Rapport de projet technologique
Licence 3

Présentation

Dans le cadre de l'UE "Projet technologique" de notre 3e année de licence informatique à l'université de Bordeaux, nous avons développé une application mobile fonctionnant sous Android intitulé "Image Editor" qui est un éditeur d'images. Elle permet de modifier certaines caractéristiques d'une image ou d'une photo comme la luminosité ou le contraste et également d'appliquer différents filtres et effets tels que le flou et le passage en niveaux de gris.

Design

L'interface utilisateur d'une application mobile doit être intuitive permettant une utilisation facile à prendre en main et compréhensive pour tous. Nous nous sommes inspirés de l'application android "Photo Director" (lien en annexe 1). En effet, nous souhaitions que les menus et icônes soient le plus minimalistes possible afin de laisser toute la place à la photo. Ainsi deux menus transparents ont été implémentés, ne laissant apparaître que l'essentiel. Nous distinguons également les fonctions liées aux modifications de l'image dans le menu situé en bas de l'écran, de celles qui relèvent de "l'utilitaire" situées en haut de l'écran tels que la sauvegarde ou la réinitialisation des modifications. Les différentes modifications sont triées en trois catégories dans le menu en bas qui sont "Ajustement", "Filtre" et "Convolution". A noter que nous avons fait le choix d'une interface en anglais pour cibler un public plus large si l'application venait à être publiée sur le Play Store.

Description des besoins implémentés

Fonctionnels

Interface

Comme expliqué précédemment, notre interface peut être séparée en deux parties distinctes, d'un côté les fonctions utilitaires et de l'autre les modifications d'image.

Comme convenu dans le cahier des charges, nous avons implémenté le chargement d'une image depuis la galerie du téléphone matérialisé par le bouton dans le menu du haut représentant une galerie. S'il s'agit de la première importation d'image et si l'utilisateur est sous Android 6.0 il lui sera demandé lors de l'exécution, d'autoriser l'application à accéder à ses fichiers. S'il utilise une version antérieure à celle-ci, les permissions de l'application décrites dans le Manifest seront acceptées par l'utilisateur lors de l'installation de l'application.
La possibilité de prendre directement une photo depuis la caméra de l'appareil est accessible par le bouton représentant un appareil photo. Une fois validé, la photo est chargée à la place de la photo précédente sans confirmation supplémentaire estimant que la confirmation de l'application caméra suffit.
Pour répondre à ces besoins, nous faisons appel à des Intent. En effet lors de l'appui sur ces boutons, notre application va en quelque sorte envoyer des messages par le biais de la classe *Intent* "je souhaite accèder à la galerie" ou "je veux prendre une photo" et ensuite toutes les applications disponibles sur l'appareil pouvant répondre à cette demande vont être proposées à l'utilisateur. L'intent démarré par l'Activity de notre application transmet des données, ici le chemin d'accès à la photo, qui vont être gérées dans la méthode *onActivityResult* dans laquelle on va donc remplacer notre image actuelle par celle envoyée par l'intent. À noter que pour les besoins du projet et des tests nous chargeons au démarrage une image stockée dans les fichiers de l'application.

Une fois que l'utilisateur applique des modifications sur son image choisie, nous lui avons donné la possibilité de les annuler afin de rétablir son image précédente ou bien directement l'image originale. Nous avons choisi d'implémenter cette fonction de réinitialisation des filtres par le biais d'un bouton situé dans le coin haut gauche afin qu'il ne soit pressé par l'utilisateur que délibérément, matérialisé par une flèche "retour". Lors d'un appui simple, c'est la dernière modification qui est annulée et ainsi de suite. Lors d'un appui long, qui est donc intentionnellement

effectué, l'utilisateur peut recommencer de zéro avec son image originale sans aucune modification.

Lorsque l'utilisateur est satisfait de son image modifiée, pour la sauvegarder dans son appareil, il doit appuyer sur l'icône en haut à droite. Après avoir spécifié le nom, l'image est sauvegardée dans la mémoire interne de l'appareil à l'emplacement *"STORAGE>Internal Storage> ImageEditor*". Cette opération peut éventuellement bloquer le thread principal de par la méthode *bitmap.compress* en fonction de l'appareil utilisé. Dans un second temps, ce sera optimisé en exécutant l'opération dans un autre thread, avec une tâche asynchrone, afin de ne pas bloquer l'interface et donc ne pas avoir la sensation pour l'utilisateur que l'application ne réponde plus.

