université de BORDEAUX

Licence 3 Informatique

Cahier des charges - Android

Ramstagram

GUILBAULT Maxime, BARRIERE Antoine, MOMPRIVÉ Rémi

20 avril 2017

Présentation du projet

Ce projet a été réalisé dans le cadre du cours de traitement d'images sous Android en 3e année de Licence Informatique à l'Université de Bordeaux. Notre groupe est composé de 3 étudiants qui sont BARRIERE Antoine, MOMPRIVÉ Rémi et GUILBAULT Maxime.

Le but de ce projet était de développer une application de traitement d'images sur smartphone avec système Android. Les images peuvent aussi bien être obtenues depuis la galerie du téléphone que directement depuis la caméra.

L'application est composée de fonctionnalités permettant d'afficher, de modifier et de sauvegarder des images. Les modifications possibles sont l'application de différents filtres simples (sépia, noir et blanc, etc) et plus avancés (filtre gaussien, filtre laplacien, filtre effet dessin, etc).

Nous avons utilisé un dépôt Github pour développer le projet. Vous pouvez le trouver à ce lien ci-dessous : https ://github.com/al-barriere/projet-RAM

Pour la création de ce projet nous avons testé notre application sur 3 téléphones différents qui sont Oneplus 3T (Android 7.1.1), LG G3 (Android 6.0) et orange phone (Android 4.4.4) ansi que sur des émulateurs de Nexus 5X (Android 7.1.1).

Table des matières

1 Implémentations réalisées **4**
1.1 1ère release . . . 4
1.2 2eme release . . . 4

2 Architecture du code **5**
2.1 Diagramme UML . . . 5
2.2 Fichiers Java . . . 5
2.2.1 Activities . . . 6
2.2.2 CustomViews . . . 6
2.2.3 Fragments . . . 6
2.2.4 Treatments . . . 7
2.2.5 Utils . . . 7
2.3 Layouts . . . 7

3 Description détaillée des algorithmes implémentés **9**
3.1 Explication du fonctionnement . . . 9
3.2 Algorithmes de couleurs . . . 9
3.2.1 Niveaux de gris . . . 9
3.2.2 Sépia . . . 10
3.2.3 Teinte choisi . . . 10
3.2.4 Garder une couleur . . . 10
3.3 Algorithmes augmentation/diminution/égalisation . . . 11
3.3.1 Exposition . . . 11
3.3.2 Contraste . . . 11
3.3.3 Égalisation d'histogramme . . . 11
3.4 Algorithmes de convolution . . . 12
3.4.1 Sobel . . . 12
3.4.2 Laplacien . . . 12
3.4.3 Flou moyenneur . . . 12
3.4.4 Flou gaussien . . . 13
3.5 Algorithmes divers . . . 13
3.5.1 Filtre médian . . . 13
3.5.2 Effet crayon . . . 13
3.5.3 Effet cartoon . . . 13

4 Tests de performances **14**

5 Problèmes rencontrés et solutions trouvées **15**
5.1 Application de l'effet sur une zone dessinée . . . 15
5.1.1 Comportement souhaité . . . 15
5.1.2 Résultat obtenu . . . 15
5.1.3 Amélioration possible . . . 15
5.2 Zoom et Scroll . . . 16
5.2.1 Problème rencontré . . . 16
5.2.2 Solution trouvée . . . 17

5.3 Compatibilité des téléphones 17

Chapitre 1

Implémentations réalisées

1.1 1ère release

Lors du lancement de l'application, nous avons la possibilité de choisir si nous voulons utiliser une photographie que l'on prend avec notre appareil photo mais aussi de pouvoir prendre une photographie qui se situe dans notre galerie.

Une fois l'image choisie, vous accédez à la page de traitement qui vous permet d'ajouter plusieurs filtres sur votre image (menu du bas). Nous retrouverons comme filtre ceux pour augmenter/diminuer la luminosité, l'égalisation d'histogramme ainsi que les filtrages de couleur et les convolutions. Enfin au dessus de notre image, nous avons un menu qui permet soit de réinitialiser l'image mais aussi de pouvoir quitter ce menu pour revenir au choix des photographie et surtout la possibilité d'enregistrer l'image dans notre galerie.

Malheureusement lors de ce premier rendu, nous n'avons pas pu réaliser toutes les fonctionnalités demandées. En effet, il nous manquait le zoom avec le déplacement ainsi que le réglage du contraste (augmentation/diminution). Du coup ces fonctionnalités ont été mises en priorité pour la préparation de l'application pour la 2eme release.

