

Le but de ce TP est de réaliser des manipulations de base sur des images au format BMP comme par exemple des filtres. Vous devez avoir bien travaillé le CM09 sur les structures, pour faire ce dernier TP9. On rappelle qu'une image au format BMP est composée de pixels dont la couleur est déterminée par 3 composantes : rouge, verte et bleue. On peut donc représenter un pixel par un triplet d'entiers entre 0 et 255, par exemple (0,0,0) représente un pixel noir et (255,0,0) un pixel rouge.

En important le fichier projet INF1\_TP9.zip dans Eclipse (ou l'éditeur de votre choix<sup>1</sup>), vous disposez des fonctions suivantes pour manipuler images et pixels :

```
- Pixel p = new Pixel(14,21,56); //Crée un pixel de composantes RVB (14,21,56)
- p.r = 34; Modifie la composante rouge du pixel. Idem avec p.v et p.b
- BitMap img = new BitMap(300,200); //Crée un objet de type BitMap qui représente
  une image de largeur 300 et de hauteur 200 (en pixels).
- BitMap img = aPartirDe("tiger.bmp");
  //Charge l'image tiger.bmp dans un objet BitMap
- int l = img.largeur(); int h = img.hauteur()
  //Récupère la largeur et la hauteur de l'image
- img.set(10, 15, p); //Transforme le pixel situé aux coordonnées (10, 15) (pour
  (hauteur, largeur)) en le pixel p
- Pixel p2 = img.get(0,1); //Récupère le pixel situé aux coordonnées (0,1)
- BitMap.enregistreBitMap(img, "image.bmp");
  //Enregistre l'image img dans le fichier image.bmp
  //Pour voir apparaître une image que vous enregistrez pour la première fois
  (ce qui crée un nouveau fichier), vous aurez besoin de rafraichir le projet
  dans votre project explorer: clic droit sur le projet puis Refresh.
```

---

### Exercice 1 : Verifier pixel

Les composantes d'un pixel doivent toujours être des entiers entre 0 et 255. Dans les exercices suivants, il est possible que la valeur d'une composante devienne négative ou supérieure à 255, ce qui résulte en un pixel invalide.

Écrire donc une fonction `void verifierPixel(Pixel p)` qui permet de corriger les valeurs du pixel si nécessaire. Si une composante a une valeur  $< 0$ , alors on la corrigera en 0. De même, si une composante a une valeur  $> 255$ , on la corrigera en 255. Par exemple, le pixel (300,-5,24) sera corrigé en (255,0,24). Attention, il ne faut **pas** renvoyer un nouvel objet de type `Pixel`, mais modifier celui donné en argument.

**Dans la suite, pensez toujours à utiliser cette fonction pour vérifier les pixels quand c'est nécessaire!**

### Exercice 2 : Figures simples

**2.a]** Écrire une fonction `simpleRectangle(int n, int m)` qui renvoie une nouvelle `BitMap` blanche de largeur  $n$  et de hauteur  $m$  qui contient un rectangle noir de 10 Pixels d'épaisseur à 10 Pixels des bords. Vous pouvez supposer que  $n > 50$  et  $m > 50$ .

Par exemple pour  $n = 100, m = 100$ , cela donne la Fig. 1 (les pixels blancs sont montrés en gris, pour les voir sur la feuille dont le fond est blanc).

---

1. Si vous compilez votre programme Java en ligne de commande, depuis le dossier INF1\_TP9/src/, il faut écrire `javac -cp "../lib/outils.jar:." TP9.java` et l'exécuter avec `java -cp "../lib/outils.jar:." TP9`, en adaptant éventuellement le chemin vers le `.jar`.

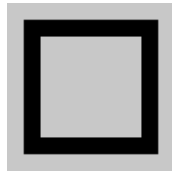


FIGURE 1 – Une simple rectangle de taille  $n = 100$   $m = 100$ .

**2.b]** Écrire une fonction `simpleCroix(int n, int m)` qui renvoie une nouvelle `BitMap` blanche de largeur  $n$  et de hauteur  $m$  qui contient une croix rectangulaire noire. Le trait vertical est de 10 `Pixels` de large si la largeur de l'image est paire et de 11 si la largeur est impaire. Le trait horizontal a la même taille. La croix à une séparation de 10 `Pixels` avec les bords. Vous pouvez supposer que  $n > 50$  et  $m > 50$ . Par exemple pour  $n = 100, m = 100$ , cela donne la Fig. 2.

