Université de Nice - Sophia Antipolis
Faculté des Sciences

DEUG MIAS MP1

Programmation 2001-02

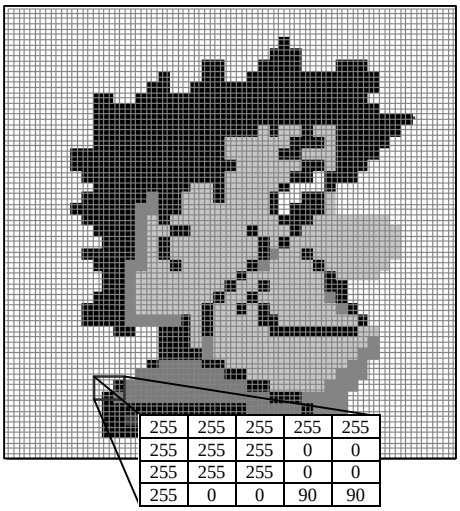
7. INTRODUCTION AUX IMAGES EN "NOIR ET BLANC" ET AU TRAITEMENT D'IMAGES

A. Création d'une classe `BWImage` (`BW` pour "***B***lack and ***W***hite").

Le premier objectif de ce TP est de créer une classe `BWImage` permettant de représenter une image en noir et blanc (ou plus exactement en niveaux de gris).

Mais tout d'abord qu'est ce qu'une image ? En informatique, une image est simplement un tableau dans lequel pour chaque case on associe une couleur.

Par exemple, l'image  est donnée par le tableau ci-contre. Une case du tableau s'appelle un *pixel*.



255	255	255	255	255
255	255	255	0	0
255	255	255	0	0
255	0	0	90	90

Par simplicité, dans ce TP nous allons seulement considérer des images en niveaux de gris. Ainsi, la couleur d'un pixel ne sera représentée que par une seule valeur: l'intensité. L'intensité varie de 0 (noir) à 255 (blanc) offrant ainsi 256 niveaux de gris.

Ces 256 niveaux de gris suffisent pour décrire les différentes intensités d'une image de bonne qualité. Ainsi, une image ne sera pour nous qu'un *tableau d'entiers compris entre 0 et 255*.

Exercice 7.1 **La classe** `BWImage`

a) Ecrivez la classe `BWImage` ayant les variables d'instance suivantes:

```
int width;  // largeur de l'image (nombre de colonnes du tableau)
int height;  // hauteur de l'image (nombre de lignes du tableau)
int data[][];  // tableau d'entiers compris entre 0 et 255.
```

L'entier `data[i][j]` représente l'intensité du pixel se trouvant à la ligne i et la colonne j.

b) Ecrivez un constructeur prenant en paramètre un tableau de type `int[][]` et initialisant correctement nos trois variables; ne copiez pas seulement la référence du tableau, copiez le contenu du tableau en entier.

Dans le constructeur, testez les valeurs du tableau et saturez les valeurs strictement supérieures à 255 et celles qui sont strictement négatives (*i.e.* si l'entier associé à un pixel est >255, vous prendrez 255 pour valeur et si l'entier associé à un pixel est <0, vous prendrez 0 pour valeur).

c) Ecrivez un deuxième constructeur qui crée une image blanche aux dimensions passées en paramètres.

d) Ecrivez aussi, en respectant les signatures, les accesseurs

```
public int getWidth()
public int getHeight()
public int[][] getData()
```

e) Pour avoir accès à la valeur d'un pixel, ou pour la modifier, écrivez les méthodes `getColor` et `setColor` de signature:

```
public int getColor(int i,int j)
public void setColor(int i, int j, int niv).
```

Attention, n'oubliez pas de vérifier si les couples (i,j) appartiennent bien au tableau. Que faire sinon ?

N'oubliez pas non plus de saturer le niveau de gris si celui que l'on vous donne n'est pas entre 0 et 255 (bornes incluses).

Pour ne pas être réduit à ne considérer que des images dessinées par nos soins, il est intéressant de pouvoir charger des images à partir de fichiers présents sur le disque dur. La classe `ImageLoader` qui vous est fournie, vous permet de charger des images au format "PPM". Après avoir ajouté et compilé cette classe dans votre répertoire, il suffit pour charger l'image "*eglise.ppm*" dans la variable `img` (de type BWImage), d'écrire à l'endroit désiré:

```
img = ImageLoader.loadImage("eglise.ppm");
```

Testez votre classe au Top-level. Essayez de charger une image; affichez ses dimensions...

