# TP C#9 : MyTinyToshop

## 1 Consignes de rendu

À la fin de ce TP, vous devrez rendre une archive respectant l'architecture suivante :

```
– login_x.zip
    |– login_x/
        |– AUTHORS
        |– README
        |– MyTinyToshop/
            |– MyTinyToshop.sln
            |– MyTinyToshop/
                |– Tout sauf bin/ et obj/
```

Bien entendu, vous devez remplacer *login_x* par votre propre login. N'oubliez pas de vérifier les points suivants avant de rendre :

- Le fichier AUTHORS doit être au format habituel : ** login_x$* où le caractère '$' représente un retour à la ligne.
- Le fichier README doit contenir les difficultés que vous avez rencontrées sur ce TP, et indiquera les bonus que vous avez réalisés.
- Pas de dossiers bin ou obj dans le projet.
- **Le code doit compiler !**

## 2 Introduction

Au cours de ce TP, nous allons faire des mathématiques appliquées ! Plus précisément, nous allons aborder des notions de traitements d'images en recréant un petit logiciel constitué de fonctionnalités relativement simples.

# 3 Cours

## 3.1 Traitement d'images

Le traitement d'images est une discipline informatique étudiant les images numériques, afin de les améliorer, les modifier, ou d'en extraire des informations.

Une image est composée de pixels (contraction de *picture element*). Un pixel est représenté par 4 composantes : son intensité rouge, verte et bleue, ainsi que sa transparence (composante $\alpha$).

On peut, grâce à certains traitements sur ces pixels, rendre une image plus floue ou plus nette, la distordre, la retourner, ou encore détecter les visages ou les portions de texte qu'elle contient. Nous allons rapidement étudier quelques uns de ces traitements.

## 3.2 Opération point à point

Une opération point à point est une opération qui calcule la nouvelle valeur d'un pixel en fonction de sa valeur originelle ou de sa position. Les opérations arithmétiques et logiques sont des opérations point à point. Par exemple, rajouter une valeur positive à chacune des composantes des pixels d'une image nous donnera une image plus claire.

Le passage en niveau de gris, la binarisation et l'inversion d'une image sont les opérations point à point que vous allez devoir implémenter.

## 3.3 Opération géométrique

Une opération géométrique change tout simplement l'arrangement ou la position des pixels. Dans cette catégorie, nous retrouvons, entre autres, les opérations de rotation ou de symétrie.

## 3.4 Opération locale - Convolution

Une opération locale calcule la valeur d'un pixel en fonction de ce dernier et de ses voisins. L'opération qui nous intéresse est la convolution.

La convolution est un procédé par lequel la nouvelle valeur d'un pixel est calculée en faisant la somme pondérée de sa valeur et de celle de ses voisins. Les voisins pris en compte sont définis par une "fenêtre" ($3\times3$, $5\times5$, ...) centrée sur le pixel à calculer. Les coefficients associés à chacun de ces pixels se trouvent dans la matrice de convolution, qu'on appelle aussi masque ou noyau. On peut écrire ce calcul sous la forme suivante, $N$ étant la taille du masque, $P$ un pixel et $M$ un masque :

