Antoine Gagnon
Cyril Niobé

Université de Bordeaux

Création d'une application Android de traitement d'image

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 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 quatre catégories dans le menu en bas qui sont "Ajustement", "Filtre", "Incrustation" 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.

Deuxième release

Lors de la première release de notre application, nous avions émis à la fin de notre premier rapport ce que nous souhaitions implémenter et améliorer pour la deuxième version. Ainsi ont été implémentés:

- L'incrustation d'objets sur des points précis du visage
- L'effet de dessin au crayon
- L'effet négatif sur une photo
- Le bon fonctionnement du filtre Laplacien
- Optimisation du code avec l'ajout des tâches asynchrones pour chaque traitement d'image
- L'utilisation de RenderScript
- L'utilisation d'openCV
- L'ajout de fichier de traduction des Strings

Besoins 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.

Chargement d'images

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 validée, 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.

Annulation des modifications

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 court, c'est la dernière modification qui est annulée et ainsi de suite. Lors d'un appui long, l'utilisateur peut recommencer de zéro avec son image originale.

Cette fonctionnalité est mise en place par la sauvegarde de chaque étape de l'image dans une pile, et un désempilement d'une ou plusieurs images en fonction de l'appui long ou court sur le bouton de retour.

Sauvegarde de l'image

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.

Lors de cette deuxième release, nous avons changé la méthode *savedImage* de la MainActivity en classe interne privée afin qu'elle puisse dériver de AsyncTask et donc performer l'opération dans une tâche asynchrone comme prévu afin de ne plus bloquer le thread principal si l'opération est un peu longue.
L'image est sauvegardée dans la mémoire interne du téléphone (aussi appelé Internal Storage) sous le chemin suivant: Internal Storage > ImageEditor > filename.png

Zoom et déplacement dans l'image

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 noter que 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 pour le rendu de la deuxième release car nous nous sommes consacrés à implémenter d'autres choses.

Traitement d'image

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

Classe abstraite

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) *AbstractImageModificationAsyncTask* qui sert de base pour tous les traitements d'image.

Son implémentation n'est pas complète et ses méthodes doivent être définies dans ses classes filles.

Ainsi, pour chaque modification d'image, une nouvelle classe est créée. La classe abstraite mère contient une bitmap source et résultat, un pointeur vers l'activité principale 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 dérive elle même d'une classe abstraite, AsyncTask. Cette interface sert à ce que ses classes dérivées voient leur code exécuté dans un autre thread que le principal.

Cette classe mère permet de ne pas surcharger le thread principal qui s'occupe notamment du rendu de l'interface, elle va réaliser les opérations de traitement d'image sur un thread différent. Si le thread principal est bloqué par une longue opération, l'utilisateur aura l'impression que l'application a reçu une erreur et ne marche plus. La tâche asynchrone s'exécute donc en parallèle et une fois que celle-ci est terminée, sa valeur de retour, ici une image va être assignée à *l'ImageView*. Ainsi l'UI thread (pour User Interface thread) n'est plus bloqué.

Luminosité et Contraste

La luminosité et le contraste proposent tous deux la même interface à l'utilisateur, lorsque l'une des deux options est choisie, une *SeekBar* va apparaître et permettre de régler l'intensité de la modification.
Cette *SeekBar* va fournir une valeur entre -100 et 100 à l'algorithme, qui va agir différemment suivant la modification.

Dans le cas de la luminosité, chacun des pixels de l'image va être converti dans les canaux HSV (Hue, Saturation, Value), la troisième valeur **Value** correspond à la luminosité, notre algorithme va donc ramener la valeur entre -100 et 100 à un ensemble [-1,1] qu'il va ensuite ajouter à la **Value** comprise entre 0 et 1, pour au final retourner le pixel modifié dans l'image.

Pour le contraste, le résultat de la *SeekBar* en -100 et 100 va être ramenée à l'ensemble [-255,255], on va ensuite utiliser cette valeur pour calculer notre facteur de contraste, afin de réduire ou augmenter la taille de l'histogramme de l'image. On va appliquer le contraste à chacun des canaux RGB des pixels de l'image avant de reformer celle ci. La formule utilisée pour modifier le contraste a été trouvée sur le site donnée en Annexe 6.