B. CREATION DU VIEWER D'IMAGES.

Maintenant que nous disposons de la classe `BWImage` et que nous savons charger des images, nous avons envie de pouvoir les visualiser. Pour se faire, nous allons créer la classe `ImageViewer` qui sera une `Frame` (ou mieux... une `MP1Frame`) puis nous lui ajouterons un `Canvas` personnalisé dans lequel nous "dessinerons" notre image.

Exercice 7.2 Création de notre canvas:

a) Créez la classe `ImageCanvas` héritant de la classe `Canvas`.

b) Ajoutez la variable d'instance: `BWImage myImage;`
cette variable contiendra la référence vers l'image à dessiner.

c) Ecrivez un constructeur prenant en paramètre une image de type `BWImage`; ce constructeur doit bien évidemment initialiser la variable d'instance.

d) Surchargez la méthode `paint` de signature public `void paint(Graphics gr)`.
Cette méthode devra dessiner l'image. Pour cela vous programmerez et utiliserez la méthode `drawPixel`:

```
public void drawPixel(int nL, int nC, int color, Graphics gr){
       Color c = new Color(color,color,color);
       gr.setColor(c);
       gr.drawLine(nC,nL,nC,nL);
};
```

e) Puis, comme dans le TP précédent, afin de bien dimensionner votre fenêtre, surchargez aussi la méthode `getPreferredSize()`

Exercice 7.3 Et enfin, le viewer ...

a) Ecrivez la classe `ImageViewer` étendant la classe MP1Frame.

b) Ecrivez un constructeur prenant en paramètre une image de type `BWImage`. A notre fenêtre portant le nom de "Viewer d'image", ajoutez un `canvas ImageCanvas`, puis redimensionnez la en utilisant la méthode `pack()`.
Demandez lui enfin d'être visible...

c) Dans la méthode main de `ImageViewer`, vous pouvez désormais charger une image, puis l'afficher à l'aide d'un `ImageViewer`. Affichez dans deux fenêtres distinctes les images "*Pau.ppm*" et "*lena.ppm*".

C. INTRODUCTION AU TRAITEMENT D'IMAGES.

Dans ce qui suit, j'espère pouvoir vous convaincre de l'utilité du traitement d'images...
Qu'est ce que le traitement d'images ?

Le traitement d'images consiste à faire subir des transformations à nos images, afin d'en tirer diverses informations ou d'obtenir d'autres images.

A quoi sert le traitement d'images ?

1. Nous pouvons faire de la "*segmentation*": Par exemple, à partir de photos satellites, on voudrait que l'ordinateur arrive à calculer la superficie de blé ou de colza cultivée dans telle région.
2. Nous pouvons faire de la "*vision par ordinateur*": Par exemple, pour donner la vue à un robot, il faut qu'il distingue les objets, puis qu'il arrive à estimer leur distance...
3. Nous pouvons faire de la "*restauration*" d'images: Nous pouvons diminuer les flous sur une photo, ou diminuer les effets de grains sur une photo mal scannée, etc.
4. Nous pouvons faire de la "*compression*" d'images: Vous connaissez certainement tous, le format "DivX" qui vous permet de compresser un DVD sur un seul CDR ! Compresser les images permet aussi par exemple, de diminuer les durées de téléchargement trop longues sur internet...
5. et bien d'autres applications encore...

Le traitement d'image est donc un domaine scientifique très varié et très utile. En photographie, un filtre est un objet placé sur l'objectif pour produire des effets spéciaux : il modifie l'image à l'entrée de l'objectif. Pour le traitement d'image en informatique, nous allons donc procéder de la même façon en créant un objet *filtre* qui prend une image et la transforme en une autre. Les filtres locaux sont une notion de base que nous tentons de vous présenter dans cette partie.

Exercice 7.4 Tests de filtres:

a) Ecrivez la classe `Filtre` qui contient la méthode suivante:

```
public BWImage apply(BWImage img){.....}
```

cette méthode applique le filtre; c'est à dire que pour chaque point (i,j) de l'image `img`, elle applique l'opérateur local

```
public int oper(BWImage img, int i, int j){
       return img.getColor(i,j);
}
```

Que fait cet opérateur `oper` ?

