
Rapport de projet

Retouche d'images sur Android

Manuel Ricardo Guevara Garban
Loïc Lachiver
Romain Pigret-Cadou
Sofiane Menadjlia

L3 Informatique
Projet technologique
2020

Université
de **BORDEAUX**

Table des matières

1	Introduction :	3
2	L'application :	4
2.1	Lancement :	4
2.2	Utilisation :	4
3	Effets :	8
3.1	Luminosité (Brightness) :	8
3.2	Contraste (Contrast) :	8
3.3	Saturation (Saturation) :	9
3.4	Hue :	9
3.4.1	Teint uniforme :	9
3.4.2	Teint non uniforme :	10
3.5	Keep hue :	10
3.6	Egalisation d'histogramme (Enhance) :	10
3.7	Convolution (Blur, Sharpen, Neon) :	11
3.7.1	Flou gaussien et moyenneur (Blur) :	13
3.7.2	Amélioration de la netteté / Masque flou (Sharpen) :	14
3.7.3	Détection des contours (Néon) :	16
3.7.4	Remarques et limites	17
4	Structure du projet	18
4.1	Structure graphique Android et navigation :	18
4.2	Classe Image	19
4.2.1	Classe ImageInfo	19
4.2.2	File d'effets	19
4.3	AsyncTasks	20
4.4	Classes d'effets	20
4.5	Fragment EffectSettingsFragment	20
4.6	Packages utilitaires	20
4.7	Modularité du code	21
5	Performances :	22
5.1	Temps d'exécution :	22
5.2	Mémoire :	23
6	Remarques et améliorations :	26
6.1	Remarques sur le code	26
6.2	Remarques sur les librairies Android	26
6.3	Améliorations à court terme :	27
7	Gestion du projet :	29
7.1	Organisation générale :	29
8	Conclusion	30

1 Introduction :

Ce rapport synthétise notre travail de développement d'une application de retouche d'image exécutable sur la plateforme mobile Android.

Réalisé dans le cadre de l'UE *Projet technologique* lors du 2ème semestre de Licence 3 informatique à l'Université de Bordeaux, les objectifs de ce rendu de groupe étaient les suivant :

- Réaliser une application graphique Android
- Permettre de charger, éditer et sauvegarder facilement une image
- Proposer des effets par simple modification de pixels
- Proposer des effets manipulant des histogrammes
- Proposer des effets de convolution
- Utiliser la technologie d'accélération RenderScript
- Gérer le développement d'un projet de groupe

➔Lien GitHub du projet : <https://github.com/picachoc/pimp-android>

NB : Pour des informations plus détaillées sur le code, générez la JavaDoc de ce dernier.

2 L'application :

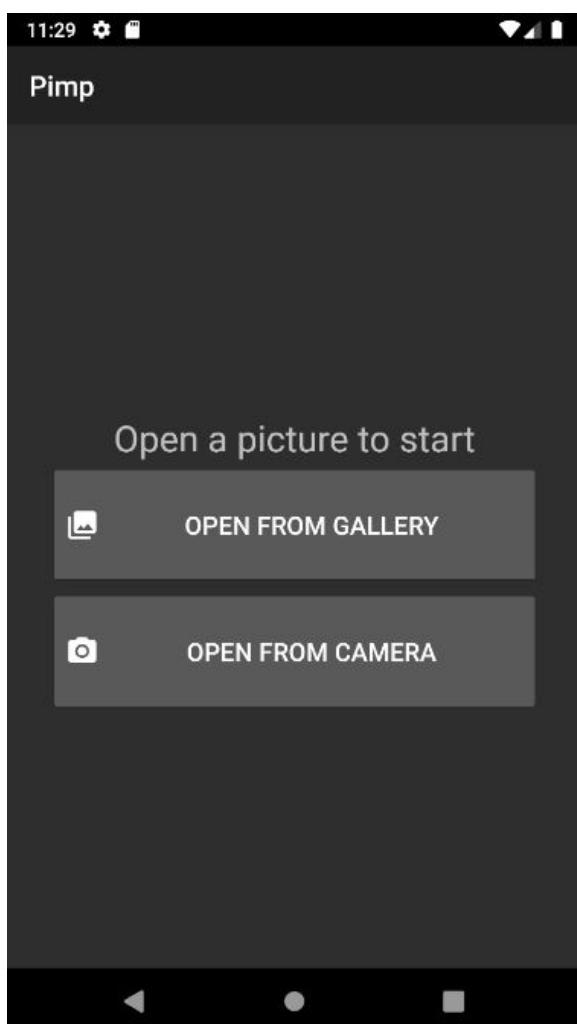
Version minimale Android requise : Oreo (8.0) (API level 26)

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 le dépôt, rendez vous sur Android Studio puis **File > New > Import Project** et sélectionnez le dossier créé par le clone.

2.2 Utilisation :

A l'ouverture de l'application vous obtiendrez l'interface suivante :



Appuyez sur le premier bouton (🖼) pour ouvrir votre galerie photo et choisir une image à importer.

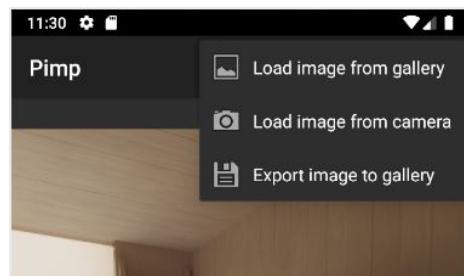
Ou appuyez sur le second bouton (📷) pour ouvrir la caméra de votre appareil Android. Capturez alors une image, elle sera sauvegardée dans votre galerie et instantanément importée dans l'application.

Il sera nécessaire à la toute première utilisation de ces fonctionnalités d'autoriser l'application à accéder à la galerie et/ou la caméra.

Après avoir choisi une image vous accéderez à la page d'édition principale :



Il vous sera possible d'importer une nouvelle image depuis le menu ☰ en haut à droite :



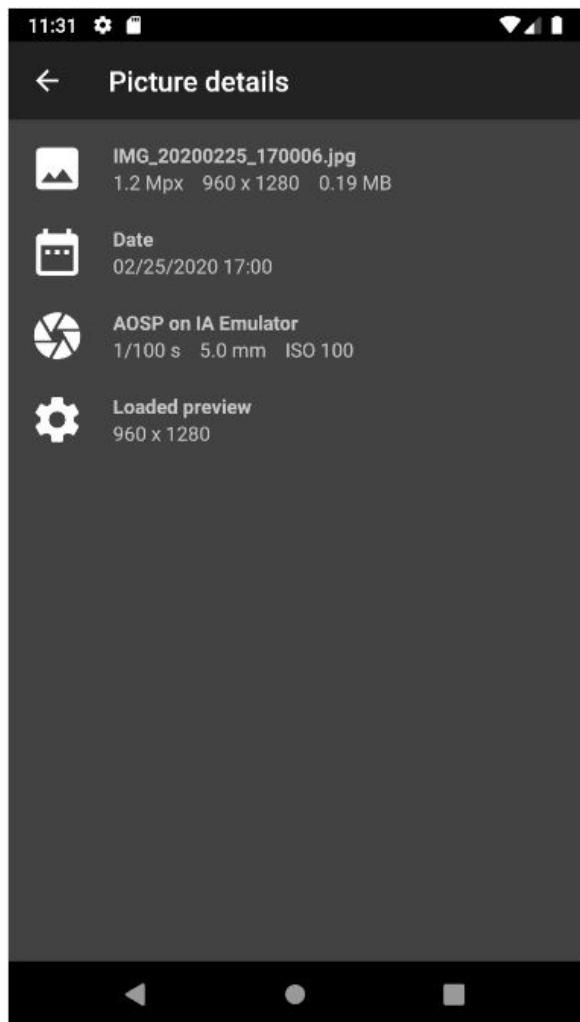
Le bouton permettra d'exporter votre image dans votre galerie, cette opération peut prendre un peu de temps car l'image exportée sera de même taille que l'image d'origine.