Sepia et Noir et Blanc

Le sepia correspond à ajouter une teinte jaunie à l'image, la force du sepia est donnée par une valeur arbitraire de 20, qui selon nous donnait un effet de bonne qualité dans la plupart des cas. Pour appliquer l'effet, pour chacun des pixels de l'image nous appliquons la formule:

```
int gry = (r + g + b) / 3;
r = g = b = gry;
r = r + (SEPIA_DEPTH * 2);
g = g + SEPIA_DEPTH;
b -= 80;
```

Ainsi chacun des pixels va obtenir une teinte plus jaunâtre, donnant un effet de sepia.

Le noir et blanc quant à lui change les valeurs RGB de chacun des pixels en leur utilisant la formule:

0.299 * r + 0.587 * g + 0.114 * b

On obtient une valeur de gris qui peut être appliquée à tous les canaux rouge vert et bleu du pixel.

Négatif

Pour l'algorithme de l'effet négatif d'une image, nous nous sommes inspirés de la documentation android officielle de la classe ColorMatrix (https://developer.android.com/reference/android/graphics/ColorMatrix.html). Celle-ci permet de modifier les couleurs et les composants alpha d'une Bitmap.

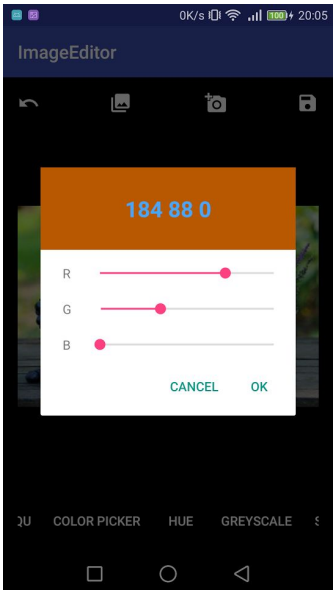
En effet, la matrice pouvant être définie comme un array:

```
[ -1, 0, 0, 0, 255,
    0, -1, 0, 0, 255,
    0, 0, -1, 0, 255,
    0, 0, 0, 1, 0 ]
```

Qui va être appliqué comme ceci pour chaque canal:

```
R' = a*R + b*G + c*B + d*A + e;
G' = f*R + g*G + h*B + i*A + j;
B' = k*R + l*G + m*B + n*A + o;
A' = p*R + q*G + r*B + s*A + t;
```

Changement de teinte



Pour cet algorithme, l'utilisateur a accès à un outil de sélection de couleur. Grâce à lui il va pouvoir choisir la teinte qu'il souhaite que son image prenne. Une fois la teinte sélectionnée, elle est transmise à l'algorithme qui va récupérer la valeur HSV de chacun des pixels, dans ce cas-ci il va utiliser la **Hue** du pixel, qui correspond à la teinte, qu'il va remplacer par la valeur choisie par l'utilisateur. Une fois la modification appliquée à tous les pixels, l'image est reformée et retournée.

Égalisation d'histogramme

La version originale d'égalisation était réalisée "à la main", elle consistait en l'utilisation des formules de distribution cumulative pour équilibrer l'histogramme.
Après avoir créé un tableau comptant le nombre de pixel en fonction de leur **Value** dans les canaux HSV, on équilibre le tableau des valeurs avec la fonction *generateCDF().* Une fois le nouveau tableau généré, on change la **Value** de chacun des pixels avec la nouvelle obtenue. Cela permet donc d'obtenir un histogramme équilibré, permettant l'image d'avoir un meilleur contraste.

La version utilisée au final utilise OpenCV, elle convertit l'image originale en un *Mat* de canaux YCrBr qui permet de récupérer uniquement l'intensité (la valeur de gris) des pixels, indépendamment de leur couleur. On utilise ensuite la fonction *Imgproc.equalizeHist* sur le canal de l'intensité **Y**, avant de reformer une image utilisant les canaux RGB depuis les canaux YCrBr modifiés.

