Université de Bordeaux

Département Informatique

Année 2018-2019

Groupe 2

Licence 3

Rapport de projet

Traitement d'image Android

Auteurs :
Rémi Lasvenes
Tom Leyrissoux
Kévin Mutima
Florian Simba

Table des matières

1 Introduction

Le but de ce projet est de développer une application mobile de traitement d'images sous **Android** dans le cadre de l'**UE Projet technologique** en L3 Informatique. Ce projet doit respecter le cahier des charges fourni par les enseignants et donc implémenter au moins les fonctionnalités obligatoires définies dans ce dernier. Ce rapport servira à détailler l'**architecture globale** de l'application, décrire les tests qui ont été effectués et préciser les **fonctionnalités supplémentaires** à implémenter pour la V2.

2 Architecture et implémentation

L'architecture de l'application est décomposé en **3 parties** ;

2.1 Écran d'accueil

La première partie étant essentiellement une page d'accueil qui ne reste affiché que très peu de temps, et à pour but de mettre en avant le logo, ainsi que l'université de Bordeaux.

2.2 Écran du choix de l'image

On est redirigé automatiquement vers cet écran après quelques secondes seulement, c'est à partir de là que l'on va pouvoir choisir si on préfère prendre une image depuis la galerie, ou bien depuis la caméra du téléphone ; si on choisi la galerie, cela ouvre directement la galerie par défaut du téléphone, et si on choisi la caméra, cela lance directement l'application photo/camera par défaut du téléphone. On a mis un bouton switch qui permet à l'utilisateur d'utiliser soit les effets impelementés renderscript ou java

2.3 Écran d'édition

C'est la partie principale de l'application, c'est depuis cet écran que l'on va afficher l'image précédemment choisie, et qu'on va pouvoir accéder à toutes les fonctionnalitées ; zoomer, scroller, appliquer un filtre, convertir une image en niveau de gris, etc... On a choisi une interface qui permet de laisser le plus de place possible à l'affichage de l'image afin d'optimiser l'expérience de l'utilisateur. On a mis une zone déroulante horizontalement en bas de l'écran contenant des boutons avec la prévisualisation de l'effet sur l'image. En haut de la zone réservée pour l'image y sont cinq boutons respectivement :

- Back : pour revenir au fragment précèdent donc l'écran du choix de l'image
- Flèche undo : pour enlever le dernier effet applique à l'image si l'effet existe.
- Boutons réinitialisation
- Flèche redo : pour réappliquer l'effet précédemment enlever

— Save : Pour sauvegarder l'image

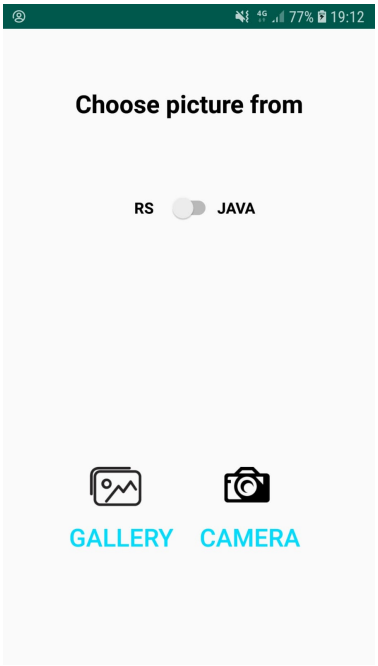

FIGURE 1 – Écran de sélection de l'image

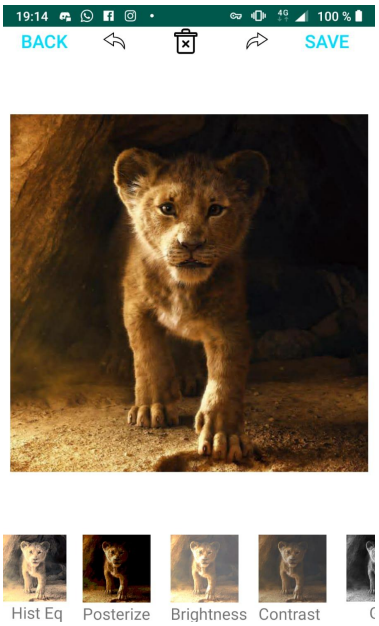

FIGURE 2 – Écran d'édition de l'image

2.4 Implémentation

Concernant l'implémentation, nous avions initialement choisis de mettre une activité par vue/écran, mais cela était trop volumineux, nous avons donc modifié notre code pour remplacer ces activités par des fragments. La grande majorité des fonctions (contraste, luminosité, égalisation d'histogramme, etc...) correspondent à une classe (la classe `Contrast`, `Brightness`, `HistogramEq`, etc...) afin de factoriser un maximum de code et d'y apporter plus de lisibilité.