Vous pouvez manipuler l'image, glissez pour la déplacer et pincez pour zoomer ou dé-zoomer.

En bas de l'écran vous trouverez la liste des effets disponibles applicables sur votre image, faites défiler de droite à gauche.

Le bouton annule tout les changements appliqués à l'image en la restaurant à son état d'origine.

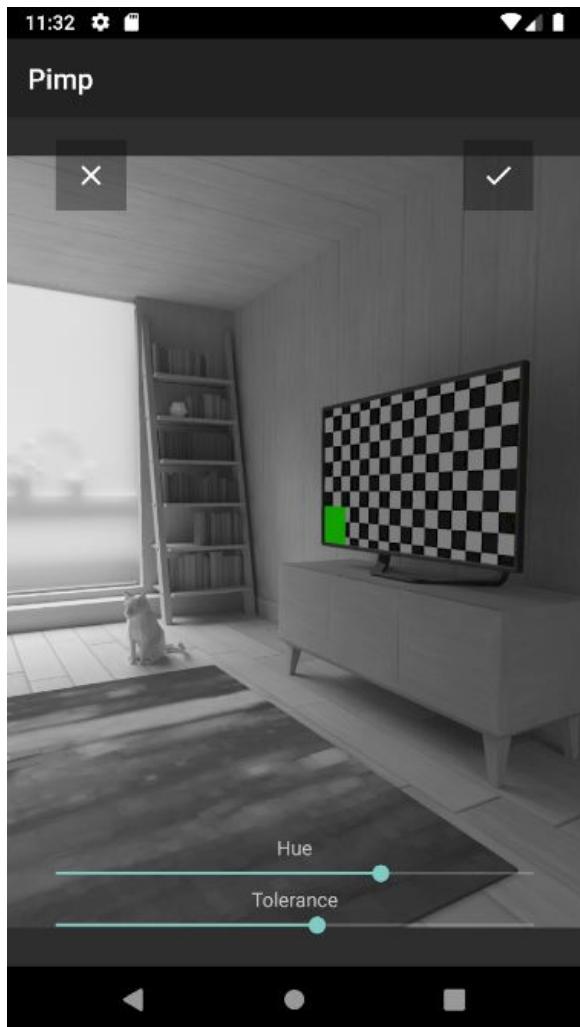
Le bouton **i** quand à lui affiche les informations suivantes à propos de l'image :



Les sections et permettent d'afficher des informations respectivement sur le fichier image, sur sa date de prise de vue, sur l'appareil source et sur ses coordonnées géographiques. Ces sections ne sont pas toujours visibles et dépendent du fichier image.

Quand à la section elle donne les dimensions de l'aperçu actuellement affiché dans l'application, il peut être plus petit que l'image d'origine afin de préserver la mémoire du téléphone et le temps d'exécution. L'image exportée en revanche sera aux bonnes dimensions.

Après avoir sélectionné un effet sur la liste en bas de l'écran, des éléments d'interface vont se superposer à votre image :



Le bouton ✕ en haut à gauche vous fait revenir à la liste d'effets et annule l'effet courant.

Le bouton ✓ en haut à droite applique l'effet et vous fait revenir à la liste d'effets.

En bas des curseurs ou des boutons peuvent apparaître, selon l'effet sélectionné. Manipulez les pour voir en temps réel l'impact sur l'image.

NB : L'entièreté de cette application est verrouillée en mode portrait.

3 Effets :

NB : Tous les effets nécessitant un histogramme utilise celui de la "valeur", soit le maximum entre les trois canaux de l'espace de couleur Rouge Vert Bleu.

3.1 Luminosité (Brightness) :



Méthode appelante : *Retouching.setBrightness()*

Script : *brightness.rs*

Ce réglage ajoute une valeur (positive ou négative) aux trois canaux RGB de l'image. Cette valeur est fixée par la seekbar. Les valeurs sont tronquées entre 0 et 255, par conséquent on perd de l'information dans les valeurs extrêmes de luminosité.

Cet effet n'utilise pas la luminosité existante de l'image, ainsi on peut obtenir des résultats qui sont parfois discutables, par exemple le noir qui s'éclaircit et inversement pour le blanc. Pour pallier à ce problème, on pourrait introduire une multiplication afin de modifier la luminosité proportionnellement à celle existante. Cependant, cette solution modifie aussi le contraste, nous avons donc choisi de laisser l'algorithme tel quel.

3.2 Contraste (Contrast) :

Méthode appelante : *Retouching.dynamicExtensionRGB()*

Script : *dynamicExtension.rs*

Ce réglage effectue une extension linéaire de dynamique. Les nouveaux extremum de l'histogramme sont définis à partir de la position de la seekbar. La dynamique est ainsi étendue autour d'une valeur se situant au milieu des deux anciens extremum de l'histogramme*. On a donc une image uniforme lorsque l'on règle le contraste au minimum. En augmentant le contraste, les extremum peuvent sortir de l'intervalle [0 ;255], ce qui provoque une distorsion de l'image.

*Il serait peut-être plus judicieux de prendre la médiane de l'histogramme cumulé afin d'avoir une valeur qui représente mieux la "valeur moyenne" de l'image.

3.3 Saturation (Saturation) :

Méthode appelante : `Retouching.setSaturation()`

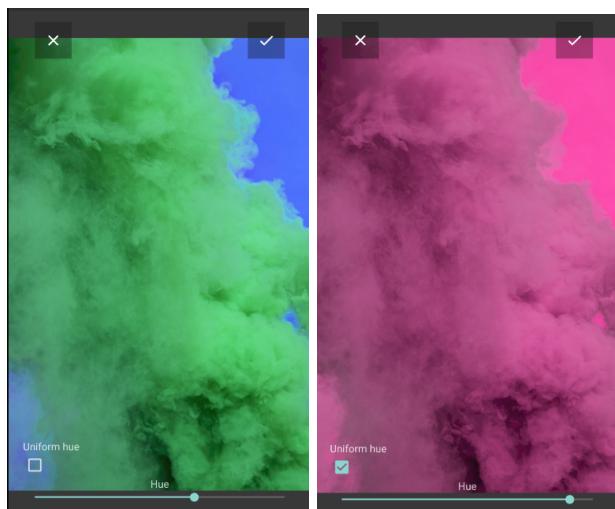
Script : `saturation.rs`

Ce réglage permet de régler la saturation de l'image. Soit S la saturation existante, S' la nouvelle saturation et F le facteur de saturation. S et S' vont de 0 à 1.

$$S' = S + F * (1 - S) * S.$$

On observe que la nouvelle saturation est proportionnelle à deux facteurs : l'espace restant avant une saturation totale (1-S) et la saturation existante S. Par conséquent, en augmentant la saturation, chaque pixel tend vers sa saturation maximale, tout en garantissant une saturation proportionnelle à celle existante, évitant ainsi de saturer le gris.

3.4 Hue :



Méthode appelante : `Color.colorize()`

Scripts : `colorize.rs, utils.rs`

Ce réglage effectue une conversion des couleurs de chaque pixel du format RGB au HSV puis change le champ Hue du pixel ce qui permet de changer le teint de l'image. Le teint en question est sélectionné par l'utilisateur grâce à la seekbar qui lui est fournie ainsi qu'une option à cocher qui lui propose deux choix :

3.4.1 Teint uniforme :

En cochant la case "uniforme" l'effet remplace le teint de tous les pixels de l'image en changeant le champ hue de chaque pixel par celui choisi par l'utilisateur grâce à la seekbar.

