TP Seam carving

Le but de ce TP est de réduire automatiquement la largeur d'une image sans toutefois changer la taille des zones les plus intéressantes de cette dernière. L'algorithme que nous allons implémenter, appelé *seam carving* est implémenté dans Photoshop sous le nom de *content aware scaling*.

1 Travailler avec des images

Dans ce TP, nous travaillerons avec des images monochrome 8 bits où chaque pixel possède un niveau de gris variant de 0, représentant le noir, à 255, représentant le blanc. Pour stocker cette valeur, nous utilisons donc le type entier non signé `uint8_t`.

Une image est donc une matrice de pixels que nous stockerons dans la structure suivante.

```
struct image {
    uint8_t **at;
    int height;
    int width;
};
typedef struct image image;
```

1. Écrire une fonction `image *image_new(int h, int w)` renvoyant un pointeur `im` vers une image allouée sur le tas de hauteur h et de largeur w. Si i et j sont des entiers tels que $0 \leqslant i < h$ et $0 \leqslant j < w$, la valeur du pixel de coordonnées (i, j) sera obtenue à l'aide de l'expression `im->at[i][j]`.
2. Écrire une fonction `void image_delete(image *im)` qui libère la mémoire allouée à l'image `im`.

Dans la suite, on utilisera les fonctions

`image *image_load(char *filename)` et `int image_save(image *im, char *filename)`

disponibles dans le fichier header `image.h`. Cette bibliothèque charge et sauve des images monochromes 8 bits au format PNG. Il est donc nécessaire d'ajouter l'option de compilation `-lpng`.

3. Écrire une fonction `void invert(image *im)` qui inverse les niveaux de gris de l'image, le noir devenant blanc et le blanc devenant noir.
4. Écrire une fonction `void binarize(image *im)` qui transforme tout pixel sombre (de valeur strictement inférieure à 128) en pixel noir et tout pixel clair en pixel blanc.
5. Écrire une fonction `void flip_horizontal(image *im)` qui effectue une symétrie de l'image par rapport à un axe vertical.



6. Écrire une fonction `image *reduce_half(image *im)` qui prend en entrée une image dont la hauteur et la largeur sont des nombres pairs et qui renvoie une nouvelle image dont la hauteur et la largeur ont été divisées par deux par rapport à l'image d'origine et dont la valeur de gris de chaque pixel est obtenue en moyennant les valeurs des 4 pixels correspondants dans l'image d'origine.

2 Détection de bords

Afin de détecter les contours des objets présents dans l'image, pour chaque pixel de coordonnées (i, j) n'étant pas sur le bord de l'image, on définit son énergie par

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

où $p_{i,j}$ est la valeur du pixel de coordonnées (i, j). Afin de prendre en compte les cas où l'on se trouve sur les bords de l'image, on définit plus généralement, pour tout $(i, j) \in [\![0, h[\![\times [\![0, w[\![$

$$e_{i,j} := \frac{|p_{i,j_r} - p_{i,j_l}|}{j_r - j_l} + \frac{|p_{i_b,j} - p_{i_t,j}|}{i_b - i_t} \quad \text{avec} \quad j_r := \begin{cases} j+1 & \text{si } j < w-1 \\ j & \text{sinon} \end{cases} \quad j_l := \begin{cases} j-1 & \text{si } j > 0 \\ j & \text{sinon} \end{cases}$$

$$i_b := \begin{cases} i+1 & \text{si } i < h-1 \\ i & \text{sinon} \end{cases} \quad i_t := \begin{cases} i-1 & \text{si } i > 0 \\ i & \text{sinon} \end{cases}$$

Afin de stocker les énergies des pixels de l'image, on définit enfin la structure

```
struct energy {
    double **at;
    int height;
    int width;
};
typedef struct energy energy;
```

1. Écrire une fonction `energy *energy_new(int h, int w)` renvoyant un pointeur `e` vers un tableau d'énergie de hauteur h et de largeur w, allouée sur le tas. Écrire la fonction `void energy_delete(energy *e)` permettant de libérer la mémoire correspondante.
2. Écrire une fonction `void compute_energy(image *im, energy *e)` prenant en entrée une image `im` et un tableau d'énergie `e` de même taille et remplissant ce tableau avec les données d'énergie de l'image.
3. Écrire une fonction `image *energy_to_image(energy *e)` prenant en entrée un tableau d'énergie et générant une image correspondante où un pixel d'énergie minimale sera représenté par un pixel noir et un pixel d'énergie maximale par un pixel blanc.

3 Deux approches naïves

Nous pouvons maintenant nous attaquer à notre problème qui consiste à réduire la largeur de l'image tout en conservant la taille des objets intéressants. Pour cela, nous allons retirer sur chaque ligne un pixel d'énergie minimale.