Concernant l'appel aux fonctions, cela ce fait dans le fragment `WorkFragment`, pour certains effets, cela ce fait lorsque l'utilisateur va lacher son doigts de la seek-bar et va ainsi appeler la méthode en question avec les bons paramètres.

Pour ce qui est de la convolution, nous avons choisis d'utiliser les `AsynTask` afin de ne pas bloquer le `thread` principale de l'application avec les calculs, cela permet également de voir l'avancement de l'algorithme au fur et à mesure du temps ; une barre de progression indique à l'utilisateur l'avancée du traitement, ainsi qu'une fenêtre l'informant qu'un traitement est en cours d'éxécution.

3 Fonctionnalités pour la V1

3.1 Charger une image

Pour charger une image depuis la galerie ou depuis la caméra nous utilisons le fragment désigné par la classe `ChoosePictureFragment` et l'activité `fragment_choose_picture`.

Dans les deux cas si la taille de l'image est superieur à un Mégapixel, elle sera automatiquement redimensionner pour obtenir une taille en dessous de un Mégapixel. Ainsi, on aura un meilleur temps d'éxecution pour les traitement complexes. Mais l'image ne sera pas perdue on s'en servira lors de la sauvegarde finale, comme ça la qualité de l'image ne sera pas désaltérer.

Pour ce, on redimensionne l'image en divisant sa hauteur et sa largeur par le logarithme en base deux du nombre de Mégapixels. Comme ça on est certain de conserver le ratio de l'image.

3.1.1 depuis la galerie

La galerie est "mise sur écoute". Puis, son activité est démarrée lors d'un clic sur le bouton galerie. L'image est finalement passée sous forme d'URI au fragment `WorkFragment`.

3.1.2 depuis la caméra

La caméra est elle aussi "mise sur écoute". Il faut cependant obtenir la permission de l'utilisateur pour avoir accès à cette dernière. Un fichier image (.jpg) est créé, il contiendra la nouvelle image, puis son activité est démarrée lors du clic sur le bouton caméra. L'image est également passée sous forme d'URI au fragment `WorkFragment`.

3.2 Afficher une image

L'URI de l'image obtenu du fragment précédent permet dans `WorkFragment` de créer le Bitmap sur lequel on va réaliser des calculs : "outBmp". Il sera une copie du Bitmap original `inBmp`, donné par l'URI, qui permettra de réinitialiser l'image. L'image sera afficher grâce à une imageView associée à `outBmp` : `img_view`.

3.3 Zoomer

Pour ce qui est du zoom, cela se fait de manière analogue au zoom sur l'image ; en utilisant deux doigts, et en les rapprochant ou en les éloignant. On peut aussi zoomer une image en double cliquant sur l'image, le zoomer encore plus en répétant le même geste et la troisième fois l'image se réinitialise. La limite sur le zoom est la zone réservée pour l'image.

3.4 Scroller

Le déplacement sur une image zoomé (scroll) va de paire avec le zoom, cela se fait avec un seul doigts, en le déplacant sur l'écran, et la limite et la taille de l'image.

3.5 Appliquer des filtres

On peut à présent empiler des effets grâce à une pile sur laquelle on empile des objets de type `Effect`.
En effet, chaque effet aura sa propre classe qui va étendre la classe abstraite `Effect` et qui à son tour contient les méthodes abstraites :

— apply : qui servira à appliquer les effets implémentés en RenderScript.
— applyJava : qui servira à appliquer les effets implémentés en Java.

Elle contient aussi quelques fonctions qui seront utiles aux classes filles tel que : CheckInRange, toString, getGrey, etc.

Ainsi, grâce à cette pile on pourra revenir annuler un effet sans devoir ré-initialiser toute l'image par exemple ou bien ré-appliquer un effet annuler. Pour cela, deux piles sont nécessaire ; l'une contiendra tous les effets appliquer sur le bitmap et l'autre contiendra les effets qu'on va dépiler de la première pile.

