Rapport de Projet 2024

Traitements d'images en niveau de gris ou en couleur

Bilâl Jaiel

Dâte : January 4, 2025

Contents

1 Introduction

Le but de ce projet est d'implémenter différents traitements sur des images en niveau de gris ou des images en couleur comme lors du TP 6.

2 Structure du projet

Le projet est constitué des fichiers ci-dessous. Ces fichiers contiennent toutes les fonctions données (ex. : le fichier filename.[h/c]) ou demandées par le sujet (hors questions bonus), ainsi que quelques fonctions supplémentaires. On trouvera également ci-dessous une présentation des fonctions supplémentaires et de celles qui nécessitent des spécifications :

./projet
|
|-> main.c
|
|-> filename.[h/c]
|
|-> lut.[h/c]
|
|-> Makefile
|
|-> pixels.[h/c] :
|
| int pixel(int comp, int nb_chan, int c)
La fonction pixel prend en paramètre trois entiers : comp, qui désigne la position du pixel dans l'image, nb_chan, le nombre de canaux pour un pixel, et c, la position de l'octet dans le canal spécifié. Elle calcule et renvoie la position de l'octet correspondant dans l'image, en fonction de ces paramètres.
|
|-> pictures.[h/c] :
|
| FILE* canopen(char* filename)
Cette fonction prend en paramètre le nom d'un fichier et vérifie si le fichier peut être ouvert en mode lecture. Si l'ouverture réussit, elle retourne un descripteur de fichier qui permet de lire le contenu du fichier. En cas d'échec, elle retourne NULL indiquant que le fichier ne peut pas être ouvert.
|
| FILE* canwrite(char* filename)
Cette fonction prend en paramètre le nom d'un fichier et vérifie si le fichier peut être ouvert en mode écriture. Si l'ouverture réussit, elle retourne un descripteur de fichier qui permet de lire le contenu du fichier. En cas d'échec, elle retourne NULL indiquant que le fichier ne peut pas être ouvert en écriture.
|
| int is_gray_picture(picture p)
Cette fonction prend en paramètre une image. Si celle-ci est une image en nuances de

gris, elle retourne 1. Sinon, elle retourne 0.
|
| int is_color_picture(picture p)
Cette fonction prend en paramètre une image. Si celle-ci est une image en couleur, elle retourne 1. Sinon, elle retourne 0.
|
| int rand_up_to(int max)
Cette fonction données dans le TP6 prend en paramètre un entier naturel et renvoie un entier aléatoire compris entre 0 et la valeur de cet entier.
|
| int byte_number(picture p)
Cette fonction prend en paramètre une image et renvoie le nombre total d'octet qu'elle contient.
|
| picture grow_size_nearest_neighbor_interpolation(picture p, double factor)
Cette fonction prend en paramètre une image et un facteur. Elle renvoie une image agrandie selon le facteur spécifié en utilisant la méthode d'interpolation du plus proche voisin.
|
| picture reduce_size_nearest_neighbor_interpolation(picture p, double factor)
Cette fonction prend en paramètre une image et un facteur. Elle renvoie une image réduite selon le facteur spécifié en utilisant la méthode d'interpolation du plus proche voisin.
|
| picture grow_size_bi_linear_interpolation(picture p, double factor)
Cette fonction prend en paramètre une image et un facteur. Elle renvoie une image agrandie selon le facteur spécifié en utilisant la méthode d'interpolation bi-linéaire.
|
| picture reduce_size_bi_linear_interpolation(picture p, double factor)
Cette fonction prend en paramètre une image et un facteur. Elle renvoie une image réduite selon le facteur spécifié en utilisant la méthode d'interpolation bi-linéaire.
|

3 Choix Techniques et Résolutions

3.1 Remarques générales

Dans une image, chaque pixel est représenté par un ou plusieurs octets en fonction du format. De manière visuel on représente cette structure sous la forme d'un tableau. Cependant dans mon code, je traite les images comme une seul ligne d'octets via une liste car je trouve cela plus confortable. Ainsi, les opérations d'accès sur les pixels s'appliquent sur une seul ligne.

