Rapport de Projet de Programmation Impérative

Julien EYRAUD

5 janvier 2025

Manipulation d'images en C

- Encadré par Guillaume BUREL -

Table des matières

[1]

1. Les parties de code contenus dans ce document ne sont que des extraits ; les fonctions ont été tronquées (...) pour mettre en exguergue certaines parties de celles-ci.

1 Introduction

Le projet consiste à implémenter des fonctions de **manipulation et filtrage d'images** en C. Ces images sont au format **PPM** ou **PGM**. Il s'agit d'images codées en données binaires.

Les principaux objectifs du projet sont :

— **Comprendre** la structure d'image PPM/PGM et apprendre à manipuler les images en mémoire.

— **Implémenter** des fonctions pour manipuler les données d'une image .

— **Optimiser** ces implémentations en utilisant une LUT (Look Up Table).

2 Structure d'une Image

Une image numérique est un tableau de pixels. Chaque pixel contient des informations de couleur ou de luminosité dont les valeurs sont comprises dans l'intervalle [0,255]. Dans ce projet, une image est représentée par la structure suivante en langage C :

```c
typedef unsigned char byte;

typedef struct {
    unsigned int width;
    unsigned int height;
    unsigned int channels;
    byte *content;
} Picture;
```

— `byte` est un alias pour `unsigned char`, utilisé pour représenter un octet, soit une valeur entre 0 et 255.

— Le tableau unidimensionnel `content` stocke les valeurs des pixels en mémoire.

— L'image dispose d'une largeur `width` et d'une hauteur `height`.

— `channels` représente le nombre de canaux d'un pixel dans l'image. 1 pour les images en niveaux de gris ; 3 pour les images RVB.

3 Description des Images

Il existe 2 types d'images : les **images en niveau de gris** (représentées par des valeurs de luminosité) et les **images en couleur dites RVB** (représentées par des valeurs de Rouge, Vert et Bleu).

- Un pixel d'une image en niveau de gris est par exemple : *128* .
  0 : noir, *128* : gris moyen, *255* : blanc *etc...*
- Un pixel d'une image RVB est par exemple : *120 56 204* .
  où *120* est l'intensité du rouge, *56* celle du vert et *204* celle du bleu.
  0 0 0 : noir, *255 255 255* : blanc *etc...*

3.1 Particularités des formats PPM/PGM

Une image *.ppm (couleur) ou *.pgm (gris) est décrite par deux parties : l'en-tête (texte) et les données - pixels - de l'image (informations binaires). Par exemple, le fichier .ppm d'une image RVB est comme suit :

P6
// des lignes de commentaires
512 512
// d'autres lignes de commentaires
255
Partie binaire

- P6 (P5 pour les images en niveaux de gris) : format de l'image.
- 512 512 : largeur puis hauteur de l'image (en pixels).
- 255 : valeur maximale pour les pixels.

4 Lecture et écriture des fichiers images

Ouvrons un fichier image dont le chemin est `filename` et créeons une structure `Picture` permettant de manipuler cette image.

Tout d'abord il faut lire l'en-tête de ce fichier image avant de lire ses données binaires qui contiennent les valeurs des pixels. Aussi il faut sauter les commentaires qu'il peut y avoir entre les données de l'en-tête.

Ce processus est implémenté dans la fonction suivante :

```
1 Picture read_picture(char* filename){...}
```

Après de potentielles modifications sur les structures `Picture` par les différentes fonctions implémentées (filtres...), il faut écrire les images (de structure `Picture` à véritable fichier *.pgm | *.ppm).