Du coup, lorsque on fait un `Undo`, on dépile la première pile et on va empiler cet effet dans la seconde pour la récupérer lorsque l'utilisateur décide de faire un `Redo` par exemple.

Il est prévu que la seconde pile se vide automatiquement si l'on fait un ou plusieurs `Undo` est puis par la suite on applique un autre effet.

Comme on travaille avec une image redimensionnée, on va également utiliser `EffectStack` lors de la sauvegarde de l'image, pour appliquer tous les effets appliqués par l'utilisateurs sur la Bitmap de taille originale cette fois-ci.

Bien évidemment, un `AsyncTask` a était mis en place lors des appels de ces méthodes.

3.5.1 Griser l'image

Pour griser l'image, il suffit d'appliquer l'effet en un seul clic. C'est la classe `toGrey` qui sert à griser l'image grâce à un appel à la fonction renderscript de `greyadv.rs`.

3.5.2 Régler la luminosité

Pour régler la luminosité, nous utilisons une barre de progression allant de -100 à 100. C'est la classe `Brightness` qui permet d'appliquer l'effet grâce à un

appel à la fonction **renderscript** de `brightness.rs`. Concernant l'implémentation en **renderscript**, il s'agit là d'une des rares fonctions qui s'écrit plus facilement en **renderscript** qu'en **JAVA**.

3.5.3 Régler le contraste

Pour régler le contraste, nous utilisons également une barre de progression allant de -100 à 100. C'est la classe `Contrast` qui permet d'appliquer une **extension/réduction dynamique/linéaire** grâce à un appel à la fonction renderscript de `contrast_lde.rs`. Notre raisonnement sur cet algorithme était d'augmenter ou de réduire.

3.5.4 Égalisation d'histogramme

Pour l'égalisation d'histogramme, nous appliquons un seul effet permettant d'augmenter grandement le contraste. Ici, c'est la classe `HistogrameEq` qui sert à effectuer l'égalisation d'histogramme grâce à un appel aux fonctions renderscript de `hist_eq.rs` qui appliquent cet effet dans le domaine **YUV**.

3.5.5 Filtrage couleur

Pour coloriser l'image, nous utilisons une barre de progression (type "arc-en-ciel" pour sa practicité) allant de 0 à 359. Nous utilisons exactement le même type de barre allant de 0 à 359 pour conserver une seule gamme de couleur dans l'image. La classe `Colorize` sert à coloriser l'image grâce à un appel à la fonction renderscript de `color.rs` tandis que la classe `KeepOneColor` conserve une gamme grâce à un appel au renderscript de `keep_color.rs`.

3.5.6 Convolution

Dans la convolution, on a amélioré la version Java mais aussi réussi à implémenter une version RenderScript fonctionnelle même si cela n'a pas était chose facile.

Premièrement, pour améliorer notre version Java, on devait trouver une manière d'utiliser un getpixels. Mais le fait de récupérer et stocker les pixels dans un même tableau, faisait que lorsque on applique des masques de détection de contours on avait des lignes bizarres dans l'image.
En plus, comme le problème n'apparaissait qu'avec la détection de contour, c'était difficile de se douter de la source de l'erreur.

Ensuite, en face de ce bug, on avait également un autre bug dans la version de renderscript cette fois-ci. Là encore tout était bon sauf la manière d'accéder au pixels qui était bon.
En effet, on essayait d'accéder aux voisins d'un pixel en se servant du pointeur du pixel qui est passé en paramètre à la fonction. Mais, tout c'est bien passé

lorsque on s'est servit de l'allocation comme tableau de pixels.

La seule chose qu'on a gardé de l'ancienne version c'est l'AsyncTask qui reste très utile même si nos convolution sont plus performantes.

