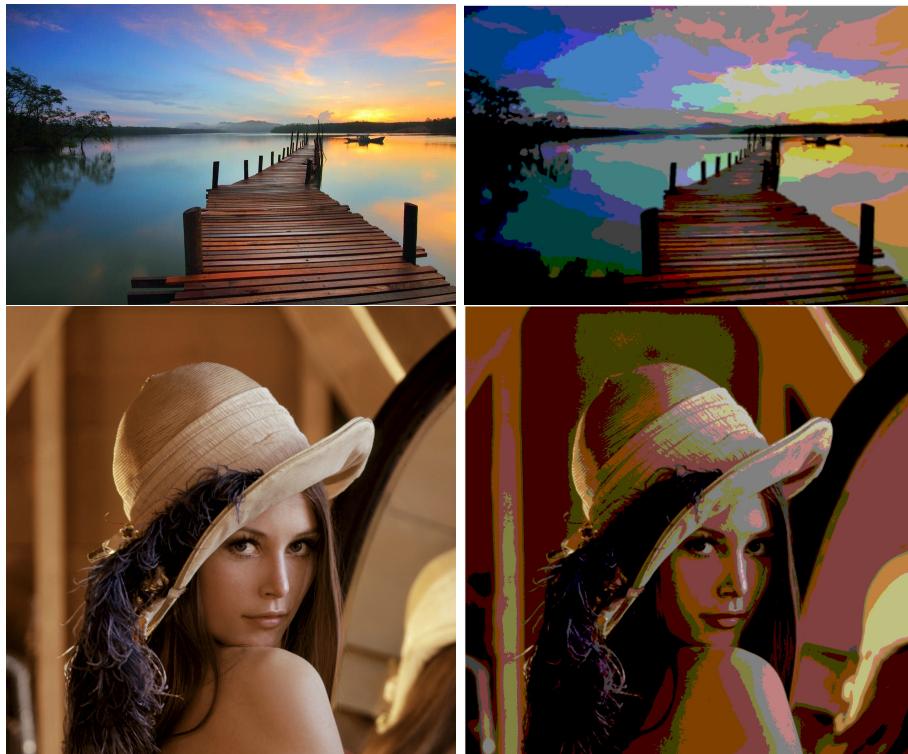
4.14.1 Présentation

Ce traitement n'ayant pas été vu en cours, la présentation sera un peu plus détaillée.

Cet algorithme permet d'obtenir un effet cartoon. Afin d'obtenir un tel effet, il faut diminuer le nombre de couleurs de l'image pour au final n'avoir qu'un nombre très restreint de valeurs possibles pour chaque pixel. L'implémentation de cet algorithme peut être variable selon le rendu attendu, plus le nombre de couleur autorisé sera grand, plus le résultat ressemblera à l'image originelle et au contraire, plus ce nombre sera petit, plus le résultat aura cet effet rappelant les vieux comics.

Ainsi, dans cette implémentation, chaque composante de chaque pixel peut avoir 5 valeurs différentes (0, 64, 128, 192, 255). Il est donc possible de trouver un maximum de $3^5 = 243$ pixels différents dans l'image retourné par cet algorithme.

4.14.2 Résultats obtenus





4.14.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java	1 s	16 s	39 s
Version Renderscript	0.2 s	0.5 s	0.9 s

4.14.4 Conclusion

Plus l'image est grande et plus le temps d'exécution est long pour la version Java alors que la version Renderscript permet d'obtenir des temps inférieurs à la seconde. Cette dernière est donc bien plus performante.

