Rapport de Projet Programmation Impérative

Ewen Expuesto

2024-2025

Table des matières

1 Contexte et Sujet du Projet

L'objectif de ce projet est d'implémenter divers traitements sur des images en niveaux de gris ou en couleur. Mon programme permet notamment de :

- **Lire** des fichiers images au format Portable Pixmap, spécifiquement le format PPM binaire pour les images en couleur, et PGM binaire pour les images en niveaux de gris.
- **Séparer** et mélanger les composantes d'images.
- **Éclaircir** et fondre des images.
- Utiliser des **Look Up Tables** pour inverser les couleurs d'une image, normaliser une image ou réduire son nombre de niveaux de pixels.
- Réaliser des **opérations arithmétiques** sur les images.
- **Ré-échantillonner** des images.

Ce document suit linéairement le sujet. Chaque titre de section précise le nom du fichier du programme évoqué.

2 Choix Techniques et Problèmes Rencontrés

2.1 Choix des types de variables `pictures.[h/c]`

Pour représenter les pixels, j'ai choisi le type `unsigned char`, permettant de limiter les valeurs des pixels entre 0 et 255.

2.2 Lecture des images `read_write.[h/c]`

J'ai d'abord adapté le programme du cours à la structure de données `picture`. Concertnant le lecture des méta-données, pour déterminer à quel moment il fallait sauter les commentaires et lire la ligne à l'aide de `fgets(buffer, 128, f);`, j'ai utilisé `printf` abondamment. Après cela, les tests de méta-données (format, taille...) fonctionnaient.

J'ai décidé de retourner une image vide si l'image donnée en paramètre est mal lue, après avoir écrit le module qui permet de créer des images vides. J'ai aussi ordonné les tests dans l'ordre qui me semblait le plus logique (voir programme).

J'ai rencontré des difficultés avec la position du curseur : le programme suivant compare la taille prévue dans les méta-données avec la taille effective en parcourant tout le fichier (sauf le header) et comptant le nombre de bytes. Cela déplace le curseur à la fin du fichier. Je ne m'en étais pas rendu compte donc quand je lisais une image, j'obtenais un fichier rempli d'espaces vides. C'est en lisant les fichiers binaires que j'ai compris qu'il fallait placer le curseur au bon endroit en ajoutant `fseek(f, current_position, SEEK_SET);` après le programme suivant :

```
long current_position = ftell(f);
fseek(f, 0, SEEK_END);
long file_size = ftell(f);
long remaining_size = file_size - current_position;
```

2.3 Écriture des images `read_write.[h/c]`

D'abord, j'utilise `fprintf` pour écrire une entrée standard comme `P5` dans un fichier.

Ensuite, c'est en résolvant la suite du sujet que je me suis rendu compte que ma fonction `write_picture` ne fonctionnait pas correctement en testant. Il fallait distinguer les formats PGM et PPM en écrivant non pas sur `img.width * img.height`, mais plutôt sur `img.width * img.height * 3` pour les images couleur. Sinon, le programme s'arrêtait à un tiers du chemin et l'image était noire sur les deux derniers tiers inférieurs. `fwrite(img.data, sizeof(byte), img.width * img.height * img.channels, f);`

2.4 Gestion des images `manage_images.[h/c]`

J'ai rencontré un bug de la même origine que précédemment en tentant de convertir une image couleur en niveaux de gris, car j'avais oublié de différencier, de la même manière que pour `write_picture`, les deux types d'images, mais cette fois-ci dans la fonction `read_picture`. En effet, il faut différencier : `(byte *) malloc(width * height * sizeof(byte));` pour les fichiers PGM, et : `(byte *) malloc(width * height * 3 * sizeof(byte));` pour les fichiers PPM. De même que : `fread(data, sizeof(byte), width * height, f);` et : `fread(data, sizeof(byte), width * height * 3, f);`. Cela causait des erreurs de mémoire en testant avec des images.

De plus, j'ai perdu du temps à devoir bien construire le Makefile à ce niveau, sans l'avoir fait au fur et à mesure car les erreurs se cumulaient sur différentes fonctions.

2.5 Manipulation directe des valeurs des pixels `handle_pixels_directly.[h/c]`

Pour la fonction d'éclaircissement `brighten_picture`, j'ai décidé de limiter la valeur d'un pixel à `MAX_BYTE` si la valeur éclaircie dépasse cette limite. Cela permet d'éviter les débordements et de maintenir les valeurs des pixels dans l'intervalle valide.

Pour la fonction `melt_picture`, j'ai utilisé une syntaxe utilisant `rand` trouvée sur Internet pour qu'on soit dans le domaine souhaité. Afin de trouver les bons indices des pixels à comparer, j'ai réalisé plusieurs essais avant de tomber sur la bonne formule.

2.6 Manipulation des pixels en utilisant une LUT (Look Up Table) `lut.[h/c]` et `pictures.[h/c]`

Il a fallu ne pas oublier d'utiliser la fonction `clean_lut`.

Pour la normalisation dans la fonction `normalize_dynamic_picture`, j'ai fait attention à ne pas diviser par 0.