3.4.2 Teint non uniforme :

En décochant la case "uniforme" ajoute le teint sélectionné par l'utilisateur grâce à la seekbar au champ hue déjà contenu dans chaque pixel donnant un nouveau teint à chaque couleur de l'image.

3.5 Keep hue :



Méthode appelante : Color.keepColor()

Scripts : keepColor.rs, utils.rs

Garde qu'un seul teint avec un degré de tolérance sélectionnés par l'utilisateur grâce aux deux seekbars "hue" et "tolerance" qui lui sont fournies et garde le reste des pixels en échelle de gris en convertissant les couleurs de chaque pixel du format RGB au HSV puis gardant que les pixels contenant le hue choisi dans le bon intervalle.

3.6 Egalisation d'histogramme (Enhance) :

Méthode appelante : Retouching.histogramEqualization()

Scripts : cumulativeHistogram.rs, assignLut.rs

Comme son nom l'indique, cet effet utilise l'égalisation d'histogramme afin d'améliorer le contraste. On calcule d'abord l'histogramme cumulé et on en déduit la LUT (Look Up Table), que nous assignons ensuite à chaque pixel.

Cependant, afin d'égaliser l'histogramme, l'algorithme éclaircit les zones sombres, ce qui donne un résultat peu convaincant sur les images de faible luminosité. Une solution à ce problème est l'égalisation d'histogramme adaptative (CLAHE), mais nous n'avons pas eu le temps de l'implémenter pour ce rendu.

3.7 Convolution (Blur, Sharpen, Neon) :

Dans cette sous-section on retrouve des effets réalisés avec des convolutions pour flouter une image en passant par deux types différents de kernel : Gaussien et moyen-neur (bouton "Blur"), des effets pour améliorer la netteté d'une image avec la fonction "Laplacian of gaussian" (bouton Sharpen) et des effets de détection de contours (bouton "Neon") en réalisant des convolutions avec des opérateurs comme Sobel, Prewitt, Kirsch ou une convolution simple avec un kernel laplacien.

Pour ces effets nous avons implémenté 3 méthodes de convolution.

- Convolution classique.
- Convolution séparable.
- Convolution détection de contours.

Convolution classique

Cette méthode de convolution suit le même algorithme vu en cours, on peut l'appliquer avec des noyaux asymétriques de dimensions impaires uniquement, cette opération a été parallélisée grâce à l'outil "renderscript" qui parallélise le calcul par CPU multithreading/GPU. En plus dans cette méthode (et dans toutes les autres) nous avons rajouté des optimisations pour parcourir seulement les index nécessaires de l'image lors de la convolution. En calculant les index des centres des deux dimensions du kernel on peut anticiper et éviter les appels de fonction inutiles sur les bords de l'image par exemple.

$X_{début}$ et X_{fin} étant le premier et dernier index à parcourir dans l'axe des abscisses respectivement et $Y_{début}$ et Y_{fin} étant le premier et dernier index à parcourir dans l'axe des ordonnées respectivement.

$$Kernel_{CentreX} = \frac{Largeur_{kernel}}{2}$$

$$Kernel_{CentreY} = \frac{Hauteur_{kernel}}{2}$$

$$X_{début} = Kernel_{CentreX}$$

$$X_{fin} = Largeur_{image} - Kernel_{CentreX}$$

$$Y_{début} = Kernel_{CentreY}$$

$$Y_{fin} = Hauteur_{image} - Kernel_{CentreY}$$

En définissant les index du début et de fin de parcours pour les deux dimensions de l'image sur le script de lancement renderscript comme ci-dessus on peut s'en passer de quelques appels de fonctions sur les bords de l'image et ainsi gagner du temps de calcul.

Convolution séparable

Pour la convolution classique avec un kernel de taille $N * M$ on doit faire $N * M$ multiplications pour chaque échantillon, cependant si le kernel est séparable on peut passer à $N + M$ opérations. Voir 9.

Or afin d'optimiser le calcul de la convolution sur des filtres séparables comme le filtre gaussien et moyenneur nous avons implémenté une méthode de convolution séparable.

$$\begin{bmatrix} A \cdot a & A \cdot b & A \cdot c \\ B \cdot a & B \cdot b & B \cdot c \\ C \cdot a & C \cdot b & C \cdot c \end{bmatrix} = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \cdot [a \ b \ c]$$

Un kernel est séparable quand sa matrice des poids peut être représentée par le produit de deux vecteurs.

Ce calcul est réalisé en faisant deux convolutions unidimensionnelles successives (en X et en Y) sur l'image d'origine en stockant le résultat dans une image intermédiaire. La propriété d'associativité de la convolution rend ce calcul possible.

$$x * (N * N) \iff (x * N) * N$$

La limite de cette méthode c'est que l'on doit passer par une image partielle pour la réalisation du calcul en utilisant plus de mémoire que pour une convolution classique.

Convolution détection de contours

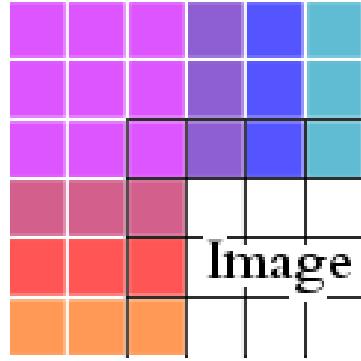
Cette méthode prend en paramètre deux kernels de même taille et réalise deux convolutions avec chaque kernel et ensuite calcule le produit des deux. Vu que ces deux convolutions sont indépendantes l'une de l'autre on peut se permettre de les réaliser au même temps dans le script et de faire l'addition des valeurs absolues résultantes des deux accumulateurs. Comme son nom l'indique cette méthode est utilisée notamment pour faire des convolutions pour la détection de contours avec des opérateurs comme Sobel, Kirsch, Prewitt, etc.

Padding

Pour éviter l'effet conséquent de la convolution qui est la perte de bords, on applique un padding par extension depuis renderscript.

Les kernels

Tous les kernels sont définis dans la classe fr.ubordeaux.pimp.util.Kernels. Les kernels de détection de contour (Sobel, Kirsch, Laplace, Prewitt) sont définis en variable



Exemple de padding par extension.

statique et finale, car leur taille est fixe, les autres sont générés par une méthode. Si on prend le cas du kernel de Gauss, la méthode gauss permet de générer la version séparable du kernel, c'est-à-dire une seule ligne. Les méthodes génératrices de kernel prennent en argument la taille de celui-ci, ce qui sert aux effets paramétrables tels que blur et sharpen.

3.7.1 Flou gaussien et moyenneur (Blur) :



Comparaison des deux filtres Gaussien(2ème image) et Moyenneur(3ème image) sur une image de taille 4000x3000px.

Méthodes appelantes : Convolution.gaussianBlur(), Convolution.meanBlur()

Scripts : convolution.rs

Dans la section blur, on réalise des convolutions avec un filtre gaussien ou moyenneur selon le choix de l'utilisateur, ces filtres sont réalisés avec une méthode de convolution séparable voir 3.7. Ces kernels sont générés en fonction du progrès de la seekbar qui définit une taille de vecteurs allant de l'intervalle [3 - 25]. Pour le filtre gaussien l'écart type noté σ est généré en fonction de la taille du kernel en suivant cette relation.

$$\sigma = \frac{\text{taille}_{kernel}}{2}$$

Cette relation a été définie arbitrairement car il n'y a pas moyen de calculer avec certitude l'écart type de la fonction gaussienne et en fonction des résultats obtenus après une série des tests on a décidé de laisser la relation de ci-dessus.

