IN 101 - TD 06 **Solution**

1 Images en noir et blanc

≪Écrivez un programme permettant transformer une image initialement en niveaux de gris en une image en noir et blanc selon un seuil. ≫

Chaque point d'une image en niveau de gris est représenté par une valeur **entière** $\in [0; max]$ pour un max donné (explicitement dans le format de représentation de l'image). La valeur d'un point représente sa luminosité ($0 \rightarrow$ noir, $max \rightarrow$ blanc, entre les deux $\rightarrow$ un niveau de gris). Pour représenter une image en noir et blanc, il suffira de n'utiliser que les deux valeurs 0 et 1 (donc max vaudra 1 pour l'image résultat).

Votre programme devra traiter des images au format `pgm`. C'est une représentation sous forme **textuelle** d'une image. Un fichier dans ce format **commence toujours** par la chaîne "P2", suivie de la **largeur** de l'image, de sa **hauteur**, puis de la valeur max dont il est question ci-dessus (séparées par des ≪blancs ≫ – retours à la ligne, espaces). Ensuite viennent, séparées par des ≪blancs ≫, les valeurs des points de l'image.

Ainsi, le fichier `ensta.pgm` contenant l'image suivante (réduite pour rentrer dans la page) :

a le contenu suivant, où l'on reconnaît la chaîne fixe "P2" suivie de la largeur de l'image (870 pixels), sa hauteur (438 pixels), la valeur maximale de niveau de gris (255, qui correspond donc à du ≪blanc ≫), puis la suite de pixels (les 3 premiers valant 194, le $4^{\text{ème}}$ 195, etc.).

```
P2
870
438
255
194
194
194
195
...
```

De nombreux fichiers au format `pgm` vous sont donnés pour tester votre futur programme. Pour visualiser une image dans votre environnement Linux, vous pouvez par exemple utiliser les commandes `eog` ou `display`. Sinon, l'explorateur de fichiers vous le permettra à la souris.

Q 1.1. Quelles sont les grandes étapes du programme ?

Solution

1. Charger le contenu du fichier pour en faire une image en mémoire.
2. Altérer la couleur des points de cette image.
3. Sauvegarder l'image obtenue dans un fichier.

Q 1.2. Quelle structure de données allez-vous utiliser pour représenter l'image ?

Solution

Une image est caractérisée par :
— sa largeur (un entier),
— sa hauteur (un entier),
— valeur *max* d'intensité (un entier),
— l'ensemble de ses points (un tableau d'entiers).

Nous allons donc grouper ces différentes informations dans une structure pour manipuler une image comme un tout :

```
struct image_t {
  int width ;
  int height ;
  int max_pix_val ;
  int *pixels ;
};
```

Q 1.3. Identifiez tous les cas où le contenu d'un fichier est incorrect vis-à-vis à vis du format `pgm`.

Solution

— Marqueur `"P2"` manquant.
— Hauteur manquante.
— Largeur manquante.
— Largeur ou hauteur ≤ 0.
— Valeur d'intensité max manquante.
— Pixels manquants
— Pixels en trop.

Souvenez-vous des fonctions de manipulation de fichiers et de la fonction `fscanf` que nous avons vues au TD 3.

Q 1.4. Décrivez l'algorithme de la fonction permettant de charger le contenu d'un fichier, d'en créer une image et de la retourner. Quels sont ses domaines d'entrée et de sortie ? Comment peut-on signaler une erreur ?

Solution

Cette fonction va prendre en argument le nom du fichier, donc une chaîne de caractères et retourner une image sous forme d'une structure comme décrit en Q1.2.

Pour signaler une erreur, on pourrait quitter le programme directement avec la fonction `exit`. Nous l'avons fait dans de précédents exercices. Néanmoins, les fonctions que nous écrivons ici peuvent être réutilisées dans d'autres programmes de manipulation d'images, donc autant les

rendre robustes aux erreurs. Ainsi, on pourra retourner une structure d'image dans laquelle le pointeur vers le tableau de pixels est nul.

```
load_pgm (nom fichier)
  Ouvrir le fichier en lecture
  Lire 1 chaîne dans le fichier
  Si lecture échouée ou chaine != "P2", fichier corrompu
  Lire la hauteur, la largeur, la valeur max d'intensité
  Si lecture échouée, fichier corrompu
  Si hauteur ou largeur <= 0, fichier corrompu
  Allouer tableau de pixels
  Lire "taille" valeurs, transformer en entier et remplir le tableau
  Si une des lectures échouée, fichier corrompu
  Fermer ficher
  Grouper les information dans une structure d'image
  Retourner  l'image
```

Q 1.5. Décrivez l'algorithme de la fonction permettant de sauvegarder dans un fichier une image passée en argument. Quels sont ses domaines d'entrée et de sortie ?

