

BE 1 : Images

1 Images BMP

Dans ce TP, nous nous intéresserons à la création d'images au format BMP.¹ Ce format de fichier a le bon goût d'être simple à manipuler (pas de compression) et être lisible (normalement) par vos navigateurs web et applications préinstallées sur vos ordinateurs.

1.1 Couleurs RGB

L'encodage RGB est un encodage courant des couleurs en informatique (pas le seul). Dans cet encodage, une couleur est déterminée par la proportion des couleurs primaires rouge (Red), vert (Green) et de bleu (Blue). Chaque proportion est codée sur un octet : 0 signifie l'absence de cette couleur primaire, et 255 est la valeur maximale pour une couleur primaire.

Par exemple, la couleur rouge est encodée avec $R = 255$, $G = 0$ et $B = 0$. Plus succinctement, on note la couleur rouge `0xff0000`. De manière similaire, la couleur verte est `0x00ff00` et la couleur bleue `0x0000ff`. De plus, le blanc est encodé `0xffffffff` et le noir `0x000000`. Vous pouvez en apprendre plus sur cet encodage ici : https://fr.wikipedia.org/wiki/Rouge_vert_bleu.

1.2 Format BMP

Le format de fichier BMP est composé (comme beaucoup de formats de fichiers) d'un premier en-tête de 14 octets contenant des informations sur le fichier (type, taille du fichier, etc.), un second en-tête de 40 octets contenant des informations sur l'image (notamment sa largeur, sa hauteur, le nombre de couleurs utilisées, etc.), suivi des données de l'image à proprement parler, c'est-à-dire pour chaque pixel, trois octets qui encodent la couleur de ce pixel. Les valeurs codées sur plusieurs octets sont encodées en *little-endian* (les octets de poids faible d'abord).

Le premier en-tête (le *Bitmap File Header*) suit le format suivant :

- les deux premiers octets correspondent aux caractères ASCII `BM`. Ces deux octets indiquent aux applications qui vont ouvrir ce fichier qu'il s'agit d'un fichier au format BMP. La plupart des formats de fichier utilisent ce procédé. On appelle ces quelques premiers octets un *magic number*.
- les quatre octets suivants encodent la taille totale du fichier : l'en-tête s'étend sur 54 octets, et on utilise 3 octets par pixel, donc ce champ vaut $54 + 3 \cdot w \cdot h$, où w et h sont la largeur et la hauteur de l'image.
- les quatre octets suivants identifient l'application qui a créé le fichier : dans notre cas, nous y mettrons la valeur 0.
- les quatre octets suivants encodent l'offset de début de l'image, c'est-à-dire 54.

Le second en-tête (le *Bitmap Info Header*) suit le format suivant :

- 4 octets : la taille de cet en-tête (40)
- 4 octets : la largeur de l'image
- 4 octets : la hauteur de l'image
- 2 octets : le nombre de *plans* de l'image (vaut 1)
- 2 octets : le nombre de bits par pixel : on utilisera la valeur 24 (8 bits pour chacun de R, G, et B)

1. https://fr.wikipedia.org/wiki/Windows_bitmap

- 4 octets : l'algorithme de compression. On n'utilisera pas de compression et on utilisera donc la valeur 0 pour ce champ.
- 5 champs de 4 octets qu'on mettra à 0.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31		
																B						M										}	Bitmap File Header
Taille du fichier : $54 + 3 \cdot w \cdot h$																																	
Identifiant application : 0																																	
Début de l'image : 54																																	
Taille du second en-tête : 40																																}	Bitmap Info Header
Largeur de l'image : w																																	
Hauteur de l'image : h																																	
Nombre de plans : 1																Nombre de bits par pixel : 24																	
Algorithme de compression : 0																																	
Taille de l'image (redondant) : 0																																	
Résolution horizontale : 0																																	
Résolution verticale : 0																																	
Nombre de couleurs dans la palette : 0																																	
Nombre de couleurs « importantes » : 0																																	
Pixels																																	

Les données de l'image sont ensuite encodées, pixel par pixel.

- Chaque pixel est encodé par sa couleur RGB dans l'ordre B, puis G, puis R. Un pixel rouge (RGB 0xff0000) sera donc encodé par les trois octets 00, 00 puis ff.
- Les pixels sont décrits ligne-par-ligne, **en commençant par la ligne du bas**.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Ainsi, les pixels de l'image ci-dessus seront encodés dans l'ordre MNOPIJKLEFGHABCD.

