Projet Fondements de l'informatique

Luc Brun

1 Informations générales

Début du projet : Semaine du 8 Novembre 2021

Remise du projet : Vendredi 7 janvier 2021

Nombre de personnes : le projet est effectué obligatoirement en binôme.

Documents à remettre : Un rapport écrit de 10 pages (max) avec une description des méthodes algorithmiques et des structures de données utilisées ainsi qu'un jeu d'essais. Ce document sera déposé sur moodle. Une mini soutenance de 10 minutes aura lieu la semaine des examens. A cet effet, un diaporama de 7 à 8 diapositives (format pdf) sera également déposé sur moodle.

2 Notations

La couleur est un objet fort complexe dont l'étude correspond à une science appelée la *colorimétrie*. Toutefois, pour nous une couleur sera stockée sous la forme de 3 octets consécutifs indiquant les quantité de bleu, rouge et vert qu'il faut surimposer pour obtenir la couleur. Chaque composante est stockée sur un octet et varie donc entre 0 et 255. Une couleur se stocke au total sur $3*8 = 24$bits.

Une image couleur est un tableau 2D de couleurs. Un élément de ce tableau est appelé un pixel (picture element). Un pixel est défini par ses coordonnées i, j qui correspondent à ses indices dans la matrice (avec l'origine en haut à gauche de l'image et l'axe y orienté vers le bas).

Une table de couleur est un ensemble de couleurs. Dans notre cas, ces tables seront stockées sur disque comme des images $1 \times nb\ couleur$.

3 Description du projet

Le but du projet est de proposer une méthode d'inversion de tables de couleur par kd arbre.

— Une méthode d'inversion de tables de couleur prend en paramètre une image 24 bits et une table de couleur. Elle produit en sortie une image où chaque pixel de l'image est affecté à sa couleur la plus proche dans la table.

— Un kd arbre est un arbre binaire codant une découpe récursive d'un ensemble de points (ou de couleurs dans notre cas). Les kd arbres sont très utilisés pour trouver les k-plus proches voisins d'un point donné.

4 Module table de couleur

On se propose de créer un module spécifique pour le stockage d'une table de couleur. Cette impléméntation doit se faire en cachant l'implémentation (struct dans le .c) et avec des fonctions courtes et claires (voir le module image).

Une table de couleur est une structure composée de :

1. Un pointeur d'entiers ou de char* (a votre convenance) qui contient la suite des couleurs de la table,
2. Un entier qui code le nombre de couleurs de la table,
3. Un booléen `owner` indiquant si la table a alloué le pointeur ou pas. Seule la table ayant alloué le pointeur a le droit de le libérer.

On considère le .h suivant :

```
#ifndef __TABLE_HH__
#define __TABLE_HH__

typedef enum {red,green,blue} axis;
typedef char color; /* ou int*/
typedef enum {false,true} boolean;
typedef color_table struct* color_table;


color_table   create_color_table(image);
boolean       destroy_color_table(color_table);

color_table   color_table_duplicate(color_table,int,int)

void          color_table_get_color(color_table,int,color*);
int           color_table_get_nb_color(color_table)
void          color_table_sort(color_table,axis);

#endif
```

1. La fonction `create_color_table` crée une table à partir d'une image composée d'une seule ligne. Il faut donc allouer le pointeur, initialiser la longueur et remplir le tableau. Le champ `owner` est positionné à `true` dans ce cas.
2. La fonction `destroy_color_table` supprime une table de couleur. Le pointeur ne doit être libéré que si `owner` est positionné à `true`. La structure contenant la table doit par contre être libérée dans tous les cas.