Enfin, L'utilisateur a la possibilité de zoomer et scroller sur son image afin de mieux visualiser les modifications. Afin de scroller, un seul doigt doit être posé sur l'image et ensuite déplacé. Pour zoomer, deux doigts sont nécessaires afin d'effectuer un "étirement" ou une "diminution" de la distance entre les doigts sur l'image qui va alors grossir. À noté que pour le moment, l'utilisateur peut scroller au-delà des limites de l'image. Malgré une compréhension théorique du problème et des tests en utilisants la classe RectF pour déterminer les bords de l'image nous n'avons pas pu régler ce problème. Il est cependant prévu d'améliorer le scroll pour le rendu final de l'application.

Traitement d'image

Conformément aux demandes du cahier des charges, nous avons implémenté différents traitements d'image.

- Les modificateurs de luminosité et contraste, qui vont faire varier ces paramètres en fonctions du choix d'intensité de l'utilisateur à l'aide d'une barre de défilement.
- Le passage en noir et blanc, avec possibilité de conserver une couleur choisie par l'utilisateur.
- Des filtres de couleur comme le mode sépia ou le changement de teinte de l'image en fonction d'une couleur choisie par l'utilisateur.
- Un filtre d'égalisation d'histogramme permettant d'ajuster le contraste d'une image avec un contraste trop bas de façon optimale.
- Des filtres de convolution ajoutant un effet de flou à l'image, soit pas un calcul de moyenne, soit en utilisant une fonction gaussienne. Ces filtres possèdent différents facteurs d'intensité rendant l'image plus ou moins floue.
- Des filtres de convolution permettant de détecter et mettre en valeur les contours d'une image en utilisant des algorithmes tels que Sobel ou Laplacien.

Le filtre Laplacien n'est pas pleinement fonctionnel. En effet, malgré une compréhension théorique de l'algorithme, nous n'obtenons pas le résultat attendu avec notre implémentation. Le résultat montre que ce ne sont pas les bons pixels sélectionnés et qu'il y a comme une sorte de décalage. Même si il n'y a pas un lissage avec le filtre Gaussien au préalable, ce n'est pas le problème détecté. Après des recherches, pour le rendu final nous allons essayer de le mettre en place en utilisant OpenCV qui est une librairie très répandue pour le traitement d'image (lien annexe 4).

A noté pour les tests que les opérations de convolution, notamment Laplacien et Sobel sont particulièrement longues.

Non fonctionnels

Pour la réalisation de ce projet, nous avons utilisé un gestionnaire de version de code source nommé Git qui permet de garder un historique des fichiers et de coordonner le travail effectué par différents membres sur ces fichiers.

Le traitement d'image est un processus qui nécessite une forte utilisation de la mémoire vive puisque ce sont des opérations lourdes à effectuer. Afin d'obtenir une application fluide sous un téléphone mobile, nous avons optimisé le code de différentes manières de part une utilisation limitée des appels pour obtenir les pixels à traiter et l'utilisation de threads expliqué ci-dessous. A noté que pour ce type d'application, nous avons agrandit la taille du tas (*heap size)* en ajoutant *android:largeHeap="true" (lien annexe 2)* dans le manifest de l'application. Ainsi le système d'exploitation va accorder un plus grand tas pour les processus de notre application.

Architecture

Parmis les fonctionnalités expliquées précédemment, nous allons rentrer plus en détail dans l'implémentation de certaine d'entre elles.

Annulation de filtre

L'annulation de filtre est une fonctionnalité présente dans la plupart des applications de traitement d'image, nous avons souhaité l'implémenter afin d'essayer de comprendre de quelle manière les autres applications la mette en place.
Il existe plusieurs manières de créer les fonctionnalités "annuler" et "refaire" pour une application, nous nous concentrerons uniquement sur l'annulation dans notre cas.

L'utilisation du patron de conception Commande permet d'exécuter, de stocker et de les annuler des commandes dans l'ordre, à condition qu'elles implémentent une fonction undo(). Dans notre cas d'édition d'image, il n'est pas possible pour la plupart des filtres de créer une fonction undo qui viendrait à inverser les modifications de façon algorithmique, il est par exemple impossible de rendre sa couleur à une image passée en noir et blanc.
Dans le cas où le patron de conception Commande ne fonctionne pas pour les undo, il était également possible de stocker l'image de départ, puis de lui réappliquer toutes les modifications sauvegardées sauf la dernière, mais dans notre cas, les modifications prennent trop de temps, et cette méthode serait très frustrante pour l'utilisateur s'il avait appliqué de nombreuses modifications.