$$\sum_{i=-N/2}^{N/2} \sum_{j=-N/2}^{N/2} P_{x+i,y+j} \times M_{i+N/2,j+N/2}$$

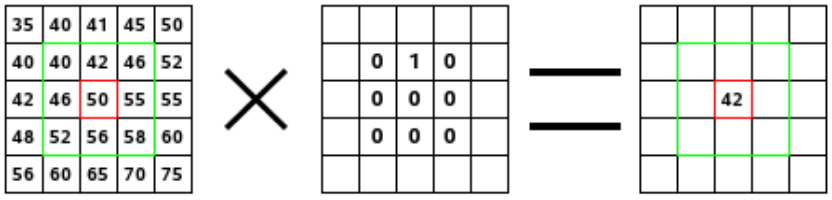



FIGURE 1 – Convolution

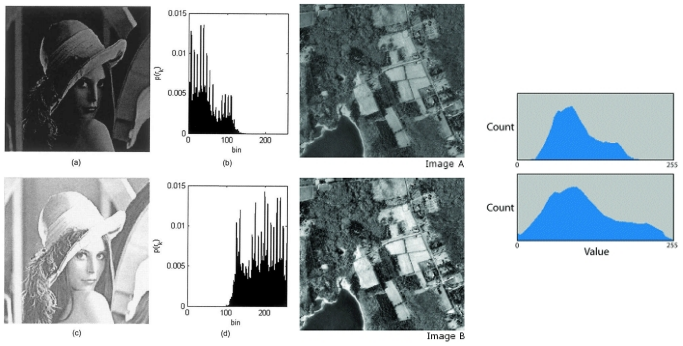
Ces calculs seront appliqués sur chaque composante du pixel et seront ramenés à 0 ou 255 lorsque les résultats dépasseront ces valeurs limites. On appelle cette dernière opération le *clamping*. Pour notre implémentation de la convolution, nous considérerons que les voisins hors-image d'un pixel sur un bord valent 0 (pixel noir).

La somme des poids du masque influe sur l'intensité de l'image résultante. Lorsque celle-ci vaut 1, l'intensité moyenne est conservée. La majorité des masques vous sont fournis dans les sources.

## 3.5 Histogramme et table de correspondance

L'histogramme d'une composante d'une image représente la distribution de l'intensité de celle-ci, c'est-à-dire le nombre de pixels présents pour chaque intensité (de 0 à 255). L'histogramme d'une image nous informe sur son contraste. En effet, une image en niveaux de gris sombre aura un plus grand nombre de de pixels proche de 0, l'histogramme sera décalé vers la gauche.

Un histogramme dont les valeurs sont très rapprochées est synonyme de mauvais contraste. Afin de corriger cela, il faut étendre l'histogramme. Nous allons appliquer le même traitement à chacune des composantes d'une image en couleur, comme si nous traitions plusieurs niveaux de gris, pour simplifier l'exercice.




(a) Contraste (b) Extension

FIGURE 2 – Histogrammes

Pour ce faire, il faut trouver, pour chaque composante du pixel, les intensités minimales et maximales de son histogramme, c'est-à-dire la plus petite et la plus grande intensité existante dans l'histogramme (en abscisse), et non pas le plus petit ou plus grand nombre de pixels présents trouvable dans l'histogramme (en ordonnée). Ensuite, il faut appliquer la formule suivante à chaque pixel, $x$ étant une intensité entre 0 et 255, $c$ étant la composante sur laquelle nous travaillons :

$$output_c(x_c) = \begin{cases} 0 & x_c < low_c \\ 255 \times (x_c - low_c)/(high_c - low_c) & low_c \leq x_c \leq high_c \\ 255 & high_c < x_c \end{cases}$$

Au lieu de calculer à chaque fois la valeur correspondante pour chacune de ses composantes, il est possible de pré-calculer les résultats de la fonction. Ces résultats seront

stockés dans des tables de correspondances (*Look-up table*, LUT, en anglais). Pour connaître les nouvelles valeurs, il suffira d'accéder à la table correspondante en utilisant la valeur originelle comme indice :

$$output_c(x_c) \iff LUT_c[x_c]$$

# 4 Exercices

Vous devez commencer par récupérer les sources disponibles sur l'intranet ACDC. [1]. Les images sont représentées par des instances de la classe *Bitmap* [2] et les pixels par des instances de la classe *Color* [3]. L'utilisation de la méthode *RotateFlip* de la classe *Bitmap* est interdite.

Pour chacune des fonctions de ce TP, vous trouverez en annexe un aperçu du résultat attendu.

## 4.1 Filtres

### MapPixels

Vous devez tout d'abord compléter la fonction *MapPixels* qui sera en charge d'appliquer une fonction spécifique (passée en argument, représentant les filtres que vous implémenterez par la suite) aux pixels de votre image [4].

```
1 public static Bitmap MapPixels(Bitmap img, Func<Color, Color> filter);
2 public static Bitmap MapPixels(Bitmap img, Func<Color, Color> filter,
3                                 int xB, int yB, int xE, int yE);
```

Vous devez écrire les deux surcharges de cette fonction :

— Les quatre derniers paramètres représentent respectivement l'abscisse et l'ordonnée du point où commence l'application de la fonction fournie, ainsi que l'abscisse et l'ordonnée du point où le traitement prend fin (non inclus, $x < xE \ldots$).