Pour le filtre moyenneur tout comme le filtre gaussien on génère un vecteur de taille dans l'intervalle [3 - 25] et on les applique verticalement et ensuite horizontalement avec la méthode de convolution séparable.

Cet effet s'applique en temps réel sur l'image et pourtant doit être assez rapide pour son utilisation, il restent encore quelques optimisations à faire pour améliorer la fluidité de ce dernier. Voir 3.7.4.

3.7.2 Amélioration de la netteté / Masque flou (Sharpen) :



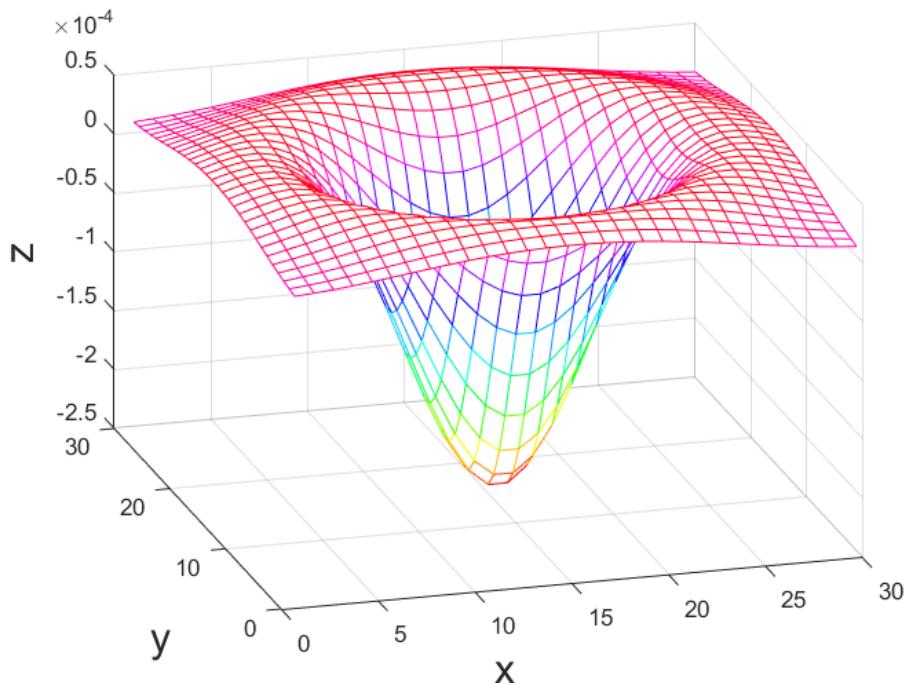
Application de filtre LoG, kernel de taille 13x13, $\sigma = 2.6$ et image de 4000x3000px



Application de filtre LoG, kernel de taille 9x9, $\sigma = 1.8$ et image de 1024x724px

Méthode appelante : `Convolution.sharpen()` Scripts : `convolution.rs`

Cet effet consiste à rehausser les contours dans une image également que d'enlever le possible flou dans une image, avec une seekbar en progression. Le filtre est réalisé par la convolution d'un filtre suivant une distribution discrète de la fonction "La-place of Gaussian" (LoG) voir 9, ce filtre est pourtant séparable en 4 convolutions mais pour histoire d'implémentation et de temps on s'est contenté de garder la version avec la convolution classique. Ce filtre est réalisé en temps réel avec des kernels de taille en allant de taille 3x3 jusqu'à 13x13.



Représentation graphique de LoG

Le filtre Laplacien étant très sensible au bruit, c'est pour ceci que généralement on applique une convolution avec un filtre gaussien avant pour supprimer ce bruit, cependant avec LoG on peut appliquer une distribution Gaussienne au filtre Laplacien afin d'éviter de faire 2 convolutions.

Par ailleurs pour reproduire cet effet de rehaussement de contours on peut flouter l'image originale avec un filtre gaussien (suppression du bruit), ensuite calculer la charte de contours avec un filtre laplacien et puis combiner l'image d'origine avec le résultat de la charte de contours. Grâce à cette fonction on peut compresser cette procédure à une génération de kernel suivi d'une seule convolution de ce dernier.

Comme pour les filtres précédents l'écart type du LoG est défini comme suit :

$$\sigma = \frac{\text{taille}_{kernel}}{5}$$

Comme pour l'écart type gaussien ce sigma a été choisi en fonction des résultats.

3.7.3 Détection des contours (Néon) :



Opérateur de Sobel sur une image de taille 4000x3000px



Opérateur de Sobel sur une image (passée en échelle de gris) de taille 860x795

Méthode appelante : *Convolution.neonSobel()*

Scripts : *convolution.rs*

Cet effet consiste à créer un effet "Neon" sur une image en utilisant des opérateurs de détection de contours (voir 9) comme Sobel, Prewitt et aussi d'un filtre Laplacien qui est également détecteur de contours. On a mis 3 boutons de type **RadioButton** pour sélectionner le type de kernel à utiliser et ainsi voir les différences de chacun sur l'image.

Les valeurs résultantes de la convolution en "Neon" ne sont pas normalisées par extension linéaire, elles sont justes tronquées entre [0 - 255] à cause de ce troncage on risque de perdre des données sur l'image, surtout avec des images avec beaucoup des contours, pour avoir donc un résultat plus fidèle, faudrait passer par une image intermédiaire, calculer leurs respectifs minimum et maximum pour ensuite normaliser l'image précédente cette solution est discutable puisque on devrait réalloquer une image de plus en mémoire et rajouter quelques parcours en plus sur l'image.

Cependant nous avons essayé de **calculer la valeur maximale théorique** pour normaliser mais ceci donne des résultats très sombres par rapport à l'image originale, nous avons décidé donc de garder l'ancienne normalisation et considérer le fait de normaliser par extension linéaire.

