TP 5 : manipulation d'images et *seam carving*

L'objectif est de vous faire travailler en C en manipulant des images.

FIGURE 1 : L'image exemple pour ce sujet

Quelques infos techniques à lire Dans ce TP, on va travailler avec des images en niveau de gris, représentés par des matrices de `uint8_t` (entiers à 8 bits non signés), où 0 représente un pixel noir et 255 pour un pixel blanc. Dans le fichier `seam_carving.h` se trouve un type représentant une telle image :

```c
struct image {
    uint8_t** at;
    int h;
    int w;
};

typedef struct image image;
```

Notez bien que le champ `at` est un pointeur vers la première case d'un tableau de pointeurs, chaque case de ce tableau pointant vers un tableau de `uint8_t`. `h` représente la hauteur de l'image et `w` sa largeur. Pour `image* im` une image, on accède donc à la valeur du pixel à la `i`ème ligne et la `j`ème colonne (en partant d'en haut à gauche) avec la syntaxe `im->at[i][j]`.
Dans la suite, on utilisera deux fonctions déjà présentes dans le fichier `squelette.c` dans lequel vous allez travailler. Ces fonctions sont de prototype :

```c
image* image_load(char* filename);
void image_save(image* im, char* filename);
```

La première permet de charger un fichier `.png` avec son adresse relative au dossier courant, et la seconde permet de sauvegarder une image de type `image*` sous le nom donné en argument dans le dossier courant. Des exemples d'utilisations sont donnés dans la fonction `main` dans le fichier `squelette.h`.
Les fonctions déjà présentes dans ce fichier sont celles demandées ici. Rien ne vous empêche d'écrire d'autres fonctions auxiliaires si vous en avez besoin, ou si elles permettent d'alléger votre code. Faites attention à

réfléchir à vos solutions avant de les implémenter : sur l'image que j'ai choisie pour illustrer, une durée de traitement de plus de 20-30 secondes veut sans doute dire qu'il y a une de vos fonctions qui est à optimiser. Enfin, avant de commencer : tous les fichiers doivent conserver la même structure que dans l'archive. Les fichiers `stb_*.h` servent à l'importation et exportation d'images, et le fichier `seam_carving.h` regroupe les prototypes des fonctions à écrire et les `#include <_>` nécessaires. On utilisera la ligne de compilation suivante :

```shell
gcc -O0 -Wall -Wextra -Wvla -fsanitize=address,undefined -lm -ldl squelette.c -o executable.exe
```

Compiler le fichier entier avec ces options peut prendre du temps quand on s'approche de la fin du sujet (jusqu'à 30 secondes chez moi, en remplaçant l'option `-O0` par `-O3`). Petit rappel / petites informations sur ces options :

- `-O?` définit un niveau d'optimisation de votre code, c'est-à-dire à quel point le compilateur prendra des libertés avec la structure de votre code pour produire un exécutable plus efficace. Les niveaux vont de 0 à 3, du moins optimisé (l'exécutable est presque identique au code que vous proposez) au plus optimisé : par exemple, l'optimisation des appels récursifs terminaux arrive au niveau 2. Attention : plus votre code est optimisé, plus les outils comme les sanitizers ont une sortie difficile à déterminer (par exemple, des appels de fonctions manquants dans la pile d'appel récursive affichée dans les erreurs sont incomplètes, car certains appels récursifs sont optimisés). Je vous conseille de commencer à `-O0` puis à augmenter une fois que votre code est dépourvu d'erreurs.
- les classiques `-Wall -Wextra -Wvla` qui activent tous les avertissements habituels. On pourrait rajouter `-Werror` qui empêche la compilation dès qu'il y a une erreur.
- les sanitizers `-fsanitize=address,undefined` détectant les accès mémoire et les comportements non définis, typiquement juste avant que l'OS le fasse et vous jette.
- `-lm` qui rajoute une librairie à la compilation (la syntaxe est en fait de la forme `-l<nom_librairie>` en général) : ici, cela concerne la librairie mathématique, utilisée par certaines fonctions déclarées dans les fichiers d'en-têtes.

