Application Android Studio
Traitement d'Images

Projet réalisé par:

CHOUROUQ Sarah

L3 Informatique
Bordeaux

Introduction

Android est un système d'exploitation tournant sur différents types d'appareils : téléphones, tablettes voire même téléviseurs. Les applications Android sont développées en Java et l'IDE de prédilection est Android Studio.

Le but de ce projet était de se familiariser avec l'environnement et d'appliquer les méthodes apprises en cours pour la réalisation de divers traitements d'image. Le téléphone utilisé tout le long de ce projet était le Galaxy Nexus (4.65 720x1280 xhdpi).

Table des matières

- 1 -
Algorithmes implémentés

Voici tous les algorithmes complètement implémentés des TP 1 et 2.

A - Ligne noire

Afin de me familiariser davantage avec `Android-Studio`, il m'a été demandé de tracer une ligne de pixels noire dans le but de tester la modification de la valeur d'un pixel. Il me suffisait simplement de diviser la largeur de l'image par deux puis de récupérer chacun des pixels situés sur cette ligne. Une colorisation en noir et le tour était joué !

B - Taille de l'image

Pour afficher la taille de l'image, il s'agissait ici de récupérer la taille de la Bitmap au lieu de l'image en elle-même. En effet, sous `Android-Studio`, il n'est pas possible de travailler directement sur l'image originale mais plutôt sur une copie parfaite. Ainsi, afin de récupérer puis d'afficher la taille de l'image:

```
// Récupération de la hauteur et de la largeur
int width = options.outWidth;
int height = options.outHeight;

//Affichage de la taille grâce à un TextView
TextView tv = findViewById(R.id.txtHello);
tv.setText("w = " + width + " h = " + height);
```

C - Noir et blanc

La partie la plus intéressante du traitement d'image était de rendre en noir et blanc une image à l'origine en couleur. En premier lieu, j'ai décidé de tester la performance de deux versions de la méthode `toGray`: les méthodess `getPixel` et `setPixel` et les méthodes `getPixels` et `setPixels` de la classe Bitmap.

La première version consistait à parcourir chaque pixel de la Bitmap, de récupérer le taux de rouge, de vert et de bleu de chacun d'entre eux, puis d'appliquer la modification à chaque couleur. Comme ci-dessous:

```
public void toGray (Bitmap bm) {
        for (int x = 0; x < bm.getWidth(); x++) {
            for (int y = 0; y < bm.getHeight(); y++) {
                int RED = Color.red(bm.getPixel(x,y));
                int GREEN = Color.green(bm.getPixel(x,y));
                int BLUE = Color.blue(bm.getPixel(x,y));
                int pxColor = (int) (0.3 * RED + 0.59 * GREEN + 0.11 * BLUE);
                bm.setPixel(x,y,Color.rgb(pxColor, pxColor, pxColor));
            }
        }
    }
```

Cependant, cette version n'est clairement pas optimisée et n'est pas assez performante puisqu'elle prend beaucoup de temps à parcourir les pixels un à un.

Parcontre, une manière plus améliorée consistait à utiliser directement un tableau de pixels. Premièrement, il me fallait créer un tableau de pixels de taille identique à celle de la Bitmap, puis d'ajouter à ce tableau les chaque pixel de l'image. Il suffisait juste d'appliquer le filtre noir et blanc et l'image se grisait parfaitement.

```
public void toGray(Bitmap bm) {
        int size = bm.getWidth() * bm.getHeight();
        int[] pixels = new int[size];
        bm.getPixels(pixels, 0,bm.getWidth(), 0 , 0, bm.getWidth(),
bm.getHeight());
        for (int i = 0; i < size; i++) {
            int RED = Color.red(pixels[i]);
            int GREEN = Color.green(pixels[i]);
            int BLUE = Color.blue(pixels[i]);

            int toGray = (int) (0.3 * RED + 0.59 * GREEN + 0.11 * BLUE);
            pixels[i] = Color.rgb(toGray, toGray, toGray);

        }
        bm.setPixels(pixels, 0, bm.getWidth() , 0 , 0, bm.getWidth(),
bm.getHeight());
    }
```

Cette dernière technique est beaucoup plus efficace et performante, elle prend moins de temps à compiler. En effet, btenir des données de pixel individuelles dû à la méthode `getPixel` est très cher en temps pour le CPU. `getPixel` doit rechercher la taille du tampon graphique, calculer l'encodage, déterminer où se trouvent les données dans la bitmap, et utiliser cela pour retourner la valeur. `getPixels` le fait en masse, et donne ainsi un ordre de grandeur plus rapide.