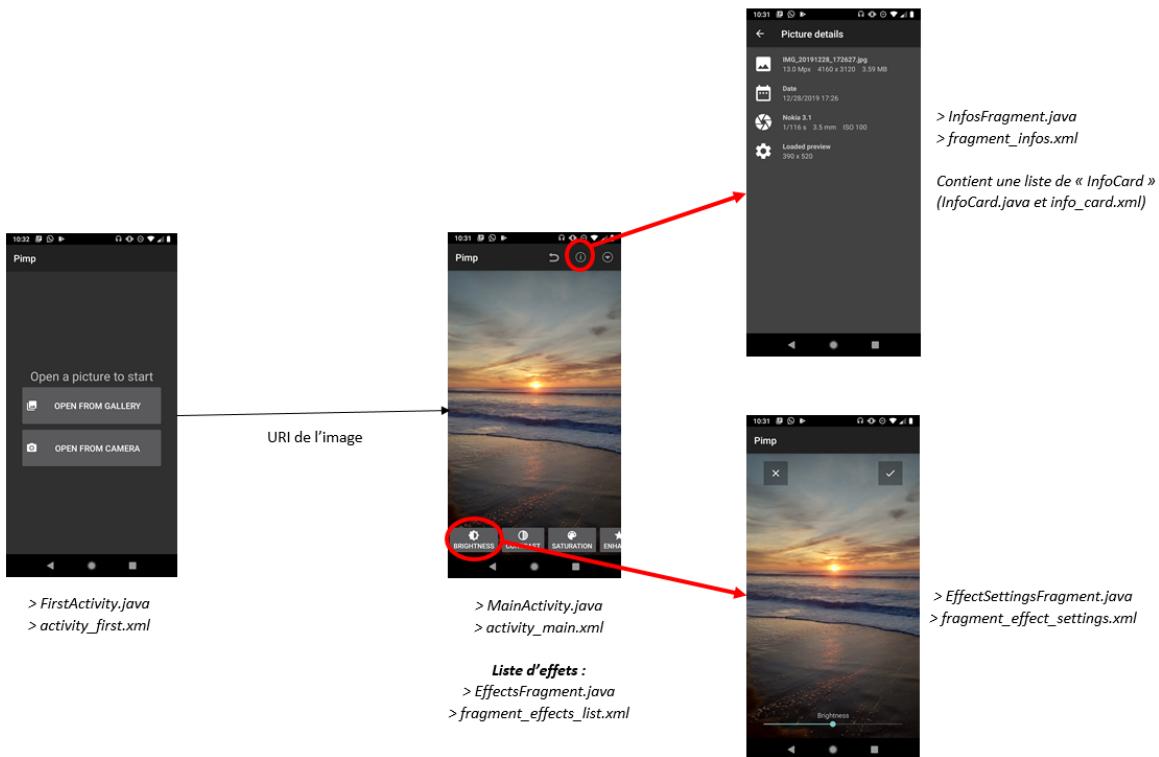
3.7.4 Remarques et limites

- Un des points de réflexion sur la convolution c'est que jusqu'à présent elle est réalisée sur les 3 canaux RGB, ce qui est assez lourd au niveau de calcul, cependant on pourrait passer l'image en échelle de gris et la faire seulement sur un des canaux ou passer pour l'espace HSV en utilisant la value ou la saturation.
- Tous les calculs de convolution sont uniquement réalisées avec des nombres "flottants", ceci est un choix d'implémentation car renderscript, possède plus de support pour réaliser des opérations flottantes que pour des opérations sur des entiers qui en plus sont codés en 8 bits dans une image. Pour éviter donc les erreurs de débordement et pour garder une bonne qualité de l'image on a décidé de travailler exclusivement avec des flottants, même si ceci peut être pénalisant au niveau de performance, cependant des tests sont à prévoir pour le prochain rendu pour essayer d'améliorer la performance sur des filtres comme "Blur"3.7.1 et "Sharpen"3.7.2.
- Pour la convolution classique et séparable les valeurs sont normalisées en prenant le résultat d'opération de voisinage pour un pixel donné divisé par la somme des poids des éléments du kernel. Ceci marche assez bien sauf pour les kernels avec des valeurs négatives notamment ceux de détection de contours comme Laplace, Sobel, Prewitt, etc (Voir 3.7).
- Il y a quelques améliorations de performances à faire pour les convolutions réalisées en temps réel comme pour les effets Blur et Sharpen, une optimisation par échantillonnage linéaire⁹ est à tester ainsi que le calcul par des entiers au lieu de flottants.
- Une des plus grandes limites des effets de convolution c'est que dû à notre implémentation de sauvegarde^{4.2.2}, on ne peut malheureusement pas garantir le même résultat sur l'image sauvegardée que sur l'aperçu. Vu que la réponse des filtres de convolution sur une image précise sont fortement liés à la taille de l'image et que l'image d'aperçu est en général plus petite que l'originale, de fois on peut se retrouver avec une image sauvegardée avec un flou moins fort ou avec une détection de bords plus faible que sur l'aperçu ainsi qu'une sensibilité plus forte au bruit.^{6.1}

4 Structure du projet

NB : La grande majorité de notre application a été réalisée à partir des librairies officielles Android. Nous avons nous mêmes implémenté nos effets. Nous avons sinon utilisé les composants graphiques fournis avec AndroidStudio (dont certains à installer, notamment **RecyclerView**). Enfin nous avons importé le composant **PhotoView** en plus des composants Android.

4.1 Structure graphique Android et navigation :



Pour afficher certains éléments d'interface comme par exemple les informations de l'image (bouton ⓘ) nous utilisons des **Fragment**. En effet une activité supplémentaire n'est pas nécessaire car cette petite partie de l'application ne correspond pas à un point d'entrée de l'application. Par ailleurs changer de fragment (plutôt que de changer directement de layout) pourrait faciliter l'implémentation d'une interface différente, pour tablette par exemple. De même, la liste d'effets et leurs paramètres respectifs sont aussi contenus dans des fragments séparés. Cela permet de gérer plus simplement leur affichage et de clarifier le code.

On notera que dans la structure actuelle de l'application, l'image actuellement éditée est contenue et manipulée depuis l'activité principale. Les fragments n'apportent à l'application que des éléments d'interface.

Une seconde activité est cependant utilisée pour la page d'accueil à l'ouverture

de l'application, cette **FirstActivity** utilise des méthodes génériques de **ActivityIO** afin de gérer l'ouverture de la galerie ou de la caméra. L'application reste dans cette activité tant qu'une **Uri** valable (\approx chemin) n'a pas été sélectionnée. Ensuite cette Uri est transférée à **MainActivity** qui va alors charger cette première Image, en cas de problème de chargement l'application peut retourner dans FirstActivity.

4.2 Classe Image

Cette classe a été conçue comme une alternative à l'utilisation directe de la classe **Bitmap** fournie par Android.

Le cœur de la classe est évidemment une instance de Bitmap, qu'il est possible de récupérer à tout moment. Par ailleurs la classe offre des fonctionnalités supplémentaires, parmi celle-ci notamment la possibilité de restaurer l'image à son état au moment de sa création ou de son chargement via la méthode **reset()**, ou d'annuler les dernières modifications apportées par un effet grâce aux méthodes **quicksave()** et **discard()**.

On notera la nécessité pour Image d'avoir la référence d'une Activité de l'application, en effet elle est requise à plusieurs moments par les librairies Android pour charger la Bitmap en mémoire.

4.2.1 Classe ImageInfo

La classe Image 4.2 génère et garde une instance de la classe **ImageInfo**, cette classe contient un grand nombre de valeurs à propos de l'Image (dimensions, coordonnées GPS, date de prise de vue, ...).

L'idée de cette classe était d'emballer toutes ces informations afin de faciliter le passage de ces informations à travers des Fragments ou des Activités (voir 4.1). On notera que tous les accesseurs appliquent des opérations de formatage sur ces données, certaines opérations pourraient être déplacées dans les constructeurs si elles venaient à être utilisées régulièrement.

4.2.2 File d'effets

Pour pouvoir exporter l'image éditée, il est nécessaire de réappliquer sur l'image d'origine tous les effets appliqués à l'image affichée dans l'application.

C'est pourquoi la classe Image permet d'emballer une Queue de **BitmapRunnable**. Les BitmapRunnable permettent de transporter des méthodes d'effets, en effet en définissant la méthode **run()** à l'instanciation de ces objets on peut emballer un effet applicable à un objet **Bitmap**. Ce qui permet de passer n'importe quel effet venant de n'importe quel auteur, ce qui renforce la modularité des effets et de la classe Image.

Dans la version actuelle cette Queue d'effets n'est qu'une liste "bloc-note" que l'utilisateur de Image peut utiliser, dans cette application les effets appliqués sont notés au fur et à mesure et cette liste est ensuite récupérée à l'export de l'image.

4.3 AsyncTasks

Les 3 classes du package **fr.ubordeaux.pimp.task** héritent de la classe Android **AsyncTask** et permettent d'exécuter certaines opérations en arrière plan, donc sans bloquer l'interface utilisateur.