Solution

Cette fonction va prendre en argument une image sous forme d'une structure comme décrit en Q1.2, le nom du fichier dans lequel sauvegarder l'image et retourner une valeur booléenne disant si la sauvegarde a bien été effectuée.

```
save_pgm (nom fichier)
  Ouvrir le fichier en écriture
  Écrire la chaîne "P2\n"
  Écrire la largeur puis un retour à la ligne
  Écrire la hauteur puis un retour à la ligne
  Écrire la valeur maximale d'intensité puis un retour à la ligne
  Parcourir le tableau des pixels et écrire leurs valeurs sur une ligne à
      chaque fois
  Fermer le fichier
```

Comme introduit au début de l'exercice, transformer une image de niveaux de gris en noir et blanc (donc **2** ≪couleurs ≫) se fait en transformant chaque pixel en l'une de ces deux couleurs. Le choix est effectué en fonction d'un **seuil**. Si un point a un niveau de gris inférieur au seuil, alors il sera transformé en ≪noir ≫ dans l'image résultat, sinon en ≪blanc ≫.

Par convention, vu du côté d'un utilisateur, une image en niveau de gris est **toujours** en **256 niveaux de gris**. Aussi, le seuil spécifié par l'utilisateur sera naturellement et **obligatoirement** $\in [0; 255]$.

Les images ci-dessous représentent le résultat du seuillage avec les seuils 50, 100, 150 et 200.

Q 1.6. Décrivez l'algorithme qui permet de transformer une image en noir et blanc en fonction d'un **seuil** passé en **paramètre**.

Solution

```
binarize (thresh, image) =
  Récupérer la largeur, la hauteur et le tableau des pixels de l'image
  Pour chaque pixel
    Si la valeur du pixel est < au seuil thresh
      Le remplacer par 0 dans le tableau des pixels
    Sinon
      Le remplacer par 1 dans le tableau des pixels
  Retourner l'image composée des largeur et hauteur initiales et de
     l'intensité maximale 1 (oui, il ne reste que 0 ou 1 comme valeurs de pixels)
```

Dans l'implémentation que nous ferons en `C`, cette fonction modifiera ≪en place≫ (par effet de bord) l'image reçue en argument. Donc nous passerons l'image en argument **par adresse**. C'est un choix, pour simplifier le code, qui évite de devoir copier l'image pour en faire une nouvelle.

Du point de vue de l'utilisateur, une image comportera toujours **256** niveaux de gris ≪logiques≫. Il devra donc fournira donc son seuil entre 0 et 255. En effet, il n'est pas censé connaître le contenu ≪technique≫ de l'image, donc le nombre réel de niveaux de gris qu'elle comporte.

Q 1.7. Comment allez-vous ramener le seuil donné par l'utilisateur dans l'intervalle propre à l'image ?

Solution

Une simple règle de 3 fait l'affaire :

$$\begin{array}{lcl} image\_max & \leftrightarrow & 255 \\ seuil\_image & \leftrightarrow & seuil\_utilisateur \end{array} \quad \Rightarrow seuil\_image = \frac{image\_max \times seuil\_utilisateur}{255}$$

Q 1.8. Écrivez la fonction `load_pgm` dont vous avez donné l'algorithme en Q1.4.