3. La fonction `color_table_duplicate` crée une sous table. Le deuxième argument correspond à l'offset [1] de la sous table par rapport à la table principale et le troisième argument correspond à la longueur de la sous table. Cette fonction initialise son pointeur à partir de celui de la table principale sans faire d'allocation. Le champ `owner` doit être positionné à `false`. Si l'on suppose que l'objet `tab` contient une table de 20 couleur, `color_table_duplicate(tab,3,5)` renvoie une table de 5 couleurs commençant à l'indice 3 dans tab.
4. La fonction `color_table_get_color` remplie le tableau de taille 3 passé en second argument à partir de la couleur dont l'indice est passé en premier argument.
5. La fonction `color_table_sort(axis)` trie les éléments d'une table de couleur par ordre croissant en fonction de leurs coodonnées sur un axe (0 pour l'axe rouge, 1 pour le vert, et 2 pour le bleu). Vous pourrez pour cela implémenter votre propre fonction de trie ou utiliser la fonction `qsort` de stdlib. Il faudra dans ce dernier cas prévoir trois fonctions de comparaison de couleurs.

On vérifiera pour chaque fonction que les arguments passés sont valides par des asserts.

5 Implémentation de la méthode triviale

Écrivez un programme qui :

1. Charge une image et une table de couleur à partir du module image (utilisez argv, argc), gérez les saisies incorrectes.
2. Transfère les couleurs de la table contenues dans une image dans une table de couleur
3. Parcours chaque couleur de l'image originale, cherche sa couleur la plus proche dans la table et écris cette couleur dans l'image.
4. Envoie l'image résultat sur la sortie standard.

6 Construction du kd arbre

Chaque noeud d'un kd arbre stocke un nombre fixé de points décidé au départ. On initialise le kd arbre avec l'ensemble global de points et chacun de ses noeuds sera découpé récursivement jusqu'à ce qu'il contienne au plus le nombre fixé de points. La découpe d'un noeud s'effectue de la façon suivante :

1. Choisir un axe de découpe (on choisira par défaut l'axe avec la plus grande projection). Pour l'axe rouge (par exemple) la longueur de la projection d'un noeud C sur cet axe est égale à $R_{max} - R_{min}$ où R_{max} (resp. R_{min})

1. décalage qui s'ajoute à une adresse mémoire

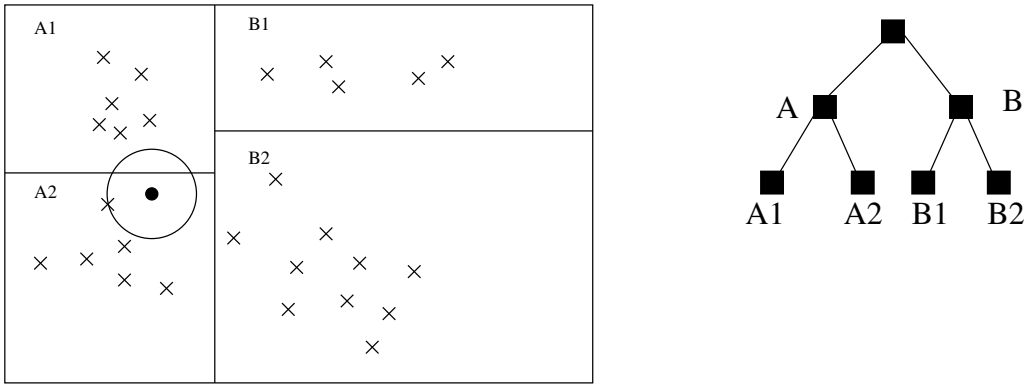

FIGURE 1 – Un exemple de kd arbre avec 4 feuilles. La recherche de la couleur (•) nous conduit sur la feuille A2. La sphère calculée sur A2 intersecte A1 ce qui va déclencher une nouvelle recherche sur la feuille A1.

représente la plus grande (resp. plus petite) valeur de R pour les couleurs contenues dans C.

2. Trier la table associée au noeud suivant l'axe choisi.
3. Déterminer un plan de coupe orthogonal à cet axe en se basant sur le tri (on choisira par défaut le plan médian : autant de points des deux cotés du plan).
4. Créer deux sous tables contenant respectivement les couleurs avant et après le plan de coupe.
5. Initialiser deux sous noeuds associés à chacune des sous tables. La découpe se poursuit alors dans les sous noeuds jusqu'à ce que chaque noeud est un nombre de couleurs inférieur au seuil.

