Licence informatique 3e année

Année 2016-2017

Projet PCA "Uniformisation d'images"

Stéphane Rubini

Le projet vise à développer un logiciel dont le but est de réduire la taille du fichier d'enregistrement d'une image, en s'appuyant sur une dégradation de sa qualité.

Le logiciel permet de visualiser la dégradation induite par une méthode d'uniformisation de l'image par moyennage des pixels voisins. Un groupe de pixels est remplacé par la valeur moyenne de ses membres si la dégradation, mesurée selon l'un des critères disponibles, est inférieure à une limite définie par l'utilisateur.

L'intérêt du logiciel s'appuie sur le postulat [1] qu'une image composée de nombreux pixels voisins de même couleur pourra être enregistrée dans un fichier de plus petite taille. La compression effective des images est prise en charge par la fonction d'enregistrement dans le format d'image ciblé (c'est à dire BMP pour le projet).

Trois critères d'évaluation de la dégradation seront proposés :

1. taille des zones uniformisées en nombres de pixels,
2. différence maximum entre les composantes de couleur des pixels couverts par une zone uniformisée,
3. zones uniformisées couvrant uniquement des pixels de faible intensité lumineuse (pour un texte "clair" sur fond "sombre").

La figure 1 montre l'interface graphique du logiciel.

L'image de gauche est l'original. Sa version uniformisée est celle de droite. Le critère utilisé pour contrôler le processus d'uniformisation est sélectionné par un groupe de trois boutons "radios". Le niveau d'uniformisation est réglé par un curseur d'ajustement. Ses valeurs minimales et maximales dépendent du critère utilisé. Sous chaque image est indiqué le nombre de zones de couleur distinctes.

Le menu déroulant `Fichier` permet de sélectionner une image à uniformiser, d'enregistrer l'image résultante au format BMP, et de quitter le logiciel. Le menu `Aide` permet d'afficher une fenêtre "à propos".

1. qu'il conviendrait de vérifier ...

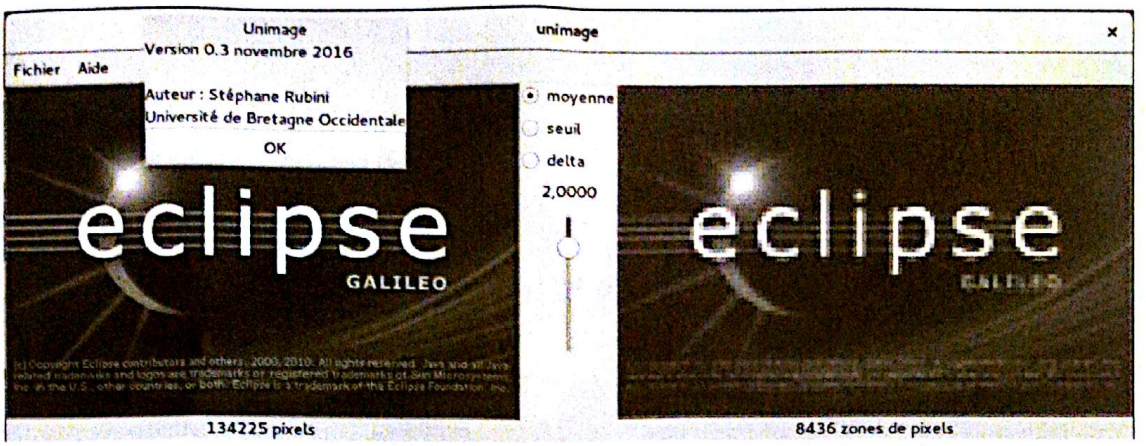

FIGURE 1 – Interface graphique du logiciel

Planning prévisionnel, organisation et notation

Le planning prévisionnel du déroulement du projet est décrit dans le tableau ci-dessous.

Séance	Taches	Durée
1	Présentation du projet	30 min
	Modules *pile*	3H
	Présentation de l'outil `gdb`	30 min
2	Module "zpixel"	3H
2 et 3	Construction de l'arbre des *zpixels*	2H
	Méthode d'évaluation de la dégradation	3H
4	GTK, présentation	30 min
	Conception de l'IHM	3H30
5 et 6	Intégration de l'ensemble	6H
6	Démonstration finale	2H
	Total	24H

Les trois premières étapes de développement seront validées par un test unitaire.

De plus, chaque étape fera l'objet d'une évaluation par l'enseignant. Le passage à l'étape suivante du développement est conditionné à une évaluation positive de la part de ce dernier.

gcc -g prog.c -o prog. → binaire executable + code C.

→ $ gdb ./prog.

→ run → lance prog

→ l affiche code.

→ break main / 10.

↑ ligne 10.

→ next executer une fonction.

→ step ~~avance si breakpoint~~ avance d'une instruction.

→ cont → jusqu'à prochain breakpoint

Séance 1

module pile Le module pile prend en charge une structure de données de type pile LIFO.

La pile enregistre des éléments de type `void *`. L'allocation des éléments accessibles à partir des pointeurs n'est pas de la responsabilité du module pile.

Le nombre maximum d'éléments est fixé au moment de la création de la pile.

Le module devra apporté les fonctionnalités minimales suivantes :

- création et destruction d'une pile,
- empilement et dépilement d'une valeur de type `void *`,
- nombre de places restantes et occupées,
- accès aux éléments de pile sans dépilement (sous la forme d'un itérateur, un seul pouvant être en cours simultanément),
- ré-initialisation de la pile.

L'interface public du module est donnée ci-dessous.

```
typedef struct pile pile_t;
pile_t * pile_creer(int nb_places);
        // retourne NULL en cas d'erreur
void pile_detruire(pile_t * p);
void pile_initialiser(pile_t * p);
int pile_places_occupees(const pile_t * p);
int pile_places_libres(const pile_t * p);
void * pile_depiler(pile_t * p);
        // retourne NULL si la pile est vide
int pile_depiler2(pile_t * p, void ** el);
        // retourne -1 si la pile est vide, 0 sinon
int pile_empiler(pile_t * p, const void * el);
        // retourne -1 si la pile est pleine, 0 sinon
void pile_initialiser_iterateur(pile_t * p);
int pile_obtenir_element_suivant(pile_t * p, void ** el);
        // retourne -1 si il n'y plus d'elements dans la pile, 0 sinon
```