Ainsi **LoadImageUriTask** et **ApplyFilterQueueTask** permettent respectivement de charger une image et de l'exporter, elles s'occupent d'afficher un petit logo de chargement et de lancer les calculs.

ApplyEffectTask est un peu différente, elle permet de mettre en arrière plan le calcul d'application d'un effet. Elle fonctionne pour n'importe quel effet, pour ce faire elle a besoin d'un paramètre, le type d'effet sous la forme d'un **Runnable**.

4.4 Classes d'effets

Les effets applicables dans notre application sur des Bitmap sont rangés dans 3 classes du package **fr.ubordeaux.pimp.filters**.

Retouching contient les effets les plus classiques d'amélioration d'une image et notamment ceux utilisant des histogrammes (luminosité, contraste, et saturation).

Color contient tout les effets ayant un impact sur la colorimétrie de l'image (modification de teinte, noir et blanc et sélection de couleur).

Convolution contient tout les effets utilisant des méthodes de convolution (flou, détection de contour etc).

NB : Tous nos effets ne sont pas implémentés directement en Java comme le reste du code mais font appel à des scripts RenderScript en C, ce qui accélèrent considérablement l'application des effets.

4.5 Fragment EffectSettingsFragment

Ce fragment un peu particulier joue le rôle le plus important dans l'application des effets sur l'image, en effet selon le bouton d'effet sélectionné il génère les curseurs nécessaires au paramétrage de l'effet.

Leur nombre peut varier d'un effet à l'autre mais également leur fourchette de valeurs, ainsi cette classe convertit les méthodes d'effets et leurs arguments en éléments d'interfaces pour l'utilisateur.

4.6 Packages utilitaires

De nombreuses classes du code permettent une factorisation et une clarification du code en offrant des méthodes pratiques, généralement statiques. Parmi ces classes on pourra retrouver :

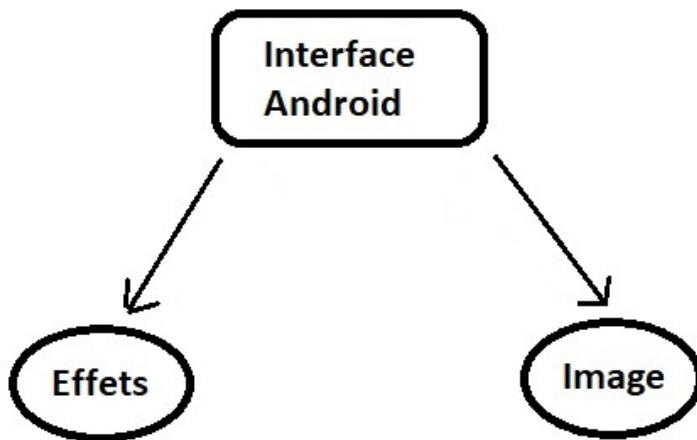
La classe **Utils** qui offre des méthodes pour récupérer la taille de l'écran, pour calculer un ratio de redimensionnement, pour manipuler des chemins d'image ou régler des problèmes d'orientation d'image.

La classe **BitmapIO** permet d'effectuer le chargement d'une Bitmap de plusieurs manières, depuis les resources ou un autre emplacement du téléphone, et avec la taille voulue.

La classe **ActivityIO** permet de gérer l'ouverture de l'application de galerie ou de caméra et d'en récupérer le retour, le tout en gérant les permissions de l'application.

Enfin la classe **Kernels** offre des méthodes de génération de noyau de convolution permettant de créer différents effets.

4.7 Modularité du code



Notre code a été conçu en 3 grandes parties, la partie principale regroupe toutes les activités, les fragments, les tâches, ... Cette partie utilise des instances de la classe Image, et fait appel à nos méthodes d'effets.

Cette structure permet une grande modularité du code, notre classe Image tout comme nos effets sont facilement exportables pour être utilisés sur un autre projet, on peut imaginer créer une autre application avec les mêmes effets mais une interface complètement différente.

A l'inverse si nous voulions récupérer des effets sur un autre projet il est tout à fait possible de les ajouter au projet, il suffira alors de rajouter en conséquence un bouton dans notre interface et un comportement pour **EffectSettingsFragment** 4.5

5 Performances :

Tous les tests de performance présentés dans cette section ont été effectués sur un Nokia 3.1. Voici un résumé de ses caractéristiques :

- **Version** : Android 9
- **Résolution** : 1440 x 720 pixels
- **Cadence processeur** : 1.5 GHz
- **RAM** : 2 Go

5.1 Temps d'exécution :

Les temps d'exécution présentés ne tiennent pas compte du premier temps de chargement du script renderscript.

Voici les temps d'exécution **sur 10 appels** de chaque effet pour une image de dimension **425x265px**.

Effet	Temps d'exécution en ms (min max moyenne écart-type)
Brightness	12.0 46.0 16.9 9.73
Contrast	16.0 28.0 20.5 3.80
Saturation	13.0 23.0 15.2 2.929
Enhance	22.0 32.0 26.4 3.00
To gray	9.0 11.0 10.0 0.63
Invert	8.0 10.0 8.4 0.66
Change hue	11.0 18.0 13.4 1.90
Keep hue	10.0 11.0 10.6 0.48
Gaussian blur 3x3	17.0 21.0 18.8 1.24
Gaussian blur 25x25	47.0 51.0 48.5 1.36
Mean blur 3x3	16.0 19.0 18.2 1.07
Mean blur 25x25	54.0 60.0 55.8 1.72
Sharpen 3x3	21.0 25.0 23.6 1.11
Sharpen 13x13	136.0 165.0 146.3 8.97
Sobel filter	21.0 31.0 24.7 2.53
Prewitt filter	24.0 27.0 25.3 1.09
Laplacian filter	22.0 26.0 23.6 1.11

Voici les temps d'exécution **sur 10 appels** de chaque effet pour une image de dimension **3400x2118px**.

Effet	Temps d'exécution en ms (min max moyenne écart-type)
Brightness	105.0 311.0 145.8 56.95
Contrast	163.0 202.0 176.9 12.73
Saturation	209.0 236.0 221.4 8.34
Enhance	256.0 268.0 261.3 3.74
To gray	125.0 138.0 129.6 3.35
Invert	84.0 105.0 94.7 8.69
Change hue	218.0 319.0 231.7 29.48
Keep hue	162.0 181.0 168.4 5.06
Gaussian blur 3x3	245.0 265.0 256.4 6.63
Gaussian blur 25x25	1453.0 1484.0 1462.2 9.08
Mean blur 3x3	245.0 652.0 358.5 146.25
Mean blur 25x25	1461.0 2447.0 1716.6 318.34
Sharpen 3x3	433.0 524.0 471.7 25.25
Sharpen 13x13	5112.0 8974.0 5679.2 1119.49
Sobel filter	450.0 497.0 469.4 14.47
Prewitt filter	452.0 490.0 472.5 10.39
Laplacian filter	411.0 533.0 460.4 40.08

Conclusion sur les temps d'exécution

Mesurer les temps d'exécution nous a permis de nous rendre compte du gain de temps de la convolution séparable. Le flou gaussien (implémenté en séparable) 25x25 prend 1462.2 ms, tandis que le sharpen 13x13 (non séparable) prend 5679.2 ms.

Sharpen 3x3, Sobel, Prewitt et Laplacian ont le même temps d'exécution car elles correspondent toutes à des convolutions classique avec un kernel 3x3.

De plus, on remarque que l'effet "Invert" est le plus rapide. Cela s'explique par le fait que c'est le seul effet qui manipule directement des entiers. Peut-être pourrions-nous appliquer cette méthode à d'autres effets tels que la convolution, afin d'optimiser les temps d'exécution.