— La première surcharge, ne possédant pas ces quatre paramètres, effectuera le traitement sur l'intégralité de l'image.

1. http://acdc.epita.it
2. http://msdn.microsoft.com/fr-fr/library/system.drawing.bitmap(v=vs.110).aspx
3. http://msdn.microsoft.com/fr-fr/library/system.drawing.color(v=vs.110).aspx
4. Rappelez-vous l'ordre supérieur en Caml

### GreyScale

Cette fonction transforme en niveau de gris un pixel donné.

### Negative

Cette fonction transforme un pixel en son inverse (le blanc devient noir, et vice-versa).

### Binarize

Cette fonction renvoie la couleur noire si le pixel donné en niveau de gris est inférieur à un certain seuil (128 par exemple), et la couleur blanche sinon.

### Tinter

Cette fonction applique une teinte à un pixel donné. La teinte et le coefficient sont désignés par les propriétés *Tint* et *Coef* de *Filters.cs*. Vous pouvez modifier le coefficient et la couleur via le *WindowsForm*.

Notez ci-dessous la formule pour appliquer une teinte à un pixel :

$$newComponent = (tintComponent - oldComponent) \times (coef/100) + oldComponent$$

```
1 public static Color Greyscale(Color color);
2 public static Color Negative(Color color);
3 public static Color Binarize(Color color);
4 public static Color Tinter(Color color);
```

### Stripes

Complétez cette fonction qui applique à une image plusieurs teintes à la suite, alignées verticalement. Contrairement aux fonctions précédentes, où l'on ne travaillait que sur un pixel, *Stripes* prend l'intégralité de l'image, lui applique les teintes données par le tableau *colors*, et renvoie l'image ainsi modifiée. Vous devrez donc utiliser les fonctions *MapPixels* et *Tinter*. N'oubliez pas de mettre à jour la propriété *Tint* au fur et à mesure et conservez le coefficient initial. Pour tester votre fonction, le filtre *French Flag* est à votre disposition.

```
1 public static Image Stripes(Bitmap img, Color[] colors);
```




## 4.2 Transformations géométriques

### Miroir

Écrivez les fonctions *HorizontalMirror* et *VerticalMirror* ci-dessous qui renvoient les symétriques de l'image, respectivement selon un plan horizontal et vertical.

```
1 public static Image HorizontalMirror(Bitmap img);
2 public static Image VerticalMirror(Bitmap img);
```

### Rotation

Écrivez les fonctions *LeftRotation* et *RightRotation* ci-dessous, qui effectuent une rotation à 90 degrés sur l'image, respectivement vers la gauche et vers la droite.

```
1 public static Image LeftRotation(Bitmap img);
2 public static Image RightRotation(Bitmap img);
```

## 4.3 Convolution

Écrivez la fonction *Convolute* qui applique une matrice de convolution carrée à une image, et renvoie l'image résultante.

```
1 public static Image Convolute(Bitmap img, float[,] mask);
```

Veillez bien à ce que la nouvelle valeur des composantes de chaque pixel soit comprise entre 0 et 255. Dans le cas contraire, la valeur 0 ou 255 leur sera attribuée selon le cas de dépassement. Enfin, assurez-vous de bien calculer vos valeurs à partir d'une copie de l'image originale, au risque de voir vos résultats faussés par votre traitement en cours.

Les matrices de convolution vous sont fournies, seule la matrice **AverageMask** est manquante, à vous de la trouver et de la remplir.

## 4.4 ContrastStretch

Afin de corriger le contraste de nos images, il va falloir calculer des histogrammes pour chacune des composantes, ainsi que leurs intensités minimales et maximales. Pour faciliter le stockage de ces informations, nous allons utiliser des dictionnaires [5].

