

TP C#7 : TinyPhotoshop

Consignes de rendu

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

```
rendu-tp-prenom.nom.zip
|-- prenom.nom/
    |-- AUTHORS
    |-- README
    |-- TinyPhotoshop/
        |-- TinyPhotoshop.sln
        |-- TinyPhotoshop/
            |-- Everything except bin/ and obj/
            |-- Auto/
                |-- Auto.cs
            |-- Basics/
                |-- Basics.cs
                |-- Geometry.cs
            |-- Convolution/
                |-- Convolution.cs
            |-- Instagram/
                |-- Filter.cs
                |-- InstaFilter.cs
                |-- MyImage.cs
                |-- Nashville.cs
                |-- Toaster.cs
            |-- Steganography/
                |-- Steganography.cs
        |-- Form/ (Do not touch files inside)
        |-- Program.cs
```

N'oubliez pas de vérifier les points suivants avant de rendre :

- Remplacez `prenom.nom` par votre propre login et n'oubliez pas le fichier `AUTHORS`.
- Les fichiers `AUTHORS` et `README` sont obligatoires.
- Pas de dossiers `bin` ou `obj` dans le projet.
- Respectez scrupuleusement les prototypes demandés.
- Retirez tous les tests de votre code.
- **Le code doit compiler !**

AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (*), un espace, votre login et un retour à la ligne. Voici un exemple (où \$ est un retour à la ligne et ↳ un espace) :

```
* ↳prenom.nom$
```

Notez que le nom du fichier est `AUTHORS` sans extension. Pour créer simplement un fichier `AUTHORS` valide, vous pouvez taper la commande suivante dans un terminal :

```
echo "* prenom.nom" > AUTHORS
```

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un **README** vide sera considéré comme une archive invalide (malus).

1 Introduction

Vous avez vu récemment le fonctionnement des fichiers et la POO. Aujourd’hui vous allez découvrir la manipulation d’images en C#.

1.1 Objectifs

Au cours de ce TP, nous allons faire des mathématiques appliqués mais, pas seulement. Le but de ce TP est de manipuler les notions que vous avez apprises jusqu’à aujourd’hui afin de créer un programme avec des fonctionnalités qui sont relativement simples mais, qui peuvent demander un peu de réflexion.

2 Cours

2.1 L’ordre supérieur

Dans ce sujet, nous allons traiter d’image. Comme vous allez le voir plus loin, vous allez devoir appliquer des filtres au image. Pour simplifier la chose, nous allons utiliser un peu d’ordre supérieur. Notre utilisation est simple et va se contenter de l’utiliser en paramètre de fonction.

Dans cet exemple la fonction **Apply** va retourner le résultat de la fonction **function** à l’entier **x**.

```
1 int Apply(int x, Func<int, int> function)
2 {
3     return function(x);
4 }
5 // Le mot Func explicite le fait que ça soit une fonction
6
7 // Le type d'entrée est le type des paramètres que prend la
8 // fonction et est le premier int
9
10 // Le type de retour est le type que renvoie la fonction et est
11 // le second int
12
13 // La fonction function est une fonction et s'utilise comme telle
```

Pour appeler notre fonction, il nous suffit ensuite de l’appeler avec comme paramètre une fonction qui correspond aux types d’entrées et de sorties.

```
1 int x = -50;
2 x = Apply(x, Math.Abs);
3 // x == 50
```

2.2 Les images

Les images sont stocké dans différents formats (.jpg, .png, .gif, ...). La bibliothèque de C# nous permet de facilement lire un fichier image et directement récupérer les données utiles dans un objet.

Les images sont divisées en pixels (De la même manière qu’une matrice). Un pixel contient quatre données, le rouge, vert et le bleu pour indiquer la couleur du pixel mais, aussi une dernière valeur qui est le champ Alpha qui permet de connaître le niveau de transparence de ce pixel.

2.3 Manipulation d'images

En C# pour manipuler des images la plus part des outils se trouvent dans :

```
1 System.Drawing // Bibliothèque générale
2 System.Drawing.Image // Objet image général
3 System.Drawing.Bitmap // La classe que nous utilisons qui hérite de image
```

Je vous invite à lire la page de documentation de l'objet Bitmap si vous avez besoin de détail spécifique à propos de cette classe (comment ouvrir/sauvegarder/modifier une image etc.).

