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 en niveaux de gris, où la valeur d'un pixel peut varier de 0 (pixel noir) à 255 (pixel 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 :

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

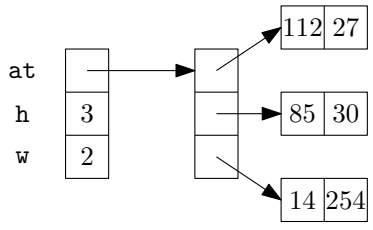

Figure XXIX.1 – Schéma mémoire pour une `image`.

Si `im` est un `image*`, on pourra donc utiliser `im->at[i]` pour accéder à la ligne `i` de l'image (qui est de type `uint8_t*`) et `im->at[i][j]` pour accéder à la valeur du pixel (i, j) (qui est de type `uint8_t`).

▶ **Question 1** Écrire une fonction `image_new` renvoyant un pointeur `im` vers une image allouée ayant la hauteur et la largeur spécifiées en paramètre. On fera bien attention aux problèmes d'*aliasing* : si le nombre d'appels à `malloc` que vous effectuez ne dépend pas de la hauteur de l'image, c'est que vous avez fait une erreur.

```
image *image_new(int h, int w);
```

▶ **Question 2** Écrire une fonction `image_delete` qui libère toute la mémoire associée à une image.

```
void image_delete(image *im);
```

Dans la suite, on utilisera les fonctions `image_load` et `image_save` disponibles dans le squelette. Ces fonctions permettent de charger et de sauvegarder une image dans un fichier au format `png` :

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

```
...
image *im = image_load("chemin/fichier.png");
// Do some processing on im...
image_write(im, "chemin/nouveau_fichier.png");
```

▶ **Question 3** Écrire une fonction `invert` qui inverse les niveaux de gris d'une image, le noir devenant blanc et le blanc devenant noir. Cette fonction travaillera en place en modifiant l'image.

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

▶ **Question 4** Écrire une fonction `binarize` qui transforme tout pixel sombre (de valeur strictement inférieure à 128) en pixel noir et tout pixel clair en pixel blanc.

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

▶ **Question 5** Écrire une fonction `flip_horizontal` qui effectue une symétrie de l'image par rapport à un axe vertical.

```
void flip_horizontal(image *im);
```



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 tous i, j vérifiant $0 \leqslant i < h$ et $0 \leqslant j < w$:

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

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

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

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

Pour alléger le code, on pourra (ce n'est pas indispensable) utiliser par la suite l'*opérateur ternaire* :

```
(cond) ? val1 : val2
```

Il s'agit d'une **expression** (et pas d'une instruction) qui vaut `val1` si `cond` est vraie, `val2` sinon. Essentiellement, c'est la même chose qu'un `if...then...else` en OCaml, mais plus limité (`cond`, `val1` et `val2` doivent être des expressions, et ne peuvent donc pas, par exemple, contenir une boucle `for`...).

```
int min(int x, inty){
  return (x <= y) ? x : y;
}
```

▶ **Question 6** Écrire une fonction `energy_new` renvoyant un pointeur `e` vers un tableau d'énergie de hauteur `h` et de largeur w, alloué sur le tas. Écrire également la fonction `energy_delete` permettant de libérer la mémoire correspondante.

```
energy *energy_new(int h, int w);
void energy_delete(energy *en);
```

▶ **Question 7** Écrire une fonction `compute_energy` 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.

```
void compute_energy(image *im, energy *en);
```

▶ **Question 8** Écrire une fonction `energy_to_image` 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.

```
image *energy_to_image(energy *en);
```

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.

▶ **Question 9** Écrire une fonction `remove_pixel` 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.

```
void remove_pixel(uint8_t *line, double *e, int w);
```

Remarque
Après l'appel, la case d'indice $w - 1$ de `line` pourra contenir une valeur quelconque.

▶ **Question 10** Écrire une fonction `reduce_one_pixel` ayant la spécification suivante :

Entrées : un pointeur `im` vers une structure `image`, un pointeur `en` vers une structure `energy`.