Plus précisément, étant donné que le rouge est 0xff0000, le jaune est 0xffff00 et le bleu est 0x0000ff, on aura l'encodage suivant :

```
00 00 ff 00 00 ff 00 00 ff 00 00 ff
ff 00 00 00 00 ff 00 ff ff 00 ff ff
ff 00 00 00 00 ff 00 ff ff 00 ff ff
ff 00 00 00 00 ff 00 ff ff 00 ff ff
```

Vous pouvez vérifier en ouvrant l'image `example.bmp` et en utilisant un éditeur hexadécimal :

```
$ hexdump -s54 -C example.bmp
```

```
00000036 00 00 ff 00 00 ff 00 00 ff 00 00 ff ff 00 00 00 |.....|
00000046 00 ff 00 ff ff 00 ff ff ff 00 00 00 ff 00 ff |.....|
00000056 ff 00 ff ff ff 00 00 00 00 ff 00 ff ff 00 ff ff |.....|
00000066
```

L'option `-s54` signifie de commencer à lire au 54-ème octet du fichier, l'option `-C` permet d'afficher octet par octet (et pas groupé par 2 octets, avec les problèmes d'endianness que cela implique).

- **Important.** Chaque ligne de pixels est complétée avec des bits de bourrage (*padding bits*), afin que chaque ligne de pixels soit encodée par un multiple de 4 octets. La valeur de ces octets de bourrage n'a pas d'importance : on pourra mettre des 0.

Par exemple, si la largeur de l'image est 30, on utilise pour chaque ligne $30 \times 3 = 90$ octets utiles, mais on doit rajouter 2 octets de bourrage pour arriver au prochain multiple de 4, c'est-à-dire 92.

Vous pouvez échapper à cette contrainte en ne considérant dans un premier temps que des images dont la largeur est elle-même un multiple de 4, pour gagner du temps.

1.3 Structures pixel et image

On modélisera une image en C grâce aux structures suivantes :

```
typedef struct pixel {
    unsigned char r;
    unsigned char g;
    unsigned char b;
} pixel;
typedef struct image {
    unsigned int width;
    unsigned int height;
    struct pixel* data;
} image;
```

Ces définitions sont écrites pour vous dans le squelette `be1-squelette.c`.

Un pixel contient donc trois champs de type `unsigned char` : un par couleur primaire de l'encodage RGB.

Une image est une structure contenant la largeur, la hauteur de l'image, ainsi qu'un tableau de pixels.

Question 1. Écrivez une fonction `image* empty_image(int w, int h)`; qui crée une image pour laquelle tous les pixels sont bleus, de largeur `w` et de hauteur `h`.

La couleur bleue est encodée `0x0000ff`.

Avant de vous lancer dans le code, réfléchissez :

- allez-vous avoir besoin d'allouer de la mémoire ? utiliserez-vous de l'allocation statique ou dynamique ?
- de combien de zones mémoires avez-vous besoin ? de quelle(s) taille(s) ?

Les pixels de l'image sont disposés les uns à la suite des autres, en commençant par tous ceux de la première ligne (indice 0, celle du bas), puis tous ceux de la deuxième ligne, etc. Ainsi, le i -ème pixel de la j -ème ligne sera à la $j * w + i$ -ème position. Par exemple, le 17ème (indice 16) pixel de la 5ème (indice 4) ligne sera à la position $4 * w + 16$.

Question 2. Écrivez les deux fonctions `void write_int(int data, FILE* f)` et `void write_byte(unsigned char data, FILE* f)` qui écrivent, respectivement un entier ou un octet, dans le fichier `f`. Ces écritures sont binaires, et non textuelles.

Question 3. Complétez la fonction `void write_image(image* img, char* filename)`; qui écrit l'image `img` dans le fichier dont le nom est `filename`, en respectant le format étudié ci-dessus.

L'écriture des en-têtes est déjà faite pour vous : vous n'avez qu'à écrire les données correspondant à chacun des pixels, dans l'ordre défini ci-dessus.

Repensez au problème des octets de bourrage discuté précédemment. Si vous souhaitez ne pas vous en préoccuper pour le moment, retenez que vous serez limités à des images dont la largeur est un multiple de 4.

Question 4. Écrivez une fonction `main` qui crée une image vide (toute bleue) de taille 100x100, et qui l'écrit dans un fichier `blank.bmp`. Vérifiez que vous pouvez ouvrir cette image avec votre visionneur préféré.

2 Dessiner des formes

Question 5. Écrivez une fonction `void draw_pixel(int x, int y, unsigned int color, image* img)` qui change la couleur du pixel aux coordonnées (x, y) en `color`, dans l'image `img`.

Remarque : la couleur est encodée comme un entier (par exemple `0xff0000`), il faut décoder cette couleur pour la transformer en un `struct pixel` et ses trois champs `r`, `g` et `b`.