Il est possible dans un objet Bitmap de directement accéder aux pixels de coordonnées (x,y). Ainsi vous pouvez utiliser ces méthodes de Bitmap pour récupérer ou définir un pixel :

```
1 GetPixel(Int32, Int32);
2 SetPixel(Int32, Int32, Color);
```

2.4 Filtres

Les filtres permettent de facilement modifier une image pour lui ajouter des effet, améliorer son rendu... Il existe différents types de filtres.

2.4.1 Point à point

Les premiers, les plus simples sont les filtres que l'on considère point à point. C'est à dire que l'application du filtre ne dépend pas du reste de l'image et peut donc être fait directement pixel par pixel. Un autre type, les filtres géométriques sont des filtres correspondant aux déplacements (par copie) des pixels sur l'image. Comme par exemple les rotations, les translations etc.

2.4.2 Convolution

Enfin, le dernier type de filtre est celui utilisant des convolutions.

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" (3×3 , 5×5 , . . .) 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 :

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

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.

Tous les masques que vous allez utiliser sont donné vide dans les sources. C'est à vous de faire des recherches sur internet pour trouvé quelles sont les poids à donner à chaque case.

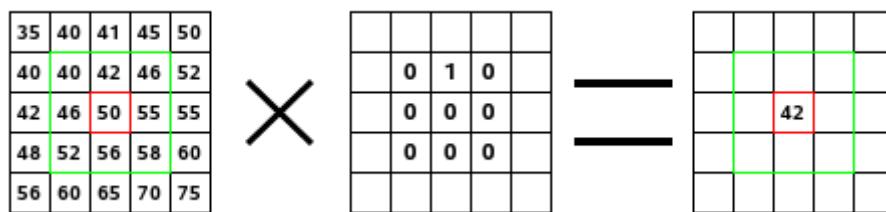


FIGURE 1 – Convolution

2.5 Stéganographie

La stéganographie est l'art de la dissimulation : son objectif est de faire passer inaperçu un message dans un autre message. Dans ce TP nous allons faire de la stéganographie de texte dans une image et d'une image dans une image.

Commençons d'abord par comprendre le principe de coder du texte dans une image. Avant de commencer il vous faut savoir qu'une composante est codée sur un byte, tout comme les caractères qui servent à constituer du texte. C'est-à-dire que un pixel est codé sur trois bytes, on a donc un byte pour le rouge, un pour le vert et un pour le bleu. On peut logiquement se dire que l'on peut remplacer un pixel par trois caractères, mais ce n'est pas le cas car on perdrait totalement la valeur d'origine du pixel. Si on transpose ça sur des grands textes, au final on va détruire une grande partie des pixels et il n'y aura plus d'image.

Une question se pose : Comment pouvons-nous faire pour perdre le moins d'information possible au niveau de l'image ? Au lieu de remplacer tous les bits de l'image on va en remplacer qu'une partie. Les bits de poids forts sont ceux qui modifient le plus les couleurs, on logiquement toucher aux bits de poids faibles. Par convention pour ce TP on choisit de ne modifier que les quatre bits de poids faibles des composantes. Il va donc falloir découper les caractères en deux pour pouvoir les insérer dans l'image.

Prenons un exemple, on souhaite coder le texte "abc" dans deux pixels gris. On va tout d'abord couper chaque caractère en deux puis on va les placer dans les composantes des deux pixels.

Lettre	a	b	c
Représentation ASCII	97	98	99
Représentation binaire	0110 0001	0110 0010	0110 0011

Pixels de référence	R: 1000 0000 G: 1000 0000 B: 1000 0000	R: 0010 1010 G: 0010 1010 B: 0010 1010
Caractères à coder	a : 0110 0001 b : 0110 0010 c : 0110 0011	
Pixels obtenus	R: 1000 0110 G: 1000 0001 B: 1000 0110	R: 0010 0010 G: 0010 0110 B: 0010 0011

Passons maintenant à la stéganographie d'image. Celle-ci est plus simple à comprendre et à mettre en place. Une composante est toujours codée sur un byte, donc un pixel toujours sur trois bytes. Mais qu'est-ce qui nous empêche de coder un pixel d'une image dans un pixel de l'autre ?

Absolument rien et c'est ce que vous devrez faire. Malgré la simplicité de ce procédé nous allons devoir, comme pour le texte, dégrader les deux images. Comme précédemment, les poids forts sont ceux qui modifient le plus la valeur, les poids forts des deux images sont donc importants. Malheureusement on ne peut pas superposer les deux poids forts alors nous allons ruser. Le principe est de basculer les poids forts de l'image à encoder dans les poids faibles de l'autre. Pour l'exemple nous allons reprendre les deux mêmes pixels que le précédent, avec eux un pixel bleu et un pixel vert. Dans ce cas là il suffit juste de prendre les poids forts des pixels à coder et les mettre dans les poids faibles des autres.