Préconditions : les dimensions du tableau image et du tableau énergie sont identiques. Les valeurs contenues dans le tableau énergie n'ont aucune importance.

Post-conditions :

- un pixel (d'énergie minimale) a été éliminé de chaque ligne de l'image ;
- les valeurs de `en->w` et `im->w` ont été décrémentées.

```
void reduce_one_pixel(image *im, energy *en);
```

▶ **Question 11** Écrire une fonction `reduce_pixels` qui retire à chaque ligne de l'image le nombre de pixels spécifié en entrée en itérant la fonction précédente. Tester cet algorithme sur les différentes images qui vous ont été fournies. Qu'en pensez-vous ?

```
void reduce_pixels(image *im, int n);
```

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

▶ **Question 12** Écrire la fonction `best_column` qui prend un tableau d'énergie et qui renvoie l'indice de la colonne dont la somme des énergies est minimale.

```
int best_column(energy *en);
```

▶ **Question 13** Écrire la fonction `reduce_one_column` 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->w` ainsi que `e->w` de 1.

```
void reduce_one_column(image *im, energy *en);
```

▶ **Question 14** Écrire la fonction `reduce_columns` qui itère la fonction précédente pour retirer `n` colonnes à l'image `im`. Testez cet algorithme sur les différentes images qui vous ont été fournies. Qu'en pensez-vous ?

```
void reduce_columns(image *im, int n);
```

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, contenant exactement un pixel de chaque ligne de l'image et commençant sur la ligne du haut. 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

▶ **Question 15** 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

▶ **Question 16** Écrire une fonction `energy_min_path` qui prend en entrée un tableau d'énergie et le transforme en un tableau des énergies minimales des chemins partiels.

```
void energy_min_path(energy *en);
```

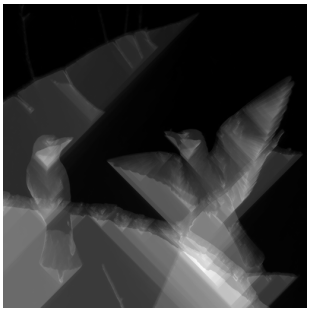

Figure XXIX.2 – Représentation graphique de l'énergie des chemins partiels.

On définit la structure suivante pour stocker un chemin de l'image :

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

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`.

▶ **Question 17** Écrire une fonction `path_new` renvoyant un pointeur vers une nouvelle structure `path`, contenant un tableau de la taille spécifiée, ainsi qu'une fonction `path_delete` libérant la mémoire associée à un `path`.

```c
path *path_new(int n);
void path_delete(path *p);
```

▶ **Question 18** Écrire la fonction `compute_min_path` prenant en entrée un tableau des énergies minimales des chemins partiels et un chemin `p` dont la taille correspond à la hauteur du tableau des énergies, et remplissant `p` avec le chemin d'énergie minimale.

```c
void compute_min_path(energy *en, path *p);
```

▶ **Question 19** Écrire enfin la fonction la fonction `reduce_seam_carving` enlevant successivement n chemins d'énergie minimale dans l'image `im`. Testez votre fonction sur les différentes images fournies.

```c
void reduce_seam_carving(image *im, int n);
```

L'algorithme présenté ici a été inventé en 2007 par Shai Avidan et Ariel Shamir. Ce sujet est basé sur le travail de Mickaël Péchaud (Lycée Joffre), adapté en C par François Fayard (Les Lazaristes).

Solutions

▶ **Question 1** Attention à bien allouer h lignes !

```
image *image_new(int h, int w) {
    image *im = malloc(sizeof(image));
    im->at = malloc(h * sizeof(int8_t *));
    for (int i = 0; i < h; i++) {
        im->at[i] = malloc(w * sizeof(int8_t));
    }
    im->h = h;
    im->w = w;
    return im;
}
```

▶ **Question 2** Le nombre d'appels à `free` doit correspondre au nombre d'appels à `malloc`.

```
void image_delete(image *im) {
    for (int i = 0; i < im->h; i++) {
        free(im->at[i]);
    }
    free(im->at);
    free(im);
}
```

