UNIVERSITÉ DE BORDEAUX
Collège Sciences et Technologies
Département Licence
Bâtiment A22, 351 cours de la Libération
33405 Talence Cedex
FRANCE
05 40 00 66 03

RAPPORT

DÉVELOPPEMENT D'UN PROJET EN TRAITEMENT D'IMAGES

BESSE Emma HONORÉ Cauli VEILLAT Emma

SOMMAIRE

INTRODUCTION

Cette introduction tiendra lieu de remerciements et de présentation globale du projet.

Nous tenons à remercier nos professeurs, Anne VIALARD et Boris MANSENCAL, de nous avoir montré comment réaliser une application de traitement. Outre le sujet qui était particulièrement intéressant, nous avons, par exemple, appris à gérer et à aller au bout d'un projet, à chercher par nous-même les solutions aux erreurs de compilation et d'exécution.

Nous tenons également à remercier nos familles et amis pour leurs précieux conseils et leur aide pour la relecture des cahier des charges, README et rapport rédigés lors des deux releases.

Le but de ce projet est de développer une application de traitement d'image sur smartphone avec système Android. Les images peuvent aussi bien être obtenues depuis la galerie du téléphone que directement depuis la caméra. Le projet est découpé en deux parties, et deux releases du logiciel seront donc à rendre. La première release est commune à tous les groupes. Elle est composée des fonctionnalités de base afin de gérer, afficher et sauvegarder les images. De plus seront intégrés quelques traitements d'images basés sur une transformation d'histogramme ou une convolution.

Avec cette première release, un cahier des charges devra être complété, avec un ensemble de fonctionnalités additionnelles choisies parmi une liste de choix.

La deuxième release devra être fournie accompagnée d'un rapport détaillé, contenant les points suivants : une description des algorithmes de traitement d'images implémentés, une description de l'architecture du code et une description des tests effectués démontrant la correction du code et le gain obtenu en terme d'optimisation.

Enfin, il faudra spécifier les versions d'Android sur lesquelles ont été testées l'application, en précisant s'il s'agit d'un téléphone physique et/ou de l'émulateur.

Cette introduction a été rédigée à partir d'extraits des exigences du cahier des charges et du rendu finale.