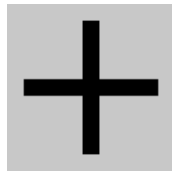


FIGURE 2 – Une simple croix de taille  $n = 100$   $m = 100$ .

---

Pour tous les exercices suivants, vous pouvez utiliser comme exemple l'image `tiger.bmp` (en Fig. 3) qui est fournie dans le fichier squelette du projet `INF1_TP9.zip`.



FIGURE 3 – La figure exemple : un tigre du Bengale.

### Exercice 3 : Extraire les composantes



FIGURE 4 – Les composantes rouges, vertes, et bleues de la figure du tigre.

Écrire une fonction `composanteRouge(Bitmap img)` qui crée une nouvelle image constituée uniquement de la composante rouge de l'image passée en paramètre (les autres composantes étant fixées à 0). De même pour `composanteVerte` et `composanteBleue`. Par exemple avec la figure du tigre, cela donne la Fig. 4 pour les trois composantes.

#### Exercice 4 : Inverser les couleurs

Écrire une fonction `inverser` qui inverse les couleurs d'une image, c'est à dire que chaque composante est transformée en sa différence à 255. Par exemple le pixel  $(10, 126, 2)$  sera transformé en le pixel  $(255-10, 255-126, 255-2) = (245, 129, 253)$ . Par exemple avec la figure du tigre, cela donne la Fig. 5. Si cela vous évoque un négatif de photo argentique, vous avez raison, c'est exactement ça !



FIGURE 5 – L'inversion des couleurs de la figure du tigre.

#### Exercice 5 : Bruit aléatoire

Écrire une fonction `bruit`, qui ajoute un bruit d'amplitude (entière) maximale  $m$  à une image `BitMap` et renvoie une nouvelle `BitMap`. Ceci est réalisé en ajoutant un entier aléatoire, uniformément compris entre  $-m$  et  $m$ , à chaque composante d'un pixel. Vous pouvez pour cela utiliser la fonction `entierAleatoire(-m, m)` qui est fournie dans le projet. Par exemple avec la figure du tigre, cela donne la Fig. 6.



FIGURE 6 – Ajout d'un bruit aléatoire d'amplitude maximale  $m = 64$  (à gauche) et  $m = 128$  (à droite) de la figure du tigre.

#### Exercice 6 : Filtres

**6.a]** Écrire une fonction `filtreMonochrome` qui transforme une image couleur en une image en noir et blanc dont la valeur de chaque pixel est déterminée ainsi : on calcule la valeur moyenne des composantes, si cette valeur est  $< 128$  alors le pixel devient noir, sinon il devient blanc. Par exemple avec la figure du tigre, cela donne la Fig. 7.



FIGURE 7 – Filtre monochrome de la figure du tigre.

**6.b]** Écrire une fonction `filtreGris` qui transforme une image couleur en une image en teintes de gris. Un pixel gris est obtenu en fixant les trois composantes à la même valeur, par exemple un pixel (5,5,5) sera gris foncé (de teinte 5) et un pixel (220,220,220) sera gris clair (de teinte 220). Pour choisir la teinte de gris que va prendre un pixel, essayez les deux manières suivantes :

- la valeur de la teinte correspond à la valeur moyenne des composantes,
- la valeur de la teinte d'un pixel  $(r, v, b)$  est calculée<sup>2</sup> avec  $0.2125 * r + 0.7154 * v + 0.0721 * b$ .

Quelle version vous semble la plus réaliste ? (challenge) essayez avec d'autres images.

Par exemple avec la figure du tigre, cela donne la Fig. 8, qui compare à gauche la première option et à droite la deuxième option.



FIGURE 8 – Deux façons d'afficher en noir et blanc la figure du tigre.

**6.c]** (challenge) Écrire une fonction `filtreSepia` qui applique un filtre sépia sur l'image. Pour cela, si on note  $(r_1, v_1, b_1)$  le pixel de l'image d'origine, et  $(r_2, v_2, b_2)$  le pixel de l'image d'arrivée, alors on a la relation (vecteur = matrice  $\times$  vecteur) suivante

$$\begin{pmatrix} r_2 \\ v_2 \\ b_2 \end{pmatrix} = \begin{pmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.686 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{pmatrix} \times \begin{pmatrix} r_1 \\ v_1 \\ b_1 \end{pmatrix}$$

C'est à dire  $r_2 = 0.393 * r_1 + 0.769 * v_1 + 0.189 * b_1$ ,  $v_2 = 0.349 * r_1 + 0.686 * v_1 + 0.168 * b_1$  et  $b_2 = 0.272 * r_1 + 0.534 * v_1 + 0.131 * b_1$ .

