ENSICAEN
École Nationale Supérieure d'Ingénieurs de Caen

TPE : Fondements de l'informatique
Inversion de couleurs

Valentin Durand, ENSICAEN – 1A Informatique (2015-2016)

11 février 2016

Table des matières

Introduction

Ce projet a permis de mettre en pratique les connaissances acquises en programmation C au cours du premier semestre, notamment les structures de données et les allocations dynamiques, afin de produire un programme limitant les couleurs d'une image à celles disponibles dans une table prédéfinie. Ce programme prend en paramètre une image et une table de couleur, puis se charge pour chaque pixel de chercher sa couleur la plus proche dans la table de couleur, de remplacer ce pixel par la couleur trouvée et de sauvegarder l'image finale. L'algorithme de recherche de la couleur la plus proche est décliné en deux versions, une méthode triviale assez lente et une méthode utilisant des kd arbres afin d'accélérer la durée d'exécution. Ce rapport a donc pour but de détailler l'implémentation de ces deux méthodes et de comparer leurs résultats.

1 Méthode de travail

Bien que ce projet ait été réalisé en monôme, un gestionnaire de version a été utilisé afin de structurer le travail à effectuer au cours des mois de décembre et janvier. Les jalons ont suivis linéairement les consignes stipulées dans l'énoncé :

- Familiarisation avec le module image *(16 Dec)* ;
- Création du module table de couleur *(16 - 22 Dec)* ;
- Implémentation de la méthode triviale *(22 Dec - 1 Jan)* ;
- Construction du kd arbre *(1 - 2 Jan)* ;
- Calcul de la couleur la plus proche dans l'arbre *(2 Jan)* ;
- Inversion de couleur par kd arbre *(2 Jan)* ;
- Comparaison des résultats *(31 Jan)*.

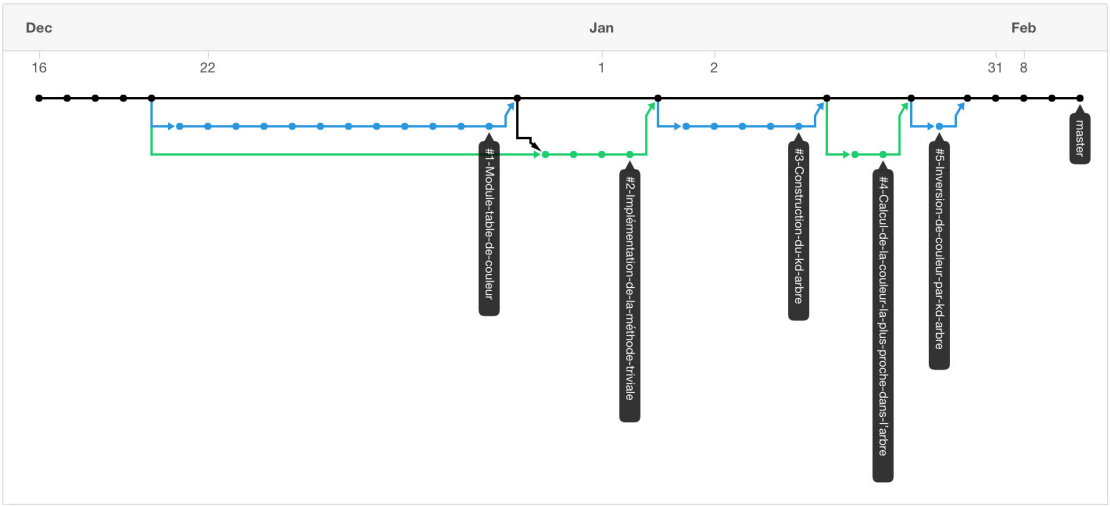

FIGURE 1 – Arbre des commits Git

L'intégralité des commits sont disponibles à l'adresse suivante : `https://github.com/vDurand/ProjetC-Inversion/commits/master`

