Informatique – MP Faidherbe

TP2 : Traitement d'Image

Corrigé

Alix Goguey - alix.goguey@gmail.com

année 2012-2013

La représentation d'une image sera une liste de tiplet (R, G, B) dont le premier élément sera $(-1, largeur, hauteur)$. Cette représentation n'est pas la plus naturelle ni la plus efficace mais elle permet de manipuer la récursion.

Une représentation plus naturelle est une matrice de triplet ou un triplet de matrice.

1 Introduction

Vous avez déjà tous ouvert au minimum 20 000 fois une photo ou une image sur un ordinateur. En y réflechissant ça peut parraître magique ces petits 0 et 1 qui se transforment en couleur et en forme. Et bien aujourd'hui, c'est vous qui transformerez cette information textuelle en information visuelle. Et vous verrez c'est hyper simple !

Dans un deuxième temps nous verrons comment modifier les images que vous afficherez avec des filtres qui améliorent ou embellissent simplement l'image.

2 Ouvrir/Sauvegarder une image

Pour simplifier les choses, des fichiers *images* vous sont fournis. La première ligne correspond à la largeur de l'image, la deuxième à la hauteur. Le reste du fichier sont les couleurs de tous les pixels. Chaque pixel est représenté par un triplet $(R, G, B) \in [\![0; 255]\!]^3$. Vous interpréterez les lignes suivantes du fichier 3 par 3. Le sens de parcours de l'image est du coin haut gauche au coin bas droite (dans le sens de lecture).

```
(* Exemple de lecture d'un fichier *)

let read_int f =                    (* recupere une ligne au *)
  int_of_string (input_line f)      (* format string et la   *)
;;                                  (* transforme en int     *)

let f = open_in "test1.txt" ;;      (* ouvre un fichier      *)
let w = read_int f ;;
let h = read_int f ;;
close_in f ;;                       (* ferme un fichier      *)
```

Exercice 1 : Implémentez la fonction `loadImg` qui lit un fichier `file` au format décrit et renvoie l'image dans une structure de données que vous choisirez. La fonction prendra en paramètre un nom de fichier.

```
let loadImg file =
  let f = open_in file in
  let w = read_int f in
  let h = read_int f in
  let rec read_line f n =
    match n with
    | 0 -> []
    | _ -> let r = read_int f in
           let g = read_int f in
           let b = read_int f in
           (r,g,b)::(read_line f (n-1))
  in
  let l = read_line f (w*h) in
  close_in f;
  (-1,w,h)::l
;;
```

```
(* Exemple d'ecriture dans un fichier *)

let write_int f i =
  output_string f (string_of_int i);  (* ecrit un int     *)
  output_string f "\n"                (* dans le fichier  *)
;;

let f = open_out "test2.txt" ;;       (* cree un fichier  *)
write_int f 34 ;;
close_out f ;;                        (* ferme le fichier *)
```

Exercice 2 : Implémentez la fonction `saveImg` qui sauvegarde dans un fichier `file` une image `img` au format de votre structure de données. La fonction prendra en paramètre une image et un nom de fichier.

```
let saveImg img file =
  let f = open_out file in
  let ign,w,h = List.hd img in
  let rec write_file f img =
    match img with
    | [] -> close_out f
    | (r,g,b)::t -> write_int f r;
                    write_int f g;
                    write_int f b;
                    write_file f t
  in
  write_int f w;
  write_int f h;
  write_file f (List.tl img)
;;
```

3 Afficher une image

Sur un écran, les pixels peuvent être représentés par une *matrice* de triplet (R, G, B). Afficher une image n'est ni plus ni moins remplir cette *matrice*. Pour afficher les images, nous utiliserons une *matrice* de pixel blanc $(255, 255, 255)$ fournie par le package `graphics` de OCaml. Cette *matrice* sera notre feuille blanche que nous modifierons en fonction des affichages à faire.

```
(* Creation d'une matrice de blanc 640 par 600 *)
#load "graphics.cma";;
open Graphics;;
open_graph "640x600";;
```

Voici quelques instructions utiles :

```
(* re-initialisation de tous les pixels *)
clear_graph();;
(* changement de la couleur courante *)
set_color (rgb r g b);;
(* affiche la couleur courante sur le pixel
   de coordonnee (x,y) *)
plot x y;;
```

Attention !! L'axe y du repère de l'image est inversé (sens : bas vers le haut). Pour plus de primitives : `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Graphics.html`

Exercice 3 : Implémentez la fonction `display` qui prend en paramètre une image `img`, une abscisse `x` et une ordonnée `y` affichant l'image à partir du point (x, y) (*i.e.* (x, y) est le coin bas gauche de l'image).

```
clear_graph();
let display img x y =
  let ign,w,h = List.hd img in
  let rec drawPixel im n w h x y =
    match img with
    | [] -> moveto 0 0
    | (r,g,b)::t -> set_color (rgb r g b);
                    plot ((n mod w)+x) (h-(n/w)+y-1);
                    drawPixel t (n+1) w h x y
  in
  drawPixel (List.tl img) 0 w h x y
;;
```

4 Traitement d'image

Traiter une image c'est lui appliquer un algorithme. En général (et ce sera notre cas), l'image résultante garde la même dimension. Seule les couleurs sont modifiées. Afin de visualiser plus facilement l'effet d'un traitement, il vaut mieux afficher à la fois l'image originale et l'image traitée.

