Dessins

1 Image RGB

Une image RBG (*Red Green Blue*) de n lignes et p colonnes peut être vue comme une matrice dont les éléments sont des triplets (r, g, b) d'entiers. Le plus souvent, et c'est ce que nous ferons ici, les valeurs r, g et b varient entre 0 et 255 (ce qui permet de les coder sur un octet chacune). Dans ce cas :

— $(255, 0, 0)$ est un rouge primaire,
— $(0, 0, 255)$ est un bleu primaire,
— $(255, 255, 255)$ est le blanc pur,
— $(0, 0, 0)$ est le noir pur,
— $(50, 50, 50)$ est un gris assez sombre,
— $(255, 0, 255)$ donne du magenta.

La synthèse des couleurs est *additive* (comme des lumières colorées qui se superposent) et non *soustractive* (comme des peintures de couleurs différentes que l'on mélange).

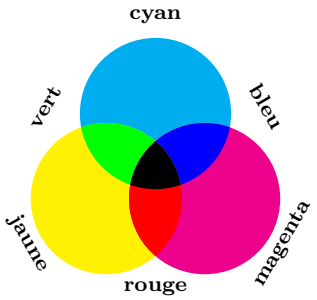


FIGURE 1 – Synthèse soustractive.

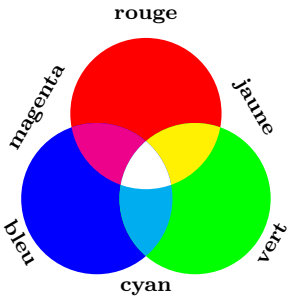


FIGURE 2 – Syntèse additive.

La manière « usuelle » d'indexer les pixels d'une image est d'utiliser des coordonnées correspondant à celles dans une matrice, avec des lignes numérotées de 0 à $n-1$ et des colonnes numérotées de 0 à $p-1$.

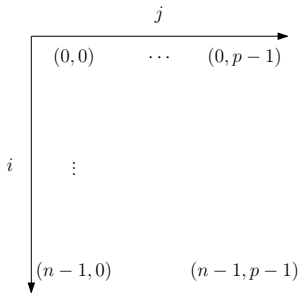


FIGURE 3 – Coordonnées image

Nous utiliserons ces coordonnées pour stocker notre tableau de pixels, mais pour tracer des lignes, cercles et autres il sera plus agréable d'utiliser les coordonnées (x, y) usuelles (avec x qui correspond à un axe des abscisses de gauche à droite et y à un axe des ordonnées de bas en haut).

2 Types utilisés

On va représenter un pixel par un tableau de 3 entiers. Il y a des types spécifiques pour les entiers de différentes tailles en C, mais pour simplifier les valeurs seront juste des `int`. On pourrait donc définir la couleur rouge, par exemple, par :

```
int red[3] = {255, 0, 0}; // red est un tableau de trois int
```

Pour alléger les notations, on va donner un nom au type qui code une couleur `rgb` : en C, il faut pour cela utiliser le mot-clé `typedef`.

```c
typedef int rgb[3]; // rgb est le *type* "tableau de trois int"
```

On peut ensuite définir des couleurs de manière plus simple :

```c
rgb red = {255, 0, 0};
rgb green = {0, 255, 0};
rgb blue = {0, 0, 255};
rgb black = {0, 0, 0};
rgb white = {255, 255, 255};
```

Pour l'image, on va utiliser un tableau bidimensionnel dont les éléments seront de type `rgb`. Ce tableau bidimensionnel sera une variable globale, et sa taille sera connue statiquement (c'est-à-dire à la compilation). On pourrait procéder ainsi :

```c
const int height = 600;
const int width = 800;

rgb canvas[600][800];
```

Cette approche est la seule à être vraiment au programme (pour un tableau statique). Elle permet d'utiliser les constantes `height` et `width` dans le code (pour des bornes de boucles par exemple), mais si l'on veut modifier la largeur de l'image il faut bien penser à le faire à deux endroits : la définition de `height` et celle de `canvas`. Nous allons utiliser la « bonne » manière de procéder, qui est d'utiliser la *directive de pré-processeur* `#define`.

```c
#define HEIGHT 600
#define WIDTH 800

rgb canvas[HEIGHT][WIDTH];
```

Essentiellement, l'effet de `#define HEIGHT 600` est de remplacer textuellement toutes les occurrences de `HEIGHT` dans le code par `600` : on parle de *macro*. C'est une pratique très courante en C, qui permet de pallier partiellement certaines faiblesses du langage, mais c'est assez piégeux et ce n'est pas au programme. Nous nous limiterons strictement à l'utilisation faite ici : définir des constantes pour pouvoir les utiliser comme tailles de tableaux statiques.