Un kd arbre est donc composé de :

1. Une table de couleur,
2. Deux pointeurs sur des kd arbres (`left_son` et `right_son`). `left_son` correspondra au kd arbre contenant les couleurs avant le plan de coupe.
3. Un axe de coupe.
4. La position du plan de coupe orthogonal à l'axe choisi.

On considère le .h suivant :

```
#ifndef __KDTREE_HH__
#define __KDTREE_HH__

#include "table.h"

typedef  struct kdtree *kdtree;
```

```
kdtree create_kdtree(color_table,int);
void   destroy_kdtree(kdtree);
#endif
```

Définissez les fonctions `create_kdtree` et `destroy_kdtree`.

- La fonction `create_kdtree` initialise sa table de couleurs et découpe l'arbre si son cardinal est supérieur au nombre max passé en second argument. Cette découpe initialise les champs restants (y compris les deux fils gauche et droite). Si le nombre de couleurs de la table est inférieur au seuil, les deux pointeurs sur les fils doivent être initialisés à NULL.
- La fonction `destroy_kdtree` commence par détruire les deux sous fils du kdtree, avant de détruire sa table, puis l'objet lui même.

Les fonctionnalités correspondant aux points 1 et 3 de la construction d'un kd arbre (cf début de section) doivent être implémentés dans des fonctions séparées privées au module [2].

7 Calcul de la couleur la plus proche dans l'arbre

On désire implémenter la fonction :

```
void   kdtree_get_nearest_color(kdtree,color*, color*);
```

Cette fonction utilise le kd arbre (1er argument) pour calculer la couleur la plus proche de la couleur passée en 2e argument. Le résultat est mis dans le 3e argument.

Cette fonction procède en plusieurs étapes :

1. Une étape descendante dans laquelle on va par une méthode récursive chercher la feuille du kd arbre qui contient la couleur passée en paramètre. Pour cela on comparera dans chaque noeud interne la couleur d'entrée avec le plan de coupe du noeud.
2. Une fois dans cette feuille, nous allons calculer la couleur la plus proche de la couleur d'entré et sa distance à celle-ci (au plus max calculs). Cela définit une sphère centrée sur la couleur d'entrée et qui contient la couleur la plus proche.
3. Lors de l'étape de remontée de la récursion (Figure 1) :
   - Si la sphère, centrée sur la couleur d'entrée intersecte le plan de coupe du noeud courant on relance la recherche sur le fils que l'on a pas exploré lors de la descente.
   - Sinon, on remonte d'un niveau dans la récursion.

2. Les fonctions privées d'un module sont des fonction `static` déclarées dans le .c du module

8 Inversion de couleur par kd arbre

Reprenez l'exercice de la section 5 en construisant un kd arbre à partir de la table de couleur. Utilisez la fonction `kdtree_get_nearest_color` pour récupérer la couleur la plus proche de chaque couleur de l'image et stocker l'image de sortie comme précédemment. Comparez les temps de calculs à l'aide de courbes en faisant varier la taille des tables de couleur. On utilisera gnuplot pour cette dernière fonctionalité.

À l'aide de l'algorithme de quantification "`quant`" disponible sur :

`/home/public/PROJET_FONDEMENTS`

générez une table de couleur adaptée à une image puis appliquez cette table à une autre image. Cette opération s'appelle un transfert de couleur.

9 Travail Facultatif

Ce travail si il est fait correctement ajoutera des points à la note finale. Il n'est cependant pas nécessaire pour obtenir le maximum de points si tout le reste est exécuté parfaitement.

On définit les moments d'ordre 0, 1 et 2 d'une table de couleur C suivant un axe k par :

$$\forall j \in \{0,1,2\}\ M_j(C)_k = \sum_{c \in C} c_k^j$$

c_k représente la composante k de la couleur c (rouge, vert ou bleu). Le moment $M_0(C)$ représente donc une somme de 1, i.e. le cardinal de C, tandis que $M_1(C)$ représente la somme de ses coordonnées (suivant l'axe k) et $M_2(C)$ la somme des carrés de ses coordonnées.