Exercice 4 : Implémentez la fonction `display` qui prend en paramètre 2 images `img1` et `img2` (ayant la même taille), une abscisse `x` et une ordonnée `y` affichant les 2 images côte à côte à partir du point (x, y).

Si vous êtes rapide, vous pouvez modifier vos fonctions `display` et ajouter un paramètre `offset` ajoutant une bordure de la couleur de votre choix autour de vos affichages. Vous pouvez aussi modifier la fonction `displayimg1img2xy` pour quelle puisse afficher côte à côte 2 images de tailles différentes.

```
let display img1 img2 x y offset =
  clear_graph();
  let ign,w,h = List.hd img1 in
  set_color (rgb 0 0 0);
  fill_rect (x-offset) (y-offset) (2*(w+offset)) (2*(h+offset));
  display img1 x y;
  display img2 (x+w) y;
;;
```

4.1 Niveau de gris

Une image en niveau de gris est une image couleur où un pixel est représenté par un triple (r, g, b) tel que $r = g = b$. Pour transformer une image couleur en niveau de gris il suffit de remplacer le triplet (r, g, b), où les valeurs ne sont pas forcément égales, par le triplet (t, t, t) tel que $t = 0,2 \times r + 0,7 \times g + 0,1 \times b$.

Exercice 5 : Implémentez la fonction `grey` prenant l'image `img` et renvoyant l'image en niveau de gris.

```
let greyRGB r g b =
  let rf = float_of_int r in
  let gf = float_of_int g in
  let bf = float_of_int b in
  int_of_float (0.2*.rf+.0.7*.gf+.0.1*.bf)
;;

let grey img =
  let ign,w,h = List.hd img in
  let rec aux img =
    match img with
    | [] -> []
    | (r,g,b)::t -> let val = greyRGB r g b in
                    (val,val,val)::(aux t)
  in
  (ign,w,h)::(aux (List.tl img))
;;
```

4.2 Normalisation d'une image

Une image vous paraissant noire (ou de couleur unie) n'est pas forcément à jeter. Il est possible que les pixels aient des valeurs tellement proches que nous ne percevons quasiment pas la différence. Regardez bien la figure 1, elle présente quelques nuances de noir. Une manière de remédier à cela est de recentrer les intervalles de valeur sur la zone utilisée grâce à la formule 1. Cela s'appelle une normalisation.

$$Val(x,y) := \frac{(Val(x,y) - \min\Big(Val(x,y)|\forall(x,y) \in Img\Big)) \times 255}{\max\Big(Val(x,y)|\forall(x,y) \in Img\Big) - \min\Big(Val(x,y)|\forall(x,y) \in Img\Big)} \quad (1)$$

Exercice 6 : Implémentez la fonction `normalisation` prenant l'image `img` et renvoyant l'image normalisée. Note : le principe est décrit avec une valeur unique, le niveau de gris, cependant il peut s'appliquer sur les canaux R, G et B séparément puis rassembler les valeurs trouvées.

FIGURE 1: Image à normaliser

```
let max_RGB img =
  let rec aux img mR mG mB =
    match img with
    | [] -> (mR,mG,mB)
    | (r,g,b)::t -> let auxR = ref mR in
                    if r>mR then
                      auxR := r;
                    let auxG = ref mG in
                    if g>mG then
                      auxG := g;
                    let auxB = ref mB in
                    if b>mB then
                      auxB := b;
                    aux t (!auxR) (!auxG) (!auxB)
  in
  aux (List.tl img) 0 0 0
;;
```

```
let min_RGB img =
  let rec aux img mR mG mB =
    match img with
    | [] -> (mR,mG,mB)
    | (r,g,b)::t -> let auxR = ref mR in
                    if r<mR then
                      auxR := r;
                    let auxG = ref mG in
                    if g<mG then
                      auxG := g;
                    let auxB = ref mB in
                    if b<mB then
                      auxB := b;
                    aux t (!auxR) (!auxG) (!auxB)
  in
  aux (List.tl img) 255 255 255
;;
```

```
let normalisation img =
  let minR,minG,minB = min_RGB img in
  let maxR,maxG,maxB = max_RGB img in
  let rec aux img minR minG minB maxR maxG maxB =
    match im with
    | [] -> []
    | (r,g,b)::t -> let r2 = ((r-minR)*255)/(maxR-minR) in
                    let g2 = ((g-minG)*255)/(maxG-minG) in
                    let b2 = ((b-minB)*255)/(maxB-minB) in
            (r2,g2,b2)::(aux t minR minG minB maxR maxG maxB)
  in
  let l = aux (List.tl img) minR minG minB maxR maxG maxB in
  (List.hd img)::l
;;
```

4.3 Etirement d'histogramme

Une image prise en contre jour semble ratée. Le soleil est trop présent, les zones à contre jour sont trop noires. Comme sur la figure 2. Comme précedemment les valeurs des pixels sont trop concentrées autour de certaines valeurs. Cependant la normalisation serait vaine, car les valeurs s'étendent quand même de 0 à 255. La solution est d'étirer les intervalles où les valeurs sont regroupées en empiétant sur les intervalles où peu de pixels prennent leur valeur.

