RAPPORT DE PROJET

PROJET PRIM

Étudiant : Arnaud GRASSIAN
Formation : ENSIIE
Date : Janvier 2025

École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise

Contents

Introduction

Ce projet a pour objectif de traiter, manipuler et transformer des images numériques (`.pgm` et `.ppm`). Cela consiste principalement à établir des fonctionnalités comme des opérations sur les pixels, des transformations géométriques, et des ajustements des niveaux de gris ou de couleur à l'aide de LUTs (*Look-Up Tables*).

Le projet est conçu pour être modulaire, avec des modules dédiés à la gestion des images, aux LUTs, et d'autres opérations spécifiques.

Organisation du Projet

Le projet est structuré en plusieurs modules pour assurer la modularité et la clarté du code. Chaque module est accompagné d'une interface, par exemple `pictures.h` (où les fonctions sont déclarées) et `pictures.c` (où elles sont implémentées).

0.1. Pictures

- Gère la création, la copie, la lecture, l'écriture et la suppression des images.
- Contient des fonctions de conversion, de transformation et également de re-échantillonnage.

0.2. LUT

- Implémente les *Look-Up Tables* (fonctions de transfert) pour transformer les valeurs de pixels.
- Permet des opérations comme l'inversion, la normalisation dynamique et la réduction des niveaux.

0.3. Filename

- Simplifie la gestion des noms de fichiers, en particulier pour générer automatiquement des noms d'images transformées.

Remarque : J'ai décidé de ne pas utiliser de module `pixel` dans ce projet puisqu'en effet, les indices `0`, `1` et `2` remplacent les constantes `RED`, `GREEN`, et `BLUE` et le code reste compréhensible et lisible avec cette notation.

Fonctionnement

0.4. Pictures

Dans `pictures.h`, j'ai commencé par définir un type `byte` pour contenir les valeurs codées par un octet. (J'ai choisi un `unsigned char` car celui-ci représente des valeurs dans la plage $[0, 255]$, ce qui est idéal ici).

```
typedef unsigned char byte;
```

J'ai défini ensuite la constante `MAX_BYTE` pour contenir la valeur maximale des octets :

```
#define MAX_BYTE 255
```

Ensuite, j'ai utilisé la structure suivante pour définir une « picture » :

```
typedef struct {
    int height; /* hauteur */
    int width;  /* largeur */
    int can;    /* nombre de canaux de l'image */
    byte* data; /* pointeur vers les données pixels de l'image */
} picture;
```

La suite du fichier `pictures.h` n'est pas intéressante à commenter puisqu'il s'agit uniquement des déclarations de fonctions.

La première fonction qui intervient dans `pictures.c` est d'ailleurs la principale :

`read_file_picture`. Elle permet, à partir d'un fichier donné en argument, de retourner l'image correspondante, au sens du type `picture` défini plus tôt.

Il est nécessaire de rappeler la structure des fichiers binaires PPM/PGM :

```
P6
512 512
255
...
```

La première ligne contient un « magic number » correspondant au type de fichier (ici `P6` identifie un fichier PPM binaire contenant une image couleur, alors que ce serait `P5` pour les fichiers PGM binaires contenant des images en niveaux de gris). La seconde ligne contient les dimensions de l'image : `512 512` correspondent ici à la largeur et la hauteur de l'image respectivement. La troisième ligne contient la valeur maximale des pixels à lire dans ce qui suit : ici `255`.

Après avoir correctement ouvert le fichier avec `fopen`, on déclare une variable `picture`, ainsi que ses éléments (`height`, `width`, un tableau de `bytes`...). On choisit de stocker ligne par ligne les informations qui nous sont utiles dans un buffer :

```
char buffer[128];
```

Il est nécessaire ensuite d'ignorer les potentiels commentaires (qui commencent par un « # »), d'où l'utilisation de la boucle suivante :

```
while(buffer[0]=='#'){
    fgets(buffer,128,f);
}
```

On analyse ensuite le contenu du buffer pour obtenir le type d'image, avec :

```
if(strcmp(buffer, "P5\n")==0)
```

Puis, on stocke les informations qui nous intéressent avec `sscanf` :

```
sscanf(buffer, "%d %d", &width, &height);
```

Je n'oublie pas de relever les erreurs potentielles lors de l'ouverture ou de la lecture des données traitées, avec des codes d'erreurs cohérents entre eux (identiques pour les mêmes erreurs).

La fonction `write_file_picture` est moins complexe, bien qu'il soit toujours nécessaire de faire attention aux potentielles erreurs lors de l'ouverture des fichiers, et de comparer le nombre de données écrites à la taille de l'image :

```
size_t data_size = p.width * p.height * p.can;
if (fwrite(p.data, sizeof(byte), data_size, f) != data_size) {
    perror("Données incomplètes");
    fclose(f);
    return -1;
}
```

Les fonctions `create_picture`, `clean_picture`, `copy_picture` ne nécessitent pas d'être particulièrement commentées, mise à part la gestion d'erreur lors de l'allocation de mémoire avec `malloc` :

```
if (image.data == NULL) {
    error("Erreur allocation de mémoire");
    exit(5);
}
```

C'est dans la boucle de la fonction `convert_to_color_picture` (et les fonctions qui suivent) que le module `pixel` aurait été utile pour gagner en lisibilité :

```
copy.data[i * 3] = p.data[i];       /* Red */
copy.data[i * 3 + 1] = p.data[i];   /* Green */
copy.data[i * 3 + 2] = p.data[i];   /* Blue */
```

Il faut enfin faire attention dans la fonction `melt_picture` à ne pas sélectionner un pixel qui se trouve sur la première ligne :

```
if (i > 0 && copy.data[ind_above] < copy.data[ind]) {
    copy.data[ind] = copy.data[ind_above];
}
```

Nous arrivons ensuite dans la partie qui utilise les fonctions de transferts (`lut`) avec la fonction `inverse_picture`.