PHOTOSHIP : LA VERSION FINALE

1. Présentation de l’application

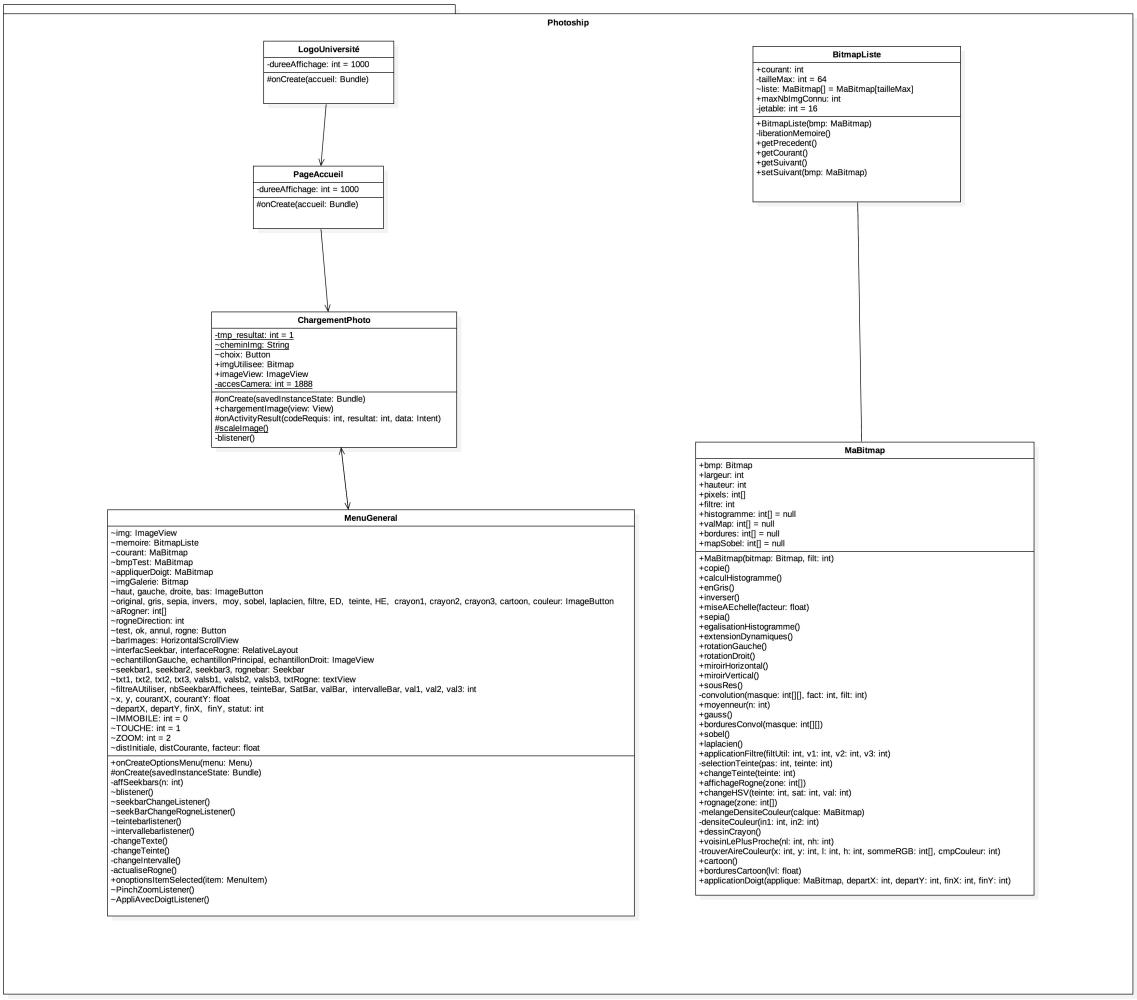
Photoship est donc une application de traitement d’images. Une fois installée sur l’appareil de l’utilisateur, celui-ci peut la lancer et l’utiliser très simplement. L’application affiche alors brièvement le logo de l’université (UB) et celui de Photoship. Elle permet à l’utilisateur d’accéder soit à sa galerie d’images, soit à la caméra de son téléphone. Il lui faut alors valider son choix pour arriver au menu principal de l’application. C’est sur ce menu que l’utilisateur peut appliquer n’importe quelle transformation à sa photo, dont il peut apercevoir le résultat grâce à des boutons-images. Cette transformation peut être soit un filtre simple comme le filtre sepia, soit un filtre nécessitant des seekbars – que l’application fait apparaître si besoin –, ou encore une modification de la structure de l’image comme une rotation de l’image. Les modifications peuvent être annulées ou rétablies grâce à l’existence d’un historique. Enfin, l’utilisateur peut enregistrer l’image transformée dans sa galerie et est ainsi renvoyé au menu de chargement de photo pour choisir une nouvelle image à modifier.

La liste des modifications disponibles grâce aux boutons-images est la suivante :
- Filtre gris
- Filtre sépia
- Filtre négatif
- Filtre flou (moyenneur ou gaussien)
- Filtre sobel
- Filtre laplacien
- Égalisation d’histogramme
- Extension de dynamiques
- Sélection de couleurs
- Filtre coloré
- 3 effets de dessin crayonné plus ou moins marqué
- Effet cartoon
- Modification de la teinte, de la saturation et/ou de la luminosité de l’image.

Elles peuvent être applicables soit intégralement à l’image ou partiellement au doigt par l’utilisateur.

La liste des modifications de la structure de l’images est la suivante :
- Zoom en écartant ou en rapprochant les doigts
- Rotation de l’image dans le sens horaire ou anti-horaire
- Rognage de l’image
- Miroir horizontal ou vertical de l’image
- Modification ou déformation de la taille de l’image.

2. Architecture définitive du projet



L’interface de l’application est représenté par les quatre activités : LogoUniversite, PageAccueil, ChargementPhoto et MenuGeneral. À chacune d’entre elles s’ajoute le fichier XML correspondant. À cela s’ajoute un fichier XML (general_menu.xml) définissant les icônes présentes dans la barre principale de l’application.

La partie « Traitement d’images » de l’application est représentée par les deux classes JAVA BitmapListe et MaBitmap. La classe BitmapListe implémente l’historique qui permettra à l’utilisateur d’annuler ou rétablir les modifications appliquées à l’image. La classe MaBitmap implémente, quant à elle, le nouveau type d’images utilisé dans l’application. Il se compose d’une **Bitmap** classique, de trois entiers représentant la largeur, la hauteur et le filtre utilisé sur l’image, et de cinq tableaux d’entiers représentant les pixels, l’histogramme, les bordures, les pixels gris et les pixels avec le filtre Sobel.