3 Format d'image utilisé

Nous n'allons pas afficher « en direct » les images que nous allons créer : nous allons sauvegarder l'image dans un fichier et utiliser ensuite une application dédiée au visionnage d'images. Nous allons utiliser le format le plus simple du monde : le format PPM. Un fichier PPM est un fichier texte obéissant à des règles très simples.

- Le fichier commence par une ligne réduite à `"P3"`. C'est ce qu'on appelle un *magic number* qui permet d'indiquer le format du fichier.
- La ligne suivante contient deux entiers séparés par une espace : le premier indique la largeur de l'image (en nombre de pixels), le second la hauteur. Attention, ce n'est pas forcément le plus naturel.
- La ligne suivante contient un entier indiquant la valeur maximale des composantes RGB. Si cette valeur est 100 par exemple, alors un pixel blanc sera codé par $(100, 100, 100)$. Nous utiliserons 255 comme valeur maximale, ce qui est le plus standard.
- Ensuite, on donne les valeurs R, G et B de chaque pixel : d'abord la première ligne (celle du haut) de gauche à droite, puis la deuxième ligne, etc. Chaque valeur doit être séparée de la précédente et de la suivante par un caractère d'espacement (espace, retour à la ligne, etc). En pratique, on mettra les trois valeurs correspondant à un pixel sur une ligne, séparées par des espaces, puis on reviendra à la ligne pour le pixel suivant.

```
P3
3 2 # 3 colonnes, 2 lignes
255
255 0 0 # pixel rouge en haut à gauche
255 0 0
0 0 255
0 255 0 # pixel vert en bas à gauche
255 255 255
150 150 150
```

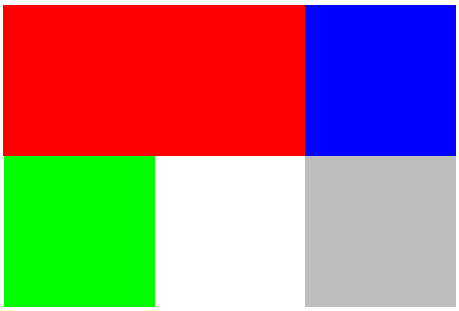


FIGURE 4 – Exemple de fichier au format PPM.

4 Création du fichier

Dans le fichier `squelette.c` fourni, vous trouverez quelques définitions de variables globales et une fonction

```
void put_pixel(int x, int y, rgb c)
```

Cette fonction agit en modifiant le tableau global `canvas`, et elle accepte des coordonnées « mathématiques » usuelles : l'origine est en bas à gauche, l'axe des x va de gauche à droit et l'axe des y de bas en haut. Pour écrire des valeurs dans `canvas`, on passera systématiquement par cette fonction.

Remarque

⇨ Si les coordonnées passées à la fonction `put_pixel` sont « hors cadre », elle ne fait rien. Cela simplifiera les choses par la suite si par exemple on veut tracer un disque dont seule une partie se trouve dans l'image.

4.1 Canvas

La fonction `print_canvas` doit afficher sur la sortie standard, à l'aide de la fonction `printf`, le contenu d'un fichier PPM correspondant au tableau `canvas`. Attention, la case `canvas[i][j]` contient le pixel de coordonnées (i, j) dans le système décrit à la figure **??**.

```
void print_canvas(void)
```

Avec les instructions présentes dans le `main` fourni, on devrait à la compilation obtenir exactement l'affichage de la figure **??**, sans les commentaires, bien sûr.

4.2 Redirection de sortie

Pour pouvoir *afficher* l'image graphiquement, nous devons la sauvegarder dans un fichier. On pourrait utiliser une variante de la fonction `printf` permettant d'écrire dans un fichier, mais nous allons choisir une autre solution, qui utilise les fonctionnalités du `shell`.

Si l'on exécute une commande que l'on fait suivre par `> nom`, alors la *sortie standard* de la commande est *redirigée* vers le fichier `nom`. Cela signifie que tout ce qui aurait normalement été affiché lors de l'exécution de la commande est à la place écrit dans le fichier `nom`. Si ce fichier n'existe pas il est créé ; s'il existe déjà, il est entièrement écrasé.