2 Architecture

```
ProjetC-Inversion/
├── bin/
├── doc/
├── include/
│   ├── arbre.h
│   ├── table.h
│   └── image.h
├── lib/
│   └── libimage.a
├── src/
│   ├── arbre.c
│   ├── graphdata.c
│   ├── image.c
│   ├── inversion.c
│   ├── table.c
│   ├── test_arbre.c
│   ├── test_table.c
│   └── trivial.c
└── makefile
```

FIGURE 2 – Structure du répertoire projet

Afin de gérer les images `.ppm`, le programme exploite le module image sous forme du `image.c` compilé en librairie statique `libimage.a` contenu dans le dossier lib. Le makefile permet de compiler trois exécutables :

- **trivial** qui réalise l'inversion des couleurs par la méthode triviale ;
- **inversion** qui réalise l'inversion des couleurs par le biais d'un kd arbre ;
- **graphdata** qui permet de générer un fichier contenant les temps nécéssaires pour l'inversion une image par la méthode trivial et la méthode kd arbre avec des tables successivement de 128, 256, 512 et 1024 couleurs.

3 Table de couleur

3.1 Structure des données

Une table de couleur est une image de 1 pixel de hauteur sur le nombre de couleurs de la table. Pour stocker cette image, on utilise une structure `color_table` qui contient un entier représentant le nombre de couleurs stockées, un booléen définissant si la table est une sous-table ou non (vrai si elle n'est pas une sous-table, faux sinon) et un pointeur de pointeur permettant d'allouer un tableau d'entiers à deux dimensions. Ce tableau a un nombre de colonnes égale au nombre de couleurs de la table et 3 lignes pour les composantes rouge, vert, bleu de chaque pixel. Cette représentation par tableau permet de trouver facilement une couleur par son index sans avoir à l'explorer.

3.2 Fonctions

— `create_color_table` : à partir d'une image on créé une table de couleur. Dans un premier temps on récupère le nombre de couleurs grâce à la fonction `image_give_largeur` du module image. On alloue ensuite les 3 lignes du tableau qui contiendront les composantes des couleurs, en fonction de la taille récupérée précédemment. Une fois ce tableau défini, on explore l'image pixel par pixel et on récupère leurs composantes RGB grâce à la fonction `image_lire_pixel` du module image, que l'on stocke dans le tableau d'entier ;

— `destroy_color_table` : nettoie l'empreinte mémoire de la table de couleur. On libère l'espace mémoire de chaque ligne du tableau dans un premier temps, puis l'espace mémoire du reste de la structure. Attention on ne libère le tableau que si la table n'est pas une sous-table sinon la table mère serait corrompue ;

— `color_table_duplicate` : crée une sous-table à partir d'une table mère. On alloue 3 pointeurs qui prendront l'adresse de la case du tableau de la table mère correspondant à la première case de la sous-table pour de chaque composante RGB. Ce qui équivaut à l'adresse de la première case du tableau de la table mère décalé du nombre de case où commence la nouvelle table fille ;

— `color_table_get_color` : retourne la couleur à une position précise dans la table sous la forme d'un tableau de taille 3 ;

— `color_table_get_nb_color` : retourne ne nombre de couleurs présentes dans la table ;

— `color_table_sort` : trie les couleurs suivant un axe RGB choisi ;

— `color_table_get_min` et `color_table_get_max` : retourne la couleur la plus petite et la plus grande en fonction d'un axe RGB. Ces fonctions étaient utilisées pour la première version du choix de l'axe de coupe du kd arbre, mais sont devenues obsolètes avec la méthode de la variance.

4 Méthode triviale

On passe en argument le chemin d'une image et d'une table de couleur, on charge ensuite ces deux images grâce à la fonction image_charger du module image. On crée la table de couleur à partir de la seconde image via la fonction `create_color_table`. Puis l'inversion d'une image par une table de couleur se fait ici en comparant la distance entre la couleur de chaque pixel de l'image et toutes les couleurs de la table. On explore donc la table de couleur en entière autant de fois qu'il y a de pixel dans l'image. La distance est calculée en utilisant la formule :

$$Distance = \sqrt{(c1[rouge] - c2[rouge])^2 + (c1[vert] - c2[vert])^2 + (c1[bleu] - c2[bleu])^2}$$

On remplace ensuite la couleur de chaque pixel par la couleur de la table ayant la distance la plus faible avec celle-ci. L'image résultat est ensuite enregistrée grâce à la fonction `image_sauvegarder` du module image.