4.15 L'effet de neige

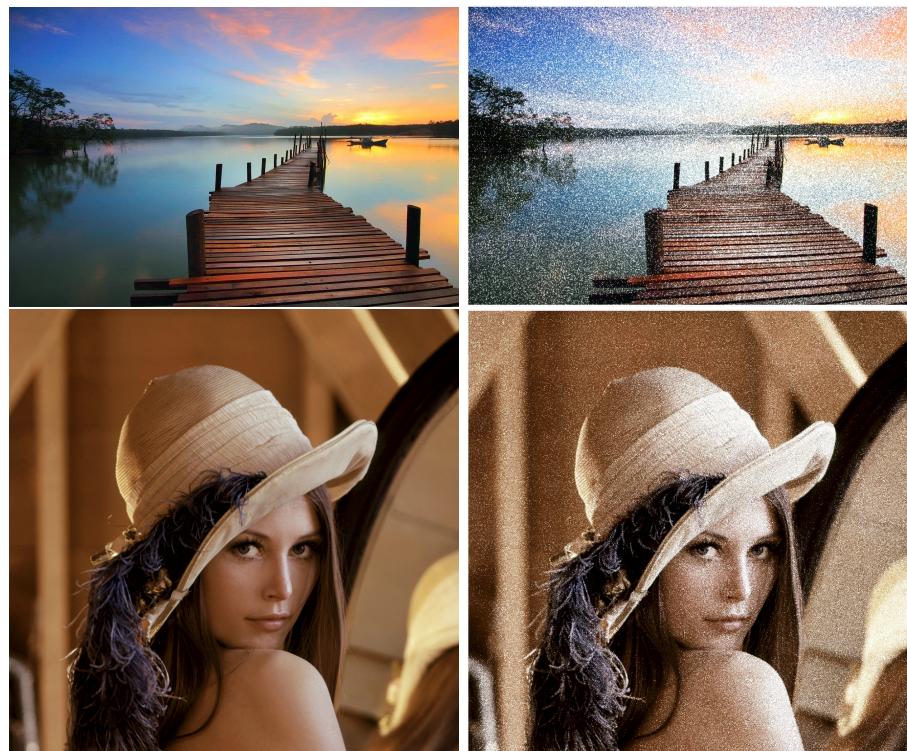
4.15.1 Présentation

Ce traitement n'ayant pas été vu en cours, la présentation sera un peu plus détaillée.

Ce traitement permet de faire apparaître un filtre sur l'image comme si c'était une photo prise alors que la neige tombait.

L'algorithme pour obtenir un tel effet est très simple. Pour chaque pixel, il faut déterminer une valeur aléatoire entre 0 et 255 et si les 3 composantes du pixel sont supérieures à la valeur aléatoire, alors on transforme le pixel en un pixel blanc.

4.15.2 Résultats obtenus





4.15.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java	0.15 s	0.3 s	0.4 s
Version Renderscript	0.2 s	1 s	2 s

4.15.4 Conclusion

Pour la première fois dans tout ce projet, le Renderscript semble vraiment moins efficace que le Java. Ce manque d'efficacité est très certainement du à l'appel de la fonction rsRand(). C'est la seule façon d'expliquer pourquoi le Renderscript atteint des valeurs aussi importantes que 2 secondes d'exécution.

Ainsi, la version Java est ici la bonne version à utiliser.

4.16 L'incrustation d'objets

4.16.1 Présentation

Ce traitement n'ayant pas été vu en cours, la présentation sera un peu plus détaillée.

Cet algorithme utilise la classe FaceDetector qui permet d'identifier des parties précises d'un visage. Si de telles parties existent sur l'image, alors l'algorithme va incruster des "objets" qui sont deux yeux et un nez à la place des anciens présents sur l'image originelle. Si l'algorithme ne détecte pas l'existence de ces détails sur l'image, il ne fera rien et affichera simplement un message indiquant qu'il n'est pas possible d'insérer d'objets sur cette image.

4.16.2 Résultats obtenus

Sur les 3 images utilisées lors de ces tests, seule celle de Lena fait apparaître un résultat visuel. Les deux autres images ne seront présentes que dans le tableau pour voir leur temps d'exécution.



4.16.3 Temps d'exécution

	Paysage	Lena	Espace
Taille bitmap	900 x 600	1960 x 1960	3840 x 2160
Version Java	1.2 s	1.3 s	1.5 s

4.16.4 Conclusion

Ce traitement n'a pas été réalisé en Renderscript et il sera donc impossible de comparer les deux versions.

Cependant, il est possible de dire que ce traitement peut être à l'origine de traitements plus poussés où l'on pourrait laisser le choix à l'utilisateur de quels

yeux ou quel nez ou n'importe quelle autre partie d'un visage il veut en lui proposant une gamme d'image en rapport avec les parties présentes sur l'image et détectées par la classe FaceDetector.

4.17 Le zoom et le scroll

Ici il ne s'agit pas d'un traitement d'image mais d'une fonctionnalité présente dans l'application.

Il est en effet possible de bouger l'image dans l'ImageView qui la contient grâce à un mouvement de slide avec un doigt sur l'image ainsi que de zoomer ou bien dézoomer avec deux doigts.

Cette fonctionnalité n'est cependant pas parfaite car il est possible de bouger l'image en dehors de l'ImageView, c'est-à-dire que les bords ne sont pas gérés.

Un autre léger problème est présent sur cette fonctionnalité. Lorsque l'utilisateur va toucher l'image pour la première fois dans le but de la déplacer ou bien de zoomer, cette dernière va remonter tout en haut de l'ImageView alors qu'elle était bien centrée auparavant.

5 Conclusion

Les attentes du dernier rapport ont été totalement dépassées, elles ne concernaient que l'implémentation de la convolution en Renderscript et la finalisation de l'incrustation d'objets. Il se trouve que l'application possède maintenant des effets supplémentaires comme l'effet crayonné, l'effet cartoon ou encore l'effet de neige (qui ont en plus également été réalisés en Renderscript).

Le seul problème trouvé étant celui concernant le zoom et scroll détaillé précédemment, cependant celui-ci n'est pas très grave étant donné que l'application fonctionne très bien malgré son existence.

Pour ce qui est des pourcentages de travail sur le projet :

- Vincent : 95
- Tom : 5
- Sarah : 0
- Firas : 0