3.2 lut.[h/c]

3.2.1 create_lut

Pour la fonction create_lut, j'ai décidé d'utiliser la fonction calloc afin d'initialiser toutes les valeurs de la LUT à 0. Cela permet de prévenir les erreurs qui pourraient survenir si les valeurs de la LUT n'étaient pas initialisées après sa création.

3.2.2 apply_lut

Pour la fonction apply_lut, déterminer l'indice correct nécessite de prendre en compte que la taille de la LUT n'est pas nécessairement égale au nombre total d'octets de l'image. Pour cela, il faut effectuer une mise à l'échelle en utilisant un produit en croix :

Octet -> 255
????? -> Taille de la LUT

Cette opération permet d'obtenir la valeur correcte de l'indice. On peut alors accéder à la LUT pour récupérer la valeur correspondante et mettre à jour l'octet de l'image.

3.3 pictures.[h/c]

3.3.1 read_picture

Pour la fonction read_picture, il faut gérer les lignes commentées qui peuvent apparaître entre la deuxième et la quatrième ligne du fichier. Pour cela j'utilise une boucle while qui lit chaque ligne, vérifie si son premier caractère est un '#', et dans ce cas, ignore la ligne pour passer à la suivante. Si la ligne n'est pas commentée elle est alors traitée.

Cependant lorsqu'une ligne est examinée le curseur avance systématiquement à la suivante. Cela pose un problème lorsqu'on souhaite se positionner précisément au début des octets de l'image correspondant à la quatrième ligne. En effet, lorsque la quatrième ligne est traiter le curseur se retrouve à la cinquième ligne.

Pour résoudre ce problème il est nécessaire de mémoriser la position du curseur avec ftell avant de lire une ligne. Ainsi lorsqu'on arrive au traitement de la quatrième ligne, on peut repositionner le curseur grâce à fseek à l'emplacement enregistré dans une variable position. Cela garantit qu'on commence correctement la lecture des octets de l'image.

3.3.2 canopen / canwrite

Pour éviter la redondance liée à l'ouverture des fichiers en lecture et en écriture j'ai décidé de créer deux fonctions : canopen et canwrite. Ces fonctions permettent de vérifier si un fichier peut être ouvert respectivement en lecture ou en écriture. Si l'opération est possible elles renvoient un descripteur de fichier sinon elles retournent NULL.

Pour la fonction canwrite, je commence par vérifier si le fichier peut être ouvert en lecture à l'aide de canopen. Si ce n'est pas le cas, la fonction retourne NULL. Ensuite je procède à un test d'ouverture en écriture pour confirmer que cette opération est possible.

3.3.3 melt_picture

Dans cette fonction il est nécessaire à un moment donné de déterminer lequel de deux pixels est le plus foncé. Pour cela j'ai décidé de calculer la somme des composantes RGB de chaque pixel. En effet, un pixel blanc correspond aux valeurs (255, 255, 255), tandis qu'un pixel noir est représenté par (0, 0, 0). On peut donc en déduire que plus la somme des composantes d'un pixel est élevée, plus celui-ci est proche du blanc et donc plus il est clair.

De plus, cette fonction doit également gérer les effets de bord. Par exemple si l'on choisit un pixel au hasard et que l'on souhaite le comparer à celui situé au-dessus, il est indispensable de vérifier que le pixel sélectionné ne se trouve pas sur la première ligne de l'image car dans ce cas il n'existe pas de pixel au-dessus.

3.3.4 normalize_dynamic_picture

Dans cette fonction, on travaille avec une image dont les octets se situent dans une plage de valeurs comprise entre un minimum (‘min‘) et un maximum (‘max‘), qui ne sont pas nécessairement 0 et 255. L'objectif est de ramener ces valeurs dans la plage standard [0, 255].