5 Kd arbre

5.1 Structure des données

Un kd arbre est une suite de pointeur sur des noeuds liés entre eux. On utilise donc une structure `kdtree` qui représente un noeud et qui stocke 2 pointeurs vers son noeud fils droit et son noeud fils gauche. Si ce noeud est une feuille, ces 2 pointeurs sont nuls. Cette structure contient aussi une table de couleur. La racine de l'arbre contient la table complète avec toutes les couleurs puis son fils gauche contient la moitié trié inférieure à un plan de coupe de cette table et le fils droit la moitié supérieure. On coupe successivement la table à chaque noeuds jusqu'à ce que chaque feuille contiennent une infime portion de la table d'origine. Pour connaître le plan de coupe de chaque noeud, la structure comprend un pointeur vers une structure stockant les données du plan de coupe et un entier spécifiant l'axe de coupe RGB. Cette structure `cutting_plane` possède un entier qui indique la position sur l'axe de coupe du noeud, de la couleur juste au dessus du plan de coupe dans la table et un entier contenant la valeur moyenne entre la composante de l'axe de coupe de la couleur juste au dessus du plan et celle juste en dessous.

5.2 Fonctions

- `choose_sorting_axis_by_max` : permet de sélectionner l'axe RGB de coupe en prenant celui qui a la plus grande différence entre sa composante de couleur maximum et sa composante de couleur minimum. Cette fonction est délaissé au profit de la méthode par la variance ;
- `choose_sorting_axis_by_var` : permet de choisir l'axe RGB de coupe en fonction de celui qui a la plus grande variance. On calcule donc la moyenne de chaque composante RGB de toutes les couleurs de la table, puis on calcule la variance de chaque axe en faisant la somme des carrés de chaque composante moins la moyenne de cette composante, le tout divisé par le nombre de couleurs ;
- `choose_axis_position` : retourne la position de la couleur juste au dessus du plan de coupe. On divise la table en deux et on vérifie bien que la valeur n'est pas la même des deux côtés du plan. Sinon on boucle jusqu'à ce que la valeur soit différence afin qu'on ait pas des couleurs à composantes identique sur un axe RGB de part et d'autre du plan de coupe ;
- `create_kdtree` : cette fonction permet de créer le kd arbre. Dans un premier temps on alloue l'espace mémoire pour le noeud racine, on lui associe la table de couleur au complet triée via la fonction `color_table_sort`, puis on définie l'axe et le plan de coupe grâce aux fonctions vues précédemment. Ensuite on définie récursivement les noeuds fils droit et gauche qui prennent respectivement la moitié supérieur et inférieur de la table de couleur du noeud père, et ce tant que le nombre de couleur au dessus ou en dessous du plan de coupe n'est pas inférieur au seuil prédéfini ;
- `destroy_kdtree` : supprime récursivement les fils des noeuds tant qu'ils ne sont pas nul, détruit la table de couleur et désalloue la structure `cuttingPlane` de chaque noeud ;
- `get_distance_between_colors` : calcul la distance entre deux couleurs avec la même formule que dans la méthode triviale ;