Utilisez les opérateurs de décalage `<<` ou `>>` et les opérateurs binaires `|`, `&` ou `^`.

Attention : vérifiez que les coordonnées (x, y) sont valides pour l'image avant d'écrire n'importe où.

Test : Dans la fonction `main`, après la création de votre image vide, et avant d'écrire votre image dans un fichier, dessinez des pixels rouges aux coordonnées $(50, 50)$, $(25, 25)$ et $(2000, 2000)$.

Une fois que vous vous êtes assurés que votre fonction dessine bien des pixels aux coordonnées indiquées, et ne provoque pas d'erreur pour le point $(2000, 2000)$, supprimez ces points de votre fonction `main`.

Question 6. Écrivez une fonction `void draw_rectangle(int x1, int y1, int x2, int y2, int color, image* img)` qui dessine un rectangle de couleur `color` dans l'image `img`, dont le coin en bas à gauche est aux coordonnées (x_1, y_1) et le coin en haut à droite est aux coordonnées (x_2, y_2) .

Test : dans la fonction `main`, ajoutez les instructions nécessaires pour dessiner un rectangle vert dont le coin en bas à gauche est à $(55, 0)$ et le coin en haut à droite à $(95, 40)$.

Question 7. Écrivez une fonction `void draw_circle(int x, int y, int color, int radius, image* img)` qui dessine un cercle de couleur `color` dans l'image `img`, dont le centre est aux coordonnées (x, y) et le rayon est `radius`.

On rappelle que les points (x, y) à l'intérieur d'un cercle de centre (x_C, y_C) et de rayon r sont caractérisés par l'équation :

$$(x_C - x)^2 + (y_C - y)^2 \leq r^2$$

Précision : on considèrera que dessiner un cercle de rayon 0 revient à dessiner son centre (et pas ne rien dessiner).

Test : dessinez un cercle jaune (0xffff00) de centre (15, 85) et de rayon 20.

Question 8. Écrivez une fonction `void draw_line(int xfrom, int yfrom, int xto, int yto, int color, image* pf)` qui dessine une ligne depuis le point $(xfrom, yfrom)$ jusqu'au point (xto, yto) .

Pour ce faire :

- Calculez le nombre de points (`int nbpoints`) à dessiner : il s'agit du nombre maximal entre $(xfrom - xto)$ et $(yfrom - yto)$.
Utilisez la fonction `int abs(int x)`, déclarée dans l'en-tête `<math.h>`, qui permet de calculer la valeur absolue d'un nombre.
- Calculez les incréments (`float dx` et `float dy`) en x et y qui vont être ajoutées à chaque nouveau point.
- Dessinez tous les `nbpoints` points.

Par exemple, pour dessiner la ligne de $(0, 0)$ à $(20, 40)$:

- `nbpoints` sera égal à 40
- `dx` sera égal à 0.5 ; `dy` sera égal à 1
- on dessinera les points $(0, 0)$, $(0.5, 1)$, $(1, 2)$, $(1.5, 3)$, ...

Comme les coordonnées des points sont entières dans notre format, les flottants seront arrondis à l'entier inférieur (par défaut).

Question 9. Dans la fonction `main`, dessinez des lignes de $(55, 40)$ à $(75, 55)$, et de $(75, 55)$ à $(95, 40)$, puis observez le résultat.

Question 10. Écrivez une fonction `void draw_arc(int xc, int yc, int color, int radius, int anglefrom, int angleto, image* pf)` qui dessine un arc de cercle :

- de centre (xc, yc) ;
- de rayon `radius`;
- de couleur `color`;
- depuis l'angle `anglefrom` (en degrés) à l'angle `angleto` (en degrés).

On utilisera les fonction `sin` et `cos` de la librairie `<math.h>`. Attention, ces fonctions prennent leurs arguments comme des angles en radians. Pour convertir un angle en degrés `deg` en radians : `float rad = (float)deg * acos(-1) / 180`²

Les coordonnées d'un point sur un arc de cercle de centre (x_C, y_C) , de rayon r et à un angle ϕ sont obtenues par :

$$(x, y) = (x_C + r \cdot \cos(\phi), y_C + r \cdot \sin(\phi))$$

2. La constante PI n'est pas fournie par la librairie C : on l'obtient avec un appel à `acos(-1)`.

Compilation : Pour compiler avec les fonctions mathématiques, il faut indiquer au compilateur que l'on veut lier l'exécutable avec la librairie mathématique. Pour ce faire, on ajoute l'option `-lm` au compilateur :

```
$ gcc be1.c -lm -o be1
```

(Cette remarque est donnée à titre informatif. En effet, dans notre grande bonté, nous avons déjà inclus cette option dans le Makefile.)