1. Écrire une fonction `void remove_pixel(uint8_t *line, double *e, int w)` prenant une ligne de pixels `line` de longueur `w` et une ligne d'énergies `e` correspondante et qui élimine un pixel d'énergie minimale, tout en décalant vers la gauche les pixels se trouvant après lui.
2. Écrire une fonction `void reduce_one_pixel(image *im, energy *e)` qui calcule d'abord l'énergie de chaque pixel, la stocke dans le tableau `e` puis retire un pixel sur chaque ligne de l'image. Le tableau `e` devra faire la même taille que l'image `im` et les valeurs qu'il contient initialement devront être ignorées. On veillera à diminuer `im->width` ainsi que `e->width` de 1.
3. Écrire une fonction `void reduce_pixel(image *im, int n)` qui retire n pixels par ligne à l'image `im` en itérant la fonction précédente. Testez cet algorithme sur les différentes images qui vous ont été fournies. Qu'en pensez-vous ?

Pour remédier à ce problème, nous allons enlever uniquement des pixels situés sur la même colonne.

4. Écrire la fonction `int best_column(energy *e)` qui prend un tableau d'énergie et qui renvoie la colonne dont la somme des énergies est minimale.
5. Écrire la fonction `void reduce_one_column(image *im, energy *e)` qui calcule l'énergie de chaque pixel puis réduit l'image en lui enlevant la colonne d'énergie minimale. Le tableau `e` devra faire la même taille que l'image `im` et les valeurs qu'il contient initialement devront être ignorées. On veillera à diminuer `im->width` ainsi que `e->width` de 1.
6. Écrire une fonction `void reduce_column(image *im, int n)` qui retire n colonnes à l'image `im` en itérant la fonction précédente. Testez cet algorithme sur les différentes images qui vous ont été fournies. Qu'en pensez-vous ?

4 Seam carving

L'idée de l'algorithme de *seam carving* est d'assouplir un peu la contrainte de réduction colonne par colonne. Pour cela, on définit un *chemin de pixels* comme une suite de pixels connectés, soit verticalement, soit en diagonale, dont le premier pixel est en haut de l'image et le dernier pixel est en bas de l'image. Ces chemins contiennent exactement un pixel par ligne. L'énergie d'un chemin est défini comme la somme des énergies des pixels le constituant. Par exemple, voici un chemin d'énergie 6 pour une image de 4 pixels par 4 pixels.

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

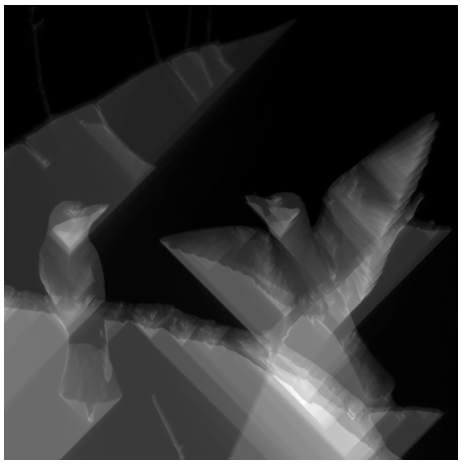
Afin de réduire l'image d'un pixel, on souhaite trouver puis enlever un chemin d'énergie minimale. Pour se faire, on définit un *chemin partiel* comme un chemin, sans la contrainte qu'il atteigne le bas de l'image. Afin de trouver un chemin d'énergie minimale, on va calculer pour chaque pixel, l'énergie minimale d'un chemin partiel terminant sur ce pixel. Par exemple, pour notre image de 4 pixels par 4 pixels dont le tableau des énergies a été donné plus haut, on obtient le tableau suivant.

1	1	0	3
5	1	2	4
2	3	3	3
6	3	4	3

1. Calculer à la main, le tableau des énergies minimales des chemin partiels pour le tableau d'énergie suivant.

2	1	1	0
3	3	2	2
2	0	1	2

2. Écrire une fonction `void energy_min_path(energy *e)` qui prend en entrée un tableau d'énergie de pixels et qui le transforme en un tableau des énergies minimales des chemins partiels.



On définit la structure

```
struct path {
    int *at;
    int size;
};
typedef struct path path;
```

utilisée pour stocker un chemin de l'image. Une telle structure aura une taille de la hauteur de l'image et `p->at[i]` désignera la colonne par laquelle le chemin passe à la ligne `i`.

3. Écrire une fonction `path *path_new(int n)` renvoyant un pointeur `p` vers un tableau de taille `n` alloué sur le tas. Écrire la fonction `void path_delete(path *p)` permettant de libérer la mémoire correspondante.
4. En déduire la fonction `void compute_min_path(energy *e, path *p)` prenant en entrée un tableau des énergies minimales des chemins partiels et un chemin `p` de même hauteur que le tableau des énergies et remplissant `p` avec le chemin d'énergie minimale.
5. Écrire enfin la fonction la fonction `void reduce_seam_carving(image *im, int n)` enlevant successivement n chemins d'énergie minimale dans l'image `im`. Testez votre fonction sur les différentes images fournies.