Chapitre 6. Manipulations d'images

6.1 Manipulations d'images

Jusqu'ici on a seulement peint des pixels, on peut également récupérer la couleur d'un pixel :

`couleurPixel(img:image, x:int, y:int)`	Retourne la couleur du pixel (x, y) dans l'image *img*, par exemple : `(r,g,b) = couleurPixel(img, 10, 10)`

Il est très courant d'appliquer un *filtre* sur une image, c'est-à-dire un calcul sur chacun des pixels de l'image. L'instruction

```
(r,g,b) = couleurPixel(img, 10, 10)
```

récupère les valeurs r, g et b de la couleur du pixel de coordonnées $(10, 10)$. Par exemple, pour ne conserver que la composante rouge du pixel $(10, 10)$, on peut utiliser :

```
colorierPixel(img, 10, 10, (r,0,0))
```

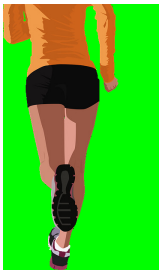
Exercice 6.1.1 Écrire une fonction `filtreRouge(img:image)` qui, pour chaque pixel de l'image, ne conserve que la composante rouge comme expliqué ci-dessus. Testez-la sur la photo de théière. Faites de même pour le vert et le bleu et affichez les trois résultats ainsi que l'image d'origine côte à côte. Remarquez notamment que pour le bleu il n'y a pas d'ombre en bas à droite. En effet, la source lumineuse en haut à gauche ne contient pas de bleu.

Exercice 6.1.2 Écrire une fonction `modifierLuminosite(img:image,facteur:float)` qui pour chaque pixel multiplie par *facteur* les valeurs des trois composantes r, g, et b. Remarquez que la fonction `colorierPixel` n'apprécie pas que l'on donne des valeurs non entières. Utilisez donc la fonction `int(truc)` qui arrondit `truc` à l'entier inférieur.

Testez les facteurs `2`, `1.2`, `0.8`, `0.5` sur la photo de théière (n'oubliez pas de recharger la photo à chaque fois pour éviter de cumuler les effets).

Exercice 6.1.3 On dispose d'une photographie d'un personnage prise sur un fond vert (sur la photocopie le vert apparaît en gris clair, l'image est disponible sur le site du cours). On souhaite incruster ce personnage sur un autre fond comme dans l'exemple suivant. On va procéder étape par étape, en commençant par une version simpliste qui ne réalisera pas complètement l'objectif, puis on va la corriger petit à petit pour obtenir le résultat voulu.

Les images `personnage.png` et `fond.jpg` sont disponibles en téléchargement sur le site du cours.



`personnage.png`

`fond.jpg`

incrustation

1. Écrire et tester une fonction `copieCoinImage(avantPlan, fond)` qui copie l'image `avantPlan` dans l'image `fond`. On place l'image en avant plan en haut à gauche de l'image de fond (on fait correspondre leurs coins supérieurs gauches). On suppose également que l'image de fond est assez grande pour contenir toute l'image à copier.

2. Écrire et tester une fonction `copieImage(avantPlan, fond, dx, dy)` qui copie l'image `avantPlan` dans l'image `fond` en faisant correspondre le pixel `(0,0)` de l'image d'avant plan avec le pixel `(dx,dy)` de l'image de fond, i.e. on place l'image d'avant-plan à un endroit voulu dans l'image de fond.

3. On suppose que le fond vert, qui ne doit pas apparaître dans l'image résultat, est composé uniquement de pixels de couleur (0,255,0). Écrire et tester une fonction `incrusterImage(avantPlan, fond, dx, dy)` qui incruste sans copier les pixels verts l'image `avantPlan` dans l'image `fond` tout en faisant correspondre le pixel `(0,0)` de l'image d'avant plan avec le pixel `(dx,dy)` de l'image de fond.

4. On souhaite fabriquer l'image donnée en exemple. Il s'agit maintenant de centrer horizontalement le coureur qui incruste l'image `imagePersonnage` dans l'image `fond`.

