

TP 3– Lecture et écriture binaires ; manipulation de bits

1 But du TP

Concepts abordés

- Lecture de fichiers binaires.
- Implémentation d’une spécification (format de fichier).
- Manipulation de bits (masques).

Fonctions et opérateurs utiles

- `fopen()`, `fread()`, `fwrite()`, `fclose()`, `fprintf()`
- `fseek()`
- `stat()`
- opérateurs de manipulation de bits : `<<` `>>` `&`

N’hésitez pas à utiliser la commande `man` pour obtenir la documentation de chaque fonction.

Contraintes et remise des TPs

- Le code de chaque exercice doit être dans un répertoire séparé (exercice1, exercice2, etc.).
- Chaque répertoire d’exercice doit contenir un fichier `Makefile` permettant de compiler les réponses à chaque question.
- Quand une question ne demande pas uniquement du code mais aussi du texte, des explications, des commentaires, etc. vous pouvez soit les rendre dans un fichier texte (.txt) dans le dossier de l’exercice, soit si ce n’est pas trop gros les mettre en commentaire dans votre code. **Si une question demande des explications ou une réponse texte et que vous vous contentez de rendre du code, vous n’aurez pas tous les points.**
- Une note de propreté et style du code est attribuée à votre travail et prise en compte lors de l’évaluation. Elle inclue : la lisibilité du code (taille des fonctions, noms de variables explicites, etc.), sa propreté (libération de mémoire, contrôle des erreurs, réemploi de variables, etc.), sa qualité algorithmique, la présence et la pertinence des commentaires, les gardes `#ifndef` dans les headers, l’indentation et mise en page. L’ensemble de ces consignes est détaillé dans le document “propreté et lisibilité du code” disponible sur la page du module.
- Le code source doit être **mis en ligne sur Moodle** sous forme d’une archive tar.gz¹ **au plus tard 15 minutes après la fin du TP**. Votre archive doit être propre : pas d’exécutables, pas de .o, pas de fichiers temporaires. En cas de problème de remise, contacter vos encadrants.
- Les supports de cours, les exemples et les fichiers utiles aux TPs sont **sur la page du cours sur Moodle**.

1. Pour faire une archive tar.gz : `tar cvzf mon_archive.tar.gz mon_repertoire`

Barème de notation

Exercice 1 : coefficient 1. Exercice 2 : coefficient 3. Style, propreté et lisibilité du code : coefficient 1.

But général

Le but de ce TP est d'écrire un convertisseur du format d'image TGA (un format utilisé dans l'industrie graphique) vers le format d'image PPM (un format très simple utilisé par de nombreuses bibliothèques de traitement d'image).

Exercice 1 – Lectures binaires

Le format TGA contient de nombreux nombres entiers écrits en binaire (pour décrire la taille de l'image, les pixel, ...); suivant la précision et l'intervalle de valeur requis, ces données sont codées sur 8, 16 ou 32 bits. Dans un premier temps, nous allons écrire quelques fonctions simples pour simplifier la lecture de chaque type d'entier.

1. **Types de données.** Dans un fichier `binary.h`, définissez les types simple suivants :
 - `ui8_t` : entier non signé sur 8 bits;
 - `ui16_t` : entier non signé sur 16 bits;
 - `ui32_t` : entier non signé sur 32 bits.
2. **Lecture de données.** Dans un fichier `binary.c`, définissez (et testez) les fonctions suivantes :
 - `ui8_t read_ui8(FILE *f)` : lecture d'un `ui8_t` dans le fichier `f`;
 - `ui16_t read_ui16(FILE *f)` : lecture d'un `ui16_t` dans le fichier `f`;
 - `ui32_t read_ui32(FILE *f)` : lecture d'un `ui32_t` dans le fichier `f`.Ajoutez ces prototypes à `binary.h`.
3. **Taille d'un fichier.** Ajoutez à `binary.c` une fonction de prototype :
`ui32_t file_size (const char* fname)`
qui renvoie la taille (en octets) du fichier de nom « `fname` ». Testez rapidement cette fonction.

Exercice 2 – Le format TGA

Spécifications

Le format TGA est un format d'image bitmap (raster) assez simple et assez souvent utilisé car il peut gérer la transparence. Il est par exemple utilisé pour stocker les textures de certains jeux video. Dans ce TP, nous allons écrire un programme permettant de :

- lire un fichier TGA simple (nous ne gérerons pas toutes les subtilités du format);
- afficher les informations sur l'image (largeur, hauteur, etc.);
- convertir cette image dans le format PPM, plus simple.

Téléchargez le fichier `image.tga` sur le site web du cours. Ouvrez ce fichier avec un visionneur d'image (par exemple la commande `display`). Ouvrez la maintenant avec `vim` : on ne distingue pas grand chose ! Pour essayer d'y voir plus clair, on peut essayer de le visualiser au format hexadécimal avec l'outil `hexdump` (`hexdump -C image.tga`) mais on ne distingue pas grand chose de plus. Le fichier ne contient peu ou pas de chaînes de caractères ; il va falloir lire les spécifications du format TGA pour comprendre comment lire ces données.