Pixels de référence	R: 1000 0000	R: 0010 1010
Pixels à coder	G: 1000 0000	G: 0010 1010
Pixels obtenus	B: 1000 0000	B: 0010 1010
Pixels de référence	R: 1000 0000	R: 0010 1010
Pixels à coder	G: 0001 0100	R: 0001 0100
Pixels obtenus	B: 0110 0001	G: 0110 0011
Pixels de référence	R: 1000 0001	R: 0010 0001
Pixels à coder	G: 1000 0001	G: 0010 0110
Pixels obtenus	B: 1000 0110	B: 0010 0001

3 Exercices

3.1 Exercice 1 : La Manipulation d'image

À présent vous savez comment encrypter des images pour pouvoir y glisser des secrets. Malgré cela votre apprentissage n'est pas terminé, vous allez à présent apprendre à manipuler des images.

3.1.1 Palier 0 : La Manipulation de pixels

Dans ce palier vous allez vous occuper des images en les modifiant pixels par pixel. Les fonctions suivantes se trouvent dans **Basic.cs**

Grey

La fonction **Grey** prend en entrée une couleur et renvoie la couleur correspondant en niveau de gris.

Voir [Figure 3](#)

```
1 public static Color Grey(Color c)}
```

Binarize

La fonction **Binarize** prend en entrée une couleur et renvoie soit du blanc soit du noir selon que la couleur est plus proche de l'un ou de l'autre.

Voir [Figure 4](#)

```
1 public static Color Binarize(Color c)
```

BinarizeColor

La fonction **BinarizeColor** fait la même chose que **Binarize** mais en traitant composante par composante c'est-à-dire que chacune des trois composantes (rouge, vert, bleu) doit être soit au maximum soit au minimum.

Voir [Figure 5](#)

```
1 public static Color BinarizeColor(Color c)
```

Negative

La fonction **Negative** doit inverser chaque composante de la couleur **c** (le blanc devient noir et vice-versa) et la retourner.

Voir [Figure 6](#)

```
1 public static Color Negative(Color c)
```

Tinter

La fonction **Tinter** doit appliquer la couleur **tint** à **c** a **factor** % et doit la retourner, c'est-à-dire que la couleur résultante est un mélange des deux couleurs.

Voir [Figure 7](#)

```
1 public static Color Tinter(Color c, Color tint, int factor)
```

Apply

Maintenant que nous avons fait tous les filtres, on va devoir les appliquer à notre image. C'est là qu'intervient l'ordre supérieur. Voir [subsection 2.1](#).

La fonction **Apply** doit appliquer la fonction **func** à chaque pixel de l'image **img** et doit la retourner.

```
1 public static Image Apply(Bitmap img, Func<Color, Color> func)
```

3.1.2 Palier 1 : La Manipulation d'Image

Dans ce palier vous allez vous occuper de modifier les formes des images. Les fonctions suivantes se trouvent dans **Geometry.cs**.

Shift

La fonction **Shift** permet de décaler l'image **img** relativement à sa position de **x**, **y** et retourne l'image résultante. Si la partie de celle-ci est coupé par les bords, elle doit être collée à l'opposé de celle-ci.

Voir [Figure 8](#)

```
1 public static Image Shift(Bitmap img, int x, int y)
```

SymmetryHorizontal

La fonction **SymmetryHorizontal** fait une symétrie horizontale de l'image **img** en son centre et retourne l'image résultante.

Voir [Figure 9](#)

```
1 public static Image SymmetryHorizontal(Bitmap img)
```

SymmetryVertical

La fonction **SymmetryVertical** fait une symétrie verticale de l'image **img** en son centre et retourne l'image résultante.

Voir [Figure 10](#)

```
1 public static Image SymmetryVertical(Bitmap img)
```

SymmetryPoint

La fonction **SymmetryPoint** fait une symétrie par point, aux coordonnées **x**, **y**, de l'image **img** et retourne l'image résultante.

Voir [Figure 11](#)

```
1 public static Image SymmetryPoint(Bitmap img, int x, int y)
```

RotationLeft

La fonction **RotationLeft** fait une simple rotation de +90° sur l'image **img** et retourne l'image résultante.

Voir [Figure 12](#)

```
1 public static Image RotationLeft(Bitmap img)
```

RotationRight

La fonction **RotationRight** fait une simple rotation de -90° sur l'image **img** et retourne l'image résultante.