- Filtre moyenneur
  On a la possibilité de définir la taille du masque, qui va être générer par les fonctions de la class `KernelMaker`.
- Gaussien
  Comme pour le filtre moyenneur, il est également possible de choisir la taille du masque mais pas la valeur du sigma qui est fixée à 1.3.
- Gaussien intrinsic
  On a gardé le script intrinsic du flou gaussien avec un radian de taille 3, pour le comparer avec notre version renderscript.
  L'intrinsic reste rapide avec une différence de temps d'exécution d'une dizaine de secondes.
- Sobel, Prewitt
  Comme ces deux effets utilisent deux masques, on a crée leur propre algorithme de convolution qui est découper en plusieurs étapes dont les plus importants sont :
  - On calcule la somme des valeurs absolues des élements de l'un des masques ensuite on divise par deux. On utilise la valeur absolue pour éviter d'avoir une somme nulle. La valeur ainsi obtenue nous servira à la normalisation plus loin
  - On recupère les sommes des valeurs de tous les canaux des deux blocs de neuf pixels
  - On normalise chaque canal pour la faire revenir entre 0 et 255 en se servant de la valeur obtenue plus haut
  - On calcule le module et teste s'il n'y a pas de dépassement de valeur pour chaque canal.

  Les bords de l'image quant à eux seront laisser intactent comme pour la convolution classique.
- Emboss
  Cette effet est resté inchanger et fait juste ressortir nettement mieux les contours, même pour les images en couleur.

3.6 Réinitialiser

L'image est aisément réinitialisée grâce au Bitmap "inBmp". Ce Bitmap possède une copie "outBmp" sur lequel sont effectués tous les calculs, il n'est donc pas modifié. Pour obtenir l'image d'origine il suffit donc de remplacer "outBmp" par "inBmp" dans l'imageView qui sert à afficher l'image lors du clic sur le bouton réinitialiser.// En plus de cela, on vide toutes les piles de l'`EffectStack`.

3.7 Sauvegarder une image

Pour sauvegarder une image, il existe la fonction "savePic" dans WorkFragment. Elle demande la permission pour créer un fichier image dans le téléphone de l'utilisateur, pour ensuite réappliquer tous les effets choisits par l'utilisateur mais cette fois-ci sur l'image non redimensionner pour conserver la résolution d'origine. Elle est appliquée lors d'un simple clic sur le bouton de sauvegarde.

4 Fonctionnalités additionnelles pour la V2

Pour cette V2, nous avons surtout pris le choix d'ajouter le maximum de fonctions implementable en Renderscript comme :

4.1 Effet de dessin au crayon

L'effet de dessin au crayon s'effectue en deux étapes :

— d'abord, la fonction apply de PencilDraw exécute le script de l'algorithme pencil.rs : pour chaque noyau, on va griser le pixel, et pour accentuer la différence entre les zones claires et les zones sombres de l'image, on réparti la luminosité en quatre, comme pour avoir quatre différents niveaux de pression par un crayon sur l'image.

— pour finir, la fonction apply réalise une convolution : un renforcement de contour grâce au filtre Emboss, ce qui va faire ressortir cet effet des quatre niveaux de pression en renforçant les contours sombres de l'image.

4.2 Effet cartoon

L'effet cartoon s'effectue également en deux étapes :

— d'abord, la fonction apply de Cartoon exécute le script de l'algorithme cartoon.rs : pour chaque noyau, le pixel va être converti en hsv. Ensuite, la saturation va être augmentée si possible (de 10 %) pour renforcer la température comme dans une bande dessinée. De la même manière que pour l'effet crayon, la luminosité va être réparti en quatre, comme pour avoir quatre feutres de clartés différentes pour une même couleur. Puis, on va convertir le pixel en rgb.

— pour finir, la fonction apply réalise une convolution : un filtre moyenneur donc pour donner un flou à l'image, cela va rendre les détails de l'image et des silhouettes moins visibles pour passer à une image moins réaliste et plus cartoon.

4.3 Saturation

Pour la saturation on a deux choix, soit passer par l'espace couleur HSV et changer la saturation, soit en calculant la chromatique à partir de la luminance et la saturation (source :Wikipedia).
La deuxième méthode est la plus éfficace est c'est elle qu'on reimplémenter en

renderscript car le fait de changer de saturation en HSL affecte également la luminosité perçue d'une couleur.

4.4 Posterization

Pour cette effet, on va redéfinir(limiter) le nombre de valeurs possibles de chaque composante du pixel. Cela donne lieu à des coupures abruptes de couleurs, comme sur un vieux poster.