La figure 1, issue des spécifications, résume le format TGA. Analysons cette figure. Comme la plupart des formats d'image, le format TGA est constitué d'un en-tête (header), contenant des méta-données comme la taille de l'image et le nombre de couleurs, puis l'ensemble des pixels sous forme de données « brutes ». L'en-tête est constitué de différents champs, de différentes tailles. Plus précisément, il est constitué des champs suivants (dans cet ordre) :

- `idlength` (8 bits) : taille du champ "image id" (variable) ;
- `colour map type` (8 bits) : type de carte de couleur ;
- `image type` (8 bits) : à ignorer ;
- `colour map origin` (16 bits) : à ignorer ;
- `colour map length` (16 bits) : nombre d'entrées dans la carte de couleurs ;
- `colour map depth` (8 bits) : taille en bits d'une entrée dans la carte de couleurs ;
- `x origin` (16 bits) : à ignorer ;
- `y origin` (16 bits) : à ignorer ;
- `width` (16 bits) : largeur (en pixels) ;
- `height` (16 bits) : hauteur (en pixels) ;
- `bpp` (8 bits) : 8, 24 ou 32 bits par pixels ;
- `image descriptor` (8 bits) : à ignorer.

1. **Type de donnée.** Dans un fichier `img.h`, définir un type structuré `img_t` contenant les champs de l'en-tête précisés dans la section précédente.

Vous devez utiliser les types `ui8_t`, `ui16_t` et `ui32_t` de `binary.h`.

2. **Lecture des méta-données.**

1. Dans `img.c` (et `img.h`), ajoutez une fonction `void read_header(FILE* f, img_t* img)` qui lit l'en-tête contenu dans le fichier `f` et complète la structure `img` (qui doit déjà exister en mémoire). Utilisez les fonctions des fichiers `binary.h` / `binary.c`. *Attention* : les champs à ignorer sont présents dans l'en-tête. Bien que leur valeur ne nous intéresse pas, il faut donc les lire.
2. Utilisez `read_header` pour implémenter un programme appelé `print_data` prenant en argument une image TGA et affichant :
 - le nom du fichier ;
 - la taille du fichier (en octets) ;
 - la taille de l'image sous la forme « 640x480 » ;
 - le nombre de bits par pixels (bpp).

3. **Lecture des pixels.** En nous restreignant à 8 bits pour chaque couleur (de 0 à 255), 24 bits sont nécessaires pour chaque pixel. On peut donc stocker chaque pixel dans un entier de 32 bits (les derniers 8 bits sont généralement réservés à l'information de transparence). On stockera les pixels dans l'ordre rouge (r), vert (g), bleu (b).

Nous allons stocker les pixels dans un tableau de `ui32_t`. Chaque `ui32_t` contient 32 bits ; nous allons utiliser les 8 bits de poids faible pour stocker les 8 bits codant pour la composante « r », (rouge) les 8 bits suivants pour le « g » (vert) et les 8 suivants pour