Avec le téléphone utilisé pour les tests, les résultats sont clairs, la version originale prend environ 4300 millisecondes, là où la version utilisant OpenCV prend environ 48 millisecondes. C'est pour cette raison que nous avons choisi de conserver la version utilisant OpenCV, les résultats sur l'image étant les même.

Noir et blanc sauf une couleur

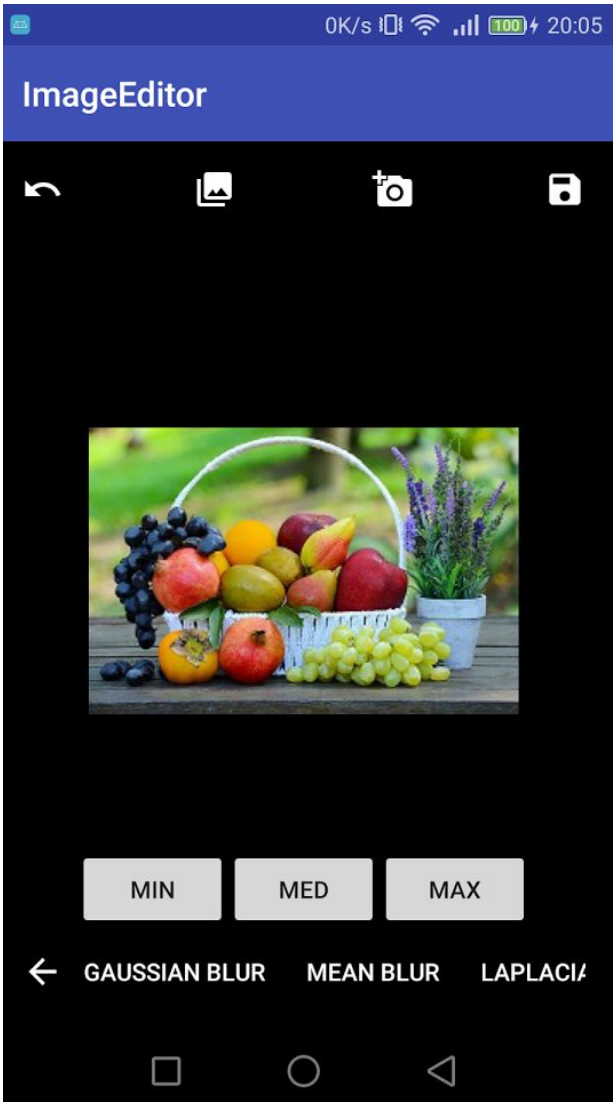
Dans ce cas, l'utilisateur va sélectionner la couleur qu'il souhaite conserver dans l'image, qui va être transmise à l'algorithme. L'algorithme va créer un intervalle d'acceptation des couleurs, dans notre cas d'une valeur arbitraire de 25 par rapport à la valeur de teinte allant de 0 à 360. Chacun des pixels de l'image va être passé en HSV, et l'algorithme va vérifier que la valeur **Hue** du pixel est comprise dans l'intervalle. Si elle l'est, le pixel est copié tel quel dans la nouvelle image, sinon c'est son équivalent en gris qui l'est.

Une fois tous les pixels vérifiés, l'image résultante est retournée.

Flou Gaussien

Lors de la sélection du flou gaussien, l'utilisateur possédait plusieurs options, lui permettant d'obtenir un flou plus ou moins puissant. Les valeurs MIN, MED et MAX étant gérée différemment suivant les versions que nous avons implémentées.

Le flou gaussien a tout d'abord été réalisé "à la main", en créant une matrice de 5 par 5 ou 3 par 3, suivant le choix d'un flou minimal ou maximal par l'utilisateur, contenant les valeurs d'une fonction gaussienne. En récupérant tous les pixels de l'image sous forme de tableau, l'algorithme venait ensuite appliquer à chaque pixel la matrice, créant ainsi une moyenne gaussienne des pixels l'entourant, permettant d'appliquer un effet de flou.