5.2 Mémoire :

Voici l'utilisation de la mémoire vive du téléphone au cours de l'utilisation de l'application. On démarre le benchmarking à partir du chargement de l'image (après `firstActivity`), puis on applique tous les effets disponibles en annulant à chaque fois.

Image de 425x265px

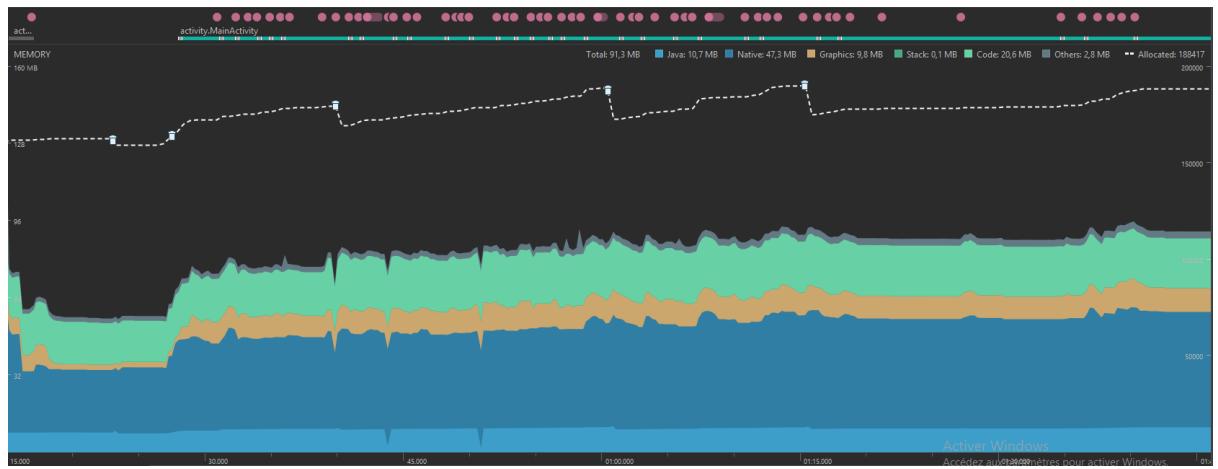
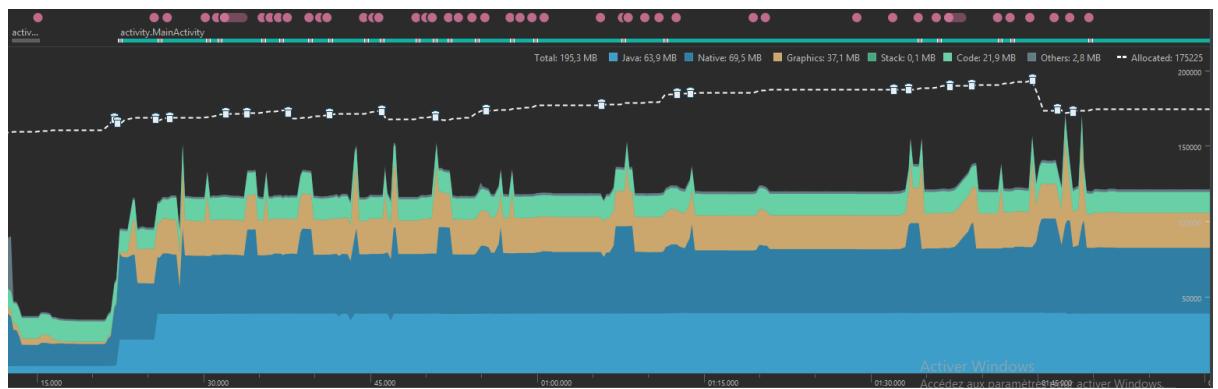


Image de 3400x2118px



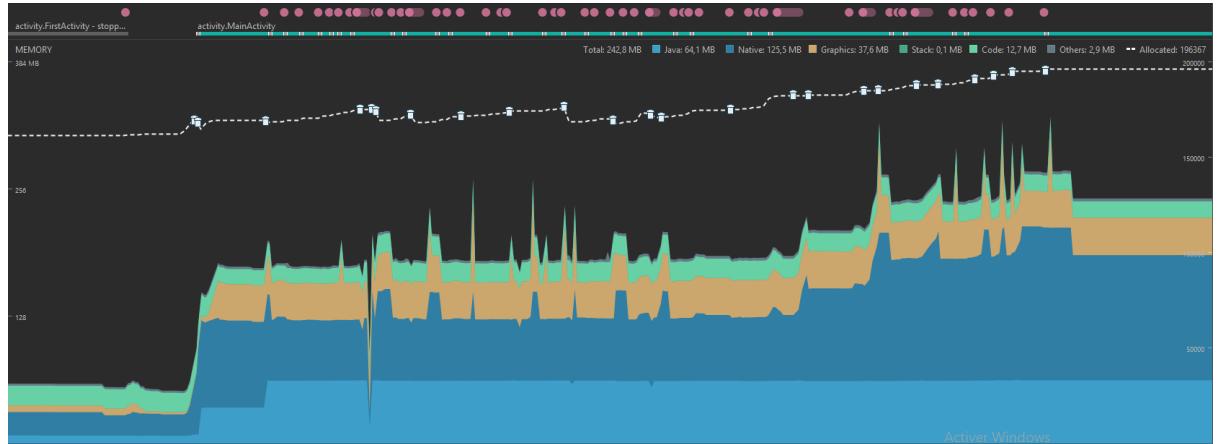
Conclusion sur l'utilisation de la mémoire

On observe que la mémoire est plutôt bien gérée et qu'elle reste constante tout au long de l'utilisation. De plus, le garbage collector vient régulièrement libérer la mémoire allouée.

En analysant le tas de l'application ("app heap" sur le profiler), on se rend compte que la sauvegarde de l'image (effectuée avec la méthode `quicksave()`) y occupe la plus grande place. La sauvegarde est stockée sous forme de tableau d'entiers, on peut donc difficilement le rendre moins lourd en mémoire. En revanche, la file d'effets (4.2.2) pourrait nous permettre d'annuler les modifications sans avoir à stocker une sauvegarde. En effet, si l'utilisateur a appliqué n effets et qu'il souhaite annuler la dernière modification, on peut ré-appliquer tous les effets jusqu'à n-1 sur l'image d'origine. Cependant, on perdrait évidemment en temps.

Bug détecté

Le profiling de la mémoire nous a permis de détecter un problème que l'on a résolu. En effet, on remarquait une augmentation constante de la mémoire lorsque l'on appliquait un flou :



C'était du à notre méthode `convolve2dSeparable`. La convolution séparée nécessite une allocation temporaire pour stocker le résultat de la première convolution (horizontale). Elle était créée puis libérée directement depuis notre script `convolution.rs` (méthode `convolutionSeparable`). Cependant, la libération ne fonctionnait apparemment pas, nous avons donc géré l'allocation de ce bitmap temporaire depuis java, ce qui a réglé le problème.

6 Remarques et améliorations :

6.1 Remarques sur le code

A propos de la classe **Image** 4.2, nous sauvegardons systématiquement une copie des pixels d'origine de l'Image, de plus si un appel à **quickSave()** est effectué une 3ème copie de l'Image est chargée en mémoire. Image offre cependant des constructeurs pour charger une image proportionnée à l'écran de l'appareil. Le risque de débordement mémoire est donc largement évité par cette limitation de taille.

