Rapport de projet - Compression par taches de couleur
Unniversité Paris 8 - Vincennes
L3 Informatique

LEGOUEIX Nicolas

10 avril 2020

Table des matières

Chapitre 1

Rappel du projet & Déviation du sujet

1.1 Le projet

Le projet à pour but de compresser une image en regroupant tous les pixels de couleur similaire en une seule et même tâche d'une couleur unique. Des pixels sont dits similaires à partir du moment où les différences de leur composante en rouge vert et bleu sont inférieures à un seuil fixé.
Ainsi avec un seuil de 100, un pixel (0 0 255) sera dans la même tache qu'un pixel (0 0 190) mais pas qu'un pixel (0 255 0).

1.2 Dévitations

A cause d'une limitation à la fois architecturale et matérielle, j'ai du traiter l'image par morceaux et pas d'un seul coup, faisant apparaitre certains pixels a piori assez proches comme appartenant à deux taches distinctes de couleur légèrement différentes. Vous pourrez voire une illustration de ces propos en annexe avec des exemples d'utilisation. Chaque "morceau" est une matrice de X par X (valeur ecrite en global en tête du fichier modif.c) qui contient une partie de l'image. Cette solution fait perdre en fidélité et dans l'aspect "bande déssinée" recherché, mais permet de gagner drastiquement en temps de calcul et d'éviter de cumuler les appels récursifs. Elle sera justifiée plus en détails plus loin. La taille de la matrice peut varier de 1 (image inchangée) à 400 (valeur la plus haute testée au dela de laquelle le temps de calcul devient très long et pose de forts risques de crash. ATTENTION : avec une matrice de 400, mon ordinateur a mis environ 10min pour parcourir tout Refuge !). Plus la taille de la matrice est grande, plus le temps de calcul sera grand mais le gommage sera efficace).
A la base, la compression est obtenue en résumant une tache aux pixels qui la bordent. Celà permet de stoquer beaucoup moins de pixels et une seule

valeur de couleur par tache au lieu d'une par pixel. Cependant, je ne suis parvenu qu'a obtenir une tache représentant tous les pixels qu'elle contient et pas seulement ses bordures. Le résultat a au final l'effet inverse d'une compression puisque le fichier texte obtenu pèse environ 2x plus que le fichier image source. Par exemple, Refuges.ppm pèse 2.4Mb initialement et le fichier obtenu pèse 4.1Mb avec des matrices de 50x50 et 4.5Mb en 100x100.

Chapitre 2

Approche générale

2.1 Coeur de l'algorithme

L'idée générale était à la base de procèder en plusieurs étapes :

- Découper l'image de départ en un tableau de Pixel. Un pixel est un groupe de 3 couleur et possède un état exploré (ou non).
- Pour chaque pixel de ce tableau
  - Si il est exploré, contiuer avec le suivant
  - Sinon, il appartien à une nouvelle tache de couleur non traitée
    - Explorer la tache
    - Calculer la couleur moyenne de la tache ainsi formée
    - La mémoriser dans un fichier
    - Remplacer les pixels du tableau par ceux de la tache.
- Une fois le tableau completement parcouru et transformé avec les taches, l'écrire dans l'image courante pour afficher le resultat.

Une tache est alors obtenue assez simplement : on appelle une fonction recursive en lui donnant le pixel initial (celui qui a causé la creation de la tache). Cette fonction se rappelle avec les pixels situés a gauche, a droite, au dessus et au dessous d'un pixel courant à condition que ces derniers existent et ne soient pas déjà explorés. A ces conditions, on ajoute une comparaison de la couleur du pixel courant à celle d'un pixel de référence, et on quitte la fonction si la différence est trop grande. Chaque pixel correspondant à ces critères est ajouté dans un tableau d'indexs et marqué comme exploré.
On peut ensuite parcourir la tache ainsi formée et le tableau de Pixels qui représente l'immage et calculer la moyenne des couleurs de chaque pixel de la tache. Une tache est donc résumée par une liste d'indexs dans l'image et un unique triplet rouge vert bleu.