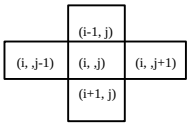
b) Dans la méthode `main` de la classe `ImageViewer`, nous allons créer différents filtres. Comment allons nous procéder ? Réfléchissez ... Vous le savez ...

c) Et pour commencer, testez le filtre ***négatif*** dont l'opérateur retourne :

```
return 255-img.getColor(i,j);
```

Testez ce filtre.

d) Ecrivez le filtre de ***lissage*** dont la méthode `oper` retourne la moyenne des intensités des cinq points représentés ci contre:



Testez ce filtre. Qu'en pensez vous? Comment faire pour augmenter sensiblement son effet... et sans se fatiguer?

Réponse: Rajouter dans la classe `Filtre` une méthode `apply` à deux paramètres appliquant *n* fois le filtre initial.

e) *Filtre de dérivation*: Peut-on dériver une image ? Cela a -t - il un sens ?

Ecrivez un filtre de dérivation par rapport à `x`, puis un filtre de dérivation par rapport à `y`.

Tester ces filtres sur les images "*eglise.ppm*" et "*lena.ppm*".

Comment faire pour avoir la dérivée seconde ? Testez; qu'en pensez vous ?

f) Pour améliorer la lisibilité des images retournées, vous pouvez créer, dans la classe `BWImage` une méthode qui normalise l'image. Plus précisément, cette méthode multipliera les intensités des points de l'image par 255/Max, où Max est l'intensité maximale de l'image. Testez.

N'est-ce pas joli tout ça ? Et nous pouvons faire mieux... nous allons essayer de détecter le contour des objets dans l'image. Ceci est une activité à la base du travail de vision robotique.

Exercice 7.5 Détecteur de contours:

a) Toujours dans la méthode `main` de la classe `ImageViewer`, écrivez le nouveau filtre `gradNorme` retournant la norme du gradient de l'image. Un gradient est une variation, orientée dans l'espace ; c'est un vecteur qui pointe dans la direction d'augmentation maximale. Le gradient a donc pour coordonnées les dérivées partielles de la fonction:

$$\nabla I(i,j) = \begin{pmatrix} \frac{\partial I}{\partial i}(i,j) \\ \frac{\partial I}{\partial j}(i,j) \end{pmatrix}$$

b) Testez ce filtre sur nos images. Normalisez les résultats.

c) Dans la classe `BWImage`, rajoutez maintenant une méthode qui seuille les valeurs de l'image : cette méthode annule l'intensité des points dont l'intensité est inférieure à un seuil `s` (donné en paramètre), et donne l'intensité maximale (255) aux points où l'intensité initiale est supérieure à ce seuil.

d) Toujours dans la classe `BWImage`, rajoutez une méthode `detectContour` effectuant les points **a)**, **b)** et **c)**. Réfléchissez...

e) Testez `detectContour` sur les images "*eglise.ppm*" et "*lena.ppm*". Trouvez le paramètre de seuillage idéal pour chaque image et admirez le résultat.

D. AMELIORATION DE L'AFFICHAGE !

En testant nos méthodes, vous avez certainement trouvé l'affichage peu efficace, voire même un peu lourd.

L'API java est très complète; elle fournit bien entendu une classe `Image`. Cette classe abstraite permet de décrire n'importe quelle sorte d'images; cependant elle nous paraît trop complexe pour être abordée sereinement dans ce TP. Nous allons quand même l'utiliser afin d'accélérer l'affichage.

Exercice 7.6 **"Amélioration" de la classe `ImageCanvas`.**

a) Nous allons créer une classe `ImageCanvasR` (R comme *R*apide) qui ne sera qu'un `Canvas` auquel on ajoutera une variable `myAWTImage` de type `Image` (classe du `package java.awt`). Dans le constructeur, vous initialiserez la variable `myAWTImage` en introduisant les instructions suivantes:

```
int[] pixData= new int[img.getWidth()*img.getHeight()];
int niv=0;
for (int i=0; i<img.getHeight () ; i++)
      for (int j=0; j<img.getWidth () ; j++){
            niv=img.getColor(i,j);
            pixData[j+img.getWidth()*i]=(new Color(niv,niv,niv)).getRGB();
      }
myAWTImage = createImage(new MemoryImageSource(img.getWidth(),
                                img.getHeight(), pixData,0,img.getWidth()));
```

b) Maintenant il ne nous reste plus qu'à faire afficher cette `image`; Comment allons nous faire cela?
En surchargeant la méthode `paint` bien sûr...

```
public void paint(Graphics gr) {
      gr.drawImage(image, 0, 0,this);
}
```

c) Que faut-il changer maintenant dans la classe `ImageViewer` pour utiliser ce qui précède? Testez...