Pour cela nous implémentons la fonction suivante :

```
1 int write_picture(Picture p, char* filename){...}
```

4.1 Lecture de l'en-tête et des données binaires

Nous ouvrons l'image avec la fonction `fopen()` en mode "rb" (lecture des données binaires) et nous utilisons les fonctions `fscanf()` pour récupérer les données de l'en-tête ainsi que `fread()` pour décrypter la partie binaire.

```
Picture read_picture(char* filename){
    ...
    FILE* fptr = fopen(filename, "rb");
    ...
    // Lire l'en-t te du fichier
    char magicNum[3];
    unsigned int width, height, channels, max_value;

    //On acc de aux valeurs en ignorant les commentaires
    fscanf(fptr,"%s\n", magicNum);
    //on va    la nouvelle ligne sans commentaires
    checkforhashtags(fptr);

    fscanf(fptr,"%d %d\n", &width, &height);
    checkforhashtags(fptr);

    fscanf(fptr,"%d\n", &max_value);
    ...
}
```

Notons que nous avons implémenté une fonction `void checkforhashtags(FILE*)` afin d'ignorer les commentaires dans l'en-tête.

```
void checkforhashtags(FILE* fptr){
    if (fptr == NULL) {
        fprintf(stderr, "Erreur lors de l'ouverture du
            fichier\n");
        return; }
    int c;
    while ((c = fgetc(fptr)) == '#') {
        // Ligne de commentaire d tect e ; Ignorer
            jusqu'   la fin de ligne
        while ((c = fgetc(fptr)) != '\n' && c != EOF);
    }
    if (c != EOF) {
        // Replacer le caract re non comment  dans le flux
        ungetc(c, fptr);
    }
}
```

4.2 Écriture des fichiers images

Il s'agit de convertir une structure `Picture` en un fichier *.pgm (gris) ou *.ppm (couleur).

Pour ce faire nous créons un nouveau fichier en mode "wb" (écriture de données binaires) dans lequel nous écrivons d'abord l'en tête avec les méta-données de l'image en utilisant la fonction `sprintf()` puis la partie binaire de l'image (les pixels de l'image) avec la fonction `fwrite()`.

```
int write_picture(Picture p, char* filename){
    ...
    FILE* fptr = fopen(filename, "wb");
    ...
    // par d faut image RGB (.ppm)
    char format[3] = {'P','6','\0'};
    if (p.channels == 1) { format[1] = '5'; }

    char header[64];
    // stocke l'en t te dans header
    int header_length = sprintf(header, "%s\n%d %d\n255\n",
       format, p.width, p.height);
    ...
    //ecriture de l'en t te
    if (fwrite(header, sizeof(char), header_length, fptr) !=
       (size_t)header_length){
        fprintf(stderr, "Erreur lors de l' criture de
           l'en-t te\n");
        fclose(fptr);
        return 6;
    }
    ...
}
```

5 Manipulation des pixels d'une image

Afin de modifier une image comme demandé dans l'énoncé, il faut accéder aux valeurs de ses pixels (gris ou couleur).

Pour procéder ainsi, deux manières nous sont proposées :
- de manière **directe**
- à l'aide d'une **LUT** (Look Up Table).

Rappel de la structure `Picture` utilisée pour représenter chaque image :

```
typedef unsigned char byte;

typedef struct {
    unsigned int width;
    unsigned int height;
    unsigned int channels;
    byte *content;
} Picture;
```

Les pixels d'une image sont stockés dans le tableau `byte* content`. Pour accéder aux valeurs, on boucle sur la hauteur et la largeur de l'image.

On utilise dans l'ensemble des fonctions implémentées, et pour nous simplifier la tâche, des indices `index_color` et `index_gray` qui nous permettent d'accéder directement aux valeurs des pixels (gris ou couleurs) dans le tableau `content` à chaque itération :

```
Picture convert_to_color_picture(Picture p){
    ...
    for (unsigned int i = 0; i < p.height; i++){
        for (unsigned int j = 0; j < p.width; j++){

            int index_color = (i * p.width + j)*3;
            int index_gray = i * p.width + j;
            ...

        }
        ...
    }
    ...
}
...
```

5.1 Manipulation directe

Prenons l'exemple de la fonction suivante, implémentée d'abord de manière directe :

```
Picture brighten_picture(Picture p, double factor){
    ...
    Picture brightened_p = create_picture(p.width, p.height,
       p.channels);
    if (is_gray(p)){
        for (unsigned int i = 0; i < p.height; i++){
            for (unsigned int j = 0; j < p.width; j++){

                    int index_gray = (i * p.width + j);

                    byte gray = p.content[index_gray];
                    int val = (int)round(gray*factor);

                    val = arrange_pixel_value(val);

                    //modification directe
                    brightened_p.content[index_gray] =
                       (byte)val;

            }
        }
    }
    ...
    return brightened_p;
}
```

Les valeurs des pixels sont modifiées de manière directe. Cette implémentation est quelque peu laborieuse.