1.2 2eme release

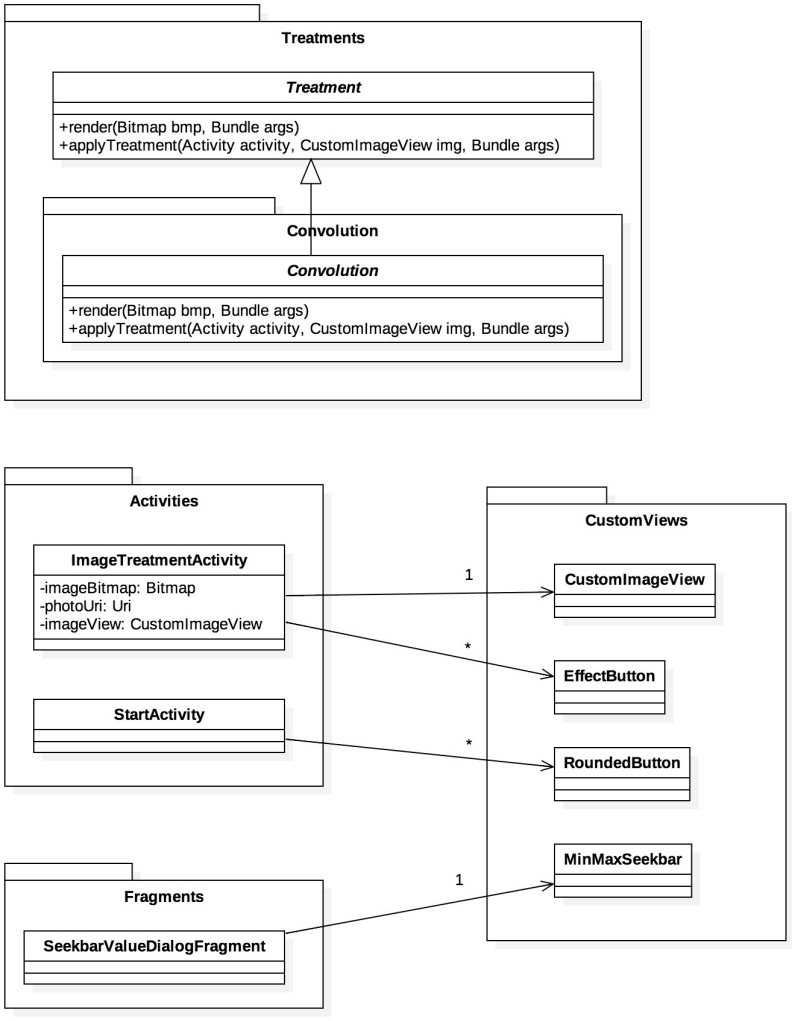
Pour cette release, nous avons d'abord réalisé les fonctionnalités manquantes. Nous avons trouvé pour le contraste un formule fonctionnelle (vous la trouverez ici : http ://www.dfstudios.co.uk/articles/programming/image-programming-algorithms/image-processing-algorithms-part-5-contrast-adjustment/). Nous avons résolu notre problème avec notre zoom et le déplacement, ce qui pose problème c'est lorsque l'on dézoom, notre image peut se trouver sur le côté et impossible de la déplacer une fois totalement dézoomé. Pour résoudre le problème nous avons créé un bouton qui permet de recentrer l'image et ainsi pouvoir continuer d'utiliser l'application correctement.

Pour les fonctionnalités demandées sur le deuxième rendu, nous avons pu faire l'effet crayon ainsi que l'effet cartoon. Pour restreindre la zone d'application d'un filtre avec le doigt, nous avons eu quelques problèmes de performances.

Chapitre 2

Architecture du code

2.1 Diagramme UML



2.2 Fichiers Java

Pour l'organisation de notre code, nous avons séparé nos fichiers dans 4 packages ayant chacun une particularité :

— **Activities** : ce package contient les 2 activités que nous utilisons dans notre application.

— **CustomView** : ce package contient les composants graphiques que nous avons créés pour l'application : une classe héritée d'ImageView qui permet de gérer le scroll et le zoom, les boutons de la première activité pour importer une image, les boutons de la seconde activité pour appliquer un effet et une classe héritée de SeekBar permettant de spécifier une valeur minimum et une valeur maximum.

— **Enums** : C'est ici, que l'on peut créer nos classes d'énumération.

— **Fragments** : lorsque l'on doit faire apparaître un dialog dans notre application c'est ici que nous mettons nos classes java.

— **Runnables** : c'est pour lancer les traitements sur un autre thread. Comme ça le thread principal (l'interface utilisateur) n'est pas bloqué et on peut afficher la fenêtre de chargement d'effet.

— **Treatment** : c'est ici que tous nos algorithmes des filtres se font. Ce package contient tous nos traitements.

— **Util** : il s'agit de nos utilitaires comme par exemple la modification de l'image ou encore l'enregistrement d'une image dans un fichier.

2.2.1 Activities

— **StartActivity** : Cette activité est affichée au lancement de l'application. C'est ici que l'on choisit la source de l'image à traiter.