Solution

```
/** Load a PGM image file.
    Set image width, height and max gray-level value by side effect in the
    related paramaters.
    Return the image as an array of integers. */
struct image_t load_pgm (char *fname) {
  char buffer[20] ;
  int size, read ;
```

```c
struct image_t image ;

/* Set to NULL in case of error. At least, this is done once for all. */
image.pixels = NULL ;

FILE *in = fopen (fname, "rb") ;
if (in == NULL) return image ;

/* Read the magic number and ensure it is the PGM format's one. */
read = fscanf (in, "%s", buffer) ;
if ((read != 1) || (strcmp (buffer, "P2") != 0)) {
  fclose (in) ;
  return image ;
}

/* Read the width, height and gray-level value. */
read = fscanf (in, "%d %d", &image.width, &image.height) ;
if (read != 2) {
  fclose (in) ;
  return image ;
}
size = image.width * image.height ;
read = fscanf (in, "%d", &image.max_pix_val) ;
if (read != 1) {
  fclose (in) ;
  return image ;
}

/* Ensure the pixels is not empty. */
if (size <= 0) {
  fclose (in) ;
  return image ;
}

/* Allocate memory for the image. */
int *pixels = malloc (size * sizeof (int)) ;
if (pixels == NULL) {
  fclose (in) ;
  return image ;
}

/* Read the bytes of the image. */
for (int i = 0; i < size; i++) {
  read = fscanf (in, "%d", &pixels[i]) ;
  if (read != 1) {
    fclose (in) ;
    free (pixels) ;
    return image ;
  }
}
```

Q 1.9. Pour vérifier la robustesse de votre fonction de chargement, testez-la avec les trois fichiers **corrompus** :

- `mangled_extra_pixels.pgm` : trop de pixels (ce cas peut ou non lever une erreur, on peut considérer que l'on ignore les pixels en trop),
- `mangled_missing_pixels.pgm` : pas assez de pixels,
- `early_bad.pgm` : entête incomplet.

Q 1.10. Écrivez la fonction `save_pgm` qui permet de sauvegarder dans un fichier une image passée en argument et dont vous avez esquissé l'algorithme en Q1.5.

Solution

```
  image.pixels = pixels ;
  return image ;
}



bool save_pgm (char *fname, struct image_t image) {

  if (image.pixels == NULL) return false ;

  FILE *out = fopen (fname, "wb") ;
  if (out == NULL) return (false) ;

  fprintf
    (out, "P2\n%d\n%d\n%d\n", image.width, image.height, image.max_pix_val) ;
  int size = image.width * image.height ;
```

Q 1.11. Écrivez la fonction `binarize` qui permet de transformer une image en noir et blanc en fonction d'un **seuil** passé en **paramètre** dont vous avez esquissé l'algorithme en Q1.6.

Solution

```
  fclose (out) ;
  return (true) ;
}



void binarize_pgm (int thresh, struct image_t *image) {
  if (image->pixels == NULL) return ;

  int  size = image->width * image->height ;
  int *pixels = image->pixels ;
  for (int i = 0; i < size; i++) {
    /* 2 colors : 0 -> black, 1 -> white. */
```

Q 1.12. Écrivez le `main` qui orchestre la transformation de l'image, depuis son chargement jusqu'à sa sauvegarde. Cette fonction récupérera sur la **ligne de commande** :

1. le nom du fichier image à charger,
2. le nom de fichier sous lequel sauvegarder l'image transformée,
3. le seuil de transformation.

Solution

```
  /* Two colors => max intensity level is 1. */
  image->max_pix_val = 1 ;
}



int main (int argc, char *argv[]) {
  int thresh ;
  bool err_code ;

  if (argc != 4) {
    printf ("Error. Usage: binarize.x img-source img-dest threshold\n") ;
    return 1 ;
  }

  /* Load image. */
  struct image_t image = load_pgm (argv[1]) ;
```

```
  if (image.pixels == NULL) {
    printf ("Error. Unable to load image.\n") ;
    return 1 ;
  }

  /* Get the threshold. */
  thresh = atoi (argv[3]) ;
  /* Simple 3-rule. */
  thresh = (image.max_pix_val * thresh) / 255 ;

  /* Convert to black and white. */
  binarize_pgm (thresh, &image) ;

  /* Save image. Its max_pix_val is now 1 since we have only 2 colors. */
  err_code = save_pgm (argv[2], image) ;
  if (! err_code) {
    printf ("Error. Unable to save image.\n") ;
    return 1 ;
```

Q 1.13. Testez votre programme avec des images qui vous sont données : `ensta.pgm`, `minions.pgm`, `flower.pgm`, `trooper.pgm`, `road.pgm` ou `marvin.pgm`.

N'essayez pas avec `rabbid.pgm` qui sert pour la partie ≪Pour aller plus loin ≫ (son format fera échouer le chargement).

S'il vous reste du temps ou pour continuer après la séance.