3. Description des algorithmes

Certaines actions se répètent pour toutes les méthodes. Nous les résumerons ici pour ne pas avoir à les redire à chaque description.

Au début de chaque méthode, on crée un tableau de pixels qui va contenir les pixels modifiés de l'image originale.

À la fin de chaque méthode de transformation, on crée une image de type **Bitmap** à partir du tableau précédent. Enfin, on crée l'image à retourner de type **MaBitmap** à partir de l'entier représentant le filtre utilisé et de la **Bitmap**.

La fonction *enGris()* permet de colorer une image en niveaux de gris correspondants. Il n'y a aucun paramètre d'entrée, et la fonction retourne l'image transformée de type **MaBitmap**.
Description de la méthode : Si nécessaire, la fonction appelle *calculHistogramme()* pour avoir l'attribut **valMap** (la tableau de pixels de l'image en niveaux de gris).
On crée un nouveau tableau dans lequel on remplace la couleur du pixel par la valeur correspondante dans **valMap**.

La fonction *inverser()* permet d'afficher une image en négatif. Il n'y a aucun paramètre d'entrée, et la fonction retourne l'image transformée de type **MaBitmap**.
Description de la méthode : On soustrait à 255 – valeur maximale de couleur disponible – la valeur de chaque canal RGB du pixel qu'on est en train de parcourir. On place ainsi la couleur obtenue dans le nouveau tableau de pixels.

La fonction *sepia()* permet d'appliquer un filtre sepia sur une image. Il n'y a aucun paramètre d'entrée, et la fonction retourne l'image transformée de type **MaBitmap**.
Description de la méthode : Dans un premier temps, on crée un entier témoin qui vaut 20. Pour chaque pixel, on remplace : le canal rouge par la valeur de **valMap** correspondante ajoutée au double du témoin, le canal vert par la valeur de **valMap** correspondante ajoutée au témoin et le canal bleu par la valeur de **valMap** correspondante. **valMap** peut être nulle ou contenir les pixels en niveaux de gris. De plus, si les canaux rouge et vert sont plus grands que 255 après chaque somme, ils sont remplacés par l'entier 255.

La fonction *egalisationHistogramme()* permet de renvoyer l'image dont on a amélioré le contraste. Il n'y a aucun paramètre d'entrée et la fonction retourne l'image de type **MaBitmap** avec un histogramme égalisé.
Description de la méthode : Si l'histogramme de l'image n'a pas été calculé auparavant, la fonction appelle calculHistogramme(). À partir de cet l'histogramme, on calcule l'histogramme cumulé. À partir de là, on calcule pour chaque pixel la valeur de luminosité correspondante avec la valeur de l'histogramme cumulé équivalente et on la remplace à la place de la « valeur » dans l'espace HSV.

La fonction *extensionDynamiques()* permet d'améliorer la luminosité de l'image originale. Il n'y a aucun paramètre d'entrée, et la fonction retourne l'image transformée de type **MaBitmap**.
Description de la méthode : Si l'histogramme de l'image n'a pas été calculé auparavant, la fonction appelle *calculHistogramme()*. À partir de là, on calcule la plus petite valeur non nulle ainsi que la plus grande valeur non nulle. On calcule alors pour chaque pixel la valeur de luminosité correspondante et on la remplace à la place de la « valeur » dans l'espace HSV.

La fonction *rotationGauche()* permet de retourner de 90° sur la gauche l'image de type **MaBitmap**. Il n'y a pas de paramètre d'entrée et la fonction retourne l'image de type **MaBitmap** sur la gauche.
Description de la méthode : On place chaque pixel de l'image originale à sa place correspondante dans l'image retournée à gauche.

La fonction *rotationDroit()* permet de retourner de 90° sur la droite l'image de type **MaBitmap**. Il n'y a pas de paramètre d'entrée et la fonction retourne l'image de type **MaBitmap** sur la droite.
Description de la méthode : On place chaque pixel de l'image originale à sa place correspondante dans l'image retournée à droite.

