
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 :	2
2	L'application :	3
2.1	Lancement :	3
2.2	Utilisation :	3
3	Effets :	7
3.1	Luminosité (Brightness) :	7
3.2	Contraste (Contrast) :	8
3.3	Saturation (Saturation) :	8
3.4	Egalisation d'histogramme (Enhance) :	8
3.5	Convolution (Blur, Sharpen, Neon) :	9
3.5.1	Flou gaussien et moyenneur (Blur) :	11
4	Structure du projet :	12
4.1	Structure graphique Android et navigation :	12
4.2	Classe Image :	13
4.2.1	Classe ImageInfo :	13
4.2.2	File d'effets :	13
4.3	AsyncTasks	13
4.4	Packages utilitaires :	14
5	Performances :	15
5.1	Temps d'exécution :	15
5.2	Mémoire :	15
6	Remarques et améliorations :	16
6.1	Remarques sur le code	16
6.2	Remarques sur les librairies Android	16
6.3	Améliorations à court terme :	17
7	Gestion du projet :	18
7.1	Organisation générale :	18
8	Conclusion :	19
9	Annexes :	19

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 :

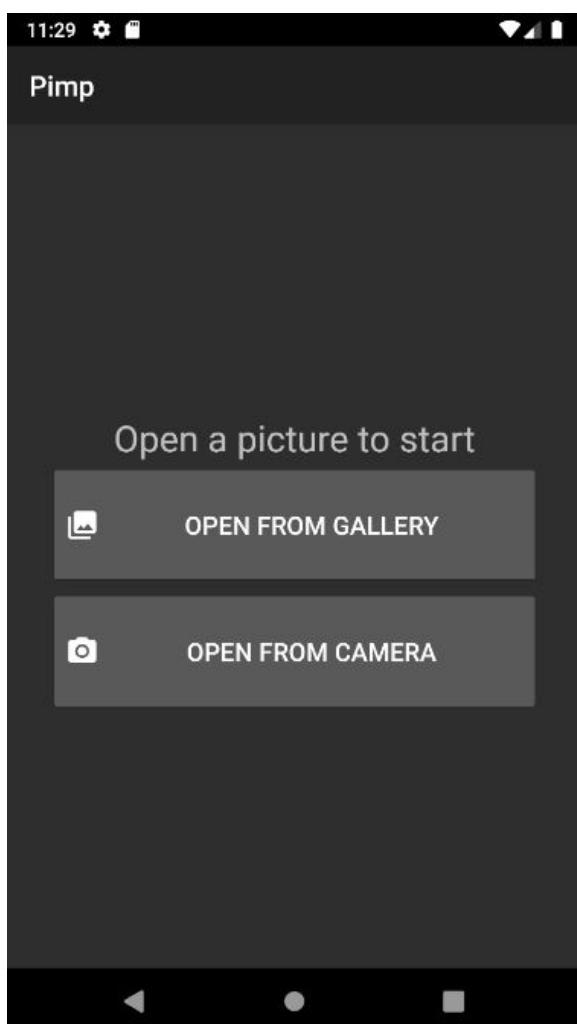
Version minimale Android requise : Oreo (8.0) (API level 26)

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.

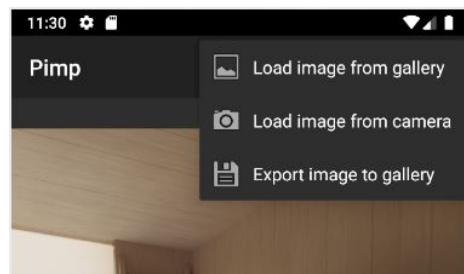
Ou appuyez sur le second bouton (📷) pour ouvrir la caméra de votre appareil Android. Capturez alors une image, elle sera sauvegardée dans votre galerie et instantanément importée dans l'application.

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 :