— **ImageTreatmentActivity** : Cette activité quand à elle correspond au référencement de tous les traitements que l'on peut faire sur notre image. C'est aussi ou l'utilisateur passe le plus de temps vu que c'est l'activité des traitement et autres options (réinitialisation de l'image de base, sauvegarde).

2.2.2 CustomViews

— **CustomImageView** : Classe permet de gérer le scroll sur une image et le pinch to zoom.

— **EffectButton** : Cette classe permet d'initialiser un bouton d'application d'effet sur une image. On lui définit des attributs "effect_button_icon" pour l'image de l'effet et "effect_button_text" pour le texte affiché en bas de l'image du bouton.

— **RoundedButton** : Cette classe permet d'initialiser un bouton de l'activité de départ. On lui définit des attributs "rounded_button_icon" pour l'image de l'effet et "rounded_button_text" pour le texte affiché en bas de l'image du bouton.

— **MinMaxSeekBar** : Classe héritant de SeekBar permettant de spécifier une valeur minimum et une valeur maximum. Il ne faut pas confondre les méthodes **getProgress()** et **getValue()** qui ont deux significations différentes :
  — **getProgress()** permet de renvoyer la valeur de la Seekbar comprise entre 0 et (max - min) / step
  — **getValue()** permet de renvoyer la valeur relative de la SeekBar comprise entre min et max. Nous utiliserons cette méthode afin de déterminer la valeur courante de notre classe MinMaxSeekBar.

2.2.3 Fragments

— **FileErrorDialogFragment** : dialog lorsque l'on a une erreur de lecture.

— **MainActivityBackButtonDialogFragment** : dialog pour savoir si l'utilisateur souhaite enregistrer ou non votre image lorsqu'il quitte la partie traitement de l'application.

— **SeekbarHueColorDialogFragment** : dialog qui demande la couleur de la teinte que l'utilisateur souhaite. Il est utilisé pour appliquer une teinte à l'image ou pour le filtrage de couleur.

— **SeekbarValueDialogFragment** : dialog qui demande la valeur que l'on veut pour l'exposition de notre image.

— **SizeMaskDialogFragment** : dialog que l'on utilise pour la création d'un filtre de convolution. Il permet de choisir la taille de ce filtre.

2.2.4 Treatments

C'est dans ce package que nous retrouvons tous nos traitements. Pour une meilleur lisibilité du code, nous avons décidé de faire une classe par traitement. Toutes nos classe héritent d'une classe abstraite **Treatment** qui possède une méthode du nom de "applyTreatment". Nos classes de traitement sont donc les classes filles de cette classe. La classe **Treatment** permet de définir le fonctionnement identique à toutes les classes qui implémentent, c'est à dire notifier la classe **CustomImageView** lorsque l'image a été modifiée.
Nous retrouvons aussi un package Convolution dans lequel nous trouvons tous nos traitements avec les filtres. Ces traitements sont des classes qui hérite de la classe abstraite "convolution" qui permet l'application du filtre sur notre bitmap. La classe abstraite **Convolution** contient la logique de la convolution, c'est à dire le code qui permet d'appliquer un filtre de convolution à une image. Les classes qui implémentent la classe **Convolution** contiendront uniquement la logique de création d'un filtre de convolution.

2.2.5 Utils

— **ImageFile** : enregistrement de l'image sur la mémoire de stockage de l'appareil.

— **ColorUtil** : classe permettant de mettre les pixels en niveaux de gris. Nous l'utilisons quand des filtres qui ne fonctionnent sur des images non colorées ou lorsque l'on doit utiliser un contraste sur notre image (produire un contraste sur tous les canaux).

— **PermissionUtil** : permet la gestion des permissions. Elle est utilisée pour la sauvegarde (Depuis andoid 6.0 il faut vérifier si l'utilisateur a bien donné les permissions, même si celles-ci sont accordées lors de l'installation de l'application).

2.3 Layouts

C'est dans ce package que nous retrouvons tous nos mises en page et aspect graphiques de notre application, ils sont tous en xml :

— **activity_image_treatment** : c'est le layout de notre activité dans laquelle nous faisons tous nos traitements.

— **activity_start** : activité démarrée au lancement de l'application.

— **effect_button_layout** : il définit le layout des boutons pour appliquer un effet dans l'imageTreatmentActivity.

— **mask_value** : lors de l'utilisation de la convolution, vous avez un dialog avec des boutons vous demandant la taille du filtre que vous voulez appliquer.

— **rounded_button_layout** : ce layout permet d'avoir des boutons arrondis. Cela peut se voir au lancement de l'application sur les boutons d'accueil.

— **seekbar_hue** : layout personnalisé lorsque que l'on veut choisir la teinte sur une image. Sur l'application, ce layout sera affiché sous la forme d'un popup.