5. On souhaite maintenant ajouter un cadre en pointillé autour de l'incrustation. Écrire une fonction `rectanglePointille(image,x1,y1,x2,y2,c)` qui dessine dans l'image le rectangle de couleur `c` défini par `(x1,y1,x2,y2)` en n'écrivant qu'un pixel sur deux.

Exercice 6.1.4 Écrire une fonction `monochrome(img)` qui pour chaque pixel, calcule la moyenne $lum = \frac{r+g+b}{3}$ des composantes r, g, b, et peint le pixel de la couleur (lum, lum, lum).

En observant bien, les éléments verts semblent cependant plus foncés que sur la photo d'origine. L'œil humain est effectivement peu sensible au bleu et beaucoup plus au vert. Une conversion plus ressemblante est donc d'utiliser plutôt $l = 0.3 * r + 0.59 * g + 0.11 * b$. Essayez, et constatez que les éléments verts ont une luminosité plus fidèle à la version couleur.

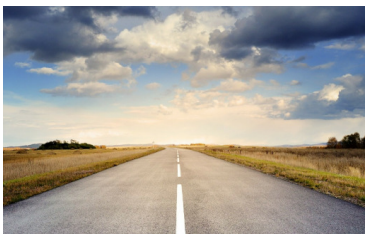
Exercice 6.1.5 Écrire une fonction `noirEtBlanc(img)` qui convertit une image monochrome, telle que produite par la fonction `monochrome` de l'exercice 6.1.4, en une image noir et blanc : chaque pixel peut valoir (0,0,0) ou (255,255,255) selon que la luminosité est plus petite ou plus grande que 127.

Cette conversion ne permet néanmoins pas de représenter les variations douces de luminosité. L'exercice difficile 6.2.5 présente une méthode plus avancée résolvant cette limitation.

6.2 Exercices de révisions et compléments

Exercice 6.2.1 On cherche à réaliser la fusion de deux images supposées de même taille, comme dans l'exemple suivant.

`img1`



`img2`

`img3`

Écrire une fonction `fusion(img1,img2,img3)` qui place dans `img3` l'image correspondant à la moyenne, pixel à pixel, des images `img1` et `img2` (supposées de même taille).

On pourra tester les deux images `ocean.png` et `road2.jpg` disponibles sur le site du cours.

Exercice 6.2.2 L'effet de postérisation est obtenu en diminuant le nombre de couleurs d'une image. Une manière simple de l'obtenir est d'arrondir les composantes r, g, b des pixels de l'image à des multiples d'un entier, par exemple des multiples de 64. Écrivez donc une fonction `posteriser(img,n)` qui arrondit les composantes des pixels de l'image à un multiple de n. Essayez sur une photo avec $n = 64, 128, 150, 200$.

Exercice 6.2.3 Une manière simple de rendre une image floue est d'utiliser l'opération moyenne. Écrire une fonction `flou(img)` qui pour chaque pixel n'étant pas sur les bords, le peint de la couleur moyenne des pixels voisins [1]. Comparer le résultat à l'original. Pourquoi faut-il plutôt passer une deuxième image en paramètre à la fonction, et ne pas toucher à la première image ?

Exercice 6.2.4 Écrire une fonction `emprisonnerJoliment(img:image, nbBarreaux:int, epaisseur:int)` qui améliore le rendu de l'exercice 5.3.6 en faisant apparaître un dégradé de gris sur chaque barreau. Pour calculer la couleur d'un pixel d'un barreau, on pourra utiliser une formule telle que

$$\text{couleur} = \frac{\text{distance au centre du barreau} * 255}{\text{épaisseur}} \ .$$

Exercice 6.2.5 Comme vous l'avez constaté dans l'exercice 6.1.5, la qualité des images produites par la fonction `noirEtBlanc` laisse à désirer. L'algorithme proposé par Floyd et Steinberg, permet de limiter la perte d'information due à la quantification des pixels en blanc ou noir. Écrire une fonction `floydSteinberg(img:image)` qui convertit une image monochrome en noir et blanc à l'aide de l'algorithme de Floyd et Steinberg (cf. page Wikipedia : http://fr.wikipedia.org/wiki/Algorithme_de_Floyd-Steinberg). Note : concernant la fonction `couleur_la_plus_proche()`, dans cet exercice on la fait retourner seulement soit du noir, soit du blanc.