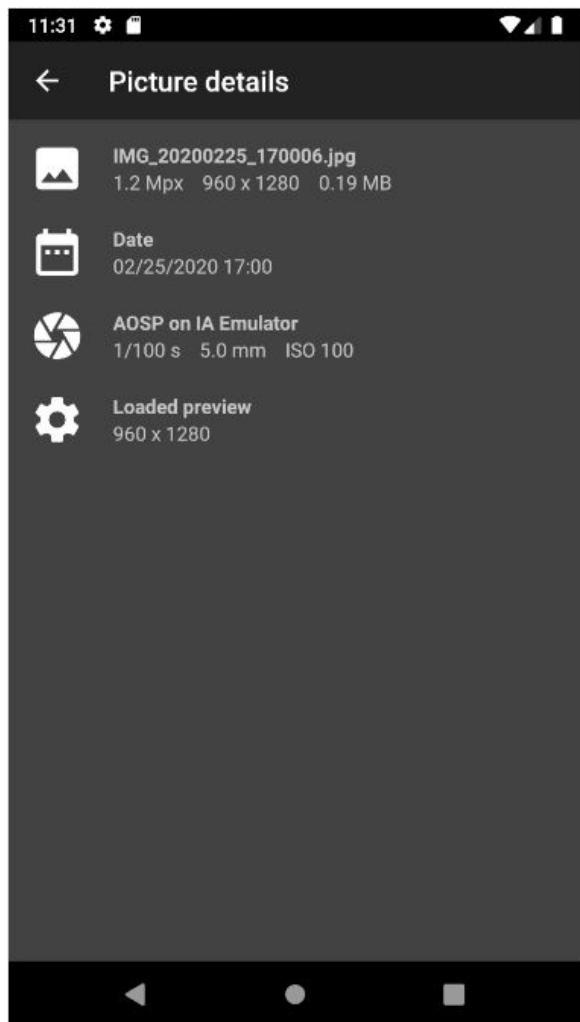


Le bouton permettra d'exporter votre image dans votre galerie, cette opération peut prendre un peu de temps car l'image exportée sera de même taille que l'image d'origine.

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.

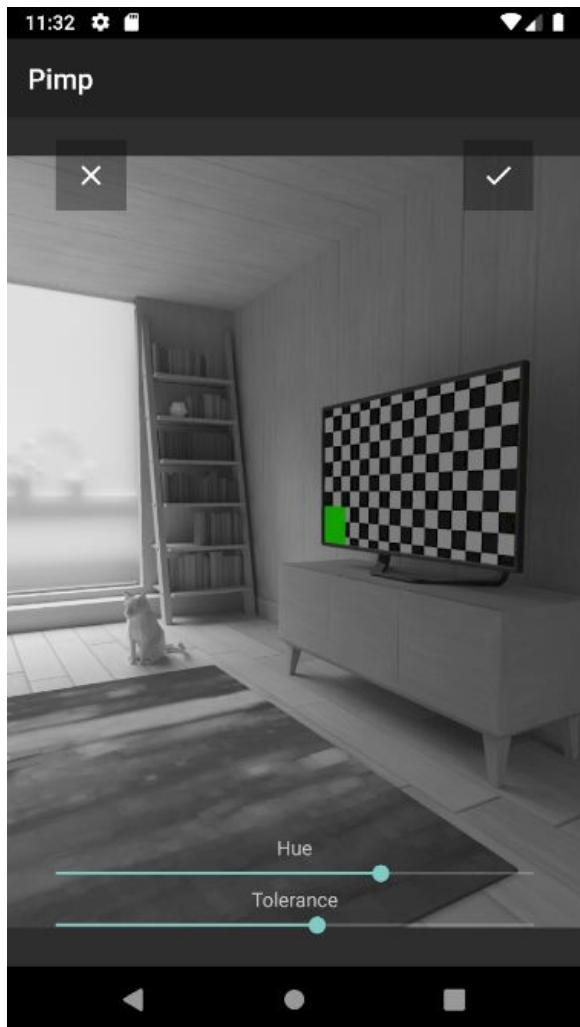
Le bouton **i** quand à lui affiche les informations suivantes à propos de l'image :



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 :

Interface :

L'utilisateur choisit un effet parmi la liste d'effets affichée en bas de l'écran, ce qui affiche les réglages disponibles (s'il y en a). L'utilisateur peut ensuite choisir de confirmer ou d'annuler la modification apportée par l'effet choisi.

Structure du code :

Les effets sont rassemblés dans le package filters (fr.ubordeaux.pimp.filters). La classe *Retouching* contient les réglages de luminosité, contraste, saturation et teinte. La classe *Convolution* contient tous les effets liés à la convolution (flou, détection de contour etc.). Toutes ces méthodes sont appelées lors de l'appui de boutons ou de glissement de seekbars. Les seekbars ont généralement une étendue allant de 0 à 255 (sauf pour les réglages de teinte), et certains effets nécessitent des valeurs pouvant être négatives. Ainsi la valeur de seekbar est modifiée dans la méthode appelante de ces mêmes effets.