— **seekbar_value** : layout personnalisé lorsque que l'on veut choisir l'exposition sur une image. Sur l'application, ce layout sera affiché sous la forme d'un popup.

Chapitre 3

Description détaillée des algorithmes implémentés

3.1 Explication du fonctionnement

Pour l'application des filtres, nous utilisons toujours la même méthode. Tout d'abord chaque filtre correspond à une classe, elles héritent tous d'une classe abstraite **Treatment**. Cette classe abstraite possède deux fonctions, la première,**render**, qui prend en paramètre un Bitmap ainsi qu'un Bundle d'arguments qui nous permet d'avoir plusieurs types d'arguments sans spécifié pour l'instant ce que l'on attend (parfois nous avons besoin que d'un imageView mais il se peut que l'on doit récupérer la valeur d'une seekbar par exemple). La seconde, **applyTreatment** qui prend en paramètre une activité, un Customview et un Bundle d'arguments. Elle nous permet de lancer notre chargement et d'appliquer le traitement en appelant **render**

Une fois nos classes créées, nous utilisons la méthode @override sur notre fonction abstraite afin de pouvoir modifier l'image comme nous avons envie selon l'effet désiré. Ainsi lors de son utilisation dans notre activité, il suffit ainsi de créer un objet de la classe (effet désiré) et d'appliquer la fonction applyTreatment avec les arguments voulus.

Pour les convolutions, c'est un peu différent. Nous avons une classe **Convolution** qui hérite de **Treatment**, mais c'est aussi une classe abstraite. Ce qui fait que les autres classes créées pour faire des convolutions héritent de cette classe **Convolution**. C'est le cas pour Laplacien, Sobel, GaussianBlur et AverageBlur.

3.2 Algorithmes de couleurs

3.2.1 Niveaux de gris

Description

Pour cet algorithme, nous voulons faire en sorte que toutes nos couleurs se retrouvent dans un gris. Il fait que l'on obtient une image avec plusieurs nuances de gris

Fonctionnement

Nous créons d'abord un tableau de pixels de la taille de notre bitmap. Puis nous récupérons toutes les valeurs de notre bitmap pour le mettre dans ce tableau de pixels. Nous allons récupérer, pour chaque pixel, la valeur de bleu, rouge et vert (formule du cours) :

- red=(red*3)/10
- bleu=(blue*11)/100
- green=(green*59)/100

Il nous suffit ensuite de récupérer la sommes des trois pour avoir une nouvelle couleur. Nous remplaçons la valeur du pixel avec cette nouvelle valeur sur les 3 canaux. Nous modifions notre bitmap avec ce tableau de pixels modifié et retournons ce nouveau bitmap.

3.2.2 Sépia

Description

Grâce à cet algorithme nous obtenons un effet jauni qui rappelle les photos d'antan.

Fonctionnement

Nous créons d'abord un tableau de pixels de la taille de notre bitmap. Puis nous récupérons toutes les valeurs de notre bitmap pour le mettre dans ce tableau de pixels. Nous allons récupérer, pour chaque pixel, la valeur de bleu,rouge et vert (formule du cours) :

- Nred=(red*393/1000)+(green*769/1000)+(blue*189/1000)
- Nbleu=(red*272/1000)+(green*534/1000)+(blue*131/1000)
- Ngreen=(red*349/1000)+(green*686/1000)+(blue*168/1000)

Nous remplaçons chaque canaux de couleur par sa nouvelle couleur (le pixel red aura donc comme nouvelle couleur Nred). Nous modifions ainsi la couleur de notre pixel. Nous modifions notre bitmap avec ce tableau de pixels modifié et retournons ce nouveau bitmap.

3.2.3 Teinte choisi

Description

Grâce à cet algorithme, l'utilisateur choisit la teinte qu'il veut donner à son image. Lorsqu'il clique sur le bouton, un menu s'affiche pour nous permettre de choisir la teinte

Fonctionnement

Tout d'abord nous avons le menu de sélection qui nous permet de choisir la teinte. Il s'agit d'une seekbar qui varie entre 0 et 255. Une fois sélectionné on va récupérer la valeur de notre seekbar dans notre fonction. Nous allons créer un tableau de pixels de la taille de notre bitmap et un tableau HSV[3] (teinte, saturation, luminosité). Puis nous récupérons toutes les valeurs de notre bitmap pour le mettre dans ce tableau de pixels. Pour chaque pixel, avec la fonction colorToHSV, nous allons modifier la valeur de la teinte hsv[0] avec la valeur que l'utilisateur a choisi avec la seekbar. Puis pour chaque pixel nous remettons la couleur avec la fonction HSVtoColor avec notre tableau hsv.Nous modifions notre bitmap avec ce tableau de pixels modifié et retournons ce nouveau bitmap.