Pour les bords de l'image, cette méthode les rendait noirs, créant ainsi une bordure d'un ou deux pixels en fonction de la taille de matrice.

L'autre version consistait à utiliser une classe native de RenderScript appelée *ScriptIntrinsicBlur* permettant d'appliquer un flou gaussien à une image avec une matrice de taille 3 à 21, permettant donc des flous bien plus puissant. La taille de la matrice en fonction des boutons correspondait à des valeurs de 3, 8 et 13.

Avec le téléphone utilisé pour les tests, les résultats sont clairs, la version originale prend environ 6500 ms pour une matrice 3x3 et 13000 pour une matrice 5x5, là où la version utilisant le RenderScript prend environ 30 ms quelle que soit la taille de la matrice. C'est pour cette raison que nous avons choisi de conserver la version utilisant RenderScript, les résultats sur l'image étant les même.

Flou moyenneur

Le flou moyenneur de la même façon, utilisait une méthode rudimentaire, en appliquant une matrice faisant la moyenne des canaux rouge vert et bleu des pixels environnant, en rendant noir les pixels de la bordure de la même façon. La taille de la matrice était également définie par des valeurs de taille de matrice de 3 et 5.

La version utilisant RenderScript fonctionne exactement de la même manière, en utilisant une matrice qu'elle va appliquer à chaque pixel en utilisant les fonctions *ScriptIntrinsicConvolve3x3* ou *ScriptIntrinsicConvolve5x5* suivant le choix de l'utilisateur. La matrice dans les deux versions étant simplement composée des valeurs 1/(tailleMatrice²).
Nous avons également choisi de conserver la version utilisant RenderScript, observant des valeurs de 28ms pour une matrice 3x3 et 42ms pour une matrice 5x5 contre 18000 et 22000 ms pour la version rudimentaire.

Laplacien

Concernant le filtre Laplacien, après avoir essayé de le coder nativement pour la première release, il en résultait un problème que nous n'arrivions pas à corriger.
Ainsi, pour la deuxième release, nous avons décidé d'utiliser openCV qui allait également être utilisé pour l'incrustation d'image. Nous utilisons donc les méthodes fournies par openCV afin d'appliquer le filtre Laplacien sur une image.

Sobel

Lors de la première release, nous avions remarqué que l'application du filtre Sobel à une image prenait un temps extrêmement long. Afin de palier à cela, l'exécution de celui-ci en tâche asynchrone réduit le temps de l'opération même s'il reste un peu plus long que les autres filtres de convolution. Nous avons utilisé la formule de Sobel vu en cours ainsi qu'une matrice de 3 par 3.

Cependant, nous avions choisi de passer l'image en niveau de gris avant afin que le contour de l'image soit en noir et blanc passant ainsi inaperçu sur le fait que l'algorithme de Sobel utilisant une matrice ne passe pas sur les pixels formant les contours de l'image.

Mais maintenant que nous utilisons des AsyncTask, il est à noter qu'il est impossible d'appeler dans une AsyncTask, une autre AsyncTask car celle-ci ne peuvent être lancées que par le thread principal, donc dans la MainActivity. Alors, il est impossible de créer une instance de la classe GreyScale qui passe une image en noir et blanc en réalisant ses opération au sein d'une tâche asynchrone.
Donc une duplication brute du code qui passe une image en noir et blanc et "nécessaire" avant l'algorithme de Sobel afin que les contours ne restent pas colorés, ce qui aurait été visible et gênant. Il y a une petite perte de temps ici mais négligeable sur l'ensemble de l'opération car passer une image en noir et blanc ne met pas beaucoup de temps comparé à appliquer l'effet Sobel.