La fonction *miroirHorizontal()* permet de retourner l'image suivant un axe de symétrie horizontal. Il n'y a aucun paramètre d'entrée, et la fonction renvoie l'image de type **MaBitmap** retournée.
Description de la méthode : On place chaque pixel de l'image originale à sa place correspondante dans l'image reflétée en haut/bas.

La fonction *miroirVertical()* permet de retourner l'image suivant un axe de symétrie vertical. Il n'y a aucun paramètre d'entrée, et la fonction renvoie l'image de type **MaBitmap** retournée.
Description de la méthode : On place chaque pixel de l'image originale à sa place correspondante dans l'image reflétée à gauche/droite.

La fonction *convolution()* permet d'appliquer les filtres gaussien et moyenneur. Cette fonction est privée et a pour paramètres le masque (matrice) définissant le filtre à appliquer, le facteur d'application donnée par l'utilisateur et l'entier représenté le filtre utilisé. Elle retourne l'image originale de type **MaBitmap** avec le filtre choisi par l'utilisateur (entre gaussien et moyenneur).
Description de la méthode : Pour chaque pixel, la fonction fait un calcul de convolution avec la matrice donnée, c'est-à-dire que la valeur des pixels voisins est multipliée par le coefficient correspondant dans la matrice, que l'on somme par la suite. Si la somme des coefficients est différente de 1, alors on divise les résultats par cette somme. Pour ne pas avoir à gérer le problème des bords, ceux-ci sont mis en blanc.

La fonction *gauss()* est la fonction qui définit le masque à appliquer pour le filtre gaussien. Il n'y a aucun paramètre d'entrée et la fonction retourne l'image de type **MaBitmap** avec le filtre gaussien.
Description de la méthode : La fonction définit un masque (un tableau à deux dimensions). Ce masque sera appliqué via la méthode *convolution()*. Celle-ci retourne l'image résultante de type **MaBitmap**.

La fonction *moyenneur()* est la fonction qui définit le masque à appliquer pour le filtre moyenneur (ce qui correspond à la moyenne des valeurs des pixels voisins). Le seul paramètre d'entrée est un entier et la fonction retourne l'image de type **MaBitmap** avec le filtre gaussien.
Description de la méthode : La fonction définit un masque entièrement composé de 1 et ayant pour taille l'entier donné par l'utilisateur en paramètres. Ce masque sera appliqué via la méthode *convolution()*. Celle-ci retourne l'image résultante de type **MaBitmap**.

La fonction *bordures Convol()* permet d'appliquer les filtres Sobel et Laplacien. Cette fonction a pour paramètres le masque (matrice) définissant le filtre à appliquer. Elle retourne l'image originale de type **MaBitmap** avec le filtre choisi par l'utilisateur (entre gaussien et moyenneur).
Description de la méthode : Cette méthode a le même fonctionnement que *convolution()* mais ne renvoie pas directement une image (des calculs supplémentaires sont nécessaires par la suite) mais un tableau contenant les sous-tableaux pour les canaux RGB de chaque pixel. Les bordures originales sont conservées en l'état.

La fonction *sobel()* est la fonction qui permet d'afficher les contours de l'image originale à partir de deux convolutions. Il n'y a aucun paramètre d'entrée et la fonction retourne l'image de type **MaBitmap** avec le filtre Sobel.
Description de la méthode : À partir de deux masques à appliquer, on calcule les tableaux correspondants par la fonction *bordures Convol()*. On calcule alors les dérivées secondes horizontale et verticale de ceux-ci, puis leurs normes. On ramène ces normes à un espace allant de 0 à 255 . On remplace enfin les valeurs dans les canaux RGB de chaque pixel.

La fonction *laplacien()* est la fonction qui permet d'afficher les contours de l'image originale à partir d'une seule convolution. Il n'y a aucun paramètre d'entrée et la fonction retourne l'image de type **MaBitmap** avec le filtre Laplacien.
Description de la méthode : À partir d'un masque à appliquer, on calcule le tableau correspondant par la fonction *bordures Convol()*. On calcule alors la dérivée seconde de celui-ci (mais pas de norme car on applique une seule convolution contrairement à *sobel()*). On ramène ces normes à un espace allant de 0 à 255 . On remplace enfin les valeurs dans les canaux RGB de chaque pixel.