Question 11. Dessinez les arcs suivants, tous de centre (100,0), de l'angle 90 à 180 degrés :

- rayon 90, couleur 0xde0000
- rayon 87, couleur 0xfe622c
- rayon 84, couleur 0xfef600
- rayon 81, couleur 0x00bc00
- rayon 78, couleur 0x009cfe
- rayon 75, couleur 0x000084
- rayon 72, couleur 0x2c009c

Prenez quelques instants pour admirer le résultat.

Question 12. On souhaite pouvoir dessiner des lignes plus épaisses qu'un pixel : rajoutez un paramètre `int width` aux fonctions `draw_line` et `draw_arc` puis utilisez la fonction `draw_circle` au lieu de `draw_pixel`.

Adaptez évidemment les appels à ces fonctions pour y ajouter le paramètre `width`. Pour obtenir une ligne d'épaisseur x pixels, quel rayon de cercle doit-on utiliser ?

Utilisez une épaisseur de 4 pixels pour les arcs de la question précédente.

3 Dessiner des visages

Nous vous proposons le code suivant pour dessiner un visage (masqué, bien entendu). Ce code est dans le squelette qui vous est fourni, à la fin du fichier, commenté.

```
void draw_face(int x, int y, struct face_description* f, image* img){
    // visage
    draw_circle(x, y, 0xffff00, 40, img);

    // oeil droit
    draw_circle(x + 20, y + 10, 0xffffffff, 10, img);
    draw_arc(x + 20, y + 10, 0x000000, 10, 0, 360, 1, img);
    draw_circle(x + 20, y + 5, 0x0000ff, 3, img);

    // oeil gauche
    draw_circle(x - 20, y + 10, 0xffffffff, 10, img);
    draw_arc(x - 20, y + 10, 0x000000, 10, 0, 360, 1, img);
    draw_circle(x - 20, y + 5, 0x0000ff, 3, img);

    // sourcils
    draw_arc(x + 20, y + 10, 0x000000, 15, 45, 135, 3, img);
```

```
draw_arc(x - 20, y + 10, 0x000000, 15, 45, 135, 3, img);

// nez
draw_circle(x, y - 5, 0x00fff0, 7, img);

// bouche
draw_arc(x, y - 10, 0xff0000, 20, 215, 325, 3, img);

// bords du masque
draw_rectangle(x - 21, y - 26, x + 21, y - 4, 0x000000, img);
// masque
draw_rectangle(x - 20, y - 25, x + 20, y - 5, 0xffffffff, img);
// liens du masque
draw_line(x - 20, y - 5, x - 38, y + 5, 0, 1, img);
draw_line(x - 20, y - 25, x - 35, y - 15, 0, 1, img);
draw_line(x + 20, y - 5, x + 38, y + 5, 0, 1, img);
draw_line(x + 20, y - 25, x + 35, y - 15, 0, 1, img);
}
```

Question 13. Appelez la fonction `draw_face` dans la fonction principale, sur une image de 100x100 à la position (50,50). Passez `NULL` pour l'argument de type `face_description` pour le moment.

Écrivez le résultat dans un fichier `visage.bmp`.

Ce visage est bien mignon, mais on aimerait pouvoir le personnaliser, changer la couleur des yeux, du visage, du nez, de la bouche, l'épaisseur des sourcils, etc.

Pour ce faire, nous allons utiliser une structure `face_description`, qui regroupe les différentes caractéristiques du visage que nous souhaitons dessiner.

La structure `face_description` est donnée ci-dessous (disponible également à la fin du fichier, commentée) :

```
struct face_description {
    int skin_color; // la couleur du visage

    int eye_bg_color; // la couleur du fond des yeux
    int eye_fg_color; // la couleur de la pupille

    int eyebrow_color; // la couleur des sourcils
    int eyebrow_width; // l'épaisseur des sourcils

    int nose_color; // la couleur du nez
    int nose_size; // la taille du nez

    int mouth_color; // la couleur de la bouche
    int mouth_width; // l'épaisseur de la bouche

    int mask; // présence du masque (0 ou 1)
    int mask_bg_color; // la couleur du masque
}
```

```
int mask_border_color; // la couleur de la bordure du masque
int mask_link_color; // la couleur des liens du masque
};
```

Question 14. Écrivez une fonction `struct face_description* default_face()` qui définit une structure `face_description` par défaut, c'est-à-dire avec les couleurs proposées dans l'exemple de la fonction `draw_face` ci-dessus.