3.2.4 Garder une couleur

Description

Grâce à cet algorithme, l'utilisateur choisit la teinte qui restera son image. Lorsqu'il clique sur le bouton, un menu s'affiche pour nous permettre de choisir la teinte à garder. Tout le reste de l'image sera en nuance de gris.

Fonctionnement

Tout d'abord nous avons le menu de sélection qui nous permet de choisir la teinte. Il s'agit d'une seekbar qui varie entre 0 et 255. Une fois sélectionné on va récupérer la valeur de notre seekbar dans notre fonction. Nous allons créer un tableau de pixels de la taille de notre bitmap et un tableau HSV[3] (teinte,saturation,luminosité). Puis nous récupérons toutes les valeurs de notre bitmap pour le mettre dans ce tableau de pixels. Pour chaque pixels, nous récupérons les 3 valeurs rouge, vert et bleu. Puis nous utilisons la fonction RGBToHSV avec nos 3 couleurs et notre tableau HSV. Nous vérifions que notre teinte hsv[0] se trouve bien entre la (valeur utilisateur-minimum) et (valeur utilisateur+minimum). Le minimum est une valeur que l'on a choisi (50 dans cette exemple). Si notre pixel

ne se trouve pas dans cet intervalle, on utilise notre fonction de nuance de gris. Nous modifions notre bitmap avec ce tableau de pixels modifié et retournons ce nouveau bitmap.

3.3 Algorithmes augmentation/diminution/égalisation

3.3.1 Exposition

Description

Grâce à cet algorithme, l'utilisateur va pouvoir assombrir ou éclaircir l'image. Lorsqu'il clique sur le bouton une barre de progression va s'afficher et il pourra choisir une valeur permettant de diminuer ou augmenter l'exposition.

Fonctionnement

Tout d'abord nous avons le menu de sélection qui nous permet de choisir l'exposition. Il s'agit d'une MinMaxSeekBar qui varie entre -255 et 255 et qui à pour valeur par défaut 0. Une fois que l'algorithme récupère la valeur nous allons ajouter pour chaque pixel au valeur rouge, vert et bleu la valeur de l'exposition que nous avons récupéré précédemment. Ainsi l'image aura un rendu plus ou moins sombre ou clair.

3.3.2 Contraste

Description

Grâce à cet algorithme, l'utilisateur va pouvoir assombrir ou éclaircir le contraste l'image. Lorsqu'il clique sur le bouton une bar de progression va s'afficher et il pourra choisir une valeur permettant de diminuer ou augmenter le contraste.

Fonctionnement

Comme pour la surexposition, on utilise une MinMaxSeekBar qui varie entre -255 et 255 et qui à pour valeur par défaut 0. Une fois que l'algorithme récupère la valeur nous allons créer un facteur qui prendra la valeur :
facteur=(259*(valeur seekbar+255))/(255*(259-valeur seekbar))
Nous allons par la suite ajouter à chaque pixel, pour chaque canaux (formule internet) :

- red=(facteur*(red-128))+128
- bleu=(facteur*(bleu-128))+128
- green=(facteur*(green-128))+128

Nous modifions notre bitmap avec ce tableau de pixels modifié et retournons ce nouveau bitmap.

3.3.3 Égalisation d'histogramme

Description

L'égalisation d'histogramme est une méthode d'ajustement du contraste d'une image qui permet de mieux répartir les intensités des pixels de l'image sur l'ensemble de la plage de valeurs possibles, en "étalant" son histogramme.

Fonctionnement

L'algorithme d'égalisation d'histogramme calcule l'histogramme cumulé de l'image en noir et blanc puis parcourt chaque pixel de l'image en couleur afin de lui appliquer une transformation.

Pour chaque couche RGB, la formule de la transformation appliquée est la suivante :
nouvelleValeur = histogrammeCumule[ancienneValeur] * 255 / tailleImage

3.4 Algorithmes de convolution

Fonctionnement

L'algorithme de convolution permet d'appliquer un ou plusieurs masque de taille diverses à une image. L'algorithme est séparé en deux partie suivant si il y a un ou deux masque à appliquer mais les deux parties fonctionnent de la même manière. Nous parcourons le tableaux de pixel puis nous appliquons le masque sur les pixels qui ne sont pas sur les bords de l'image afin d'éviter les problèmes des pixels voisins qui sortent du tableaux. Ensuite pour chaque pixel nous cherchons ses voisins, le nombre de voisins dépendant de la taille du masque. Le pixel prend alors la somme de ses voisins qui ont étaient multipliés par la valeur du masque.

3.4.1 Sobel

Description

L'effet sobel permet de détecter les contours, l'image devient noire avec les contours blancs.