La fonction *selectionTeinte()* permet de ne faire ressortir qu'une seule couleur sur l'image originale, couleur choisie par l'utilisateur. Cette fonction est privée et a pour paramètres l'entier représentant la teinte à conserver et l'entier représentant l'intervalle de tolérance à appliquer. Elle retourne l'image de type **MaBitmap** en niveaux de gris sauf pour les zones correspondant à la teinte plus ou moins le seuil indiqué.
Description de la méthode : Pour chaque pixel, la fonction vérifie si la teinte est dans l'intervalle qu'on demande : si oui, la couleur est conservée, sinon elle devient grise. Le code est volontairement redondant avec ses trois boucles similaires car cela évite d'avoir à faire des calculs plus compliqués à chaque pixel.

PHOTOSHIP : LA VERSION FINALE

La fonction *changeTeinte()* est une fonction permettant de remplacer la couleur de tous les pixels par la teinte choisie par l'utilisateur. Un entier représentant la teinte est passée en paramètres et la fonction retourne une image de type **MaBitmap** avec ce filtre coloré uni.
Description de la méthode : Grâce à l'utilisation de l'espace HSV, on modifie chaque pixel en remplaçant la teinte par celle donnée en paramètres. Ainsi, ni la saturation ni la luminosité ne sont affectées par ce changement.

La fonction *miseAEchelle()* permet de conserver les proportions de la taille de l'image quand l'utilisateur souhaite l'agrandir ou la réduite. Le float représentant le facteur de modification est passé en paramètres et la fonction retourne l'image de taille modifiée de type **Bitmap**.
Description de la méthode : Les nouvelles hauteur et largeur sont calculées à partir du facteur donné, puis une nouvelle image est créée à partir de ces nouvelles valeurs.

La fonction *changeHSV()* permet de modifier de manière indépendante la teinte, la saturation et la luminosité de l'image à l'aide de seekbars. Les paramètres d'entrée sont les nouvelles valeurs de teinte, saturation et luminosité à remplacer et la fonction retourne l'image modifiée de type **MaBitmap**.
Description de la méthode : La méthode additionne pour chaque pixel ses valeurs dans l'espace HSV avec celles données en paramètres. Si les résultats des sommes se trouvent en dehors du bon espace, ils sont alors ajustés de façon circulaire.

La fonction *rogner()* permet de créer une image rognée par l'utilisateur à partir de l'image originale. Elle contient en paramètres la zone à conserver et retourne l'image rognée de type **MaBitmap**.
Description de la méthode : À partir de la zone à conserver, la fonction calcule la nouvelle taille de l'image et recrée une image à partir de la zone à conserver (qui est le nouveau tableau de pixels) et de la nouvelle taille.

La fonction *dessinCrayon()* sert à donner un effet crayonné à l'image à transformer. Il n'y a aucun paramètre d'entrée et la fonction retournée l'image modifiée de type **MaBitmap**.
Description de la méthode : Pour cette fonction, on doit appliquer trois filtres successifs : le filtre de niveaux de gris, le filtre négatif et le filtre gaussien. Enfin, on retourne le résultat de la fonction *mélangeDensiteCouleur()* (fonction dont le sens et l'utilité nous a en partie échappé et qu'on ne détaillera pas dans ce rapport car trouvée sur un forum de développement).

La fonction *voisinLePlusProche()* est une fonction qui peut modifier la taille de l'image. Elle peut également la déformer car l'utilisateur peut choisir de ne pas conserver les proportions. Les paramètres d'entrée sont deux entiers servant à calculer les facteurs de déformation, et la fonction retourne l'image de type **MaBitmap** déformée.
Description de la méthode : À partir des deux entiers passés en paramètres, on calcule deux float correspondant aux facteurs de déformation (en divisant les entiers passés en paramètres par la largeur ou la hauteur). Puis on place les pixels aux positions équivalents dans un nouveau tableau de pixels. Enfin, on crée la **Bitmap** servant à retourner l'image **MaBitmap** avec l'entier correspondant au filtre utilisé.

