TP n°7

Graphismes et compilation

Vous aurez besoin pour ce TP du module `Graphics` (*c.f.* le TP precedent) et : de la librairie `camlimages` si elle est installee sur votre machine ; du chier `ppm.ml` de la page du projet sinon.

Chargement et sauvegarde d'images avec camlimages.

Le chargement de la librairie `camlimages` dans l'interpreteur se fait en ecrivant :

```
#use "topfind";;
#require "camlimages";;
```

ou eventuellement, selon la version de camlimages installee, en ecrivant :

```
#use "topfind";;
#require "camlimages.all_formats";;
#require "camlimages.graphics";;
```

Avec cette librairie, les images pourront être chargees en memoire a l'aide de la fonction suivante, prenant en argument un nom de chier avec son chemin d'acces complet :

```
let load filename = Graphic_image.array_of_image (Images.load filename []);;
```

La sauvegarde pourra se faire a l'aide de la fonction suivante prenant en argument un nom de chier (avec son chemin complet et son extension) et la representation en memoire d'une image :

```
let save filename m =
  Images.save filename None [Images.Save_Quality 255]
    (Images.Rgb24(Graphic_image.image_of (Graphics.make_image m)))
;;
```

Chargement et sauvegarde sans camlimages.

Suivez les directives (1), (2) et (3) de la derniere section de la page du projet. Les fonctions a utiliser pour charger et sauvegarder une image (uniquement au format `ppm`) sont `Ppm.load` et `Ppm.save`.

Accès au contenu des images

Les deux fonctions de chargement ci-dessus renvoient une image sous la forme d'une matrice de type `Graphics.color array array`. Une telle matrice `m` est un tableau dont les elements sont des tableaux de même longeur, chaque element representant une ligne de pixels. Pour a cher cette image dans la fenêtre graphique deja ouverte, il su t d'ecrire :

```
Graphics.draw_image (Graphics.make_image m) 0 0
```

La hauteur et largeur de l'image sont respectivement egales a `Array.length m` et `Array.length (m.(0))`. Le pixel de coordonnees (i, j) est encode par `m.(j).(i)` { ligne n^o j, colonne n^o i, noter l'inversion. Le type `Graphics.color` de `m.(j).(i)` est en fait le type `int` : les composantes r, g, b d'un pixel (entre 0 et 255) sont encodees par l'entier ($256 \times 256 \times r + 256 \times g + b$). Ces trois composantes peuvent être extraites en de nissant :

```
let color_to_rgb c =  (c asr 16, (c asr 8) land 255, c land 255);;
```

et en ecrivant par exemple, dans une autre fonction :

```
let (r,g,b) = color_to_rgb m.(j).(i) in ...
```

Inversement, la fonction `Graphics.rgb : int -> int -> int -> Graphics.color` permet de construire l'entier representant une couleur a partir de ses composantes r, g, b, et de rea ecter un element de `m` :

```
m.(j).(i) <- Graphics.rgb r g b; (* action, de type unit *)
```

2. Transformations locales

Exercice 2. A l'aide de deux boucles `for`, ecrire une fonction generique

```
filter : ('a -> 'a) -> 'a array array -> unit
```

telle que `filter f m` donne a chaque `m.(j).(i)` la valeur de `(f m.(j).(i))`. Cette fonction peut être utilisee pour transformer chacun des pixels d'une image. Pour de nir par exemple une fonction isolant la composante rouge de chaque pixel (Figure ??), on peut ecrire :

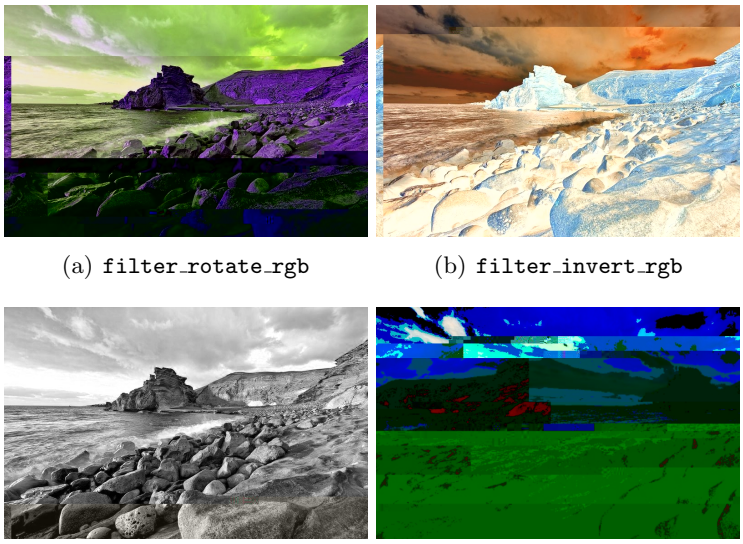
```
let isoler_rouge m =
 (* definition du traitement *)
 let f =
   (fun p ->
         let (r,g,b) = color_to_rgb p in     (* on extrait les composantes *)
         Graphics.rgb r 0 0)                 (* on ne garde que le rouge   *)

 (* appel effectif *)
 in filter f m
;;
```

(a) image d'origine (b) après `isoler_rouge`

FIGURE 1 { Application de la fonction `filter` de l'exercice ??