▶ **Question 3**

```
void invert(image *im) {
    for (int i = 0; i < im->h; i++) {
        for (int j = 0; j < im->w; j++) {
            im->at[i][j] = 255 - im->at[i][j];
        }
    }
}
```

▶ **Question 4**

```
void binarize(image *im) {
    for (int i = 0; i < im->h; i++) {
        for (int j = 0; j < im->w; j++) {
            im->at[i][j] = im->at[i][j] < 128 ? 0 : 255;
        }
    }
}
```

▶ **Question 5** L'un des rares cas où je vous conseille une boucle `while` même si un `for` est possible :

```c
void flip_horizontal(image *im) {
    for (int i = 0; i < im->h; i++) {
        int jl = 0;
        int jr = im->w - 1;
        while (jl < jr){
            uint8_t value = im->at[i][jl];
            im->at[i][jl] = im->at[i][jr];
            im->at[i][jr] = value;
            jl++;
            jr--;
        }
    }
}
```

▶ **Question 6**

```c
energy *energy_new(int h, int w) {
    energy *e = malloc(sizeof(energy));
    e->at = malloc(h * sizeof(double *));
    for (int i = 0; i < h; i++) {
        e->at[i] = malloc(w * sizeof(double));
    }
    e->h = h;
    e->w = w;
    return e;
}

void energy_delete(energy *e) {
    for (int i = 0; i < e->h; i++) {
        free(e->at[i]);
    }
    free(e->at);
    free(e);
}
```

▶ **Question 7**

```c
void compute_energy(image *im, energy *e) {
    for (int i = 0; i < im->h; i++) {
        int it = (i > 0) ? i - 1 : i;
        int ib = (i < im->h - 1) ? i + 1 : i;
        for (int j = 0; j < im->w; j++) {
            int jl = (j > 0) ? j - 1 : j;
            int jr = (j < im->w - 1) ? j + 1 : j;
            double delta_i = (double)(im->at[it][j]) - (double)(im->at[ib][j]);
            double delta_j = (double)(im->at[i][jr]) - (double)(im->at[i][jl]);
            e->at[i][j] = fabs(delta_i) / (ib - it) + fabs(delta_j) / (jr - jl);
        }
    }
}
```

▶ **Question 8** Il faut commencer par déterminer les valeurs extrêmes de l'énergie, puis effectuer une transformation affine qui envoie le minimum sur zéro et le maximum sur 255.

```
image *energy_to_image(energy *e) {
    int h = e->h;
    int w = e->w;
    double v_min = e->at[0][0];
    double v_max = e->at[0][0];
    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            if (e->at[i][j] < v_min) {
                v_min = e->at[i][j];
            }
            if (e->at[i][j] > v_max) {
                v_max = e->at[i][j];
            }
        }
    }
    image *im = image_new(h, w);
    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            double v = e->at[i][j];
            im->at[i][j] = (uint8_t) (255 * (v - v_min) / (v_max - v_min));
        }
    }
    return im;
}
```

► **Question 9** On fait exactement ce que nous demande l'énoncé (en particulier, on ne modifie donc pas le tableau d'énergie).

```
void remove_pixel(uint8_t *line, double *e, int w) {
    double e_min = e[0];
    double j_min = 0;
    for (int j = 1; j < w; j++) {
        if (e[j] < e_min) {
            e_min = e[j];
            j_min = j;
        }
    }
    for (int j = j_min; j < w - 1; j++) {
        line[j] = line[j + 1];
    }
}
```

► **Question 10**

```
void reduce_one_pixel(image *im, energy *e) {
    compute_energy(im, e);
    for (int i = 0; i < im->h; i++) {
        remove_pixel(im->at[i], e->at[i], im->w);
    }
    im->w--;
    e->w--;
}
```

▶ **Question 11** On alloue un seul tableau d'énergie, que l'on réutilise pour toutes les étapes :

```
void reduce_pixels(image *im, int n) {
    energy *e = energy_new(im->h, im->w);
    for (int k = 0; k < n; k++) {
        reduce_one_pixel(im, e);
    }
    energy_delete(e);
}
```