5. https://msdn.microsoft.com/fr-fr/library/xfhwa508(v=vs.110).aspx




Il s'agit tout simplement d'un ensemble de couple clé-valeur. Dans notre cas, la clé sera un caractère représentant la composante (*'R'*, *'G'*, *'B'*). La valeur sera soit un tableau d'entiers pour représenter un histogramme, soit un simple entier pour les intensités minimales ou maximales. Voici comment les manipuler :

```
// Initialize histogram dictionary
Dictionary<char, int[]> hist = new Dictionary<char, int[]>
                                            { { 'R', new int[256] },
                                              { 'G', new int[256] },
                                              { 'B', new int[256] } };

// Initialize lowest intensity dictionary
Dictionary<char, int> low = new Dictionary<char, int>
                                            { { 'R', 0 },
                                              { 'G', 0 },
                                              { 'B', 0 } };

/* Access to the histogram at 0 for red component
   (number of pixels with red component equal to 0) */
hist['R'][0];
// Access to lowest intensity for green component
low['G'];
// Get collection of keys for hist, usable in foreach
hist.Keys;
```

### GetHistogram

Complétez cette fonction qui renvoie un dictionnaire contenant les différents histogrammes des composantes rouges, vertes et bleues d'une image.

```
public static Dictionary<char, int[]> GetHistogram(Bitmap img);
```

### FindLow, FindHigh

Complétez ces fonctions qui renvoient respectivement les intensités minimales et maximales d'un histogramme donné. Par défaut, on considérera qu'elles valent 0 et 255.




```
1 private static int FindLow(int[] hist);
2 private static int FindHigh(int[] hist);
```

### FindBound

Complétez cette fonction qui renvoie un dictionnaire contenant les valeurs de chaque composante associée au résultat de la fonction $f$ appliquée à son histogramme. On pourra l'utiliser plus tard avec les fonctions précédentes pour récupérer les dictionnaires des intensités minimales et maximales.

```
1 public static Dictionary<char, int>
2 FindBound(Dictionary<char, int[]> hist, Func<int[], int> f);
```

### ComputeLUT

Complétez cette fonction qui renvoie la table de correspondance de la fonction d'ajustement du contraste, décrite dans la partie cours [6].

```
1 public static int[] ComputeLUT(int low, int high);
```

### GetLUT

Cette fonction renvoie le dictionnaire associant chaque composante à sa table de correspondance. Il faudra réutiliser les fonctions précédentes.

```
1 public static Dictionary<char, int[]> GetLUT(Bitmap img);
```

### ContrastStretch

Cette fonction renvoie l'image résultante de l'extension de l'histogramme. Pour cela, il faudra récupérer les tables de correspondance. Pour chaque pixel, vous pourrez alors récupérer les nouvelles composantes via la relation exprimée à la fin de la partie cours [7].

```
1 public static Image ConstrastStretch(Bitmap img);
```

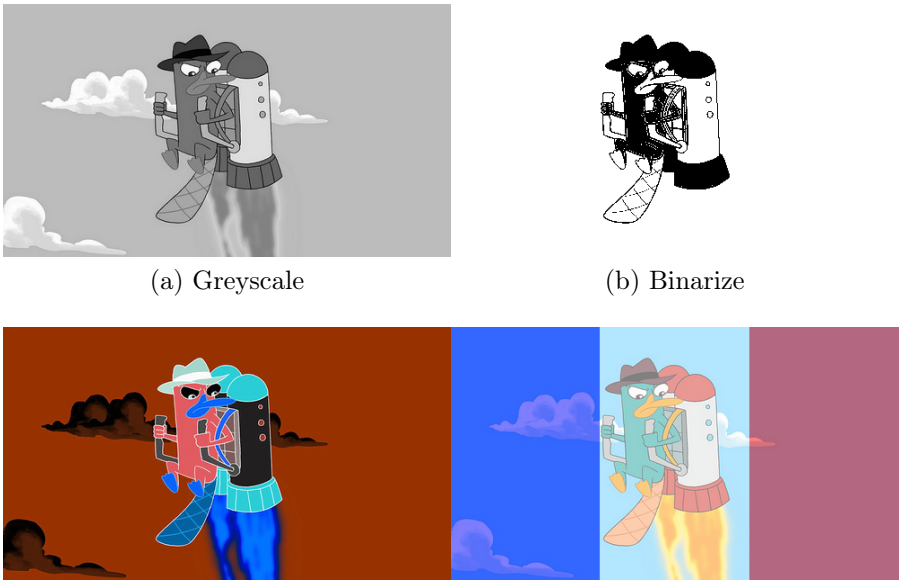
6. Cf. Histogramme et table de correspondance, p.4
7. Cf. Histogramme et table de correspondance, p.5