2.2 Le problème de cette approche et solution

Le problème est qu'une approche recursive, bien que très simple en apparence pose de gros problèmes une fois qu'elle est appliquée à une grande quantité d'appels. En effet, si cette approche fonctionnait parfaitement pour des images de 100x100 ou 200x200, il était impossible de faire tourner le programme sur des images de taille plus réaliste comme les images données en exemple et que vous pouvez voir en Annexe dont les tailles sont de 1024*768. Même des images de 800x800 finissaient inévitablement par générer une Segfault. Après pas mal de recherches, je me suis rendu compte que celle ci ne survenait pas pendant un accès mémoire mais sur un appel à explore(). Il s'avère que sur ces grandes images, les taches plus grandes imposaient un très grand nombre d'appels récursifs ce qui finissait par remplir la callStack. Cette hypothèse a été confirmée grace à l'utilisation de la commande :

```
1 ulimit -s 16384
```

Cette commande permet de changer la taille de la callStack (par defaut a 8192 octets) pour la valeur entrée par l'utilisateur. Celà a permis au programme d'aller plus loin dans la recursion prouvant l'origine du problème.Vous pouvez consulter les échanges que j'ai eu et qui m'ont permis d'arriver a cette conclusion en annexe.
Partant de l'observation que j'avais un programme fonctionnel sur de petites images mais pas sur des grandes, j'ai eu l'idée de découper le tableau de Pixels un autant de blocs de 200x200 (taille maximale pour la quelle le programme tournait a ce moment là) qu'il n'en faut pour le couvrir totalement, et d'appeller ma fonction de génération des taches sur ces blocs au lieu de l'image complète.

2.3 Le problème de la solution

Le gros désavantage de cette solution est que du coup chaque tache est générée localement, sans considération pour ce qui se trouve autour du bloc sur lequel le programme est en train de travailler. Cela veut dire qu'une tache peut s'arreter car elle a atteint les bords du bloc, alors que dans un monde idéal, elle continurait a s'étendre plus loin. L'effet produit est que l'on peut voir chaque bloc au sein d'une même zone de l'image avec des couleurs légèrement différentes (voir annexe) ce qui fait perdre en fidèlité à l'image de sortie. On peut compenser cet effet en utilisant des matrices plus grandes mais cela ne permettera jamais d'avoir des taches complètes. Les matrices plus petites permettent elles aussi de diminuer la visibilité des frontières entre blocs, mais regroupent beaucoup moins de pixels dans chaque tache. Il s'agit de trouver un compromis. Je trouve que des matrices de 15 - 30 donnent un bon compromis lissage / visibilité des fontières / rapidité d'execution.

2.4 Tentative de solution au problème de la solution

J'ai essayé de mitiger le problème en effectuant plusieurs passages avec des matrices de taille différentes (de manière à ce qu'elles ne se chevauchent pas) afin de lisser les frontières, mais cela n'a pas donné de resultat convaincant et pour une raison que j'ignore les frontières ne bougaient pas et restaient aussi visibles qu'avec un seul passage, j'ai donc abandonné l'idée.

Chapitre 3

Implémentation

3.1 Structures

Listing 3.1 – Explore

```
1  struct Pixel {
2      GLubyte r;
3      GLubyte g;
4      GLubyte b;
5      char explored;
6  };
7  typedef struct Pixel Pixel;
8
9  struct Tache {
10     int * limits;
11     GLubyte r;
12     GLubyte g;
13     GLubyte b;
14 };
```

Un Pixel est une structure regroupant les 3 composantes de couleur d'un pixel d'une image (initialement, chaque pixel de l'image occupe 3 valeurs à la suite dans un tableau ce qui est très compliqué a traiter) ainsi qu'un status explored, qui permettera de savoir si ce pixel a dejà été exploré par la fonction explore qui étend les taches de couleur.
Chaque tache est elle aussi représentée par ses 3 composantes de couleur, et par un tableau qui contient l'index de tous les pixels qui la composent.