— `search_nearest_color_in_node` : permet de trouver la couleur la plus proche dans la table spécifique d'un noeud à une couleur d'une image ;
— `kdtree_get_nearest_color` : cette fonction explore l'arbre pour trouver la feuille contenant la couleur la plus proche de la table de couleur à une couleur d'une image. La recherche se fait récursivement, si la couleur cherchée est sous le plan de coupe du noeud actuel on cherche à nouveau dans son fils gauche en rappelant `kdtree_get_nearest_color` et si elle est au dessus du plan de coupe on cherche dans le fils droit. On descend ainsi dans l'arbre jusqu'à être dans la feuille qui pourrait contenir la couleur recherchée. On explore la partie de la table spécifique à cette feuille via la fonction `search_nearest_color_in_node` pour trouver la couleur la plus proche. Une fois cette couleur trouvé, on remonte l'arbre et on vérifie que la distance entre la couleur recherchée et la couleur trouvée est bien inférieure à la distance entre la couleur recherchée et la projetée de cette couleur sur le plan de coupe. Si ce n'est pas le cas, on est obligé de réeffectuer la recherche de la couleur la plus proche dans la table spécifique au noeud père, afin d'être sûr qu'il n'y a pas une couleur encore plus proche.

6 Exécution

```
$ ./inversion IMAGES/lenna.ppm IMAGES/TABLES/table_house_1280.ppm
```

FIGURE 3 – Image Lenna avant/après inversion par table House 1280 couleurs et seuil 10

```
$ ./inversion IMAGES/fleurs.ppm IMAGES/TABLES/table_lenna_53.ppm 5
```

FIGURE 4 – Image Fleurs avant/après inversion par table Lenna 53 couleurs et seuil 5

7 Conclusion

La recherche par kd arbre assure un temps d'exécution nettement plus court pour les inversions par tables à grands nombres de couleurs. La différence ce note particulièrement lors de l'inversion par une table aux couleurs proches, comme par exemple l'inversion de l'image Lenna par la table Lenna générée à partir de cette même image, l'image n'est donc pas réellement inversée mais on a limité le nombre de couleurs utilisées. Les courbes de temps d'exécution ci-dessous ont été réalisés pour des tables de 128, 256, 512 et 1024 couleurs via `graphdata.c` et tracées par `gnuplot`.

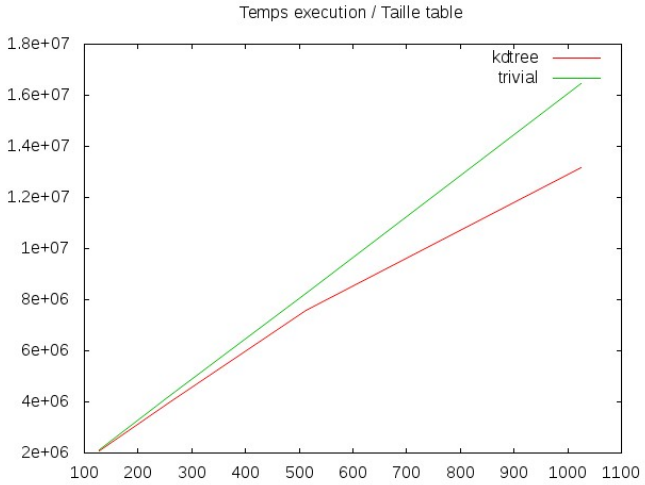

FIGURE 5 – Image et table de couleur provenant de sources différentes

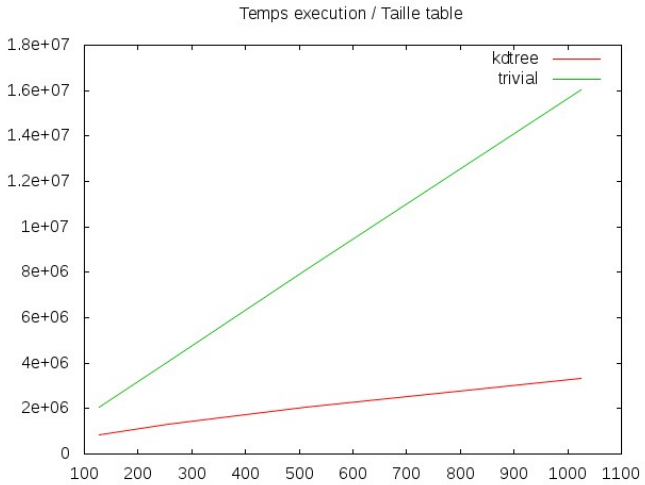

FIGURE 6 – Image et table de couleur provenant de la même source