Question 15. Modifiez la fonction `draw_face` pour prendre en compte l'argument `f` de type `struct face_description*`, et adaptez le code pour utiliser les champs de cette structure au lieu des valeurs constantes.

Vérifiez que tout fonctionne en appelant la fonction `draw_face` avec le résultat de `default_face`.

Nous souhaitons pouvoir changer les caractéristiques du visage sans avoir à recompilier notre programme à chaque fois. Pour ce faire, nous allons utiliser un fichier de description de visage, dont voici un exemple (le fichier `face.txt`) :

```
skin color 3d 25 2f
eye color 33 22 22
eye bg color ff dd ff
eyebrow color ff ff 2d
eyebrow width 5
nose color 00 ff ff
nose size 1
mouth color dc 86 f3
mouth width 1
mask yes
mask color 00 dc dc
mask border color aa bb cc
```

Nous allons lire ce fichier, remplir une structure `face_description` (décrite ci-dessous), et utiliser cette structure pour dessiner le visage correspondant.

Question 16. Écrivez une fonction `struct face_description* read_face(char* file)` qui lit une description de visage depuis le fichier de nom `file`.

- Ouvrez le fichier (`fopen`).
- Lisez le fichier ligne par ligne (`fgets`).
 - `fgets` a besoin d'un buffer (un tableau de caractères) où stocker la ligne lue. Allouez une variable locale à cet effet, de taille 256 caractères. (On supposera que les lignes ne peuvent pas être plus longues que cela)
 - Donnez cette taille (256) en deuxième paramètre à `fgets`.
 - `fgets` renvoie un pointeur vers la chaîne de caractères lue si on a pu lire une nouvelle ligne depuis le fichier, ou `NULL` si on est arrivé à la fin du fichier. La boucle principale pour lire chaque ligne peut donc être de la forme : `while (fgets(line, 256, fd)) { ... }`.
- Examinez chaque ligne en utilisant la fonction `sscanf`

```
int i1, i2;
char c;
int res = sscanf(ligne, "toto %hhx %d %d\n", &c, &i1, &i2);
```


Si `ligne` correspond au format demandé, la variable `res` contiendra le nombre de valeurs trouvées, *i.e.* le nombre de spécificateurs de format effectivement trouvées, soit 3 dans ce cas. Si la ligne ne commence pas par `toto`, `res` vaudra 0.

Le spécifieur `"%hhx"` désigne un caractère noté en hexadécimal.

- Une fois toutes les lignes traitées, fermez le fichier (`fclose`).

4 Bonus

Si vous êtes très rapide, et que vous voulez aller plus loin :

- prenez le nom du fichier de description de visage, et le nom du fichier BMP de sortie en arguments de votre programme principal, de sorte que l'on puisse appeler votre programme comme cela :

```
$ ./be1 face.txt mon-visage.bmp
```

- Ajoutez des lunettes, des cheveux, des oreilles aux visages.
- Le fichier `font.c` contient une description de chaque caractère imprimable sous forme de bitmap. Par exemple, le *bitmap* correspondant au caractère 0 est la ligne suivante :

```
{ 0x3E, 0x63, 0x73, 0x7B, 0x6F, 0x67, 0x3E, 0x00}, // U+0030 (0)
```

Il s'agit d'un tableau de 8 caractères décrivant l'affichage du caractère 0. Chaque caractère (8 bits) correspond à une ligne de pixels, chaque bit du caractère correspond à un pixel.

Par exemple, `0x3E = 0b00111110`, `0x63 = 0b01000011`, `0x73 = 0b01110011`, `0x7b = 0b01111011`, `0x6f = 0b01101111`

`0x3E = 0b00111110`

`0x63 = 0b01100011`

`0x73 = 0b01110011`

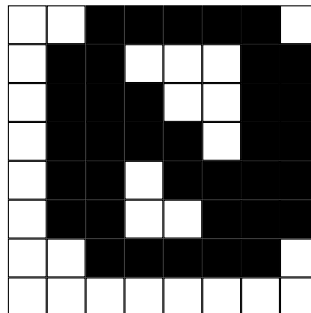
`0x7B = 0b01111011`

`0x6F = 0b01101111`

`0x67 = 0b01100111`

`0x3E = 0b00111110`

`0x00 = 0b00000000`



Écrivez une fonction `draw_letter(unsigned char bitmap[8], int x, int y, image* img)` qui affiche un bitmap `bitmap` à la position (x, y) dans l'image `img`.

Écrivez une fonction `draw_string(char* s, int x, int y, image* img)` qui affiche une chaîne de caractères `s` sur une image, à une position donnée.

Affichez votre prénom sous un visage.

- Laissez libre cours à votre imagination !