3.2 explore

Listing 3.2 – Explore

```
1  void explore(Pixel * array, int j, Tache * cur_pound,
       int * it, Pixel previous, int sizeX, int sizeY){
2    if(abs((int)array[j].r - previous.r) > SEUIL || abs((
       int)array[j].g - previous.g) > SEUIL || abs((int)
       array[j].b - previous.b) > SEUIL){
3      return;
4    }
5    array[j].explored = 1;
6    cur_pound->limits[* it] = j;
7    (* it)++;
8
9    if(j +1 < sizeX * sizeY && array[j+1].explored != 1){
10     explore(array, j + 1, cur_pound, it, previous,
         sizeX, sizeY);
11   }
12   if(j > 1 && array[j-1].explored != 1){
13     explore(array, j - 1, cur_pound, it, previous,
         sizeX, sizeY);
14   }
15   if(j + sizeX < sizeX * sizeY && array[j + sizeX].
       explored != 1){
16     explore(array, j + sizeX, cur_pound, it, previous,
         sizeX, sizeY);
17   }
18   if(j - sizeX > 0 && array[j - sizeX].explored != 1){
19     explore(array, j - sizeX, cur_pound, it, previous,
         sizeX, sizeY);
20   }
21 }
```

La fonction explore prends comme paramètre :

- array : un tableau de Pixel représentant le bloc de pixels sur lequel on travaille
- j : l'indexe du pixel courrant par rapport a array
- cur_pound : la tache de couleur qu'on est en train d'étendre
- it : un itérateur permettant l'ajout de valeurs dans les limits de cur_pound
- previous : le Pixel de référence pour les comparaisons de couleur
- sizeX et sizeY : les dimensions du bloc de pixel sur lequel on travaille

Elle a pour but d'étendre une tache de pixels donnée en arguments. Si le pixel courrant (array[j]) a une couleur similaire à celle de previous, on peut ajouter ce pixel aux limites de la tache courante et le marquer exploré. On peut ensuite étandre ce pixel. Cette tache est réalisée par des appels récursifs sur les pixels voisins au pixel courant dans array, c'est à dire array[j]. Ces

appels sont effectués si le pixel en question n'a pas déjà été exploré (auquel cas il est déjà attribué à une tache, que ce soit la tache actuelle ou une autre) et si le voisin se trouve bien dans le bloc. Cela signifie que son index doit etre inférieur à sizeX * sizeY pour le pixel suivant (aka celui de gauche), supérieur à 0 pour le précédant (aka celui de droite). Pour les pixels du dessus et du dessous, on regarde la valeur de l'index actuel soustrait et additionné de sizeX, c'est à dire une ligne complète.

3.3 tachesInit

Listing 3.3 – tachesInit

```
1  void tachesInit (Image * i){
2      int j, k;
3      int sizeX, sizeY, size, matrix, nbmatX, nbmatY;
4      GLubyte * im;
5      Pixel * array;
6      Pixel * section;
7
8      sizeX = i->sizeX;
9      sizeY = i->sizeY;
10     size = sizeX * sizeY;
11     matrix = matrixSize;
12     nbmatX = 0;
13     nbmatY = 0;
14     k = 0;
15     array = malloc(size * sizeof(Pixel));
16     section = malloc(matrix * matrix * sizeof(Pixel));
17     im = i->data;
18
19     for(j = 0; j < 3 * size; j+= 3){
20         array[k].explored = 0;
21         array[k].r = i->data[j];
22         array[k].g = i->data[j + 1];
23         array[k].b = i->data[j + 2];
24         k++;
25     }
26
27     nbmatX = sizeX / matrix;
28     if(sizeX % matrix != 0){
29         nbmatX = (sizeX / matrix) + 1;
30     }
31     nbmatY = sizeY / matrix;
```

```
32      if(sizeY % matrix != 0){
33          nbmatY = (sizeY / matrix) + 1;
34      }
35      for(int y = 0; y < nbmatY; y++){
36          for(int x = 0; x < nbmatX; x++){
37              int toget = (x * matrix) + sizeX * (y *
                    matrix);
38              section = getSection(array, toget, sizeX,
                    sizeY, section);
39              section = tachesDeCouleur(section, matrix,
                    matrix);
40              rewirteImage(im, section, toget, sizeX, sizeY
                    );
41          }
42      }
43      free(array);
44      free(section);
45  }
```