La commande `echo` permet d'afficher quelque chose directement depuis le `shell`. Par exemple, `echo hello` affichera `hello` à l'écran :

```
mpi@lazos:~$ echo hello
hello
```

En redirigeant la sortie, on peut écrire dans un fichier :

```
mpi@lazos:~$ echo Hello, World! > hello.txt
```

Cette commande n'affiche rien, mais on a bien créé le fichier. On peut l'éditer avec un éditeur de texte, ou simplement afficher son contenu à l'aide de la commande `cat` :

```
mpi@lazos:~$ cat hello.txt
Hello, World!
```

1. En utilisant la même technique de redirection, créer un fichier `test.ppm` en exécutant votre programme compilé.
2. Ouvrir ce fichier avec la visionneuse d'image `eog` (Eye Of Gnome). Il faudra sans doute zoomer pour voir quelque chose car l'image fait 3 pixels de large.
3. Modifier le programme pour créer une image 600×400 entièrement rouge, et vérifier que vous arrivez bien à l'afficher.

5 Primitives simples

Dans toute la suite du sujet, on fera bien attention à traiter correctement tous les cas, sans supposer abusivement que tel point est a gauche de tel autre, ou qu'ils ne sont pas l'un au dessus de l'autre, ou que sais-je encore ... D'un autre côté, on essaiera d'écrire du code raisonnablement concis.

1. Écrire une fonction permettant de tracer une ligne horizontale. Cette fonction prendra en entrée une ordonnée, deux abscisses et une couleur. Attention, rien ne dit qu'on a $x_0 \leqslant x_1$.

```
void draw_h_line(int y, int x0, int x1, rgb c)
```

2. Écrire de même une fonction pour tracer une ligne verticale.

```
void draw_v_line(int x, int y0, int y1, rgb c)
```

3. Écrire une fonction permettant de tracer un rectangle aligné avec les axes : deux côtés verticaux et deux côtés horizontaux. Le rectangle sera spécifié par la donnée de deux coins opposés ainsi que par une couleur.

```
void draw_rectangle(int x0, int y0, int x1, int y1, rgb c)
```

4. Écrire une fonction permettant de « remplir » un rectangle du même type qu'à la question précédente.

```
void fill_rectangle(int x0, int y0, int x1 int y1, rgb c)
```

5. Écrire une fonction permettant de remplir un disque spécifié par son centre et son rayon.

```
void fill_disk(int xc, int yc, int radius, rgb c)
```

6 Mélange de couleurs

Si on dispose de deux couleurs $c := (r, g, b)$ et $c' := (r', g', b')$, on peut pour $\alpha, \beta \in \mathbb{R}$ définir

$$\mathrm{mix}(c, c', \alpha, \beta) := \left| \begin{array}{lcl} r_m & := & \mathrm{clamp}(\alpha r + \beta r') \\ g_m & := & \mathrm{clamp}(\alpha g + \beta g') \\ b_m & := & \mathrm{clamp}(\alpha b + \beta b') \end{array} \right.$$

La fonction clamp étant définie par

$$\mathrm{clamp}(x) := \begin{cases} 0 & \text{si } x < 0 \\ 255 & \text{si } x > 255 \\ x & \text{sinon.} \end{cases}$$

1. Écrire la fonction `clamp` :

```
int clamp(double x)
```

2. Pour écrire la fonction `mix`, on est confronté à un problème. En effet, on voudrait que cette fonction renvoie un tableau, ce qui n'est pas directement possible. On va donc écrire une fonction ayant le prototype suivant :

```
void mix(rgb c0, rgb c1, double alpha, double beta, rgb result)
```

On fournira donc à `mix` un tableau `result` qu'elle modifiera pour y mettre son résultat.

3. Écrire une fonction ayant le prototype suivant :

```
void draw_h_gradient(int y, int x0, int x1, rgb c0, rgb c1)
```

Cette fonction tracera une ligne horizontale, avec le point d'abscisse x_0 de couleur `c0`, le point d'abscisse x_1 de couleur `c1`, et les points intermédiaires coloriés grâce à une interpolation linéaire de ces deux couleurs.

4. Écrire de même une fonction :

```
void fill_disk_gradient(int xc, int yc, int radius,
                        rgb c_center, rgb c_edge)
```

Cette fois, l'interpolation de couleur se fera suivant la distance entre le point et le centre : couleur `c_center` si elle vaut 0, couleur `c_edge` si elle vaut `radius`, et une interpolation linéaire entre.