Exercice 1 – Premières manipulations

▶ **Question 1** Compléter la fonction `image* image_new(int h, int w)` qui alloue (entièrement sur le tas) la mémoire nécessaire pour stocker un élément de type `image` et la mémoire nécessaire pour contenir la matrice `at` de cette image et renvoie un pointeur vers cette image nouvellement crée.

▶ **Question 2** Écrire la fonction `void image_free(image* im)` libérant toute la mémoire allouée sur le tas pour `im`.

Maintenant que l'on sait créer et libérer des éléments de type `image`, on peut faire quelques opérations élémentaires sur les images :

▶ **Question 3** Écrire une fonction `invert` inversant les pixels d'une image (le noir devient blanc, le blanc devient noir, le gris foncé devient gris clair, etc.).

```c
void invert(image* im);
```

▶ **Question 4** Écrire une fonction `binarize` qui convertit une image en noir et blanc en changeant chaque pixel pour la couleur la plus proche entre le noir et le blanc.

```c
void binarize(image* im);
```

▸ **Question 5** Écrire une fonction `filp_horizontal` qui opère une symétrie de l'image sur un axe vertical.

```c
void binarize(image* im);
```

Et pour tester ces fonctions, il suffit de compléter la fonction `main` du squelette et voir les images obtenues par l'exécutable !

FIGURE 2 : Image obtenue après `flip_horizontal`

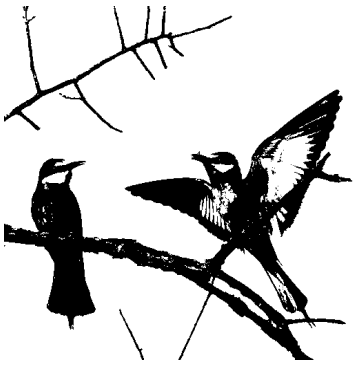

FIGURE 3 : Image obtenue après `binarize`

FIGURE 4 : Image obtenue après `invert`

Exercice 2 – Détection de bords

On va maintenant essayer de mettre en évidence les parties d'une image ayant la plus grande *énergie*, c'est-à-dire les parties avec de gros contrastes (un contour d'objet, par exemple). On définit l'énergie du pixel de coordonnées (i, j) par :

$$e_{i,j} = \frac{|p_{i,j+1} - p_{i,j-1}|}{2} + \frac{|p_{i+1,j} - p_{i-1,j}|}{2}$$

Cette formule n'a aucun sens aux bords de l'image. Dans ce cas, on utilise la formule suivante (qui est valable pour tous les pixels, au bord de l'image ou non) :

$$e_{i,j} = \frac{|p_{i,j_r} - p_{i,j_l}|}{j_r - j_l} + \frac{|p_{i_b,j} - p_{i_t,j}|}{i_b - i_t} \text{ avec } \begin{array}{ll} i_b = \min(i+1, h-1) & i_t = \max(i-1, 0) \\ j_r = \min(j+1, w-1) & j_l = \max(j-1, 0) \end{array}$$

Pour stocker l'énergie de tous les pixels d'une image, on va utiliser un type concret très proche de celui de type `image`. Cette fois-ci, à chaque coordonnée on associe un flottant contenant l'énergie du pixel associé.

```c
struct energy {
    double** at;
    int h;
    int w;
};

typedef struct energy energy;
```

▶ **Question 1** Écrire les fonctions de prototype `energy* energy_new(int h, int w)` `void energy_free(energy* e)` de rôles similaires aux fonctions des Questions 1 et 2.

▶ **Question 2** Écrire une fonction `void compute_energy(image* im, energy* e)` calculant la matrice d'énergies d'une image et l'écrivant dans l'argument `e`.

▶ **Question 3** Écrire une fonction `image* energy_to_image(energy* e)` prenant en entrée un tableau d'énergie et générant une image de mêmes dimensions, où un pixel d'énergie minimale sera représenté par un pixel noir et un pixel d'énergie maximale par un pixel blanc.