Voir [Figure 13](#)

```
1 public static Image RotationRight(Bitmap img)
```

Resize

La fonction **Resize** utilise l'interpolation linéaire pour redimensionner l'image **img** en **x**, **y**. Il faut déjà créer une image résultante de taille **x**, **y**.

Ensuite, il faut parcourir tous les pixels de l'image résultante. Sur chaque pixel il va falloir aller chercher le pixel correspondant sur l'image de départ (en utilisant le ratio d'agrandissement sur les dimensions des images, l'image résultante divisé par l'image originale). Si cela tombe sur un pixel on applique directement sa valeur. Sinon on fait la moyenne pondérée de la valeur des pixels voisins et on applique le résultat à notre pixel.

Voir https://fr.wikipedia.org/wiki/Interpolation_numerique

Voir [Figure 14](#)

```
1 public static Image Resize(Bitmap img, int x, int y)
```

3.1.3 Palier 2 : La Convolution

Dans ce palier vous allez coder la convolution sur des images.

Les fonctions suivantes se trouvent dans **Convolution.cs**. Vous pouvez voir plusieurs exemples avec différents masques ([Figure 15](#) - [Figure 20](#)). Seul un masque est fourni, pour les autres vous devez trouver les implémenter et trouver les bons réglages pour correspondre à la référence.

Clamp

La fonction **Clamp** prend en entrée un **float** quelconque et doit retourner en sortie l'entier le plus proche compris entre 0 et 255.

```
1 public static int Clamp(float c)
```

```
1 Clamp(-4563232.1254) // 0
2 Clamp(25.1249) // 25
3 Clamp(20201997.4269) // 255
```

IsValid

La fonction **IsValid** vérifie si **x** et **y** sont dans l'intervalle défini par **Size**. La fonction doit retourner **true** si c'est le cas, **false** sinon.

```
1 public static bool IsValid(int x, int y, Size size)
```

Convolute

La fonction **Convolute** calcul le filtre **mask** sur l'image **img** et retourne le résultat. Pour cela la fonction va parcourir chaque pixel de l'image et pour chaque pixel va faire un produit de celui-ci avec le masque. Pour bien réussir cette fonction vous avez à disposition les deux fonctions que vous venez d'écrire, il faut les utiliser.

Attention, pensez à ne pas faire les calculs et lire les pixels sur la même image !

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

3.2 Exercice 2 : La Stéganographie

Le but de cet exercice est de faire de la stéganographie sur des images. Pour cela vous allez implémenter deux types différents de stéganographie : sur des textes et sur les images.

3.2.1 Section image : That's a beautiful picture

Passons aux images ! Toutes les explications se situent dans la partie cours mais vous allez devoir respecter les règles suivantes :

- Vous devez commencer par écrire l'image dans le pixel ($y = 0, x = 0$), puis vous continuez sur le pixel en $x + 1$ jusqu'à arriver au bord d'une des deux images et recommencer sur la ligne suivante.
- Les bits forts de la composante rouge remplacent les bits poids faibles du rouge.
- Les bits forts de la composante verte remplacent les bits poids faibles du vert.
- Les bits faibles de la composante bleu caractère remplacent les bits poids faibles du bleu.

EncryptImage

La fonction EncryptImage doit suivre ce protocole pour encrypter le bitmap **enc** dans le bitmap **img** et la retourner. Vous devez superposer chaque composante de chaque pixel de l'image à encoder sur les composantes de chaque pixel de l'image résultante. Si l'image à encoder est plus grande que l'image où on l'encode, la fonction ne doit rien faire.

Voir [Figure 21](#)

```
1 public static Image EncryptImage(Bitmap img, Bitmap enc)
```

DecryptImage

La fonction DecryptImage doit suivre le protocole inverse pour décrypter l'image **img** et retourner l'image correspondante.

```
1 public static Image DecryptImage(Bitmap img)
```

3.2.2 Section texte : Are you talking to me ?

L'algorithme peut être divisé en deux parties. Pour la première partie, le but est de réaliser un tableau qui va nous servir de buffer pour contenir tous les caractères que l'on souhaite encoder. Vous allez devoir diviser chaque caractère de la chaîne de caractère en deux : les quatre bits de poids forts et les quatre bits de poids faibles et les mettre dans le tableau en commençant par les bits forts. La fin d'une chaîne de caractère est codé par le byte nul. Pour nous il nous suffit de rajouter dans notre buffer deux 0 à la suite. Pour le texte **abc** qui est codé respectivement en ASCII par **97**, **98** et **99**, on s'attend à avoir dans notre buffer **6, 1, 6, 2, 6, 3, 0, 0**.