Le choix qui a été fait est d'utiliser un objet représentant une pile, dans lequel le résultat d'un filtre, sous forme d'image, est stocké. Ainsi, la fonction setImageBitmap de notre ImageView, en plus d'afficher l'image dans l'ImageView vient la placer dans un objet de type Stack, tout en haut de la pile.
A chaque clic sur le bouton Undo, une fonction undo() est appelée, elle va retirer la dernière image placée en haut de la pile à l'aide de la fonction Stack.pop(), et placer la nouvelle dernière récupérée via la fonction Stack.peek() comme image affichée par l'ImageView.
Cette méthode est quasi instantanée, elle ne bloque pas l'interface et suit une complexion constante, quel que soit le filtre annulé. En revanche son plus gros problème est le stockage d'image, les images sont stockées en mémoire vive via l'utilisation de Stack, et peuvent prendre une place importante en mémoire si beaucoup de filtres sont appliqués.

Pour la future version, nous souhaitons implémenter une manière de compresser les images stockées en premières dans la pile et les décompresser au fur et à mesure des annulation en utilisant des outils proposés par l'équipe Android tels que Glide permettant une mise en cache efficace des images Bitmap.

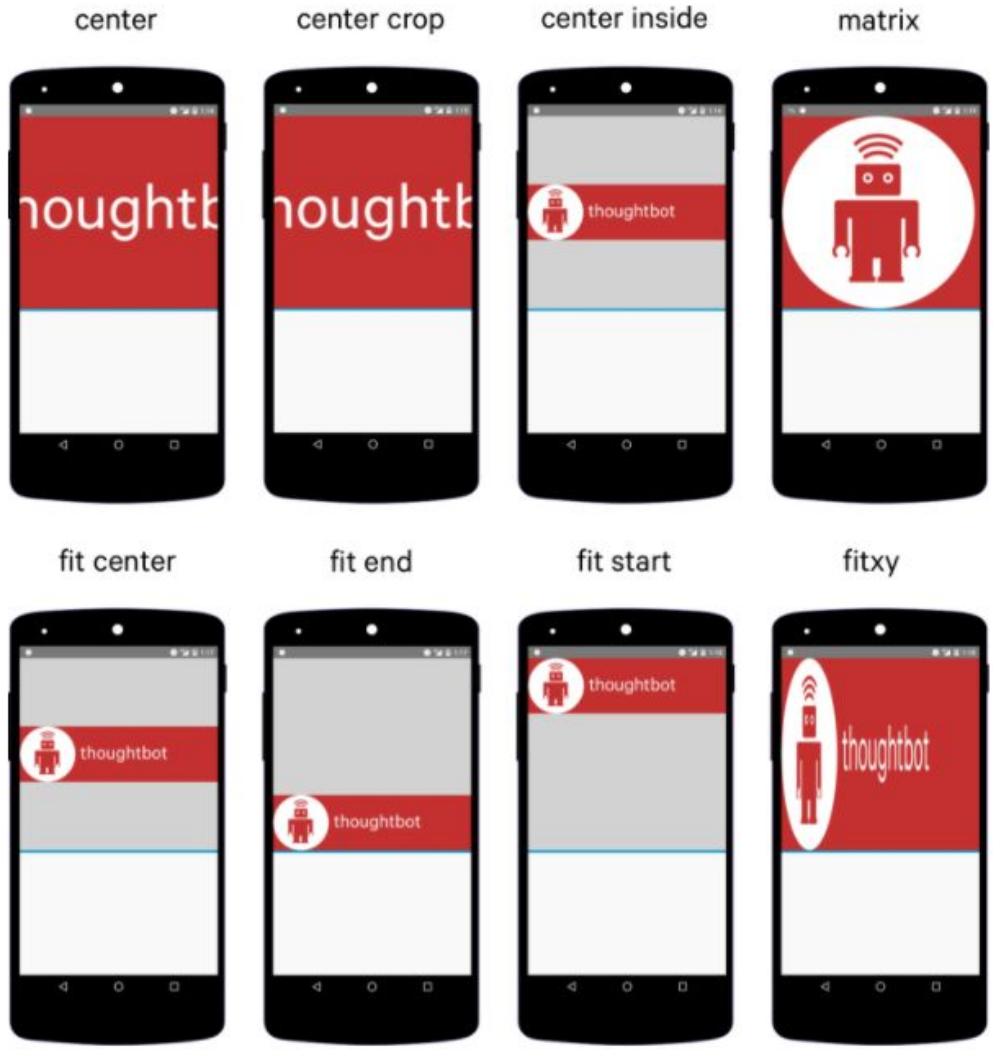
Nous allons également mettre en place la fonction "refaire" pour permettre à l'utilisateur de naviguer dans les deux sens entre ses modifications.