Pour `set_levels_pictures`, j'ai d'abord imaginé la formule de discrétisation `(int) (((((i / MAX_BYTE) * nb_levels))) * (MAX_BYTE / (nb_levels)))`. Mais en affichant les valeurs ainsi obtenues à l'aide de `printf`, j'ai remarqué qu'il fallait plutôt utiliser `(int) ((((int) (((float) i / MAX_BYTE) * nb_levels))) * (MAX_BYTE / (nb_levels)))` afin de profiter de la conversion implicite sans quoi on obtenait toujours 0. Enfin, pour que le résultat se situe entre 0 et 255, la formule obtenue au final est `(int) ((((int) (((float) i / MAX_BYTE) * nb_levels))) * (MAX_BYTE / (nb_levels -1)))`.

2.7 Opérations arithmétiques sur les images `image_arithmetic.[h/c]`

Pour la fonction `mult_picture`, j'ai décidé de multiplier les pixels de mêmes coordonnées entre eux pour former l'image finale `p_mult.data[i] = (int) ((p1.data[i] * p2.data[i]) / MAX_BYTE);`. Ainsi l'image résultante est une multiplication simple, mais comme ses pixels se répartissent pour la plupart autour de la moyenne de 127, l'image est "plate". De plus, j'ai remarqué après coup en construisant le programme principal qu'on voulait aussi pouvoir multiplier une image PPM avec une PGM. Pour cela, il fallait distinguer alors trois cas. On fait comme si l'image PGM était une image couleur avec trois fois la même valeur pour chaque pixel. J'aurais pu réutiliser la fonction `convert_to_color_picture`, mais pour des raisons de lisibilité des instructions, j'ai adapté son code directement dans la fonction `mult_picture`.

Pour `mix_picture`, il faut pouvoir accepter tout type d'image en entrée (PPM ou PGM). Donc l'image résultante est une image qui a pour channel le maximum des channels des trois images d'entrée. Après avoir créé une telle image, j'ai converti les images d'entrée en images de PPM si une d'entre elles l'est. Ensuite, j'ai appliqué la formule donnée dans le sujet.

2.8 Re-échantillonnage d'images `image_resample.[h/c]`

Je traduis les fonctions données dans le sujet. De même que précédemment, c'est en testant et en affichant que j'ai remarqué que pour calculer les facteurs de redimensionnement, il fallait écrire `double factor = (double) p.width / width;` et non `image.width / width;` pour utiliser la division réelle et non entière.

2.9 Programme principal `main.c`

Pour obtenir `Lenna_gray_dynamic.pgm`, j'ai utilisé la fonction de normalisation. De manière similaire, pour obtenir `Lenna_color_dynamic.pgm`, j'ai appliqué la fonction split, la normali-

sation puis merge le tout.

L'exécution du `main.c` m'a permis de détecter plusieurs erreurs :

— Oubli de `* channels` dans la boucle for de `brighten`, c'est l'utilisation d'images en couleurs qui m'a permis de le détecter. En effet, seulement un tiers de l'image était éclairci auparavant.

— Dans `set_levels_picture`, je déclarais par erreur `lut l = create_lutnb_levels;` au lieu de `lut l = create_lut(MAX_BYTE);`.

— Pour les fonctions `melt_picture` et aussi `brighten_picture`, je me suis rendu compte qu'il fallait copier l'image car sinon on modifiait directement par référence la valeur vers laquelle pointe le pointeur data, donc l'image de départ. J'ai ensuite rencontré le même problème avec les fonctions utilisant `apply_lut`. Ainsi, j'ai modifié `apply_lut` directement. Puis de même pour `set_levels_picture`.

— J'avais oublié plusieurs tests `is_empty_picture`.

3 Conclusion et Limites

— Le programme ne prend en charge que les formats PPM et PGM. Une extension pour d'autres formats aurait pu être implémentée.
— Les valeurs des pixels sont limitées à l'intervalle [0, 255], ce qui limite la précision pour certaines applications.
— Certaines fonctions, comme `split`, ne fonctionnent qu'avec trois composantes et ne gèrent pas par exemple la transparence.
— La fonction `resample_picture_nearest` ne fonctionne que pour des images carrées.

Annexe : Manuel d'utilisation de l'exécutable

Pour exécuter le programme, compiler le programme puis lancer l'exécutable avec les options suivantes :

1. `-h, --help` : Afficher un message d'aide
2. `--conversion` : Traiter la conversion d'image
3. `--split` : Traiter la séparation d'image
4. `--brighten` : Traiter l'éclaircissement d'image
5. `--melt` : Traiter la fusion d'image
6. `--inverse` : Traiter l'inversion d'image
7. `--dynamic` : Traiter la normalisation dynamique d'image
8. `--levels` : Traiter les niveaux d'image
9. `--resample` : Traiter le ré-échantillonnage d'image
10. `--difference` : Traiter la différence d'image
11. `--product` : Traiter le produit d'image
12. `--mixture` : Traiter le mélange d'image
13. `--all` : Traiter toutes les images