0.5. LUT

Il est tout d'abord nécessaire de définir le type `lut` dans `lut.h` pour pouvoir implémenter dans `lut.c` les fonctions de création, de nettoyage et d'application de `lut`.

```
typedef struct {
    int n;          /* Nombre de niveaux dans la LUT */
    int *values;    /* Tableau des valeurs de la LUT */
} lut;
```

J'ai ensuite décidé d'implémenter les 3 procédures suivantes directement dans `lut.c` :

```
void lut_inverse(lut *l);
void lut_normalize_dynamic(lut *l, int min, int max);
void lut_set_levels(lut *l, byte nb_levels);
```

En effet, ces 3 procédures me seront très utiles pour les fonctions permettant d'inverser ou normaliser les valeurs d'une image, ou bien pour la réduction du nombre de niveaux pour les pixels d'une image.

- **Pour l'inverse**, l'affectation suivante permet d'effectuer ce que l'on souhaite :

```
l->values[i] = 255 - i;
```

- **Pour la normalisation dynamique**, j'ai choisi d'affecter à chaque pixel la valeur suivante :

```
l->values[i] = 255 * (i - min) / (max - min);
```

En effet, on soustrait `min` à la valeur d'origine pour déplacer la plage pour qu'elle commence à 0. Pour se ramener ensuite dans l'intervalle $[0, 1]$, on divise par $(max - min)$. Enfin, pour se ramener à l'intervalle $[0, 255]$, il suffit de multiplier le tout par 255.

- **Pour la discrétisation de l'intervalle en un nombre de niveaux** `nb_levels`, j'ai appliqué la formule suivante à chaque pixel :

```
l->values[i] = (i / pas) * pas + 1;
```

avec `pas` défini comme $\frac{256}{nb\_levels}$.

Explications : `pas` correspond à l'amplitude d'un niveau de la discrétisation, donc $\frac{256}{nb\_levels}$. On « ramène » ensuite chacune des valeurs dans son intervalle inférieur. On obtient ainsi la valeur discrétisée pour chaque pixel.

Il ne reste ensuite qu'à utiliser ces procédures dans les fonctions de `pictures.c`.

Les fonctions de différence, multiplication et mélange d'images s'implémentent en appliquant les formules et en faisant attention de ne pas dépasser `MAX_BYTE`.

```
byte mult = p1.data[i] * p2.data[i];
if (mult > MAX_BYTE) {
    mult = MAX_BYTE;
}
result.data[i] = mult;
```

0.6. Re-échantillonnage d'images

0.6.1 La politique de plus proche voisin

Pour commencer, on calcule le ratio des dimensions de l'image d'entrée et de celle de sortie :

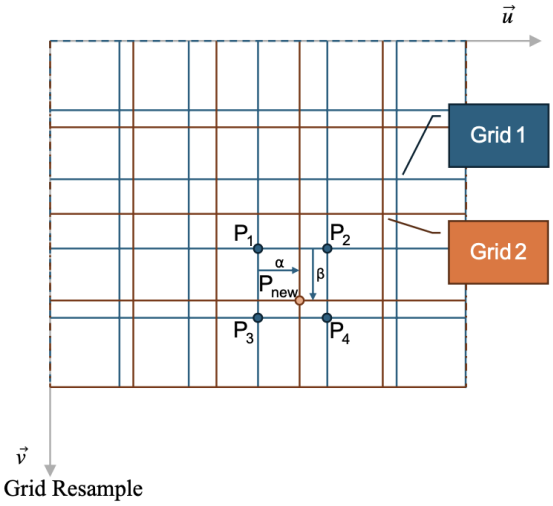
```
float ratio_x = (float)image.width / width;
float ratio_y = (float)image.height / height;
```

Ensuite, on calcule pour chaque pixel de l'image redimensionnée sa position dans l'image d'origine. Ces coordonnées sont ensuite tronquées en entier, correspondant au plus proche voisin :

```
int x_source = x * ratio_x;
int y_source = y * ratio_y;
```

0.6.2 Une politique d'interpolation bi-linéaire

On commence de la même manière que pour la politique du plus proche voisin, mais ici on calcule les coordonnées des pixels adjacents sur l'image d'origine :



On applique ensuite la formule, en ayant calculé les coefficients d'interpolation α et β :

$$P_{new} = (1-\alpha)(1-\beta)P_1 + \alpha(1-\beta)P_2 + (1-\alpha)\beta P_3 + \alpha\beta P_4$$

Cette formule permet d'interpoler la valeur du pixel redimensionné en fonction de ses quatre voisins les plus proches sur l'image d'origine.

Limites du programme et axes d'amélioration

Tout d'abord, je pense que la gestion des erreurs aurait pu être mieux structurée, en utilisant pourquoi pas des explications plus détaillées pour chacune d'entre elles. Ensuite, l'utilisation d'un module `pixel.h/c` aurait aidé mon programme à gagner en modularité (pour pourquoi pas une seconde version).

Enfin, je ne me suis pas intéressé à la complexité des programmes, ni même à la comparaison de ces derniers. Par exemple, je pense qu'il aurait été judicieux de comparer les fonctions `brighten_picture` et `brighten_picture_lut` pour comprendre l'intérêt des LUTs.

Je n'ai pas non plus passé assez de temps sur la dernière question bonus pour obtenir un résultat satisfaisant.

Conclusion

J'ai trouvé le sujet très intéressant, et grâce à lui, j'ai pu revoir toutes les notions abordées pendant le semestre en PRIM, tout en renforçant mes compétences en programmation C et en structuration de projets