Lors du chargement d'une nouvelle image, nous ré-instancions un objet de la classe Image. Ce qui veut techniquement dire que jusqu'au prochain passage du ramasse miette Android, deux images sont en mémoires, donc deux Bitmap et deux tableaux de pixels (la copie originale des Images, voir 4.2). C'est un élément discutable cependant notre application limite la taille des Images chargées. Ce qui évite largement les dépassemens mémoire.

Dans la partie 4.1 nous créons une **MainActivity** après avoir récupéré une **Uri**, il aurait été idéal de rester dans **FirstActivity** jusqu'à chargement complet de la première image. Ce qui éviterait un éventuel aller-retour entre les activités en cas d'erreur. Cependant les limites de **Parcelable** 6.2 ainsi que l'utilisation d'une **AsyncTask** (donc un Thread différent) nous ont contraint à garder ce fonctionnement.

Comme dit précédemment nous chargeons une image plus petite que l'originale afin d'optimiser la fluidité de l'application, à l'export en revanche le fichier image d'origine est chargé pour subir la même suite d'effets. Cette méthode ne permet pas d'empêcher un crash mémoire si l'image est immense (image d'appareil photo par exemple). On pourrait à l'avenir complexifier l'export pour charger et sauvegarder par morceaux le fichier image. Cependant ce n'est pas une priorité, la majorité des utilisateurs de l'application vont éditer des images venant d'appareils mobiles qui dépassent rarement les 20Mpx. Nous avons défini une limite de 5000x5000px comme taille d'image en entrée, les images au dessus de cette taille seront redimensionnées en facteur x^2 à la sauvegarde pour assurer la fluidité de l'application.

Le fait de charger l'image d'origine pour refaire les effets a comme conséquence que certains effets, notamment ceux de convolution aient une différence entre l'aperçu et l'image sauvegardée. Une possible solution à ce problème pourrait être d'afficher ou travailler avec un "crop" de l'image d'origine pour ces effets là.

6.2 Remarques sur les librairies Android

Lors de la construction des instances d'**Image** 4.2, nous devons passer la référence de l'activité contextuelle à l'Image, bien que pas très intuitif cette référence est nécessaire car utilisée par les méthodes de **Bitmap** de chargement d'image.

Pour manipuler des objets d'une activité à l'autre ou entre fragments, Android utilise des **Intent** ou des **Bundle**, passer des objets entiers devient assez lourd dans le code et nécessite l'utilisation de l'interface **Parcelable**, de plus passer un objet trop gros entraîne une **RuntimeException**. Finalement au sein d'une même application on peut se demander s'il est bien nécessaire de systématiser leur utilisation ou s'il ne serait pas plus simple de passer une référence ou simplement faire des accès statiques (au risque de perdre un peu la modularité du code).

6.3 Améliorations à court terme :

Il pourrait être intéressant de montrer des aperçus des effets en bas de l'écran dans la liste d'effets, pour cela il suffira de créer des instances d'`Image` générées à partir de l'image éditée mais en dimension inférieure. Les constructeurs sont déjà disponibles.

Il peut être possible dans les classes d'effets de rajouter des surcharges aux méthodes d'effets pour leur passer directement une instance d'`Image`. Il sera alors nécessaire de rajouter à `Image` une référence de son activité contexte, ce qui se fait facilement puisque l'image est chargée directement depuis `MainActivity` et y restera toujours.

Cette simplification anodine offrirait un tout petit peu plus de lisibilité du code, cependant son ajout est débatteable puisqu'elle rendrait les effets dépendants de la classe `Image`, ce qui romprait la modularité du code.

Dans la classe `Image` nous utilisons une file d'effets 4.2.2, pour l'instant cette file fournie par `Image` n'est pas vraiment reliée à la classe elle même dans le sens où l'utilisateur de la classe `Image` a le contrôle total de cette file.

Il pourrait être tout à fait possible de masquer la gestion de cette file, qui se remplirait de manière automatique, ainsi il sera possible de faire une méthode `export()` dans la classe `Image` pour sauvegarder l'image, sans se soucier de gérer cette file. De même il serait possible d'annuler les derniers effets en ré appliquant sur l'image d'origine tout ceux que l'on souhaite conserver.

Il faudrait cependant modifier le comportement de `quickSave()` et `discard()` afin de supprimer ou ré-ajouter la tête de file lors de l'aperçu des effets en glissant les curseurs.

Cependant la principale problématique de cette fonctionnalité serait de passer d'un système où l'on appelle les effets en passant l'`Image` (ou la `Bitmap`) en paramètre, à un système avec une méthode `applyEffect(...)` dans la classe `Image`. La question se pose alors de savoir comment préciser l'effet en question sans contraindre la classe `Image` à utiliser une batterie d'effets précise, ce qui romprait la modularité du code en rendant `Image` dépendant de nos classes d'effets. Ce problème est en réalité déjà réglé avec l'utilisation des `BitmapRunnable`, en effet il suffira de passer ces derniers à la méthode ce qui mettra une couche d'abstraction entre le terme "effet" et les classes de calculs des effets.

L'utilisation d'un historique d'effet pourra aussi permettre à l'utilisateur de sauvegarder des séries d'effets pour créer des effets personnalisés.

7 Gestion du projet :

7.1 Organisation générale :

Développer à plusieurs un code de taille moyenne sur une période de temps limitée nécessite une bonne organisation du projet pour limiter les conflits de code. En particulier sur un développement Android où des bugs propres à certains appareils apparaissent régulièrement.

Nous avons évidemment travaillé avec git pour gérer notre code, un workflow utilisant des branches a été mis en place, même si finalement les branches ont été moins nombreuses et plus volumineuses que prévu, leur utilisation a permis le développement en parallèle de beaucoup de fonctionnalités. Les parties concernant les effets, l'interface, la navigation et la structuration des classes ont ainsi pu être développées indépendamment.

Certaines fonctionnalités dépendaient cependant du développement d'autres fonctionnalités, c'est pourquoi en plus du *GitHub*, nous avions une liste de tâches à effectuer sur la plateforme *Trello*. De plus nous communiquons et débâtons sur la structure du code régulièrement par messagerie.

8 Conclusion

Bien que nécessitant une bonne rigueur de travail, travailler à plusieurs sur ce projet a donné un résultat très complet et très fonctionnel, notre application rivalise avec les autres applications du même type sur nos appareils personnels.

En plus d'offrir déjà de nombreux effets, notre application repose sur une base claire et modulaire, cela facilitera l'ajout d'effets supplémentaires d'ici le prochain rendu.

Nous allons nous concentrer sur l'optimisation des effets actuels et le peaufinage de la structure. Notre défi principal sera ensuite d'implémenter des effets très différents comme l'incrustation d'objets ou la possibilité de manipuler un masque sur l'image.

9 Annexes

Librairies utilisées :

<https://github.com/chrisbanes/PhotoView>

Cahier des charges :

<https://dept-info.labri.fr/~vialard/ANDROID/cahierDesCharges.pdf>

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

Convolution :

Convolution séparable

http://www.songho.ca/dsp/convolution/convolution2d_separable.html

Filtre gaussien

https://en.wikipedia.org/wiki/Gaussian_filter

Laplacian of gaussian

<https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm>

Noyaux

[https://fr.wikipedia.org/wiki/Noyau_\(traitement_d%27image\)](https://fr.wikipedia.org/wiki/Noyau_(traitement_d%27image))

Détection de contours

https://en.wikipedia.org/wiki/Edge_detection

Amélioration de netteté

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

Échantillonnage linéaire

<http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/>

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>

Divers :

Fichiers images et tags EXIF :

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