Manipulations géométriques

Exercice 6.2.6 (extrait DS 2014-15) Un reporter a pris plusieurs photographies d'un paysage. Elles sont de même hauteur mais montrent des points de vue décalés. Il souhaite maintenant les mettre côte à côte pour obtenir une image panoramique de ce paysage (sans se soucier des problèmes de perspective).

1. Écrire une fonction `assembler(imgGauche, imgDroite, imgPano)` qui place dans `imgPano` l'image correspondant à l'assemblage horizontal des images `imgGauche` et `imgDroite`. Quelle doit être la largeur de l'image `imgPano` passée en paramètre ?

2. Écrire une fonction `assemblerTrois(imgGauche, imgCentre, imgDroite, imgPano)` qui assemble trois images horizontalement dans `imgPano`, en utilisant la fonction précédente.

Exercice 6.2.7 (extrait DST 2015-16) L'objectif de cet exercice est d'écrire le code de deux fonctions Python permettant d'effectuer la transformation miroir (symétrie axiale) d'une image. Voici un exemple :

1. Idéalement, pour un résultat réellement correct, il faudrait en fait pour chaque composante prendre la racine carrée de la moyenne des carrés des pixels voisins, voir https://www.youtube.com/watch?v=LKnqECcg6Gw, mais contentez-vous d'une simple moyenne, cela sera déjà bien.

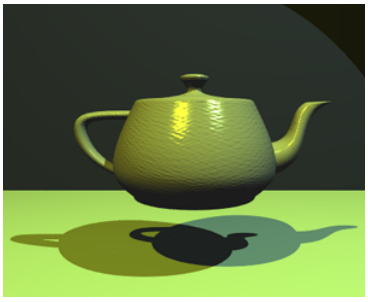
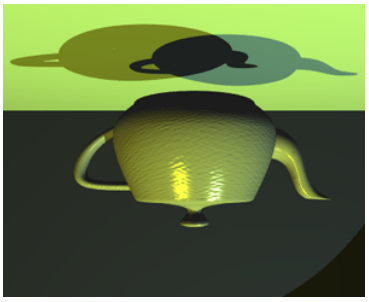
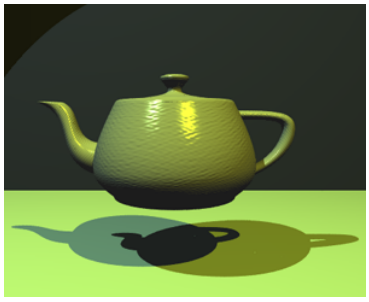


image initiale


miroir vertical


miroir horizontal

1. Écrire une fonction `miroirVertical(img1:image, img2:image)` qui place dans `img2` l'image correspondant au miroir vertical de `img1` (`img1` et `img2` sont supposées de même taille).

2. Écrire une fonction `miroirHorizontal(img1:image, img2:image)` qui place dans `img2` l'image correspondant au miroir horizontal de `img1`.

Exercice 6.2.8 Écrire une fonction `rotation90(img1:image,img2:image)` qui place dans `img2` l'image `img1` tournée de 90 degrés vers la droite, c'est-à-dire dans le sens horaire.

Pourquoi est-on obligé d'utiliser une deuxième image, plutôt que faire la rotation « en place », c'est-à-dire en n'utilisant qu'une image ?

Exercice 6.2.9

1. Écrire une fonction `rotationCoincoin(img1:image,img2:image,angle:float)` qui stocke dans `img2` l'image `img1` tournée de `angle` radians vers la droite par rapport au coin supérieur gauche de l'image. Quelques conseils :

   - Parcourir les pixels de l'image `img2` et calculer la position du pixel source correspondant dans l'image `img1` ; s'il est en-dehors de l'image, stocker un pixel noir.