2 S'il vous reste du temps : le vrai format `pgm`

≪Écrivez une fonction permettant de charger une image au véritable format `pgm`. ≫

Dans la réalité, un fichier `pgm` peut contenir des lignes de commentaire commençant par le caractère `#`. Le fichier `rabbid.pgm` illustre cette structure.

```
P2
# CREATOR: GIMP PNM Filter Version 1.1
300 437
255
255
255
      # Comment in the middle not starting in column 0.
255
255  # Comment after a value.
255
255 255 255 255
255 255 255 255 255 255 255
255
255
(...)
```

Q 2.1. Que change la présence de commentaires dans la manière dont vous allez lire le contenu du fichier ?

Solution

Désormais, une ligne peut soit contenir un commentaire, soit une ou des valeurs de pixels, voire un commentaire après des valeurs de pixels. Comme un commentaire est une chaîne ar-

bitrairement longue, il est impossible de lire avec `fscanf` car on ne saurait pas combien de ≪mots≫ de commentaire il y a après un `#` de début.

Il ne nous reste pas beaucoup de solutions autres que de lire caractère par caractère pour constituer la chaîne à retourner. De plus, comme l'on peut retourner la chaîne `"P2"` ou des chaînes représentant des valeurs entières, notre fonction de lecture ne peut directement retourner un entier, il faut qu'elle retourne une chaîne. Nous transformerons cette chaîne en valeur entière selon le besoin.

La bibliothèque standard de `C` fournit la fonction :

int fgetc(FILE ∗stream)

qui permet de lire 1 caractère dans le fichier `stream`. Elle renvoie la valeur `EOF` lorsqu'elle ne peut plus lire car la fin du fichier a été atteinte.

Q 2.2. Nous souhaitons trouver l'algorithme de notre fonction `my_scanner` qui va remplacer `fscanf` utilisé jusqu'à présent et retourner une chaîne de caractères. Ainsi, `my_scanner` devra trouver des caractères consécutifs qui ne sont pas des ≪blancs≫ ni des commentaires.

Réfléchissez aux différents cas de détection de début et de fin de chaîne. Esquissez l'algorithme de `my_scanner`.

Petite indication : si vous détectez un problème lors de la détermination de fin de chaîne à cause d'un début de commentaire, il existe la fonction `ungetc` qui permet de réinjecter un caractère dans un fichier à la position actuelle. Utilisez la commande `man` pour obtenir des détails sur cette fonction (`man 3 ungetc`).

Solution

Début de chaîne : lorsque la fonction est appelée, elle peut rencontrer des ≪blancs≫ avant le début d'une chaîne significative. Elle peut aussi rencontrer un commentaire. Au pire, elle peut atteindre la fin de fichier. Donc, tant que l'on n'a pas atteint la fin de fichier, il falloir ignorer les caractères ≪blancs≫ et si l'on rencontre un `#`, tous les caractères jusqu'à prochain `\n`.

Fin de chaîne : elle se termine lorsque l'on rencontre la fin de fichier ou un ≪blanc≫ . . . ou un `#` marquant le début d'un commentaire. On va donc accumuler les caractères tant que ce ne sont pas la fin de fichier, un ≪blanc≫ ou un `#`.

Problème : si la fin de chaîne est détectée en trouvant un `#`, celui-ci aura été consommé dans le fichier et au prochain appel à la fonction, on ne saura plus que ce qui suit était un commentaire ! Une solution simple est de remettre le caractère dans le fichier. Et il existe une fonction à cet effet :

int ungetc (**int** c, FILE ∗stream)

qui injecte le caractère `c` dans le fichier `stream` à la position courante de lecture.

Fin de fichier : si l'on arrive en fin de fichier sans avoir pu lire de chaîne, on n'aura qu'à retourner la chaîne vide.

La structure de l'algorithme est donc la suivante, qui ≪retourne≫ la chaîne. Si l'on est arrivé en fin de fichier, on retournera la chaîne vide.

```
bool my_scanner =
  chaîne résultat = ""
  dans_commentaire = faux
  c <- lire 1 caractère
```

```
tant que (c <> fin fichier) et que
         (c = blanc ou que l'on est dans un commentaire)
  si c = '#' dans_commentaire <- vrai
  sinon si (c = '\n') dans_commentaire <- faux
  c <- lire 1 caractère

si c = fin fichier, retourner la chaîne vide car on n'a rien trouvé
mettre c dans la chaîne résultat
c <- lire 1 caractère
tant que (c <> fin fichier) et que (c <> blanc)
  mettre c dans la chaîne résultat
  c <- lire 1 caractère

si c = '#', remettre c dans le fichier
clore la chaîne résultat par '\0'
retourner la chaîne résutat d'une façon ou d'une autre
```

Q 2.3. Pensez à comment vous aller écrire cette fonction en `C`. Quel vont être les domaines des entrées et des sorties de cette fonction ?