5. Écrire une fonction `get_pixel` qui remplit le tableau `result` avec le contenu de la case de `canvas` correspondant aux coordonnées (x, y). On s'inspirera de `put_pixel`, et on ne fera rien dans le cas où les coordonnées données sont hors-cadre.

```
void get_pixel(int x, int y, rgb result)
```

6. Écrire une fonction `mix_pixel` qui remplace le pixel de coordonnées (x, y) par le mélange de sa valeur actuelle (avec coefficient β) et de la couleur `c` fournie (avec coefficient α).

```
void mix_pixel(int x, int y, double alpha, double beta, rgb c)
```

7. Écrire une fonction `add_disk` permettant d'obtenir facilement un dessin ressemblant à la figure **??**.

```
void add_disk(int xc, int yc, int radius, rgb c)
```

7 Tracé de lignes par l'algorithme de Bresenham

Pour les algorithmes qui suivent, il faut clarifier la relation entre les coordonnées entières `x` et `y` d'un pixel et les coordonnées mathématiques dans le plan. On considère que c'est le centre du pixel qui a pour coordonnées (x, y) dans le plan.

7.1 Segment oblique

Pour l'instant, on ne sait tracer que des lignes verticales ou horizontales, ce qui est un peu limité. On souhaite à présent tracer la ligne reliant un point (x_0, y_0) au point (x_1, y_1), sans hypothèses particulières sur ces points.

Pour commencer, on traite le cas où la droite possède une pente comprise entre 0 et 1 et où $x_0 \leqslant x_1$. Il faut alors sélectionner exactement un pixel dans chaque colonne de x_0 à x_1 :

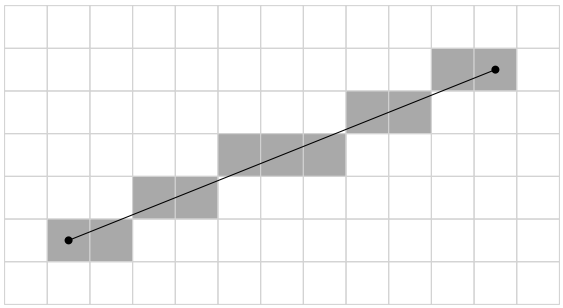


FIGURE 5 – Résultat de l'algorithme de Bresenham

Si l'on suppose que l'on traite les pixels de gauche à droite, il y a donc deux pixels candidats à chaque étape :
— celui situé immédiatement à droite du pixel actuel, de coordonnées $(x + 1, y)$
— celui situé en diagonale en haut à droite, de coordonnées $(x + 1, y + 1)$.

On veut choisir celui des pixels situé le plus près de la ligne idéale, c'est-à-dire celui dont le centre est situé le plus près de cette ligne. Une première option est de mettre l'équation de la droite sous la forme $y = y_0 + m(x - x_0)$. Comme on connaît systématiquement la valeur de x (qui est incrémenté à chaque étape), on peut calculer y et l'arrondir à l'entier le plus proche pour choisir quel pixel « allumer ». Pour réaliser un tel arrondi en C, on pourra utiliser la fonction `double round(double)`, et transformer son résultat en un entier :

```
double x = 3.76;
double x_rounded = round(x) // x_rounded vaut 4.0
int n = x_rounded           // n vaut 4

// Ou en une seule étape :
int n = round(3.76)         // n vaut 4
```

1. Écrire une fonction utilisant cet algorithme pour tracer une ligne de pente comprise entre 0 et 1.

   ```
   void draw_line(int x0, int y0, int x1, int y1, rgb c)
   ```

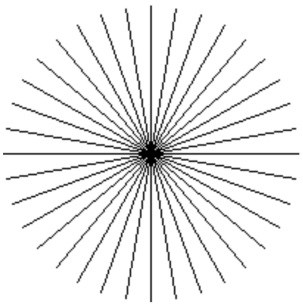
2. Compléter cette fonction pour qu'elle traite correctement tous les autres cas. On essaiera de ne pas se perdre dans le code !
3. Écrire une fonction ayant le prototype suivant :

   ```
   void draw_spokes(int xc, int yc, int radius, int nb_spokes, rgb c)
   ```

   Cette fonction tracera `nb_spokes` rayons du cercle de centre (x_c, y_c) et de rayon `radius`. Le premier de ces rayons sera horizontal et les autres seront régulièrement espacés en terme d'angle.