Classe personnalisée ImageView

Afin de représenter une image, en Android, celle-ci est placée dans un vue de la classe ImageView. Dans le but déterminer comme nous le souhaitons le comportement de l'imageView, nous avons créée une classe personnalisée. Pour ce faire, nous avons implémenté la classe ZoomAndScrollImageView dérivant de la

classe ImageView. Ainsi, nous pouvons *override* les méthodes qui nous servent pour implémenter notre propre zoom et scroll ainsi qu'y placer nos méthodes utilisées pour le *undo*, la réinitialisation des filtres.

Pour le Zoom et Scroll, nous utilisons des objets de la classe Matrix qui est utilisée pour manipuler des coordonnées. Pour ce faire, il faut passer le scaling mode de l'imageView qui contrôle comment est dessinée celle-ci, en mode Matrix lorsque l'utilisateur touche l'image avec s*etScaleType(ImageView.ScaleType.MATRIX).*
L'inconvénient d'avoir utilisé une matrice pour réaliser ces opérations, c'est que la valeur par défaut de ScaleType d'une imageView est FIT_CENTER. Donc lorsque nous passons en Matrix voici ce qui se produit:

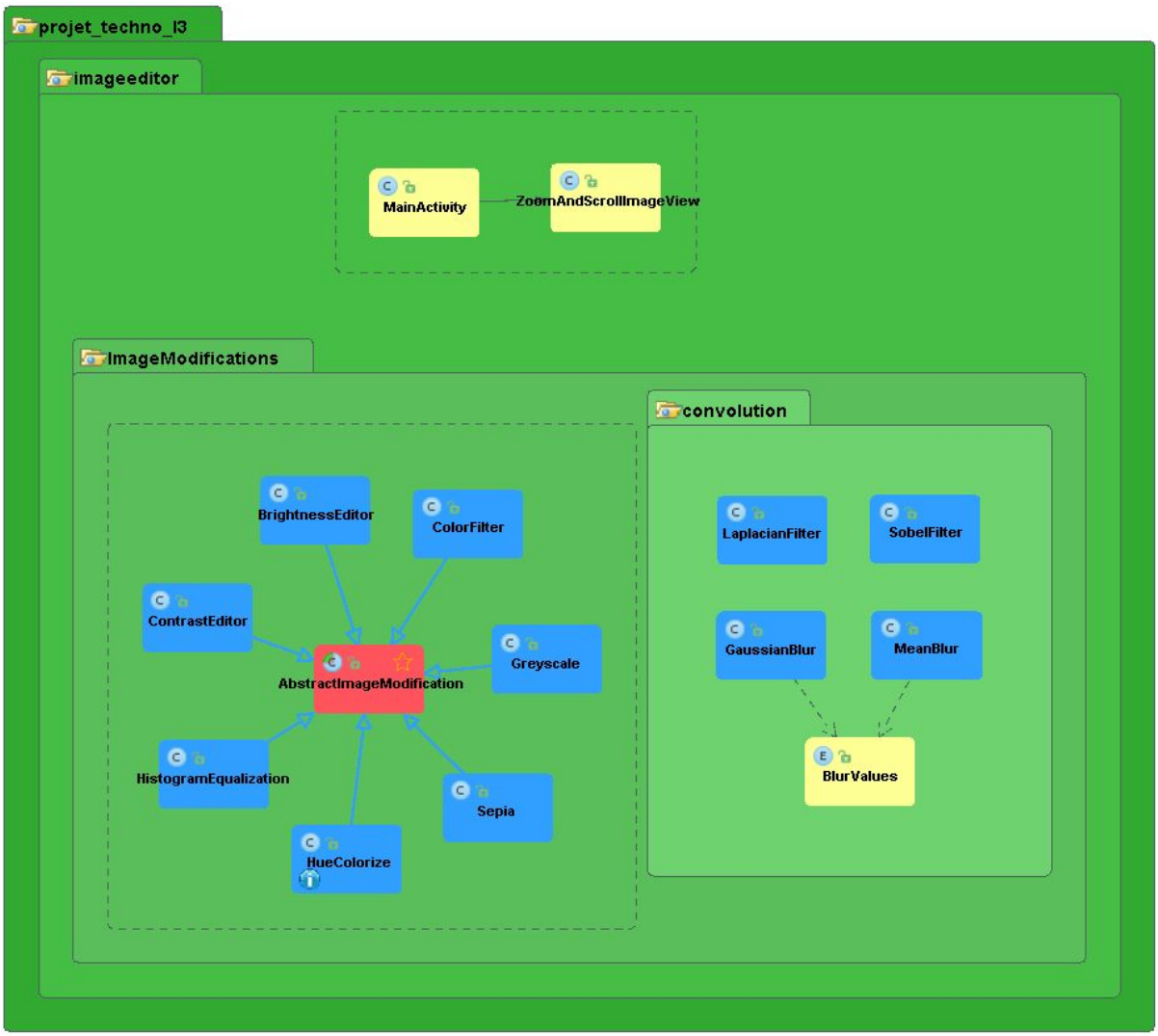