Nous allons essayer de l'améliorer dans la sous-section suivante.

5.2 Manipulation à l'aide d'une LUT

Une LUT associe des valeurs d'entrée à des valeurs de sortie déjà calculées.Cela évite d'effectuer des opérations à chaque fois, on stocke les résultats dans un tableau et on effectue une simple recherche dans ce même tableau.

On implémente une LUT avec la structure `Lut` :

```
typedef struct {
    unsigned int n;
    byte* tab;
} Lut;
```

— `n` : taille du tableau `tab`
— `tab` : tableau unidimensionnel stockant les valeurs de la LUT

Une `Lut` est appliquée aux données d'une image avec la fonction `apply_lut` :

```
Picture apply_lut(Lut lut, Picture p){
    ...
    unsigned int size = p.width*p.height*p.channels;
    Picture lut_p = create_picture(p.width, p.height,
       p.channels);

    for (unsigned int i = 0; i < size; i++){
        lut_p.content[i] = lut.tab[p.content[i]];
    }
    return lut_p;
}
```

Avec le même exemple qu'à la sous-section 5.1, l'usage d'une structure de `Lut` nous permet de rendre notre code plus efficace et plus concis.

```
Picture brighten_picture_lut(Picture p, double factor){
    ...
    Lut brighten_lut = create_lut(256);

    for (unsigned int i = 0; i < brighten_lut.n; i++) {
        int val = (int)round(i * factor);
        val = arrange_pixel_value(val);
        brighten_lut.tab[i] = (byte)val;
    }

    Picture brightened_p = apply_lut(brighten_lut, p);
    clean_lut(brighten_lut);
    return brightened_p;
}
```

6 Problèmes et choix personnel : arrondi entier des valeurs

Je n'ai rencontré qu'un réel problème : lors de la manipulation des valeurs des pixels, j'ai dû appliquer par exemple des formules mathématiques sur ces valeurs modifiant ainsi leur type de byte à double. Comme dans cette formule calculant la nuance de gris à partir des valeurs RVB d'un pixel :

$$G = 0.299 * R + 0.587 * V + 0.114 * B$$

Afin de respecter mon choix qui est d'utiliser des `byte` *i.e* des entiers compris dans l'intervalle [0,255] pour représenter un pixel ; j'ai donc dû, dans certaines des fonctions implémentées, utiliser les méthodes suivantes :

- cast explicite `(byte)`
- la fonction `round()` issue de la bibliothèque *math.h* pour faire les arrondis aux entiers des valeurs traitées.
- la fonction `int arrange_pixel_value(int)` pour m'assurer que les valeurs de pixels soient comprises entre 0 et 255.

```
int arrange_pixel_value(int pixel_value){
    if (pixel_value < 0){
        return 0;
    }
    else if (pixel_value > 255){
        return 255;
    }
    else{
        return pixel_value;
    }
}
```

7 Conclusion

Ce projet m'a permis de découvrir les principes fondamentaux du traitement d'images en C, en me concentrant sur la manipulation des formats PPM et PGM.
J'ai codé des fonctions permettant de lire, modifier et écrire des images, en manipulant directement les valeurs des pixels via des structures de données adaptées.

Le choix du type byte pour les pixels et l'utilisation de techniques comme l'arrondi des valeurs ont clarifié la gestion des données des images.
Aussi l'utilisation de LUT a grandement facilité la manipulation des pixels.

Bien que ce projet ait été un peu long, je l'ai trouvé complet et intéressant.