La fonction *cartoon()* permet de donner un effet « cartoon » à l'image originale. Il n'y a aucun paramètre d'entrée et la fonction retourne l'image transformée de type **MaBitmap**.
Description de la méthode : Grâce à la méthode *sobel()*, on détecte les bordures dans l'image. On crée alors une grille permettant d'identifier la composition de l'image : -1 correspond à une bordure, 0 à un pixel non visité et un entier positif différent pour chaque zone. Pour chaque pixel à 0, la fonction *trouverAireCouleur()* est appelée et renvoie tous les voisins du pixel tant qu'il existe et qu'il est lui même à 0. Cette fonction récursive sert à marquer tous les pixels d'une même zone avec le même numéro, et calcule ainsi la couleur moyenne de la zone (en additionnant toutes les valeurs et en divisant par le nombre de pixels de la zone). Une fois la couleur de la zone trouvée, on cherche dans notre palette de couleur si elle est proche d'une de nos couleurs utilisées. Si oui, on dit que sa couleur est celle de la couleur proche, sinon on ajoute cette couleur à notre palette. Ainsi, on réduit le nombre de couleurs utilisées si les couleurs des zones sont proches. La fonction *cartoonBorders()* servira ensuite à dessiner les contours selon la valeur donnée en entrée par la seekbar (plus la valeur d'entrée est haute, plus la fonction est tolérante et trace des contours considérés comme moins importants sur le filtre Sobel).

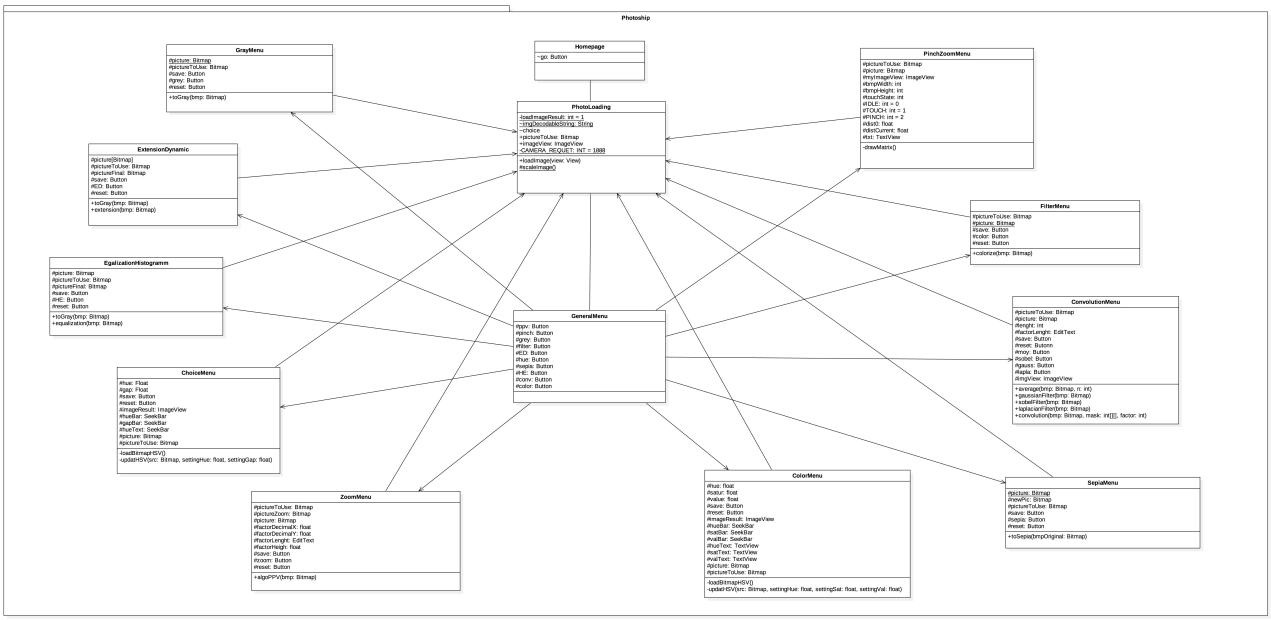
Il nous faut aussi signaler qu'après de nombreuses recherches, nous nous sommes vus dans l'incapacité de pouvoir implémenter la possibilité d'appliquer chaque filtre au doigt. Cela pourra être implémenté prochainement lors d'une pratique personnelle d'Android Studio, c'est pourquoi nous laissons le bouton switch permettant d'activer cette option ainsi que nos traces de recherches dans le code de l'application.