   - Dans un premier temps, on pourra calculer la distance du pixel au coin supérieur gauche et l'angle par rapport à l'horizontale puis calculer les coordonnées du pixel ayant la même distance au coin supérieur gauche mais l'angle diminué.
   - Dans un deuxième temps, on pourra utiliser la transformation mathématique :

$$\begin{aligned} x' &= x \cdot \cos(\varphi) + y \cdot \sin(\varphi) \\ y' &= -x \cdot \sin(\varphi) + y \cdot \cos(\varphi) \end{aligned}$$

2. Écrire une fonction `rotationCentre(img,angle)` qui retourne une image correspondant à l'image `img1` tournée de `angle` radians par rapport au centre de l'image.

Complément : retourner une nouvelle image

Jusqu'ici les fonctions que vous avez écrites ont systématiquement modifié une image leur étant passée en paramètre, par exemple l'image `img` passée à la fonction `filtre(img)`. Vous avez également placé le résultat d'une fonction dans une seconde image également passée en paramètre, par exemple dans l'image `img2` passée à la fonction `filtre2(img1,img2)`.

Une alternative est de créer l'image de résultat directement à l'intérieur de la fonction (avec la fonction `nouvelleImage`) et de la retourner après y avoir placé le résultat. Par exemple :

```
def filtre3(img:image) -> image:
  imgRes = nouvelleImage(largeurImage(img),hauteurImage(img))
  ... # calcul des pixels de imgRes a partir de img
  return imgRes
```

L'avantage d'une telle approche est de pouvoir facilement appeler une fonction sur le résultat de la précédente, par exemple `filtre3(filtre3(img))`.

Image floue

Exercice 6.2.10 Modifiez la fonction `flou(img:image)->image` de l'exercice 6.2.3 pour qu'elle retourne une version floutée de l'image `img`. Créez des images de plus en plus floues en appelant de façon répétée cette fonction.

Ajoutez à la fonction des paramètres `x1,x2,y1,y2` qui désignent les coordonnées d'une portion rectangulaire de l'image qui doit être floutée plutôt que toute l'image. Floutez ainsi seulement un visage apparaissant sur une photo de votre choix (on pourra par exemple utiliser le logiciel *gimp* pour déterminer les coordonnées à utiliser).

Agrandissement d'une image

Exercice 6.2.11 Écrire une fonction `agrandirFacteur2(img:image)->image` qui retourne une image correspondant à l'agrandissement d'un facteur deux de l'image `img`. Pour réaliser cet agrandissement, chaque pixel de l'image source donnera un carré 2×2 pixels dans l'image destination.

Tester cette fonction sur l'image `teapot.png`. Appeler plusieurs fois la fonction pour produire un agrandissement d'un facteur 8 de cette même image. L'image apparait quelque peu *pixelisée*, comment atténuer facilement ce phénomène ?

Une image peut en cacher une autre – stéganographie

On peut utiliser le fait que l'œil ne distingue pas les faibles différences de couleur pour dissimuler une image dans une autre. Prenons deux images de mêmes dimensions. Pour chacune des coordonnées (x, y), on va mélanger une partie du pixel (x, y) de l'image A avec une partie du pixel (x, y) de l'image B en faisant en sorte que l'image obtenue A' apparaisse presque comme l'image A.