Incrustations d'images

L'incrustation d'images utilise la technologie des Haar Cascades proposée par OpenCV.
Les Haar Cascades sont des fichiers xml permettant de décrire des éléments de façon générale. Dans notre cas nous les utilisons pour décrire l'oeil, la bouche et le nez d'un humain, afin de pouvoir les détecter dans une image.
Ces fichiers fournis pas OpenCV sont dont chargés en mémoire, puis l'image source va être transformée en niveau de gris, une égalisation d'histogramme va être également appliquée, afin de faciliter la détection d'éléments. Pour chacun des types d'éléments recherchés, une taille minimale en rapport avec la taille de l'image est définie, évitant ainsi de rechercher des éléments infimes ralentissant l'algorithme. Dans notre cas nous avons estimé que les photos utilisées représentaient principalement le visage, pouvant être considérées comme des "selfies". La méthode *CascadeClassifier.detectMultiScale* va donc trouver les éléments correspondant aux critères des Haar Cascades et retourner une liste de rectangle les localisant dans l'image.

Ensuite, à l'aide de la position et de la taille de ces rectangles, l'algorithme vient redimensionner notre image à incruster à la bonne taille, avant de la copier à l'emplacement du rectangle de détection. Ainsi l'image d'oeil, de nez ou de bouche va venir remplacer celle de la personne sur l'image.

Effet dessin au crayon

Afin d'implémenter l'effet de dessin au crayon sur une image, nous avons cherché d'abord quel était l'algorithme nécessaire.
Il faut d'abord convertir en niveau de gris l'image, inverser les couleurs de cette image. Ensuite appliquer l'effet de flou Gaussien et ensuite mélanger avec la technique de densité de couleur l'image obtenue avec l'image de base en niveau de gris.

Ici, pour appliquer des effets comme le niveau de gris et l'inversion de couleur sont des algorithmes déjà utilisées ailleurs mais comme expliqué précédemment, une duplication de code est nécessaire car nous ne pouvions pas lancer la tâche asynchrone du niveau de gris par exemple, à partir de la tâche asynchrone du dessin au crayon.

L'application du filtre gaussien a pu être appliqué par openCV afin de gagner en temps d'exécution.

Enfin, l'algorithme du mélange d'image par densité de couleur afin d'obtenir l'effet "dessin au crayon" a été implémenté en s'aidant de ce lien https://stackoverflow.com/questions/9841845/color-dodge-blend-bitmap.

Il aurait été possible d'utiliser openCV pour le niveau de gris et l'inversion de couleur (négatif) afin de ne pas dupliquer du code mais l'effet obtenu était plus lumineux et avait donc un rendu moins bien de l'effet crayon au dessin.

Besoin Non fonctionnels

Gestionnaire de version

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.

Allocation mémoire

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 noter 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.

Traduction Anglais/Français

Afin de permettre l'utilisation de l'application par une plus grande population, nous avons extrait tous les textes de l'application, et avons ajouté une traduction en français via l'utilisation des fichiers *strings.xml*.

Architecture

Notre architecture a été expliquée en détails dans le rapport de notre premiere release. Nous vous invitons à se répertorier à celui-ci pour cette partie.

Tests

Les tests de performances ont été réalisés sur un Honor 7, avec 3 GB de Ram, ainsi qu'un processeur Octa-core (4x2.2 GHz Cortex-A53 & 4x1.5 GHz Cortex-A53) et un chipset graphique Mali-T628 MP4.

L'utilisation du même téléphone lors de tous les tests a permis de vérifier l'amélioration du temps d'exécution des algorithmes lors de l'utilisation de RenderScript ou d'OpenCV.
Lors des tests, le téléphone n'avait aucune application non-native qui tournait en arrière plan, uniquement l'Image Editor, cela permettait d'être sûr que l'utilisation était sensiblement la même à tout moment.

Étant en binôme, nous avons pu tester tous nos traitements d'images ainsi que les fonctions utilitaires (chargement d'une image, undo, sauvegarde d'une image...) sur deux téléphones différents. Celui cité précédemment ainsi qu'un Oneplus 3 avec 6 GB de ram, un processeur Snapdragon 620, Quad-core, 2.2 Ghz et une GPU Adreno 530. Le système d'exploitation est une rom personnalisé du constructeur, basé sur Android Nougat 7.0, qui se nomme Oxygen.

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/

Annexe 6: Formule du contraste

http://www.dfstudios.co.uk/articles/programming/image-programming-algorithms/image-processing-algorithms-part-5-contrast-adjustment/