Exercice 3. A l'aide de la fonction precedente, ecrire les fonctions suivantes :

- { `filter_rotate_rgb : Graphics.color array array -> unit`
  transformant, pour chaque pixel, ses composantes (r, g, b) en (g, b, r)
- { `filter_invert_rgb : Graphics.color array array -> unit`
  transformant, pour chaque pixel, ses composantes (r, g, b) en $(255 - r, 255 - g, 255 - b)$
- { `filter_to_bw : Graphics.color array array -> unit`
  transformant, pour chaque pixel, ses composantes (r, g, b) en (c, c, c), ou c est la plus grande des trois composantes.
- { `filter_low_cut_rgb : int -> Graphics.color array array -> unit`
  telle que `filter_low_cut_rgb v m` transforme, pour chaque pixel, ses composantes (r, g, b) en (r', g', b'), ou pour chaque composante c, la composante c' vaut c si $c \geq v$, et 0 sinon.



(a) `filter_rotate_rgb` (b) `filter_invert_rgb`

(c) `filter_to_bw` (d) `filter_low_cut_rgb 180`

FIGURE 2 { Transformations sur place de l'exercice ??

3. Transformations globales

Exercice 4. Ecrire une fonction `sweep`, telle que pour toute representation d'image `m`, l'evaluation de `sweep g m` e ectue les operations suivantes :

- { creer en memoire une representation d'image (noire) `m'` de même taille que `m`,
- { pour chaque `j`, `i`, donner a `m'.(j).(i)` la valeur de `(g j i m)`,
- { puis, une fois ce traitement termine, a nouveau pour chaque `j`, `i`, donner a `m.(j).(i)` la valeur de `m'.(j).(i)`.

Exercice 5. A partir de la fonction precedente, ecrire :

- { `filter_pixelise : int -> Graphics.color array array -> unit` telle que `filter_pixelise v m` e ectue sur sa matrice temporaire `m'` le traitement suivant : `m'.(j).(i)` recoit la valeur de `m.((j/v) * v).((i/v) * v)`.
- { `filter_flip_h : Graphics.color array array -> unit` inversant l'image par symetrie d'axe vertical,
- { `filter_flip_v : Graphics.color array array -> unit` inversant l'image par symetrie d'axe horizontal.

Exercice 6. Un peu plus di cile, ecrire :

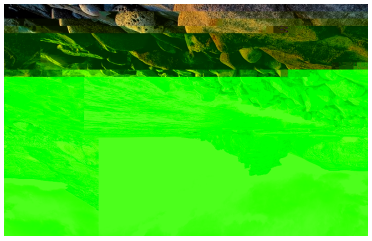
```
filter_blur : int -> Graphics.color array array -> unit
```

telle que `filter_blur v m` e ectue sur sa matrice temporaire `m'` le traitement suivant.

S'il y a moins de `v` pixels au dessus, en dessous, a gauche ou a droite de celui encode par `m.(j).(i)`, alors `m'.(j).(i)` recoit une valeur nulle ($i.e.$ devient noir). Sinon, pour chaque composante c parmi r, g, b, on calcule la moyenne c_0 des composantes des pixels encodes par `m.(l).(k)` pour `k` compris entre `i - v` et `i + v` et pour `l` compris entre `j - v` et `j + v` (servez-vous de deux boucles imbriquees et de trois references - il y a `(2*v + 1) * (2*v + 1)` pixels en tout). La valeur donnee a `m'.(j).(i)` est celle de l'encodage de (r_0, g_0, b_0).

(a) `filter_pixelise 20`

(b) `filter_flip_h`



(c) `filter_flip_v`

FIGURE 3 { Transformations par balayage de l'exercice ??

FIGURE 4 { Apres deux iterations de `filter_blur 8`

4. Compilation

Le but de cette derniere partie est de creer a partir du code ecrit precedemment un executable prenant en argument un nom de chier d'image, chargeant cette image en memoire, e ectuant une transformation d'image speci ee par une ou plusieurs options, puis sauvegardant l'image transformee { eventuellement sous un autre nom.

Si les librairies `camlimages` et `ocamlbuild` sont correctement installees sur votre machine, la compilation de votre chier source en executable devra se faire via `ocamlbuild`, suivant les directives fournies sur la page du projet, avec le même chier `_tags` { sinon, suivez la directive 4 (avec `ocamlbuild`) ou 5 (sans `ocamlbuild`) de la derniere section.

Commentez dans votre chier toutes les directives (en `#`) et tous les tests de fonctions. Ajoutez ensuite les elements necessaires pour lire la ligne de commande ($c.f$ le cours n^o 9 et, si necessaire, la documentation du langage). Il vous faudra choisir des noms d'options et determiner la maniere dont votre programme interprete ses arguments : ltre choisi, valeur passee lorsque le ltre en necessite une, nom de l'image a produire, etc. L'executable pourra par exemple être invoque de la maniere suivante :

```
$ mon_programme -f pixelise -v 20 -o image_pixelisee.jpg image_source.jpg
```