●●●

D - Filtre coloré

J'ai eu un peu plus de mal lorsqu'il s'agissait d'exploiter la colorisation. Il y avait diverses méthodes mises à dispositions mais il m'était impossible de choisir laquelle était la mieux. J'ai dû demandé de l'aide à mon professeur pour connaître davantage l'utilisation de la méthode `RGBToHSV`. Après maintes essais, j'ai finalement compris l'utilisation de cette dernière et ai pu l'implémenter de manière à ce qu'un filtre coloré s'applique sur mon image. D'abord convertir les couleurs `RGB` en `hsv` (= Teinte, saturation, luminosité), puis convertir le hsv en couleur à appliquer sur tous les pixels de l'image.

```
[...]

Color.RGBToHSV(RED, GREEN, BLUE, hsv);

[...]

int newColor = Color.HSVToColor(hsv);
pixels[i] = newColor;
```

Pour l'instant, le choix de la couleur du filtre coloré n'est pas modifiable par l'utilisateur. La valeur est donnée par défaut, mais on peut la changer à tout moment. Ici, par exemple, je l'ai laissé à 25, ce qui donne à l'image un teint légèrement orangeâtre.

```
hsv[0] = 25;
```

●●●

E - Une couleur

Sur une image colorée, il fallait garder uniquement une seule couleur et griser toutes les autres. Il était encore une fois nécessaire de re-convertir les valeurs `RGB` en `hsv`.

J'ai posé une condition d'intervalle pour toutes les valeurs de pixels qui ne sont pas celles de la couleur choisie, puis je les ai grisé.

```
if (hue - 10 >= hsv[0] || hsv[0] >= hue + 10) {
    int pxColor = (int) (0.3 * RED + 0.59 * GREEN + 0.11 * BLUE);
```

- 2 -
Outils et Performances

——— (五) ———

A - Outils utilisés

L'implémentation des divers algorithmes de traitement d'images a pu se réaliser grâce au très complet IDE nommé `Android-Studio` sous le langage `Java`. Un fichier `.xml` était nécessaire afin de gérer l'emplacement des divers outils tels que les boutons, les images ou bien les zones de textes. Par exemple, voici un morceau du code xml pour la création du bouton `Grayscale`:

```
<Button
   android:id="@+id/btnGrayscale"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_marginTop="16dp"
   android:text="Grayscale"
   app:layout_constraintEnd_toStartOf="@+id/btnColorize"
   app:layout_constraintStart_toStartOf="parent"
   app:layout_constraintTop_toTopOf="parent" />
```

Il fallait impérativement donner un identifiant - ici id - au bouton crée. Les parties layout - width height et marginTop - servent à positionner dans l'espace le bouton. Biensûr, sans les contraintes, le bouton changerait de place si un autre élément devait être ajouté. Pour palier à ce problème, il était impératif d'ajouter des contraintes par rapport à chacun des éléments présents sur la surface.

Diverses images de tailles et de couleurs différentes se trouvant dans le dossier `res/drawable` ont été utilisées pour ce projet. A l'origine, je me servais d'une image assez sombre puis j'ai dérivé vers une image nettement plus colorée pour exploiter au maximum les capacités du traitement d'images sous Android.

B - Petite image

Voici les résultats pour une image de taille **103*69**:

- **Niveau de gris:**

Vitesse d'exécution observé : *Instantané*

- **Filtre de couleur:**

Vitesse d'exécution observé : *Instantané*

- **Une couleur:**

Vitesse d'exécution observé : *Instantané*

C - Image moyenne

Voici les résultats pour une image de taille **458*458**:

- **Niveau de gris:**

Vitesse d'exécution observé : *Instantané*

- **Filtre de couleur:**

Vitesse d'exécution observé : *Latence d'environ 1,5 seconde*

C - Grande image

Voici les résultats pour une image de taille **2500*2500**:

- **Niveau de gris:**

Vitesse d'exécution observé : *Latence d'environ 0.5 seconde*

- **Filtre de couleur:**

Vitesse d'exécution observé : *Latence de plus de 15 secondes!*

J'ai délibérément afficher que certains des algorithmes sur chaque image puisque certains mettaient trop de temps à fonctionner et il fallait attendre plus de 20 secondes (voir vraiment plus) pour un résultat. Visible avec le filtre sur la très grande image, le temps d'exécution était loin d'être instantané. Plus l'image est grande, moins l'exécution est performante.

- 3 -
Bugs et limites
Conclusion

Je n'ai malheureusement pas pu aller plus loin dans ce projet. J'avais prévu de proposer une interface plus ergonomique, où l'utilisateur pourrait réinitialiser autant de fois qu'il veut les modifications apportées à l'image. Je souhaitais ajouter en amont davantages de boutons qui serviraient à donner une meilleure utilisation à l'utilisateur.

Cependant, j'ai quand même tenté d'implémenter un histogramme mais je n'arrivais pas à le faire apparaître correctement sans avoir d'erreurs.

Je n'ai pas pu suivre les cours de Renderscript et lorsque j'ai essayé d'apprendre ce langage chez moi, il m'était impossible de comprendre. Je suis donc restée en Java afin d'implémenter les méthodes que j'étais capable de faire.

En outre, j'aurais voulu proposer une application plus complète, plus conviviale et surtout plus ergonomique, mais suite à des problèmes assez personnels, je n'ai pas pu continuer le reste des TPs malgré mon implication.