L'inclusion de l'entête `math.h` permet d'accéder aux fonctions `double cos(double)` et `double sin(double)` et à la constante `M_PI`.

```
...
#define HEIGHT 201
#define WIDTH 201
...
int main(void){
    fill_rectangle(0, 0, 200, 200, white);
    draw_spokes(100, 100, 100, 36, black);
    return 0;
}
```



4. Le « vrai » algorithme de Bresenham ne travaille qu'avec des entiers (à l'époque où il a été inventé, les calculs sur les flottants étaient extrêmement coûteux) et minimise les calculs à effectuer. Il se base sur l'observation suivante (toujours dans le cas d'une pente comprise entre 0 et 1) : pour choisir entre le point $(x + 1, y)$ et le point $(x + 1, y + 1)$, il suffit de déterminer si le point $(x + 1, y + 1/2)$ est situé au-dessus ou en dessous de la droite.

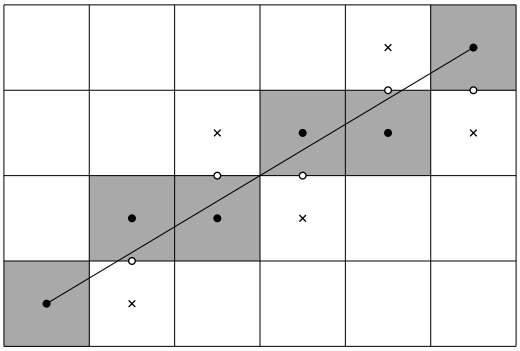


FIGURE 6 – À chaque étape, on détermine si le point marqué par un cercle est au-dessus ou en dessous de la droite. On élimine un candidat (marqué par une croix) et l'on choisit l'autre (marqué par un disque).

(a) On note $\Delta_x \coloneqq x_1 - x_0$ et $\Delta_y \coloneqq y_1 - y_0$. On se place pour l'instant dans le cas où $0 \leqslant \Delta_y \leqslant \Delta_x$, et l'on définit

$$f(x, y) \coloneqq 2\big(\Delta_x y - \Delta_y x\big)$$
$$D(x, y) \coloneqq f(x + 1, y + 1/2) - f(x_0, y_0)$$

i. Montrer que le point $(x{+}1, y{+}1/2)$ est au-dessus de la droite (au sens large) si et seulement si $D(x, y) \geqslant 0$.
ii. Calculer $D(x_0, y_0)$.
iii. Déterminer $D(x + 1, y)$ et $D(x + 1, y + 1)$ en fonction de $D(x, y)$.
iv. En déduire une fonction

```
void bresenham_low(int x0, int y0, int x1, int y1, rgb c)
```

Cette fonction aura pour précondition $0 \leqslant \Delta_y \leqslant \Delta_x$, travaillera uniquement avec des entiers, et n'effectuera que des additions, soustractions et multiplications par 2.

(b) Modifier la fonction `bresenham_low` pour que sa précondition devienne $|\Delta_y| \leqslant \Delta_x$.
(c) Écrire une fonction `bresenham_high` ayant comme précondition $|\Delta_x| \leqslant \Delta_y$.
(d) Écrire finalement une fonction

```
void bresenham(int x0, int y0, int x1, int y1, rgb c)
```

permettant de traiter correctement tous les cas.

7.2 Cercle

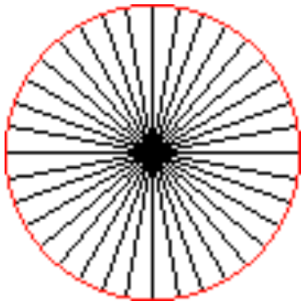
On peut adapter l'algorithme de BRESENHAM pour tracer un cercle (dont le centre est à coordonnées entières et le rayon entier). L'idée est la suivante :

— n part du point le plus à droite, que l'on allume ;
— on considère le point situé juste au-dessus, et celui situé en diagonale en haut à gauche ;
— pour chacun de ces points, on calcule le carré de la distance au centre ;
— on choisit le point pour lequel cette valeur est la plus proche du rayon au carré, et l'on recommence le même processus à partir de ce point ;
— on s'arrête au bon moment, après avoir tracé une partie du cercle ;
— les autres parties du cercle sont tracée par symétrie.

1. Quand faut-il arrêter le processus ?
2. Écrire une fonction

```
void draw_circle(int xc, int yc, int radius, rgb c)
```

```
...
#define HEIGHT 101
#define WIDTH 101
...
int main(void){
    fill_rectangle(0, 0, 100, 100, white);
    draw_spokes(50, 50, 50, 36, black);
    draw_circle(50, 50, 50, red);
    return 0;
}
```



Remarque

⇨ Tout comme pour une droite, le vrai algorithme évite de recalculer entièrement les distances au centre à chaque étape, et utilise le point situé au milieu du segment formé par les candidats pour trancher. Il n'est pas très difficile de trouver les relations rendant cela possible.