Fonctionnement

Nous appliquons le filtre Sobel sur des images en nuances de gris, donc avant même de commencer à traiter le filtre sobel nous faisons passer l'image en niveaux de gris. Une fois cela fait il suffit d'appliquer deux masque différents sur le tableaux de pixel. Différents masque de Sobel existent et ils se ressemblent plus ou moins. Nous avons choisi de prendre ceux-ci :

— {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}} ;
— {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}} ;

3.4.2 Laplacien

Description

L'effet Laplacien est lui aussi un filtre qui permet de détecter les contours mais l'image devient cette fois grisée avec les contours en noir.

Fonctionnement

Nous appliquons le filtre Laplacien sur des images en nuances de gris, donc avant même de commencer à traiter le filtre laplacien nous faisons passer l'image en niveaux de gris. Une fois cela fait il suffit d'appliquer un masque sur le tableaux de pixel. Différents masque Laplacien existent et ils se ressemblent plus ou moins. Nous avons choisi de prendre celui-ci :

— {{1, 1, 1}, {1, -8, 1}, {1, 1, 1}} ;

Une fois le masque appliqué sur tout l'image nous faisons une égalisation d'histogramme qui permet de rendre l'image plus nette.

3.4.3 Flou moyenneur

Description

Grâce à cet algorithme, l'utilisateur peut effectuer un flou sur l'image. Cela peut permettre d'homogénéiser l'image au niveau des couleurs ou en supprimant des "impuretés". Lorsque l'utilisateur clique sur le bouton, un menu s'affiche pour nous permettre de choisir la taille du filtre qui sera appliqué.

Fonctionnement

Un menu propose la taille du filtre qui sera appliqué (3*3,5*5,7*7,9*9). En récupérant la taille du filtre le flou moyenneur va définir les pixels voisins à chaque pixel. Une fois qu'on à un tableau de la taille du filtre choisi représentant les voisins du pixel on définit la valeur du pixel comme étant la moyenne de la valeur de tout ses voisins.

3.4.4 Flou gaussien

Description

Le flou gaussien permet de flouter une image de manière plus lisse que le flou moyenneur. En effet, le flou moyenneur utilise une moyenne arithmétique alors que le flou gaussien utilise une moyenne pondérée.

Fonctionnement

Un menu propose la taille du filtre qui sera appliqué (3*3,5*5,7*7,9*9). Une fois la taille choisie, l'algorithme parcourt les pixels et calcule une moyenne pondérée de la valeur des pixels voisins puis applique cette valeur au pixel courant.

3.5 Algorithmes divers

3.5.1 Filtre médian

Description

Le filtre médian permet d'appliquer un léger flou à l'image en enlevant des détails de couleur. Lorsque les couleurs sont proches, ces dernières sont fusionnées.

Fonctionnement

L'algorithme parcoure les pixels de l'image et sauvegarde les composants (tente, valeur, saturation) des 8 voisins du pixel courant ainsi que les composants du pixel courant sur les 3 canaux de couleurs. Après cela, il ordonne les composantes et prend la valeur médiane de chacune d'entre-elles. Ces valeurs médianes sont appliquées au pixel courant.

3.5.2 Effet crayon

Description

Cet algorithme permet de donner à l'image l'impression qu'elle a était dessiné au crayon à papier.

Fonctionnement

Tout d'abord pour effectuer ce traitement on applique un filtre Sobel qui va nous permettre de récupérer les contours de l'image. On obtient donc une image noir avec les contours en blanc. Ensuite l'algorithme utilise un autre filtre qui permet d'inverser les couleurs ce qui va nous permettre d'avoir désormais les contours de couleur noir. Enfin l'algorithme va effectuer un léger flou gaussien de taille 3*3 ce qui donnera un meilleur effet à notre image et obtiendra ainsi l'effet voulu.

3.5.3 Effet cartoon

Description

Cet algorithme permet de donner à l'image un effet dessin animé cartoon.

Fonctionnement

Le principe de l'algorithme est dans un premier temps de récupérer les contours de l'image puis de fusionner une image contenant seulement les contours avec la même image à laquelle on a appliqué un filtre. Afin d'obtenir les contours de l'image nous appliquons le filtre Sobel. Sur une copie de l'image nous appliquons un filtre median qui réduit les détailles de celle-ci. Pour fusionner les deux images nous allons prendre les pixels de l'image Sobel lorsque le pixel en question représente un contour sinon on récupère le pixel de l'image avec le filtre médian.

Chapitre 4

Tests de performances

Nous avons réalisé un test de performances avec différents appareils : un émulateur de **Nexus 5X** sous Android 7.1.1 et un téléphone **LG G3** sous Android 6.0.