Tous les effets nécessitant l'histogramme utilisent celui de la *valeur*, soit le maximum entre les trois canaux RGB.

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.

$$\text{On a } S' = S + F * (1 - S) * S.$$

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 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.

3.5 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.

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.

Un des points de réflexion sur la convolution c'est que jusqu'à présent on la réalise 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.

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 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 séparée 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.

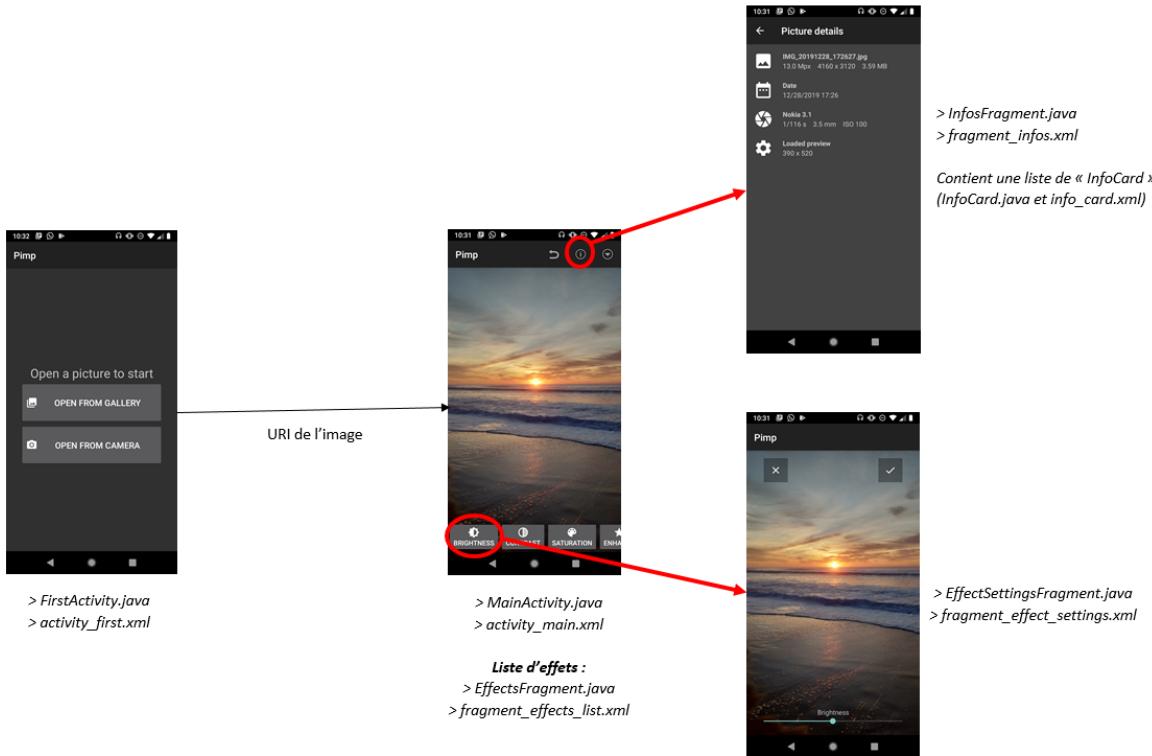
3.5.1 Flou gaussien et moyenneur (Blur) :



Méthode appelante : *Convolution.gaussianBlur()*
Scripts : *convolution.rs*

4 Structure du projet :

4.1 Structure graphique Android et navigation :



Pour afficher certains éléments d'interface comme par exemple les informations de l'image (bouton ❶) 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 (≈ 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'empaqueter 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 ré appliquer sur l'image d'origine tout les effets appliqués à l'image affichée dans l'application.

C'est pourquoi la classe Image permet d'empaqueter 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 empaqueter 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 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.

5 Performances :

5.1 Temps d'exécution :

Effets	Nokia 3.1	
	1024px	3072px
Brightness	0	0
Contrast (extension linéaire de dynamique)	0	0
Saturation	0	0
Enhance (égalisation d'histogramme)	0	0
Niveaux de gris	0	0
Invert	0	0
Hue	0	0
Keep hue	0	0
Gaussian blur	0	0
Mean blur	0	0
Sharpen	0	0
Neon - Sobel	0	0
Neon - Prewitt	0	0
Neon - Laplace	0	0
Restore	0	0

5.2 Mémoire :

TODO montrer l'ancienne instance d'image se faire garbage collecter ? voir remarques

TODO

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.

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'utilisations 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.

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 :

TODO

9 Annexes :

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

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