HISTORIQUE DE DÉVELOPPEMENT

1. Rappel de la première release

1.1- Architecture de la première release

L'architecture de la première release du projet est représentée par le diagramme UML suivant :



Elle comprenait autant de fichiers XML que de fichiers JAVA : ainsi 13 activités composaient cette première version de Photoship

1.2- Description du fonctionnement de la première release

Lors de l'ouverture de l'application, l'utilisateur était face à un écran d'accueil permanent où il devait cliquer sur un bouton pour pouvoir passer au menu de chargement de photo. Sur celui-ci, il pouvait accéder à la galerie ou à la caméra de son téléphone pour choisir l'image à transformer, puis valider son choix à l'aide d'un bouton. Un menu entièrement composé de boutons s'affichait alors, et l'utilisateur pouvait accéder au filtre qu'il désirait appliquer en cliquant sur le bouton correspondant. Pour chacune des activités correspondant à un filtre, trois choix lui étaient offerts :

- Appliquer le filtre choisi grâce à un bouton ou à des seekbars qui lui montraient le résultat pendant la manipulation,
- Sauvegarder son image dans la galerie de son téléphone,
- Réinitialiser son image en annulant les modifications effectuées.

En sauvegardant son image, l'utilisateur était directement renvoyé au menu de chargement de photo. En effet, les développeurs laissaient à l'utilisateur la possibilité de gérer les images engendrées par l'application et servant d'historique de modification.

2. Comparaison entre les deux releases

2.1- Comparaison de l'architecture

La différence entre les deux architectures est très nette : de 13 fichiers JAVA et 13 fichiers XML, on passe à 6 fichiers JAVA et 5 fichiers XML.

Ainsi, dans la première release, chaque classe JAVA avait son ficher XML associé et représentait une activité à ouvrir dans l'application, soit 13 activités à ouvrir. Le fonctionnement de cette première version était basée sur l'ouverture d'une activité correspondant au(x) filtre(s) que l'utilisateur souhaitait appliquer. Beaucoup trop d'activités pouvaient être ainsi ouvertes, ce qui rendait le menu principal illisible tellement le nombre de boutons menant à ces activités était important. Dans la deuxième release, il n'y a au total que 4 activités à ouvrir : le logo de l'université, la page d'accueil, la page de chargement de photo et le menu principal, ce qui rend la tâche beaucoup plus facile.

Par ailleurs, toutes ces activités distinctes ne permettaient pas à l'utilisateur d'appliquer plusieurs filtres à une même image avant de la sauvegarder dans sa galerie. Dorénavant, l'utilisateur peut accéder à n'importe quel filtre sans avoir à lancer une nouvelle activité.

2.2- Comparaison des algorithmes

Certaines méthodes comme *cartoon()* n'existaient pas lors du dépôt de la première release : elles ne peuvent donc pas être comparées à une nouvelle version et n'ont pas subi d'amélioration en terme d'optimisation.

Toutes les méthodes de traitement d'image nécessitant le parcours de tous les pixels de l'image utilisent le tableaux **pixels[]** à la place de **pixel[]**.

Tous les boutons permettant d'appliquer des filtres définis ont été remplacés par des boutons-images, permettant à l'utilisateur d'avoir un aperçu des modifications qu'il peut appliquer à l'image choisie.

Le plus gros changement d'un point de vue algorithmique concerne les filtres Laplacien et Sobel. Tous deux utilisaient la même fonction de convolution que les filtres moyenneur et gaussien. Ce n'est plus le cas dans cette seconde version où ils possèdent leur propre fonction de convolution addaptée : *borduresConvol()*.

De plus, les filtres Sobel et Laplacien ont eux-même été améliorés. Alors que dans la première release, toutes valeurs inférieures à 0 étaient ramenées à 0 et toutes valeurs supérieures à 255 étaient ramenées à 255. Maintenant l'intervalle possible est compris entre -8*255 à 8*255.

Les méthodes egalisationHistogramme() et extensionDynamique() sont maintenant à même de traiter des images en couleur, contrairement à la première release où elles étaient contraintes aux images en niveaux de gris.

Notons, pour terminer, que les fonctions *changerTeinte()*, *changerHSV()* et *selectionTeinte()* n'ont pas subi de modification, car elles nous semblaient déjà optimales lors de la première release.

3- Gain de l'optimisation réalisée

L'amélioration de l'architecture et l'optimisation des algorithmes ont permis des gains certains en temps et en mémoire. Mais ce ne sont pas les gains les plus notables pour cette seconde release.