Le protocole était le même sur chaque périphérique :

1. Chargement d'une image de **222ko**
2. Application du filtre sépia
3. Scroll de la **BottomBar**
4. Clic sur le bouton "Flou gaussien" et affichage du **DialogFragment** permettant de choisir la taille du filtre
5. Application d'un flou gaussien de taille 7*7

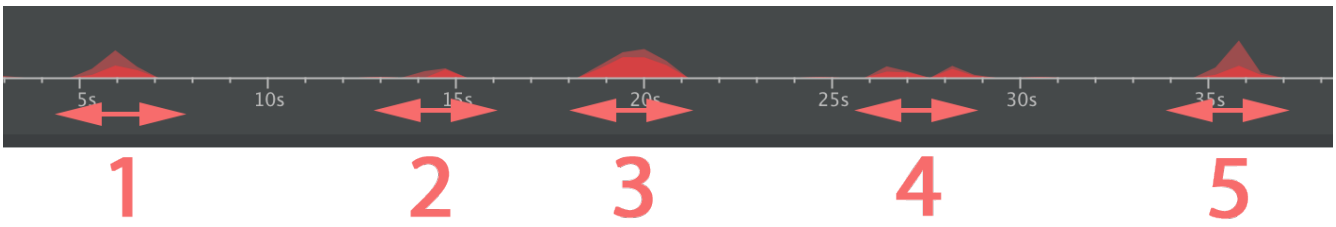


FIGURE 4.1 – Test de performances CPU sur un émulateur de Nexus 5X

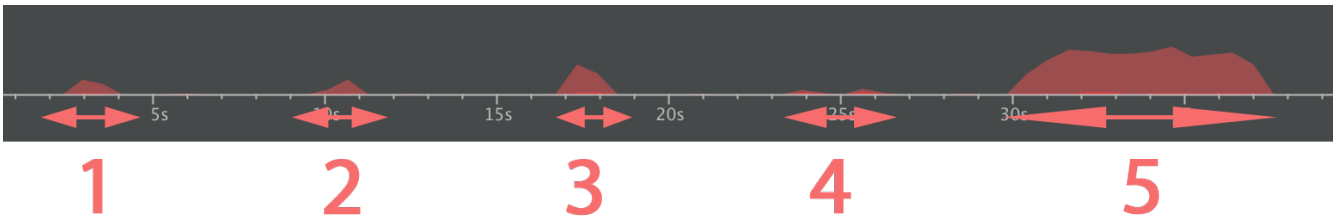


FIGURE 4.2 – Test de performances CPU sur un LG G3 physique

Les figures 4.1 et 4.2 représentent les performances de l'application selon l'appareil utilisé.
Ainsi, nous remarquons que les actions comme le chargement d'une image, l'application du filtre sépia, le scroll de la **BottomBar** et l'affichage du **DialogFragment** nécessitent un temps similaire sur les deux périphériques utilisés. Cependant, la différence de temps d'application du filtre gaussien s'explique par le fait que le Nexus 5X dispose d'un processeur Octa-core de fréquence 1,8 Ghz alors que le LG G3 dispose d'un processeur Quad-core de fréquence 2,5 Ghz. La lenteur de traitement du LG G3 peut aussi s'expliquer par le fait que des applications en tâche de fond le ralentissent alors que le Nexus 5X est un émulateur sur lequel aucune application ne tourne en tâche de fond.

Chapitre 5

Problèmes rencontrés et solutions trouvées

5.1 Application de l'effet sur une zone dessinée

5.1.1 Comportement souhaité

Nous avons voulu implémenter deux modes de fonctionnement à notre classe **CustomImageView** :

— Par défaut, l'effet choisi s'applique sur toute l'image

— Sinon, l'utilisateur dessine sur l'image la où il vaut que l'effet s'applique puis il sélectionne un effet pour que ce dernier s'applique

Pour ce faire, nous avions ajouté à la classe **CustomImageView** une variable de classe **mode** de type **CustomImageViewModeEnum** et pouvant prendre les valeurs MODE_ALL et MODE_SELECTION.
Le changement de mode de la classe CustomImageView se faisait par le biais de la toolbar de l'activité principale.

Voir figure 5.1

5.1.2 Résultat obtenu

Le résultat obtenu ne correspondait pas à nos attentes ; en effet, nous avons rencontré des latences lorsque nous touchions l'ImageView. Nous pensons que ces latences sont dues au fait que l'ajout de pixels rouges dans l'ImageView afin d'indiquer les zones sélectionnées par l'utilisateur n'était pas instantané.
En fonction de la vitesse de défilement du doigt sur l'écran, les points rouges sont plus ou moins espacés.

5.1.3 Amélioration possible

Si nous avions eu plus de temps, nous aurions testé d'effectuer l'ajout de pixels rouges dans l'image à l'aide de **RenderScript**. Nous pensons que cette technique aurait été moins coûteuse en temps et aurait permis d'atténuer l'effet de latence que l'on peut ressentir.