Pour ce faire, nous allons calculer l'histogramme cumulé de l'image $hist_{cumul}(i) =$

$Card((x,y)|Val(x,y) \leq i)$ avec $i \in [\![0;255]\!]$. Ensuite nous appliquerons la formule 2.

$$Val(x,y) := \frac{hist_{cumul}(Val(x,y)) \times 255}{largeur \times hauteur} \tag{2}$$

FIGURE 2: Image à étirer

Exercice 7 : Implémentez la fonction `etirement` prenant l'image `img` et renvoyant l'image ayant subit un étirement d'histogramme. Note : le principe est décrit avec une valeur unique, le niveau de gris, cependant il peut s'appliquer sur les canaux R, G et B séparément puis rassembler les valeurs trouvées.

```
let hist_cumul_RGB img =
  let histR = Array.make 256 0 in
  let histG = Array.make 256 0 in
  let histB = Array.make 256 0 in
  let rec hist_RGB img histR histG histB =
    match img with
    | [] -> (histR,histG,histB)
    | (r,g,b)::t ->  histR.(r) <- histR.(r)+1;
                     histG.(g) <- histG.(g)+1;
                     histB.(b) <- histB.(b)+1;
                     hist_RGB t histR histG histB
  in
  let cumul_RGB histR histG histB =
    for i = 1 to 255 do
      histR.(i) <- histR.(i)+histR.(i-1);
      histG.(i) <- histG.(i)+histR.(i-1);
      histB.(i) <- histB.(i)+histR.(i-1);
    done
  in
  let hR,hG,hB = hist_RGB (List.tl img) histR histG histB in
  cumul_RGB hR hG hB;
  (hR,hG,hB)
;;
```

```
let eval i hist ratio =
  int_of_float ((float_of_int hist.(i))*.ratio)
;;

let etirement img =
  let hR,hG,hB = hist_cumul_RGB img in
  let ign,w,h = List.hd img in
  let ratio = 255.0/.(float_of_int (w*h)) in
  let rec calcul img ratio hR hG hB =
    match im with
    | [] -> []
    | (r,g,b)::t -> let r2 = eval r hR ratio in
                    let g2 = eval g hG ratio in
                    let b2 = eval b hB ratio in
                    (r2,g2,b2)::(calcul t ratio hR hG hB)
  in
  (ign,w,h)::(calcul (List.tl img) ratio hR hG hB)
;;
```

4.4 Filtres diverses

Exercice 8 : Implémentez la fonction `imgRGB` prenant l'image `img` et 3 booléens `dispR`, `dispG` et `dispB` et renvoyant l'image affichant les couleurs en adéquation avec les booléens.

```
let imgRGB img dispR dispG dispB =
  let rec aux img dispR dispG dispB =
    match img with
    | [] -> []
    | (r,g,b)::t -> let auxR = ref 0 in
                    if dispR then
                      auxR := r;
                    let auxG = ref 0 in
                    if dispG then
                      auxG := g;
                    let auxB = ref 0 in
                    if dispB then
                      auxB := b;
                (!auxR,!auxG,!auxB)::(aux t dispR dispG dispB)
  in
  (List.hd img)::(aux (List.tl img) dispR dispG dispB)
;;
```

Le filtre sépia est un simple calcul sur chaque pixel de valeur (R, G, B) (à l'instar du

niveau de gris). Il suffit de calculer la moyenne $m = \frac{R+G+B}{3}$ et d'appliquer les formules 3, 4 et 5.

$$R := m + 2prof \text{ si } (m + 2prof < 255) \text{ sinon } R := 255 \tag{3}$$

$$G := m + prof \text{ si } (m + prof < 255) \text{ sinon } G := 255 \tag{4}$$

$$B := m - intens \text{ si } (m - intens \geq 0) \text{ sinon } B := 0 \tag{5}$$

Exercice 9 : Implémentez la fonction `sepia` prenant l'image `img`, un entier `prof` et un entier `intens` et renvoyant l'image en sépia.

```
let sepia img prof intens =
  let rec aux img prof intens =
    match img with
    | [] -> []
    | (r,g,b)::t -> let m = (r+g+b)/3 in
                    let auxR = ref m+2*prof in
                    if auxR>255 then
                      auxR := 255;
                    let auxG = ref m+prof in
                    if auxG>255 then
                      auxG := 255;
                    let auxB = ref m-intens in
                    if auxB<0 then
                      auxB := 0;
                (!auxR,!auxG,!auxB)::(aux t prof intens)
  in
  (List.hd img)::(aux (List.tl img) prof intens)
;;
```

Exercice 10 : Implémentez vos propres filtres nuançant les couleurs, modifiant la géométrie de l'image (agrandissement, retrecissement, gradient, $\cdots$). N'hésitez pas à le faire sur papier et à me demander de l'aide.