4.5 Incrustation

En ce qui concerne l'incrustation d'image, un seul type d'incrustation a été ajouté ; il s'agit de l'incrustation d'**yeux vert**. Pour procéder ainsi, on a utilisé une bibliothèque externe de *Google* qui nous permet de détecter des visages (si il y en a), ainsi, on récupère la largeur du rectangle encadrant le visage, ce qui nous permet de redimensionner l'image à incruster afin d'obtenir un résultat plus réaliste (le coéfficient de redimenssionnement a été obtenu manuellement sur une image, en testant plusieurs valeurs).

5 Difficultés

5.1 AsyncTask

Il est vrai que l'`AsyncTask` est bien pratique dans certain cas, mais dans bien d'autre il pose problème. Des fois il faut certaines sections de codes dans le `onPostExecute`, comme les `Try|Catch` et bien d'autres pour que ça soit executé proprement sinon il y aura des sortes de décalages.

5.2 Renderscript

Efficace mais difficile à débugger surtout s'il faut faire beaucoup d'essaies, ce qui a était notre cas. En effet, des fois quand on récompilais il fallait désinstaller complètement l'application pour que les changements éffectués dans le code soit bien pris en compte.

6 Améliorations

— On aurait pu éviter de faire deux fonctions de convolutions.
— Afficher un histogramme
— Ajout d'effets plus simples mais utiles comme la rotation ou l'effet miroir,
— Page d'aide.

7 Tests

7.1 Tests en temps (millisecondes)

Ces tests ont été effectués avec l'émulateur Galaxy Nexus (API 28), sur deux images : la première en couleur, de taille 949x650, la seconde en noir et blanc, de taille 512x512.

Effet \ Méthode	RS Couleur	RS N&B	JAVA Couleur	JAVA N&B
Égalisation histogramme	84	45	39	22
Posteriser (à 100%)	158	20	72	50
Luminosité (à 100%)	122	10	22	9
Contraste (à 100%)	27	32	199	200
Griser	140	13	26	7
Négatif	124	4	13	8
Saturation (à 180)	23	29	0.	0.
Coloriser (à 100)	36	19	838	364
Garder couleur (à 50)	249	226	11456	4699
Filtre moyenneur (25)	504	222	131090	53499
Filtre Gaussien (25)	288	127	130509	52304
Sobel	70	32	2361	1058
Prewitt	55	14	2236	973
Laplace (cx8)	42	33	2185	898
Emboss	39	7	2102	968
Effet cartoon	91	59	X	X
Effet dessin au crayon	76	53	X	X

On observe que les temps d'exécutions sont plus longs en général sur l'image ayant la plus grande taille. En renderscript, les temps d'exécution varie assez peu d'un effet à l'autre, on ne dépasse jamais la seconde. En java, les temps d'exécution varie bien plus d'un effet à l'autre, les effets les plus "basiques" sont très courts, souvent plus courts qu'en renderscript, mais les effet plus "complexes" : ceux qui effectuent une conversion en hsv ou une convolution sont beaucoup plus longs, on dépasse les deux minutes de calcul sur les filtres moyenneur et Gaussien sur l'image en couleur par exemple.

7.2 Tests en mémoire (nombre d'allocations)

Ces tests ont été effectués avec l'émulateur Galaxy Nexus (API 28), sur l'image en couleur, de taille 949x650.

Effet \ Méthode	RS Couleur	JAVA Couleur
Égalisation histogramme	97	3
Posteriser (à 100%)	155	65
Luminosité (à 100%)	103	14
Contraste (à 100%)	59	19
Griser	93	3
Négatif	93	2
Saturation (à 180)	349	4
Coloriser (à 100)	205	6
Garder couleur (à 50)	135	58001
Filtre moyenneur (25)	256	85
Filtre Gaussien (25)	270	98
Sobel	231	99
Prewitt	89	100
Laplace (cx8)	199	217
Emboss	189	147
Effet cartoon	420	X
Effet dessin au crayon	232	X

On observe un phénomène assez similaire de celui des tests en temps : en renderscript, les résultats sont homogènes : on dépasse souvent les cents allocations, tous les effets demandent un grand nombre d'allocations même les "basiques" ; tandis qu'en java, les résultats sont plus hétérogènes : les effets sans convolution demandent peu de mémoire alors que les convolutions ont besoin d'une centaine d'allocations ou plus. La seule exception du tableau est la version java de l'effet qui permet de garder une couleur. Cet effet devrait prendre peu de mémoire mais il fait 58001 allocations, ce qui est trop. Une amélioration du logiciel consisterait à optimiser cet algorithme en mémoire.