Pour cela, on recalcule la valeur de chaque octet. Tout d'abord, les pixels ayant la valeur ‘min‘ seront convertis en 0, tandis que ceux ayant la valeur ‘max‘ seront transformés en 255. Plus généralement, pour un pixel de valeur ‘i‘ situé entre ‘min‘ et ‘max‘, sa nouvelle valeur sera calculée à l'aide de la formule suivante :

$$\text{Nouvelle valeur} = \frac{(i - \min)}{\max - \min} \times 255$$

En soustrayant le minimum j'ai recentré les valeurs à partir de 0. Puis en divisant par l'écart max-min j'ai normalisé les valeurs dans la plage [0, 1]. Enfin je multiplie le tout par 255 pour obtenir la valeur de l'octet correspondant à la nouvelle plage de valeurs qui est [0, 255].

3.3.5 mult_picture

Dans cette fonction on réalise la multiplication des octets de deux images. Cependant, cette opération peut entraîner un dépassement de la plage initiale des valeurs. Pour éviter ce problème je commence par stocker temporairement toutes les valeurs issues des multiplications dans une liste intermédiaire. Je cherche ensuite les valeurs minimale et maximale parmi cette liste. Enfin, je normalise les valeurs de la liste des produits que je mets ensuite dans l'image finale.

3.3.6 grow_size_nearest_neighbor_interpolation

Pour cette fonction, pour chaque pixel de l'image agrandie on détermine le pixel le plus proche dans l'image initiale (le plus proche voisin). Le pixel dans l'image de sortie prend alors la valeur de ce pixel qui est le plus proche.

3.3.7 grow_size_bi_linear_interpolation

Dans cette fonction, pour chaque pixel de l'image agrandie on calcule la position du pixel correspondant dans l'image de base (plusieurs pixels de l'image agrandie peuvent être associés à un seul pixel de l'image de base). Ensuite, les quatre pixels voisins de ce pixel dans l'image de base sont interpolés pour déterminer la valeur du pixel agrandi.

L'interpolation est réalisée en calculant des coefficients d'interpolation a et b puis en appliquant la formule suivante :

$$\text{Pixel agrandi} = (1-a)(1-b)p_1 + a(1-b)p_2 + (1-a)bp_3 + abp_4$$

où p_1, p_2, p_3, et p_4 représentent les quatre pixels voisins dans l'image de base.

3.3.8 reduce_size_[nearest_neighbor, bi_linear]_interpolation

Pour ces deux fonctions, j'utilise leur fonction analogue pour l'agrandissement en lui passant la même image mais avec un facteur égal à 1/facteur. Cela permet de simplifier le code et d'éviter les répétitions.

3.4 main.c

La fonction main parcours la liste d'argument donnée en paramètre pour effectuer - selon l'extension des fichiers - une succession de tâche donnée dans le sujet. De plus, j'ai décidé dans cette fonction de prendre le moins de place possible en mémoire en libérant au fur et à mesure la mémoire.

4 Conclusion

En conclusion, on peut commencer par dire que l'interpolation avec la méthode du plus proche voisin laisse vite apparaître des pixels ce qui n'est pas très fidèle à l'image de base. Pour la fonction de multiplication, parcourir une première fois tous les octets pour faire les multiplications puis rechercher le minimum et le maximum et enfin terminer la normalisation est très coûteux. De plus, on peut dire la même chose pour la fonction de normalisation qui effectue la même chose sauf l'étape de multiplication. J'ai essayé au maximum de faire en sorte que les fonctions puissent s'adapter à différents formats. De plus, on pourrait encore beaucoup compléter ce programme au vu de toutes les possibilités de modifications et de retouches photographiques. Enfin, la gestion de la mémoire a été aussi un défi pour moi à la fin de mon code, j'ai essayé de libérer toute la mémoire que j'utilisais.