La fonction tachesInit prend comme argument l'Image que l'ont souhaite traiter.
Tout d'abord, des lignes 19 à 25, elle remplis un tableau de pixels avec les données de l'image : chaque composante est extraite en utilisant les décalages (1 pixel = 3 valeurs dans l'Image). On calcule ensuite le nombre de sous matrices nécessaires en largeur et en hauteur pour couvrir toute l'image.
Enfin, la boucle ligne 35 extrait le bloc de pixels correspondant à chaque matrice nécessaire à partir de son index de début (ligne 38) puis génère toutes les taches de couleur de ce bloc (ligne 39) et remplace les données de l'image par les données contenues dans la section obtenue (ligne 40).

3.4 getSection

Listing 3.4 – getSection

```
1   Pixel * getSection(Pixel * picture, int startPos, int
        sizeX, int sizeY, Pixel * section){
2       Pixel empty = {-1, -1, -1, -1};
3       int startX, startY, x, y, it;
4
5       it = 0;
6       startY = startPos / sizeX;
7       startX = startPos - (startY * sizeX);
8
9       for(y = startY; y < startY + matrixSize; y++){
```

```
10          for(x = startX; x < startX + matrixSize; x++){
11              if(x != startX && x >= sizeX){
12                  section[it] = empty;
13                  it++;
14              }
15              else if(y >= sizeY){
16                  section[it]= empty;
17                  it++;
18              }
19              else {
20                  section[it] = picture[x + sizeX * y];
21                  it++;
22              }
23          }
24      }
25      return section;
26  }
```

La fonction getSection prend comme argument :

- picture : un tableau de Pixel representant la totalité de l'image à traiter
- startpos : l'index du premier élément de la section à découper dans picture
- sizeX et sizeY : les dimensions de picture
- section : la section à remplire (un tableau de Pixel)

Elle retourne la section remplie avec les données correspondant.
On commence par convertir startpos en position 2D, plus faciles à manipuler (lignes 6 et 7). Puis, pour chaque element de picture entre la position de départ et la taille de notre matrice, on ajoute à la section le pixel courant (re converti en coordonnées 1D). Si la section ne couvre qu'en partie picture, on remplis l'index correspondant avec un pixel "vide".

3.5 tachesDeCouleur

Listing 3.5 – tachesDeCouleur

```
1  Pixel * tachesDeCouleur(Pixel * array, int sizeX, int
       sizeY){
2      int y, it, j, k;
3      int size = sizeX * sizeY;
4      for(y = 0; y < size; y++){
5        if(array[y].explored == 1){
6          continue;
7        }else{
```

```
 8           Tache * new_pound;
 9           new_pound = malloc(sizeof(Tache));
10           new_pound->limits = malloc(size * sizeof(int));
11
12           int x= 0;
13           while(x < size){
14             new_pound->limits[x] = -1;
15             x++;
16           }
17           it = 0;
18           explore(array, y, new_pound, &it, array[y],
                sizeX, sizeY);
19
20           new_pound->r = getAvgRed(array, new_pound->
                limits, size);
21           new_pound->g = getAvgGreen(array, new_pound->
                limits, size);
22           new_pound->b = getAvgBlue(array, new_pound->
                limits, size);
23
24           if(toFile(new_pound, size) == 1){
25             perror("Could not write this pound to output
                    file.");
26           }
27           k = 0;
28           for(j = 0; j < size; j++){
29             for(k = 0; new_pound->limits[k] != -1; k++){
30               if(j == new_pound->limits[k]){
31                 array[j].r = new_pound->r;
32                 array[j].g = new_pound->g;
33                 array[j].b = new_pound->b;
34               }
35             }
36           }
37           free(new_pound->limits);
38           free(new_pound);
39         }
40       }
41       return array;
42   }
```

La fonction tachesDeCouleur prends pour arguments :