L'image est donc bouger d'un coup au premier touché par l'utilisateur, parfois "grossit" en fonction de la taille de l'image.

Pour l'utilisateur, nous avons préféré laisser FIT_CENTER par défaut afin qu'il puisse avoir un rendu global de son image.

Classe abstraite Imagemodif

Afin de garder un code évolutif par la suite et d'éviter la redondance de code, nous avons implémenté une classe abstraite (qui ne peut être instanciée) *AbstractImageModification* qui sert de base pour ses classes dérivées. Son implémentation n'est pas complète et ses méthodes doivent être implémentées dans ses classes filles. Le diagramme de classe simplifié ci-dessous explique notre architecture.



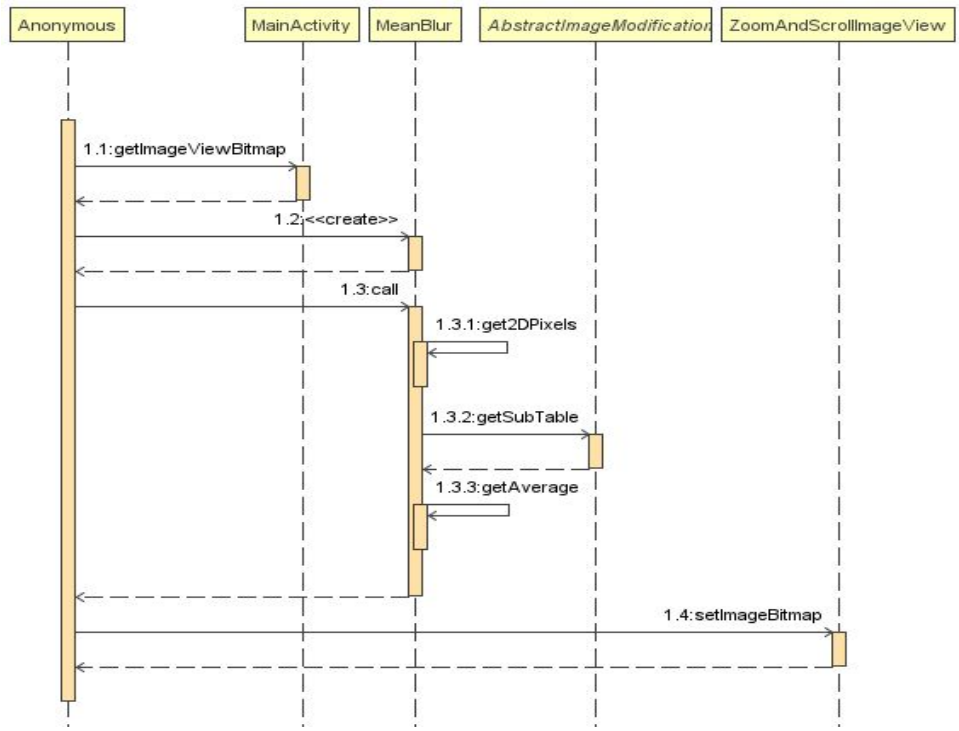
Ainsi, pour chaque modification d'image, une nouvelle classe est créée. La classe abstraite mère contient une bitmap source et des méthodes en communs qui servent à plusieurs type de modification. Si dans le futur nous souhaitons ajouter un nouveau type de flou par exemple, il suffit de créer une nouvelle classe qui implémente la classe abstraite qui n'aura plus qu'a finir d'implémenter les méthodes définis dans la classe mère.