Solution

Notre fonction doit ≪retourner≫ la chaîne. On ne sait pas à l'avance quelle taille elle va faire donc l'allouer dynamiquement va être pénible (redimensionnement en cas de besoin, etc.). Donc, on va se simplifier la tâche et prendre en argument un pointeur sur une chaîne qu'aura alloué la fonction appelante. C'est donc un passage **par adresse** (ce qui est toujours le cas lorsque l'on passe un tableau, or une chaîne n'est qu'un simple tableau de caractères).

Pour simplifier encore, on ne va **pas gérer** le possible dépassement de la chaîne servant d'emplacement de stockage. On considérera qu'elle est ≪suffisamment grande≫. Si ce n'est pas le cas, notre programme plantera bien entendu.

Notre fonction doit également prendre en argument le descripteur de fichier dans lequel lire. Puisqu'elle ≪retourne≫ sa chaîne en utilisant le passage par adresse, son type de retour sera ≪vide≫.

Elle aura donc pour prototype : **void** my_scanner (FILE ∗in, **char** ∗dest)

Q 2.4. Quel va être l'impact de l'utilisation de `my_scanner` dans votre fonction `load_pgm` ?

Solution

Il va être minime. On remplace les appels à `fscanf` par des appels à `my_scanner`. Pour vérifier si la lecture à échoué on regarde si la chaîne obtenue est vide. À cet effet, il suffit de vérifier si son premier caractère est `'\0'` (pas la peine d'appeler `strlen`). Lorsque l'on attend une valeur entière, on convertit la chaîne lue avec la fonction `atoi`.

Notons que contrairement à `fscanf` qui levait une erreur si ce qui était lu ne représentait pas une chaîne en accord avec le format spécifié, nous ne pouvons plus nous assurer que nous avons bien lu une chaîne représentant un entier. Il faudrait pour ce faire, complexifier notre fonction `my_scanner`.

Q 2.5. Implémentez la fonction `my_scanner` et modifiez votre `main` en conséquence. Testez ensuite votre programme avec le fichier `marvin.pgm`.

Solution

`my_scanner`

```
void my_scanner (FILE *in, char *dest) {
  bool in_comment = false ;
  int read = fgetc (in) ;

  /* Consume and ignore whitespaces and comments */
  while ((read != EOF) &&
         ((read == ' ') || (read == '\n') || read == ('\t') || (in_comment))) {
    if (read == '#') in_comment = true ;
    else {
      if (read == '\n') in_comment = false ;
    }
    read = fgetc (in) ;
  }

  /* If end of file reached without reading a significant character, no more
     job to do. */
  if (read == EOF) {
    dest[0] = '\0' ;
    return ;
  }

  /* We have a significant starting character. Let's go on in the input until
     white space, the end of file or a new comment. */
  dest[0] = read ;
  int i = 1 ;
  read = fgetc (in) ;
  while ((read != EOF) && (read != ' ') && (read != '\n') && read != ('\t')) {
    dest[i] = read ;
    i++ ;
    read = fgetc (in) ;
  }

  if (read == '#') ungetc (read, in) ;
  dest[i] = '\0' ; /* Close the result string. */
}
```

`load_pgm`

```
  /* Read the width, height and gray-level value. */
  my_scanner (in, buffer) ;
  if (buffer[0] == '\0') return image ;
  image.width = atoi (buffer) ;
  my_scanner (in, buffer) ;
  if (buffer[0] == '\0') return image ;
  image.height = atoi (buffer) ;
  my_scanner (in, buffer) ;
  if (buffer[0] == '\0') return image ;
  image.max_pix_val = atoi (buffer) ;
```

```
  /* Read the bytes of the image. */
  for (int i = 0; i < size; i++) {
    my_scanner (in, buffer) ;
    if (buffer[0] == '\0') {
      fclose (in) ;
      free (pixels) ;
      return image ;
    }
    pixels[i] = atoi (buffer) ;
  }
```