En effet, le gain le plus remarquable est celui concernant la lisibilité de l'application. Lors de la première release, les relations entre la partie interface et traitement d'images n'étaient pas toujours très bien comprises, ce qui donnait un développement de l'application « par tâtonnements ». Cela explique entre autre la « désorganisation » dont faisait preuve le menu principal de l'application. Les boutons y étaient disposés de façon chronologique et suivant la place restante.

Dans cette seconde release, la séparation entre l'interface et le traitement d'images est lisible immédiatement : l'interface se compose de toutes les classes associées à un ficher XML, le reste compose le traitement d'images.

CONCLUSION

L’objectif du projet était d’implémenter une application de traitements d’images. Ainsi, notre application de traitements d’images Photoship se compose :
- d’une activité (composée d’un fichier JAVA et d’un fichier XML) projetant le logo de l’université,
- d’une activité (composée d’un fichier JAVA et d’un fichier XML) projetant le logo d l’application,
- d’une activité (composée d’un fichier JAVA et d’un fichier XML) permettant à un utilisateur de choisir une image à partir de sa galerie de photos ou de la caméra de son appareil,
- d’une activité (composée d’un fichier JAVA et d’un fichier XML) représentant le menu principal de l’application et permettant d’y appliquer n’importe quel filtre,
- d’une classe JAVA implémentant un historique de modifications grâce à une liste d’images,
- d’une classe JAVA implémentant notre propre version d’une image,
- d’un fichier XML décrivant les icônes contenues dans la barre de l’application.

Malgré nos recherches et nos tâtonnements, nous ne sommes pas parvenus à développer l’option d’application des filtres avec le doigt.

Toutes les méthodes obligatoires ont été implémentées et fonctionnent dans l’application. Nous avions choisi cinq fonctions supplémentaires : simuler un effet dessin au crayon, simuler un effet cartoon, restreindre la zone d’application d’un filtre avec le doigt, changer l’orientation de l’image et rogner l’image. Sur ces cinq fonctions, quatre ont été développées avec succès. Seule l’option pour restreindre la zone d’application d’un filtre avec le doigt n’a pu aboutir. En revanche, nous avons proposé dans l’application d’autres fonctions comme simuler deux effets supplémentaires dessin au crayon plus ou moins marqués que le premier, appliquer un filtre négatif sur l’image ainsi qu’appliquer un effet miroir horizontalement et verticalement.

Ce projet était très instructif à plusieurs titres. Il nous a permis d’apprendre à développer une application, ce que nous ne savions pas faire avant cette réalisation. Nous avons découvert les outils liés au traitement d’images comme les Bitmaps, les Seekbars et la gestion des activités entre les classes JAVA et les fichiers XML.

La recherche de solutions aux erreurs de compilation et d’exécution nous a permis de développer des qualités telles que l’endurance et la persévérance dans le travail.

Ce projet nous a finalement donné le goût du développement d’applications Android.

AUTEURS ET MATÉRIEL UTILISÉ

1. Les auteurs de l'application

Les trois auteurs de ce projet sont trois étudiantes en L3 Mathématiques-Informatique.

Vous pouvez les joindre via leurs profils Github ou par les adresses mail suivantes :

- Cauli HONORÉ
quentin.honore@etu.u-bordeaux.fr **https://github.com/cauli33**

- Emma VEILLAT :
emma.veillat@u-bordeaux.fr **https://github.com/EmmaVeillat**

- Emma BESSE :
emma.besse@etu.u-bordeaux.fr

2. Le matériel utilisé tout au long du projet

Cette application a été développée grâce au logiciel Android Studio. Elle a également pu être testée grâce aux machines suivantes :
- portable Motorola G2
- portable Samsung S7
- émulateur Asus Zenfone 2
- émulateur Nexus S
- émulateur proposé par défaut par Android Studio.

L'API minimal utilisé pour tester l'application est l'API 22. Mais l'application fonctionne très bien pour toute API supérieure à 3.

La rédaction du cahier des charges a été réalisée en langage LaTex, grâce au logiciel Emacs.

Les diagrammes présentés dans ce rapport ont été réalisés en langage UML, avec l'aide du logiciel StarUML.

Certains sites ont été particulièrement utiles pour le développement de Photoship notamment :
- le forum **http://stackoverflow.com/**
- le site de cours en ligne **https://openclassrooms.com/**