Dans la suite, l'objectif va être de *diminuer la largeur* d'images tout en conservant le plus de détails possibles : le ballon, les joueurs, etc. On va explorer dans la suite plusieurs approches pour faire cela : à chaque fois, on va écrire une stratégie pour retirer *une colonne* et itérer la même méthode jusqu'à obtenir l'image voulue.

Exercice 3 – Énergie minimale ligne par ligne

La première stratégie sera d'enlever, pour chaque ligne, le pixel de plus petite énergie.

▶ **Question 1** Écrire la fonction de prototype `void remove_pixel(uint8_t* line, double* e, int w)` prenant en entrée une ligne de pixels, la ligne d'énergie correspondante et leurs longueurs, qui supprime le pixel de plus petite énergie dans l'image en décalant vers la gauche les pixels suivants.

▶ **Question 2** En utilisant la fonction précédente, on va, à partir d'une image et de sa matrice d'énergie, retirer de l'image chaque pixel de plus petite énergie de chaque ligne. Implémenter cette stratégie en une fonction de prototype `void reduce_one_pixel(image* im, energy* e)` qui retire un pixel par colonne, puis `void reduce_pixels(image* im, int n)` appliquant n fois la fonction précédente. On fera attention à mettre à jour la valeur de `im->w` et `e->w`. Que remarque-t-on pour de grandes valeurs de n ?

Exercice 4 – Colonne d'énergie minimale

Pour simplifier les calculs, on va retirer une colonne entière de l'image en choisissant la colonne de plus petite énergie de l'image.

▶ **Question 1** Écrire une fonction de prototype `int best_column(energy* e)` retournant l'index de la colonne de plus basse énergie de l'image.

▶ **Question 2** En déduire une fonction `void reduce_one_column(image* im, energy* e)` qui enlève tous les pixels de la colonne de plus basse énergie de l'image, puis une fonction `void reduce_column(image* im, int n)` retirant n colonnes à la suite. On fera attention à mettre à jour les valeurs de `im->w` et `e->w`.

Exercice 5 – Seam carving

Cet algorithme récent cherche à réduire d'une colonne une image tout en préservant les détails importants. La solution trouvée est alors une sorte d'intermédiaire entre les stratégies des Exercices 3 et 4 : on va déterminer le *chemin de plus petite énergie* reliant un pixel de la première ligne à un pixel de la dernière en passant par exactement un pixel par ligne. Dans ce chemin, on ne peut se déplacer que vers en bas à gauche, en bas, ou en bas à droite.

Exemple 1

Dans l'image de 4×4 pixels suivante dont on a indiqué les énergies, on met en valeur un chemin possible d'énergie $1 + 2 + 2 + 1$. Notez que ce n'est pas un chemin de plus petite énergie, qui est d'énergie 3 :

1	**1**	0	3
4	1	**2**	4
1	2	**2**	1
4	**1**	1	0

▶ **Question 1** Écrire une fonction `void energy_min_path(energy* e)` prenant en entrée une matrice d'énergies et qui a pour effet de remplacer les valeurs de la matrice `e->at[i][j]` par l'énergie minimale des chemins partiels partant de la première ligne et allant jusqu'à la case de coordonnée (i, j) incluse.

Pour stocker un tel chemin, on pourra utiliser le type suivant :

```c
struct path {
    int* at; // tableau de taille size
    int size;
};

typedef struct path path;
```

▶ **Question 2** Écrire les fonctions `path* path_new(int n)` et `void path_free(path* p)` allouant et libérant sur le tas des chemins.

▶ **Question 3** Écrire la fonction `void compute_min_path(energy* e, path* p)` qui, à partir d'une matrice d'énergies des chemins partiels `e` (dans le sens de la Question 1) écrit dans `p` le chemin d'énergie minimale.

▶ **Question 4** Écrire la fonction `void reduce_seam_carving(image* im, int n)` qui retire successivement n colonnes de l'image `im`. On fera attention à mettre à jour au fur et à mesure des modifications la valeur de `im->w`.