1. Définissez une fonction qui prend un noeud d'un kd arbre et renvoie son axe de plus grande variance. Notez que le calcul de la variance sur un axe k peut être effectué par :

$$var_k(C) = \frac{1}{M_0(C)} \left(M_2(C)_k - \frac{M_1(C)_k^2}{M_0(C)} \right)$$

2. Définissez une fonction qui prend un kd arbre et renvoie la position de coupe qui minimise la somme des écarts à la moyenne de chacune des deux sous tables de part et d'autre du plan de coupe. Si on note par C la table du kd arbre courant, par t la position de coupe et par $C_1(t)$, $C_2(t)$ les deux tables qui correspondront au fils gauche et droit, il s'agit de trouver t_{opt} qui vérifie :

$$t_{opt} = argmin_{t \in [min,max]} M_0(C_1(t)) var(C_1(t)) + M_0(C_2(t)) var(C_2(t))$$

où $var(C_i(t))$ représente la variance de $C_i(t)$ sur l'axe choisi.

On peut montrer (mais ce n'est pas demandé) que cette dernière formule est équivalente à :

$$t_{opt} = argmax_{t\in[min,max]} \frac{\delta(t)}{1-\delta(t)} \|\mu_1(t) - \mu\|^2$$

où $\delta(t) = \frac{M_0(C_1(t))}{M_0(C)}$ est la proportion de données dans $C_1(t)$ et

$$\mu_1(t) = \frac{M1(C_1(t))}{M_0(C_1(t))}$$

est la couleur moyenne de $C_1(t)$ et μ la couleur moyenne de C.

3. Définissez dans le .c deux tableaux de pointeurs de fonction de taille 2 correspondant respectivement au choix de l'axe de coupe et au choix de la position de coupe. Ces tableaux sont initialisés à partir des 4 fonctions que nous avons définies (les deux de la section 6 ainsi que les deux nouvelles méthodes de cette section).
4. On se propose de modifier le .h de kd arbre de la façon suivante :

```
#ifndef __KDTREE_HH__
#define __KDTREE_HH__

#include "table.h"

typedef enum {largest_interval,largest_variance} splitting_axis;
typedef enum {median,minimal_SE} splitting_position;

typedef  struct kdtree *kdtree;

kdtree create_kdtree(color_table,int,splitting_axis,splitting_position);
void   destroy_kdtree(kdtree);
#endif
```

Modifiez la méthode `create_kdtree` pour quelle utilise les pointeurs de fonctions avec les indices passés en paramètre pour le choix de l'axe de coupe et de la position de coupe. Comparez les temps de calculs avec différents choix.

10 Remarques

1. L'objet du rapport n'est pas de dupliquer les commentaires (qui doivent être présent dans le code) mais d'expliquer les différents choix possibles pour chaque problème et les avantages, inconvénients de la méthode retenue.
2. L'originalité et l'efficacité des solutions algorithmiques (montrées dans le rapport) ainsi que la qualité du code seront des éléments déterminant de la note. Des exemples d'exécutions ou une démonstration du projet sont attendus.

3. Vous pourrez vous appuyer sur :
   — Une implémentation du module image disponible sous :
   `/home/public/PROJET\_FONDEMENTS/MODULE_IMAGE`
   Notez que le main est juste fourni à titre d'exemple pour montrer le fonctionement du module.
   — Un programme de quantification permettant de générer ses propres tables de couleur :
   `/home/public/PROJET\_FONDEMENTS/quant`
   Notez que ce programme a été compilé sous cybele et nécessitera peut être de se connecter à cette machine pour fonctionner correctement.
   — Des images disponibles sous :
   `/home/public/PROJET\_FONDEMENTS/IMAGES`
   Vous êtes également libre d'utiliser vos propre images au format ppm (pensez à utiliser convert si ce n'est pas le cas).
   — Un ensemble de tables de couleurs :
   `/home/public/PROJET\_FONDEMENTS/IMAGES/TABLES/`
   Vous pouvez encore une fois générer vos propres tables en utilisant le programme quant.