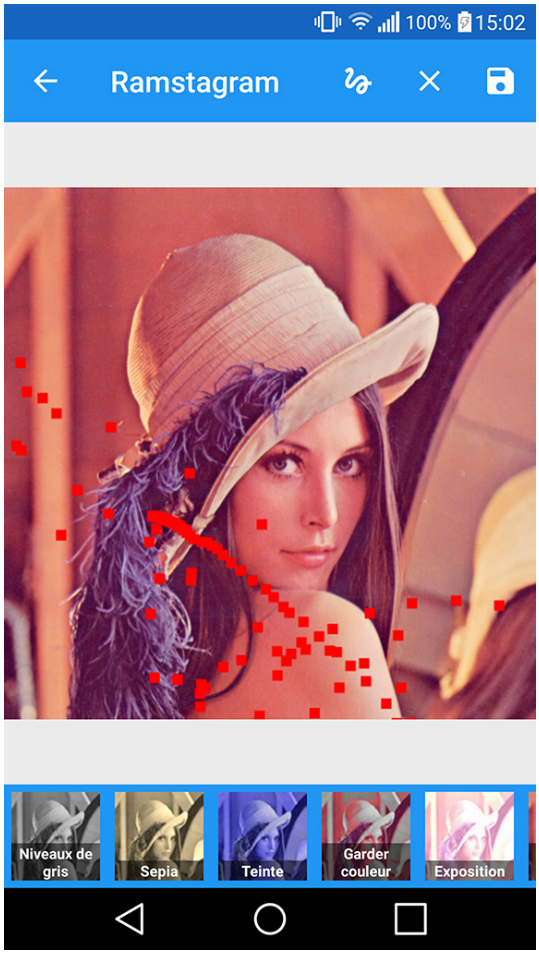


FIGURE 5.1 – Application d'un effet sur une zone sélectionnée par l'utilisateur

5.2 Zoom et Scroll

5.2.1 Problème rencontré

Nous avons remarqué un comportent anormal de notre classe **CustomImageView** qui survient dans un cas particulier : après avoir zoomé sur l'image et décalé cette dernière sur un côté, il faut dézoomer sur l'image. Cela va "bloquer" l'image sur un côté de l'écran et il sera impossible de la déplacer.

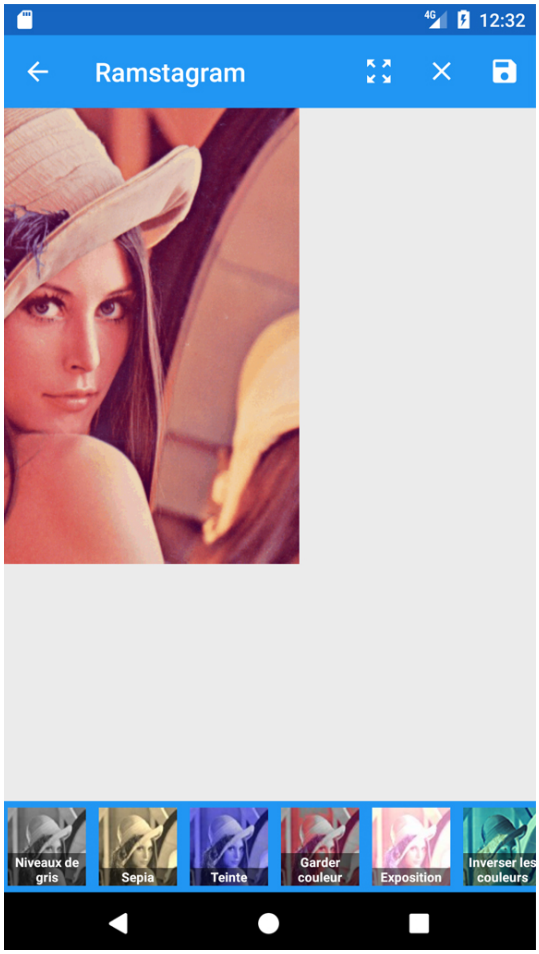


FIGURE 5.2 – Image "bloquée" sur un côté

5.2.2 Solution trouvée

Au début nous pensions qu'enlever les blocages haut, bas, droite et gauche du scroll étaient une bonne idée mais nous nous sommes rendus compte du fait que l'image pouvait être envoyée en dehors de l'écran et cela n'aurait pas été compréhensible par l'utilisateur.
La solution que nous avons décidé de retenir consiste en l'ajout d'un bouton dans la toolbar permettant de remettre à zéro le zoom et le scroll de l'image.

5.3 Compatibilité des téléphones

Selon les téléphones utilisés, nous avons rencontré des problèmes différents :

1. Avec le téléphone Orange Roya, l'application ne permet pas de capturer une photo
2. Avec le téléphone LG G3 et certains émulateurs, les photos qui sont prises dans un mode de vue (paysage, portrait) sont retournées lorsqu'elles sont chargées par l'application
3. Avec tous les appareils, si on charge une image trop lourde, le traitement ne pourra pas s'effectuer car il n'y aura pas assez d'espace en mémoire vive