Si vous avez un doute sur le calcul à effectuer, un test unitaire est à votre disposition. Pour l'utiliser vous devez créer une fonction `Pixel calculSepia(Pixel p)` qui transforme le `Pixel p` en son résultat par le filtre sépia. Attention, dans le test unitaire l'arrondi ce fait à la valeur inférieure.

Par exemple avec la figure du tigre, cela donne la Fig. 9.



FIGURE 9 – Filtre sépia, qui donne une apparence de “photo ancienne”.

---

2. Ces valeurs viennent de recherches sur l'anatomie de l'œil humain. La composante verte est multipliée par une constante bien plus grande que la composante bleue, pour refléter le fait que l'œil humain est bien plus sensible au vert qu'aux autres couleurs. Beaucoup expliquent cela en disant que l'homme est un primate et ses ancêtres ont vécu des millions d'années dans des forêts, dans des environnements où il était très important de pouvoir différencier les différentes teintes de verts de la végétation. De même, l'œil humain serait moins sensible au bleu qu'au rouge car moins présent dans la nature.



## Exercice 7 : Postérisation

**Postériser** une image revient à réduire le nombre de couleurs de l'image. Pour cela, on arrondira les composantes rouge, verte et bleue de chaque pixel au multiple le plus proche d'un entier donné. Écrire une fonction `posteriser(Bitmap img, int n)` qui postérise une image en arrondissant chaque composante à un multiple de `n`. Par exemple, avec `n = 64`, si un pixel a une composante rouge de 78, alors on changera cette valeur à 64. Si la composante vaut 127, alors on changera cette valeur à  $128 = 2 \times 64$ .

Par exemple avec la figure du tigre, cela donne la Fig. 10.

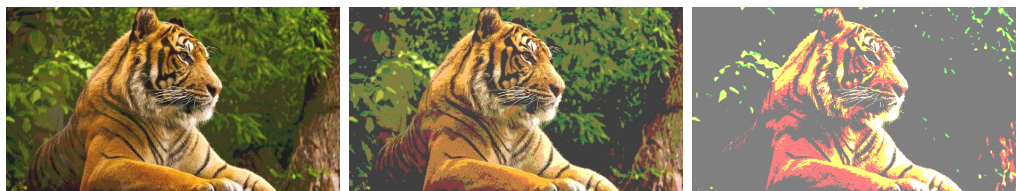


FIGURE 10 – Trois exemples de postérisations avec  $n = 32$  (à gauche),  $n = 64$  (au centre) et  $n = 128$  (à droite) de la figure du tigre. Cette technique de postérisation est à la base de la méthode de compression d'image utilisée dans le format GIF, peut-être avez-vous déjà observé que des GIF trop compressés ont des couleurs toutes similaires ?

## Exercice 8 : Flou (Challenge)

Une manière simple de flouter une image est de transformer chaque pixel en la moyenne de ses 8 pixels adjacents (ainsi que lui même). C'est à dire que pour calculer la nouvelle valeur de la composante rouge d'un pixel, on calculera la moyenne de l'ancienne valeur du pixel plus les valeurs des composantes rouges de chaque pixel adjacent. De même pour les composantes vertes et bleues. Écrire donc une fonction `flou` qui permet de réaliser cette transformation. Pour faciliter les choses, on pourra simplement recopier les pixels présents sur la bordure. Par exemple cela donne la Fig. 11.



FIGURE 11 – Comparaison entre la figure originale et la figure floutée, pour le tigre : en zoomant on voit que la figure floutée est moins nette que l'originale. A droite, le résultat d'avoir appliqué deux fois la fonction `flou`. **Remarque** la version précédente du sujet montrait une image de plus en plus blanche quand on appliquait `flou`, c'était une erreur dans la correction, merci à l'élève l'ayant signalée.

## Exercice 9 : Bordure (Challenge)

Écrire une fonction `bordure(Bitmap img, int n)` qui ajoute une bordure de `n` pixels à l'image (en créant donc une image plus grande). Commencer par faire une bordure d'une couleur unie (en passant par exemple un pixel en paramètre supplémentaire), puis essayer de faire des bordures plus jolies (rayures horizontales/verticales/diagonales, dégradé, etc.).