Compression d'images avec quadtrees

Anri Kennel* (L2-X)
Algorithmique et structures de données 2 · Université Paris 8

Année universitaire 2021-2022

Table des matières

*Numéro d'étudiant : 20010664

1 Brève présentation

J'ai réalisé le projet seul. Mon projet est de compresser une image avec une structure *quadtree*.

(a) Non compressé (3.6Mo)

(b) Compressé / 7 (56ko)

1.1 Objectifs

— Compiler facilement le programme
— Réussir à compresser l'image
— Code clair et commenté
— Ne pas utiliser OpenCV

2 Explications de la réalisation

Makefile

La compilation est simple avec le `Makefile`. Il est possible de faire `make` pour compiler avec la SDL et `O3`. Il est aussi possible de faire `make dev` et ainsi compiler avec la SDL et les flags de développement :

— `Wall` et `Wextra` pour les warnings
— `Wshadow` pour le nom des variables
— `pedantic` pour la compilation
— `g` pour Valgrind
— `Wold-style-cast` et `Wsign-conversion` pour bien utiliser les casts

Libraries

J'ai utilisé les librairies

— `fstream` pour vérifier empêcher d'écraser une image existante
— `SDL_image` pour utiliser `SDL_Surface` de la `SDL`
— `array` pour stocker et donner aux méthodes les 4 morceaux qui compose l'image quand divisé

Classe

Ma classe `QuadTree` est déclarée dans `includes/quadtree.hpp` et définie dans `src/quadtree.cpp`, elle contient :

— Une variable qui stocke la qualité de l'image (`niveau`)
— Une variable qui stocke la couleur majoritaire dans l'image (`couleur`)
— Une variable `std::pair` qui stocke les dimensions de l'image (`dimensions`)
— 4 variables représentant les enfants du noeud (`nord_ouest`, `nord_est`, `sud_ouest`, `sud_est`)
— Une variable qui définie si le noeud est final ou non (`final`)
— Une variable qui stocke le format utilisé par l'image (`format`)

— Une méthode qui permet de calculer la couleur majoraitaire dans l'image (`calculeCouleur`)
— Une méthode qui permet savoir lors de l'exportation si elle est finie (`verificationEgalitee`)
— Une méthode qui permet de séparer en 4 l'image (`coupeEnQuatre`)
— Une méthode qui permet de rassembler 4 morceaux d'image en une seule (`colleQuatreImages`)
— Une méthode qui permet d'exporter la surface avec un certain niveau de compression (`image`)

Cette classe permet de diviser récursivement l'image en 4 parts et d'en extraire la couleur qui y est majoritaire dans chaque morceaux d'image.

Constructeur

Dans le constructeur de ma classe, j'initialise `format` à `SDL_PIXELFORMAT_RGB888` au lieu du format de mon image (`image->format`) car il y a un *bug* dans la méthode `colleQuatreImages` qui préserve mal les couleurs. `SDL_PIXELFORMAT_RGB888` rend donc l'image en noir et blanc.

Surfaces

Je verrouille et déverrouille ma surface à chaque utilisation même si c'est probablement inutile mais ça m'évite de vérifier si `SDL_MUSTLOCK` est égale à 0 à chaque fois.

```
1  if(SDL_LockSurface(surface) == 0) {
2      /* ... */
3
4      SDL_UnlockSurface(surface);
5  }
```

Format des surfaces

Toutes les surfaces que je crées ont le même format, ça m'évite d'utiliser les masks en fonction de l'endian.

`calculeCouleur`

Pour calculer la couleur dans `calculeCouleur` je fait une moyenne `RGBA` de tout les pixels de la surface.

`verificationEgalitee`

Dans `verificationEgalitee` je regarde si tout les pixels `RGB` de la surface sont identiques (j'ignore le canal alpha).

`coupeEnQuatre`

Quand je coupe en quatre mon image dans `coupeEnQuatre`, je commence par définir les coordonnées de mes 4 morceaux (ici `s` est la surface mère) :

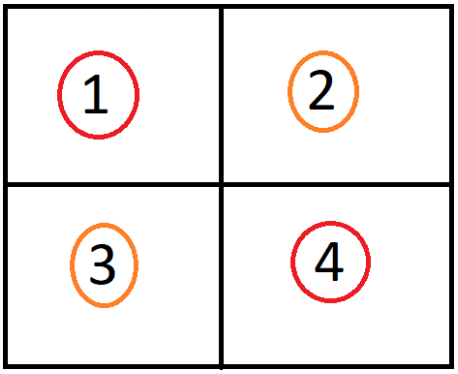
```
1  std::array<std::array<int, 4>, 4> coordonnes_quadrants;
2  coordonnes_quadrants[0] = {0       , 0       , s->w / 2, s->h / 2};
3  coordonnes_quadrants[1] = {0       , s->h / 2, s->w / 2, s->h    };
4  coordonnes_quadrants[2] = {s->w / 2, 0       , s->w    , s->h / 2};
5  coordonnes_quadrants[3] = {s->w / 2, s->h / 2, s->w    , s->h    };
```

Puis je créer tour à tour mes 4 surfaces qui je rajoute dans une `std::array`. Dans ses surfaces je vais recopier pixel-par-pixel de la grande surface vers la plus petite (ici `x`/`y` varient respectivement en fonction de la largeur/longueur du morceau d'image) :

```
1  int x1 = x * nouvelle_image->format->BytesPerPixel,
2      y1 = y * nouvelle_image->pitch,
3      x2 = (coordonnes_quadrants[i][0] + x) * s->format->BytesPerPixel,
4      y2 = (coordonnes_quadrants[i][1] + y) * s->pitch;
5  *reinterpret_cast<Uint32 *>(static_cast<Uint8 *>(nouvelle_image->pixels) + y1 + x1) =
6  *reinterpret_cast<Uint32 *>(static_cast<Uint8 *>(s->pixels) + y2 + x2);
```

`colleQuatreImages`

Quand je rassemble mon image dans `colleQuatreImages`, je commence par récupérer les dimensions de mon image originale en regardant des morceaux en diagonale (sur l'image je compare soit 1 et 4, soit 2 et 3).



Si les dimensions des 2 morceaux sont différents alors je prend la plus grande dimensions - `1`.

Une fois les dimensions récupérer, je copie pixel-par-pixel les morceaux sur ma grande surface, comme dans la méthode `coupeEnQuatre`. Je libère les morceaux de la mémoire une fois rassemblé.

3 Ajouts

— Possibilité de préciser le niveau de compresser (0 très compressé et 10 pas compressé)
— Utilisation de la SDL pour gérer l'image

3.1 Piste d'amélioration

Je n'ai malheureusement pas réussi à garder la couleur lorsque je compresse l'image, j'aimerais réussir à rendre ça fonctionnelle.