- array : un bloc de Pixel quelconque
- sizeX et sizeY : les dimensions de ce bloc

Pour chaque pixel dans array, et si ce pixel n'est pas marqué comme exploré, la fonction créé une nouvelle tache dont les limites seront au pire des cas tous les pixels de array. Les limites sont initialisées à -1. Ligne 18, cette tache est ensuite étandue via la fonction explore. Des lignes 20 à 22, la couleur de la tache est recupèrée via les fonctions getAvgColor, puis sa couleur et son contenu sont ecrits dans un fichier via la fonction toFile. Enfin, ligne 28, array est ré écris avec les données de la tache pour remplacer les pixels de inclus dans la tache actuels par la valeur moyenne des 3 couleurs.

3.6 getAvg(Red/Green/Blue)

Listing 3.6 – tachesDeCouleur

```
1  GLubyte getAvgC(Pixel * array, int * indexs, int size){
2    int tmp = 0;
3    GLubyte avg = 0;
4    int i;
5    for(i = 0; indexs[i] != -1 && i < size; i++){
6      tmp += array[indexs[i]].C;
7    }
8    avg = tmp / i;
9
10   return avg;
11 }
```

Les fonctions getAvg calculent la couleur moyenne des pixels d'une tache pour pouvoir donner une valeur en rouge vert et bleu à cette dernière. Elles prennent en argument :

- array : un tableau de pixel quelconque
- indexs : un tableau d'indexes (les indexs des pixels appartenant a la tache en cours de traitement)
- size : la taille de array.

On parcours chaque element de indexs et on ajoute la composante de la couleur actuelle (ici, C qui est remplacé par r g ou b celon la fonction) du pixel se trouvant à cet index dans array à une variable temporaire. On divise ensuite cette variable par le nombre d'indexs stoqués dans le tableau pour obtenir la moyenne.

Chapitre 4

Sythèse & Guide d'utilisation

4.1 Sythèse

Pour résumer :

— Le programme est capable de détecter des taches de couleur et de les lisser
— Ces taches sont écrites dans un fichier "converted_img.txt" a la racine du projet.
— En revanche, je n'ai pas trouvé le moyen de limiter les taches à leur contour, le fichier obtenu est donc plus lourd que le fichier original, perdant l'aspect compression du projet. Ce point a été abordé en cours depuis et avec du temps, je devrai etre en mesure de pouvoir changer cela. C'est malheureusement quelque chose dont je ne dispose plus.
— Le bouton Load taches de couleur ne fonctionne pas totalement non plus et vous gratifiera d'une jolie segfault si vous l'actionnez ! Mais c'est une feature...
— De plus, à cause de problèmes d'architecture (approche recursive au lieu d'itérative) le programme ne peut pas traiter toute l'image en une seule fois et dois la découper en plusieurs morceaux ce qui entraine des problèmes d'uniformité des couleurs.

Bien qu'ayant conscience que mon projet ne répond que partiellement au sujet, je suis satisfait du résultat. Dans l'état atuel des choses, je pense etre en mesure de le terminer mais n'ai malheureusement plus le temps de m'en occuper avant les présentations. Je devrai donc me contenter de cette version incomplète.

4.2 Utilisation

— make pour compiler
— ./palette chemin/vers/image/source

— click gauche sur la fenetre puis Generate Taches de couleur
— pour modifier la taille des matrices de traitement, changer la valeur de la variable matrixSize dans modif.c puis recompiler. Note : une valeur au dessus de 400 risque de faire planter le programme et prendra environ 10 min (sur ma machine) pour venir à bout d'une image de 1024x768. Des valeurs plus raisonables comme 25 seront venues à bout d'une image de 2880x1620 en moins que ce laps de temps.
— pour changer la valeur du seuil a partir duquel un pixel est considèré différent de la référence, modifier la constante SEUIL dans ima.h puis recompiler.

Chapitre 5

Annexe

Lien vers le sujet StackOverflow qui m'a permis de comprendre d'origine de mes segfault dans le design original : stack.

Lien vers le dépot gihub de mon projet : github.