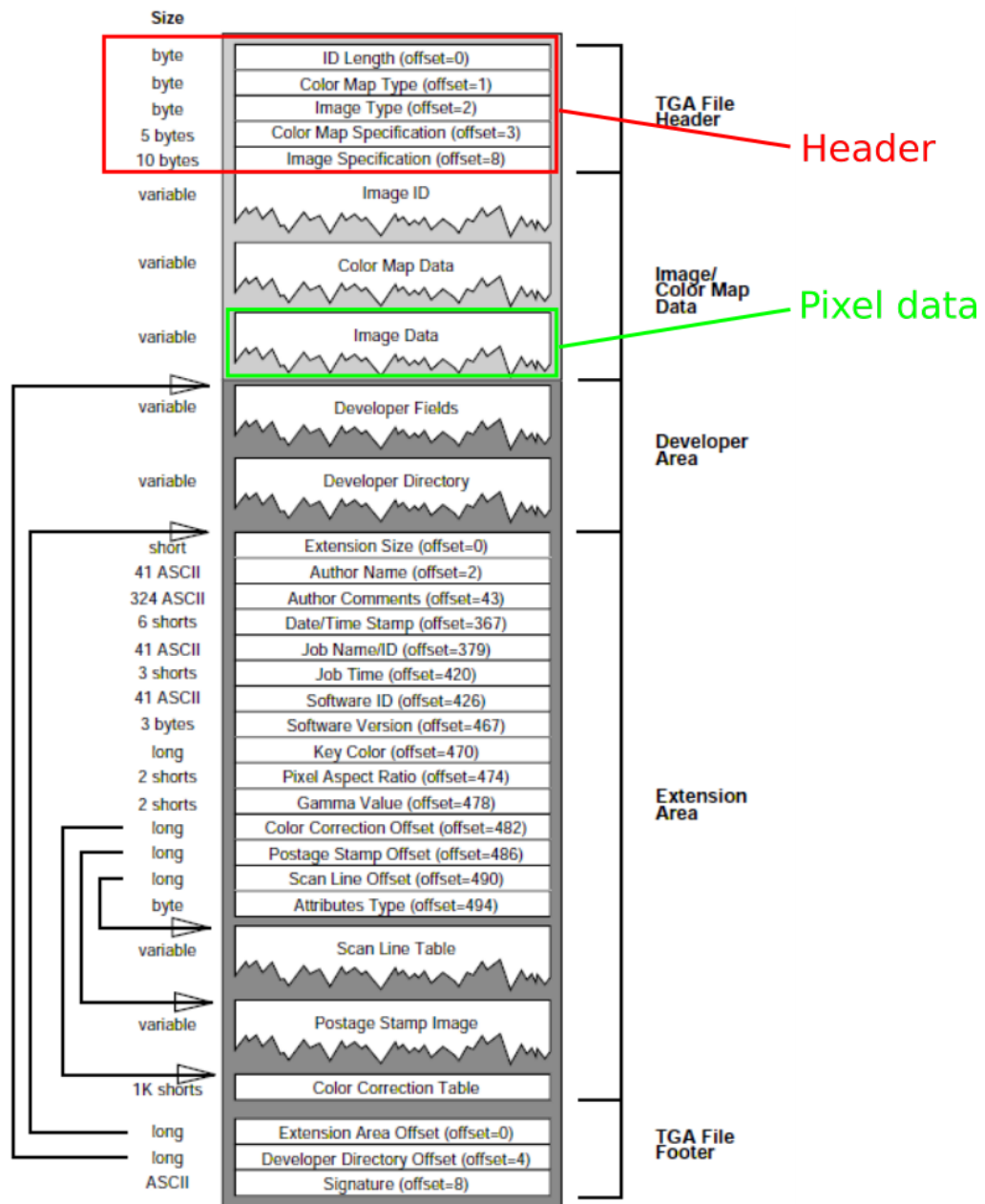


FIGURE 1 – Résumé du format TGA

le « b » (bleu). Les 8 bits restants sont généralement utilisés pour stocker une valeur de transparence. Nous les laisserons à zéros pour le moment.

1. Ajoutez un champ `pixels` de type `ui32_t*` à `img_t`.

2. Ajoutez une fonction

`ui32_t* get_pixel(const img_t* img, int row, int col)`

qui renvoie le pixel de la ligne `row` et de la colonne `col`.

3. En utilisant les opérateurs de manipulation de bits, écrivez une fonction

`ui32_t make_pixel(ui8_t r, ui8_t g, ui8_t b);`

qui construit un pixel sur 32 bits à partir des valeurs `r,g,b` chacune codées sur 8 bits.

4. Écrivez une fonction

`read_pixels(FILE* f, img_t* img);`

qui lit les pixels contenus dans `f` et remplit le champ `pixels` de `img`. *Attention :*

— avant de lire les pixels, il est nécessaire d'ignorer les champs situés entre l'en-tête et les données des pixels (les champs "Image ID" et "Color map data"); pour cela, il faut « sauter » `img->id_length + img->colourmaptype * imgcolourmaplength` octets (utilisez la fonction `fseek`);

— n'oubliez pas d'allouer le tableau de pixels à la bonne taille;

— dans un fichier TGA, les couleurs sont stockées dans l'ordre « b,g,r ».

4. **Écriture PPM.** Le format PPM est plus simple. Il est lui aussi constitué d'un en-tête, mais en ASCII. Il prend la forme :

```
P6
taille_x taille_y
255
rgbrgbrgb...
```

où `taille_x` et `taille_y` sont à remplacer par la taille de l'image et `rgbrgb...` correspond à l'ensemble des données de l'image, dans l'ordre « r, g, b ».

1. Écrivez une fonction `write_ppm(const char* filename, const img_t* img)` qui écrit l'image `img` dans le fichier de nom `filename`. Vérifiez que l'image obtenue est visuellement identique au fichier TGA original.

2. Incorporez-la dans un programme que vous appellerez `tga2ppm` qui doit permettre de convertir une image au format tga en une image au format ppm.

5. **Bonus : ligne de commande et options.** On souhaite maintenant rendre notre convertisseur un peu plus facile à utiliser en ajoutant des arguments du type : `./tga2ppm --width --height --bpp --in test.TGA --out test.ppm`

Dans ce cas, le programme doit écrire la largeur et la hauteur de l'image puis effectuer la conversion. Afin de rendre la gestion de ce qui doit être affiché la plus souple possible, nous allons utiliser les masques de bits.

1. Écrivez et testez une fonction `void print_info (const img_t *img, ui32_t data)`. On veut pouvoir appeler la fonction des manières suivante :

— `print_info(&img, WIDTH)` (affiche la largeur);

— `print_info(&img, WIDTH|HEIGHT)` (affiche la largeur et la hauteur);

— `print_info(&img, WIDTH|HEIGHT|BPP)`
— ...

2. Écrivez une fonction `ui32_t parse_args(int argc, char** argv, char** input, char** output)` qui alloue et remplit `input` (nom du fichier d'entrée) et `output` (nom du fichier de sortie) et renvoie un entier sur 32 bits compatible avec la fonction `print_info()`.
3. Insérez les appels à ces fonctions dans votre fonction `main()`.