## 4.5 Bonus

— Plus de filtres (éclaircissement, assombrissement, ...)
— Filtre Arc-En-Ciel (référez-vous au filtre "French Flag")
— Plus de masques de convolution
— Rotation d'un angle de n'importe quelle valeur (renseignez-vous sur l'interpolation bilinéaire)
— Agrandissement/réduction d'une image par l'interpolation au plus proche voisin
— Réduction d'une image par moyenne

***Brace yourselves, Deadline is coming.***

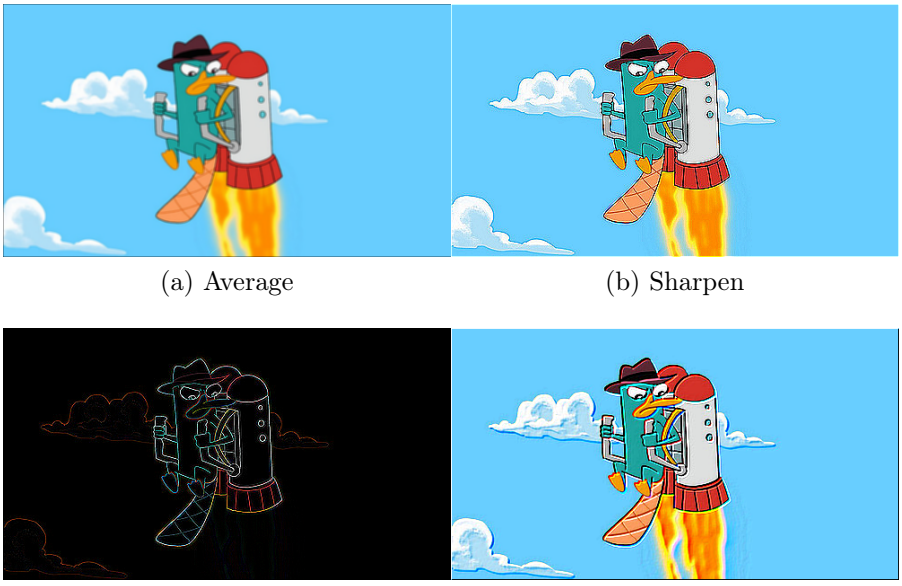
# 5 Annexes



(a) Greyscale (b) Binarize

(c) Negative (d) French Flag

FIGURE 3 – Filtres

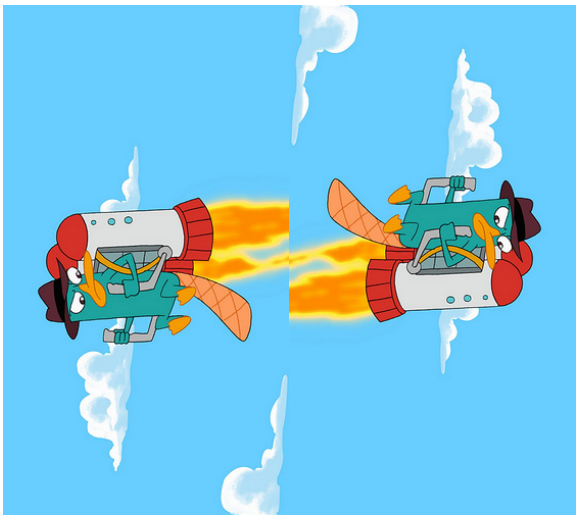


(a) Average (b) Sharpen

(c) Edge Detect (d) Emboss

FIGURE 4 – Convolutions

(a) Horizontal Mirror (b) Vertical Mirror



(c) Left Rotation (d) Right Rotation

FIGURE 5 – Transformations géométriques

(a) Original (b) Résultat

FIGURE 6 – Contraste automatique