FIGURE 5.1 – Matrice 15x15

FIGURE 5.2 – Matrice 30x30

FIGURE 5.3 – Matrice 90x90

FIGURE 5.4 – Matrice 300x300

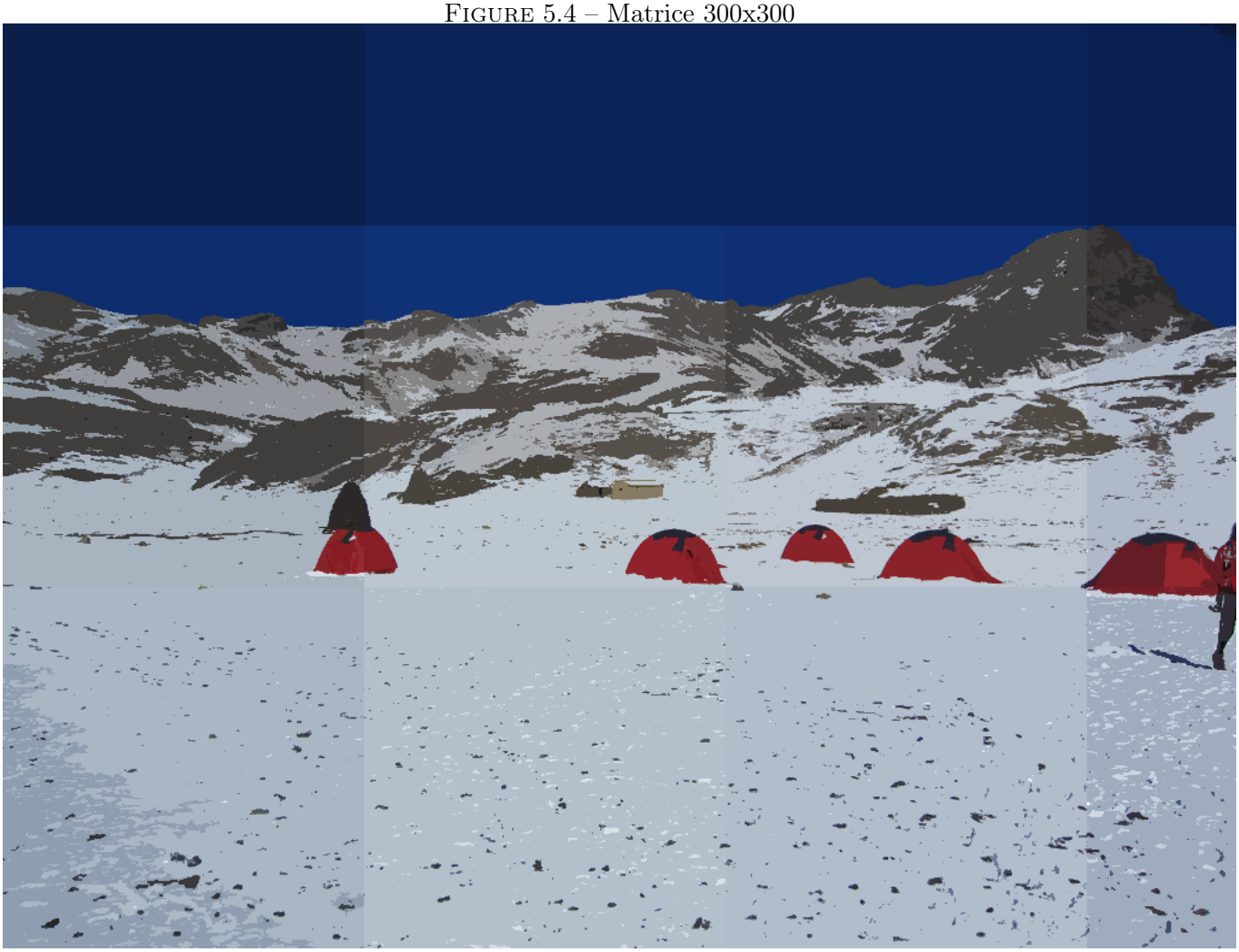

FIGURE 5.5 – Matrice 400x400