Notre classe abstraite implémente elle-même une interface, Callable. Cette interface sert à ce que les instances de classe implémentant voient leur code exécuté dans un autre thread

que celui principale. Ainsi, ses classes filles implémente l'interface Callable également et doivent définir leur méthode call().
Cette façon permet de ne pas surcharger le thread principale qui s'occupe notamment du rendu de l'interface. Si celui est bloqué par une longue opération, l'utilisateur aura l'impression que l'application aura planté ou *freeze*. Cependant, nous avons remarqué que l'interface se bloquait lors de nos grosses opérations. Il en résulte que nous allons utiliser pour le rendu final des AsyncTask, des tâches asynchrones à la place de Callable. Puisque lorsque nous lancions une grosse opération dans un autre thread, le thread principal qui l'a lancé est dans l'attente du code de retour de celui-ci. Donc contrairement à ce que l'on pensait, la grosse opération est bien effectuée dans un second thread, mais le principale attends que celui-ci se termine pour ensuite reprendre son activité. Ce n'est pas ce qui est voulu. Alors l'utilisation ici de tâches asynchrones est évidente puisque qu'une fois lancée par le thread principal, il n'attend rien en retour des ces tâches et poursuit son exécution. La tâche asynchrone s'exécute donc en parallèle et une fois que celle-ci est terminée, c'est elle même qui va faire le rendu visuel. Ainsi l'UI thread (pour User Interface thread) n'est plus bloqué.

Filtre moyenneur

Le filtre moyenneur, appelé MeanBlur dans notre application, permet d'appliquer un effet de flou à l'image, il est dérivé de la classe AbstractImageModification et implémente donc la méthode call() et l'attribut Bitmap src.
Le diagramme suivant décrit les étapes du filtre



Une fois le filtre créé avec l'image à modifier comme paramètre, l'appel à call() va lancer les modifications.

La fonction call va récupérer une liste des pixels contenus dans l'image à l'aide de la fonction getPixels, puis grâce à get2DPixels, cette liste va être transformée en tableau à deux dimensions.
Pour chacun des pixels de ce tableau la fonction getSubTable va récupérer les pixels l'entourant dans un tableau à deux dimensions, de taille 3, 5 ou 7, défini lors du clic sur l'interface par l'utilisateur, représentés par les boutons Min, Med et Max.
La fonction getAverage va parcourir ce sous-tableau, et réaliser la moyenne des canaux rouge, vert et bleu de chacun des pixels présents, créer un nouveau pixel correspondant à ces moyennes et le renvoyer à la fonction call qui va stocker ce pixel, dans une nouvelle liste de pixels ne contenant que ceux ayant été transformés.

Une fois l'opération réalisée pour tous les pixels, la fonction call va créer une nouvelle Bitmap à l'aide de la liste des pixels transformés, et renvoyer cette Bitmap à la MainActivity qui va envoyer l'image à l'ImageView pour qu'elle soit affichée et stockée dans la pile.

Choix des nouvelles fonctionnalités

Pour la deuxième release de notre projet prévue dans un mois, nous souhaitons ajouter les fonctionnalités suivantes:

- Incruster des objets sur des points précis d'un visage (yeux, oreilles, tête, nez, bouche)
- Simuler un effet dessin au crayon
- Transformer une photo en négatif
- Ajout de traductions français/anglais pour l'interface
- Optimiser le code avec implémentation de tâches asynchrones pour les opérations lourdes afin de ne pas bloquer le thread principale de l'interface.
- Corriger l'implémentation du filtre Laplacien (en utilisant OpenCV)
- Améliorer le scroll

Annexes

Annexe 1: PhotoDirector
https://play.google.com/store/apps/details?id=com.cyberlink.photodirector&hl=en

Annexe 2: Documentation à propos de Heap size
https://developer.android.com/reference/android/R.styleable.html#AndroidManifestApplication_largeHeap

Annexe 3: Outil de gestion de Bitmap
https://github.com/bumptech/glide

Annexe 4: OpenCV
http://opencv.org/