Voyons comment traiter une seule composante de couleur, disons le rouge. Pour simplifier plus encore supposons que l'intensité de la composante rouge soit codée par un entier compris entre 0 et 99 999. Imaginons que l'on veuille mélanger un pixel de l'image A dont la composante rouge vaut 17 234 avec un pixel de l'image B dont la composante rouge vaut 45 678. La technique à mettre en œuvre consiste à recombiner les trois premiers chiffres de la composante issue de A avec les deux premiers chiffres de celle issue de B en plaçant les chiffres issus de A en tête. Ce mélange des deux composantes produit le nombre 17 245. On remarque que la valeur de la composante du pixel de l'image A' est proche de celle de A car on a gardé ses trois chiffres les plus significatifs. Sur notre exemple, la différence est seulement 11 : la valeur obtenue est 17 245 contre 17 234 à l'origine. Au pire, cette technique de camouflage transforme les deux derniers chiffres de 00 à 99, ou vice-versa, ce qui fait une différence de 99, au plus.

Maintenant pour décoder une image cachée dans une autre, on peut retrouver une couleur proche de chaque pixel de la seconde image grâce aux 2 derniers chiffres de chaque composante. La composante de valeur 17 245 indique que la valeur de la composante cachée se situe entre 45 000 et 45 999 soit une erreur au maximum de 999. En ajoutant 500 à cette valeur, et donc en estimant la valeur de la composante cachée à 45 500, on réduit l'erreur maximale à 500.

Exercice 6.2.12 Écrire une fonction `decoder_composante(n:int)->int` qui, à partir d'une valeur `n` comprise entre 0 et 99 999 retourne la valeur de la composante cachée.

Appliquons maintenant cette technique à une composante d'un pixel dont la valeur est comprise en 0 et 255. Pour ce faire on exploite l'écriture en base 2 d'une composante : une telle valeur est codée sur un octet en binaire, soit 8 chiffres binaires. Pour camoufler une image on conserve les 5 premiers chiffres de la composante issue de A et les trois premiers de celle issue de B. Ainsi à partir de deux octets $a_7a_6a_5a_4a_3a_2a_1a_0$ et $b_7b_6b_5b_4b_3b_2b_1b_0$ on obtient le nombre binaire $a_7a_6a_5a_4a_3b_7b_6b_5$. Réciproquement pour découvrir la valeur cachée dans une composante, il s'agit d'isoler les 3 derniers chiffres binaires de celle-ci puis de les *décaler* de 5 chiffres. À partir du nombre binaire $a_7a_6a_5a_4a_3b_7b_6b_5$, on isole $b_7b_6b_5$ pour obtenir $b_7b_6b_500000$. Maintenant on sait que la composante originale doit avoir une valeur comprise entre $b_7b_6b_500000$ et $b_7b_6b_511111$: l'erreur maximale possible vaut donc 11111 (soit 31 en base 10). Aussi, en ajoutant 10000 (soit 16) à la composante décodée on réduit l'erreur maximale à 16.

Exercice 6.2.13 L'objectif est d'écrire une fonction permettant de découvrir une image cachée à partir d'une image source.

1. Écrire une fonction `decoder_composante(n:int)->int` qui, à partir d'une valeur `n` comprise entre 0 et 255 retourne la valeur de la composante cachée.
2. Écrire une fonction `devoiler_image(img:image)->image` qui retourne l'image cachée.
3. Utiliser cette fonction pour décoder l'image cachée dans `stegano.png` disponible sur le site du cours.

Exercice 6.2.14 Écrire une fonction `dissimuler_image(image1:image,image2:image)->image` qui retourne une nouvelle image où l'`image1` cache l'`image2`. On supposera que les images sont de même taille (on peut facilement utiliser un outil externe pour redimensionner une des deux images au besoin).

6.3 L'essentiel du chapitre

On peut récupérer la couleur d'un pixel ainsi :
`(r,g,b) = couleurPixel(img, 10, 10)`
On peut alors recalculer les valeurs de `r`, `g`, `b`, les coordonnées, etc. avant d'appeler `colorierPixel` de nouveau.

Ainsi une fonction typique qui retravaille une image est :

```
def f(img:image, n:int):
    l = largeurImage(img)
    h = hauterImage(img)
    for x in range(l):
        for y in range(h):
            (r,g,b) = couleurPixel(img, x, y)
            r = [...]
            x2 = [...]
            [...]
            colorierPixel(img, x2, y2, (r, g, b))
```