Le résultat est catastrophique. Voilà ce qu'on obtient en réduisant de 100 pixels :

▶ **Question 12** Pas de difficulté particulière :

```
int best_column(energy *e) {
    double e_min;
    double j_min = -1;
    for (int j = 0; j < e->w; j++) {
        double e_col = 0.0;
        for (int i = 0; i < e->h; i++) {
            e_col += e->at[i][j];
        }
        if (j_min == -1 || e_col < e_min) {
            e_min = e_col;
            j_min = j;
        }
    }
    return j_min;
}
```

▶ **Question 13**

```
void reduce_one_column(image *im, energy *e) {
    compute_energy(im, e);
    int j_min = best_column(e);
    for (int i = 0; i < im->h; i++) {
        for (int j = j_min; j < im->w - 1; ++j) {
            im->at[i][j] = im->at[i][j + 1];
        }
    }
    im->w--;
    e->w--;
}
```

▶ **Question 14**

```c
void reduce_columns(image *im, int n) {
    energy *e = energy_new(im->h, im->w);
    for (int k = 0; k < n; k++) {
        reduce_one_column(im, e);
    }
    energy_delete(e);
}
```

Le résultat est nettement meilleur, mais l'on introduit quand même des discontinuités visibles (dans les branches entre les deux oiseaux) :

▶ **Question 15** On obtient :

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

▶ **Question 16**

```c
void energy_min_path(energy *e) {
    for (int i = 1; i < e->h; i++) {
        for (int j = 0; j < e->w; j++) {
            int jl = j > 0 ? j - 1 : j;
            int jr = j < e->w - 1 ? j + 1 : j;
            double e_min = e->at[i - 1][jl];
            for (int jj = jl + 1; jj <= jr; jj++) {
                if (e->at[i - 1][jj] < e_min) {
                    e_min = e->at[i - 1][jj];
                }
            }
            e->at[i][j] += e_min;
        }
    }
}
```

▶ **Question 17**

```c
path *path_new(int n) {
    path *p = malloc(sizeof(path));
    p->at = malloc(n * sizeof(int));
    p->size = n;
    return p;
}

void path_delete(path *p) {
    free(p->at);
    free(p);
}
```

► **Question 18** On commence par déterminer l'extrémité inférieure du chemin, puis l'on remonte, en sachant que l'une des trois (ou deux) cases situées juste au-dessus du point actuel permet d'étendre le chemin.

```c
void compute_min_path(energy *e, path *p) {
    int h = e->h;
    double e_min = e->at[h - 1][0];
    int j_min = 0;
    for (int j = 1; j < e->w; j++) {
        if (e->at[h - 1][j] < e_min) {
            e_min = e->at[h - 1][j];
            j_min = j;
        }
    }
    p->at[h - 1] = j_min;
    int j = j_min;
    for (int i = h - 2; i >= 0; i--) {
        int jl = j > 0 ? j - 1 : j;
        int jr = j < e->w - 1 ? j + 1 : j;
        e_min = e->at[i][jl];
        j_min = jl;
        for (int jj = jl + 1; jj <= jr; jj++) {
            if (e->at[i][jj] < e_min) {
                e_min = e->at[i][jj];
                j_min = jj;
            }
        }
        p->at[i] = j_min;
        j = j_min;
    }
}
```

► **Question 19** Le plus gros du travail a été fait :

```c
void reduce_seam_carving(image *im, int n) {
    energy *e = energy_new(im->h, im->w);
    path *p = path_new(im->h);
    for (int k = 0; k < n; k++) {
        compute_energy(im, e);
        energy_min_path(e);
        compute_min_path(e, p);
        for (int i = 0; i < im->h; i++) {
            for (int j = p->at[i]; j < im->w - 1; ++j) {
                im->at[i][j] = im->at[i][j + 1];
            }
        }
        im->w--;
        e->w--;
    }
    energy_delete(e);
    path_delete(p);
}
```

Le résultat est de très bonne qualité :