Pour la seconde partie, il faut mettre chaque élément de notre buffer dans une composante de pixels. faut néanmoins respecter la règle suivante :

- Vous devez commencer par écrire l'image dans le pixel ($y = 0, x = 0$), puis vous continuez sur le pixel en $x + 1$ jusqu'à arriver au bord et recommencer sur la ligne suivante.

EncryptText

La fonction **EncryptText** doit suivre ce protocole pour encrypter la string **text** dans le bitmap **img** et doit retourner l'image. La fonction ne doit rien faire si l'image est trop petite pour accueillir le texte.

Voir [Figure 22](#)

```
1 public static Image EncryptText(Bitmap img, string text)
```

DecryptText

La fonction **DecryptText** doit suivre le protocole inverse pour décrypter le texte contenu dans l'image **img** et doit retourner la chaîne de caractères correspondant.

```
1 public static string DecryptText(Bitmap img)
```

3.3 Exercice 3 : ContrastStretch (Bonus)

Dans ce palier vous allez faire une autocorrection d'image. Les fonctions se trouvent dans **Auto.cs**.

Préambule

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 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.

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 :

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

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 :

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

Implémentation

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.

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 :

```
1 // Initialize histogram dictionary
2 Dictionary<char, int[]> hist = new Dictionary<char, int[]>
3 { { 'R', new int[256] },
4 { 'G', new int[256] },
5 { 'B', new int[256] } };
6
7 // Initialize lowest intensity dictionary
8 Dictionary<char, int> low = new Dictionary<char, int>
9 { { 'R', 0 },
10 { 'G', 0 },
11 { 'B', 0 } };
12
13 /* Access to the histogram at 0 for red component
14 (number of pixels with red component equal to 0) */
15 hist['R'][0];
16 // Access to lowest intensity for green component
17 low['G'];
18 // Get collection of keys for hist, usable in foreach
19 hist.Keys;
```

BuildHistogram

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

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

Find Low/High

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 public static int FindLow(int[] hist)
```

La fonction **FindHigh** doit trouver la valeur maximale du tableau **hist** de la composante ligne **rgb** et doit la retourner.

```
1 public 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.

```
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)
```

ContrastStretching

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. Voir [Figure 23](#)

```
1 public static Image ContrastStretching(Bitmap img)
```

3.4 Exercice 4 : Instagram (Bonus)

Dans cette dernière partie vous allez essayer de refaire des Filtres instagram.

Nashville

Pour réaliser ce filtre vous aurez d'abord besoins de créer cette fonction :

Dans le fichier **InstaFilter.cs** :

```
1 public Bitmap ColorTone(Bitmap image, Color color)
```

Cette fonction consiste à changer la balance des couleurs dans l'image.

Pour réussir cette fonction voici quelques indices :

- https://en.wikipedia.org/wiki/Color_balance
- **ColorMatrix**
- **ImageAttributes**
- **Graphics**

Ces outils sont très pratiques pour modifier une image et vous pouvez retrouver la documentation sur MSDN.

En utilisant cette fonction remplissez le fichier **InstaNashville.cs**

Le filtre consiste en deux appels de **ColorTone** avec les bonnes valeurs à vous de chercher !

Voir [Figure 24](#)

Toaster

Enfin **Toaster** est un des filtres les plus compliqués d'instagram. Pour réaliser celui-ci vous devrez implémenter cette fonction :

```
1 public Bitmap Vignette(Bitmap b, Color color1, Color color2, float crop = 0.5f)
```

Cette fonction permet de faire un dégradé de la **color1** (couleur du centre) vers la **color2** (sur les bord) en forme de cercle.

Pour vous aider :

- <https://en.wikipedia.org/wiki/Vignetting>
- **GraphicsPath**
- **PathGradientBrush**

Ce filtre utilise plusieurs balances de couleurs différentes ainsi que deux différentes vignettes. Encore une fois, c'est à vous de trouver les bonnes couleurs !

Voir [Figure 25](#)

MyCorrection

Vous vous souvenez du fichier **Auto.cs**? Oui celle du bonus de l'exercice **ContrastStretching**. Maintenant c'est à vous de rajouter de nouvelles corrections d'image. Dans **Auto.cs** rajoutez autant de fonctions que vous voulez, le but est qu'elles embellissent les images. Une fois cela fait utilisez-les dans la fonction **MyCorrection** avec l'image d'entrée **img**.

```
1 public static Image MyCorrection(Bitmap img)
```

4 Exemples



FIGURE 2 – Originale

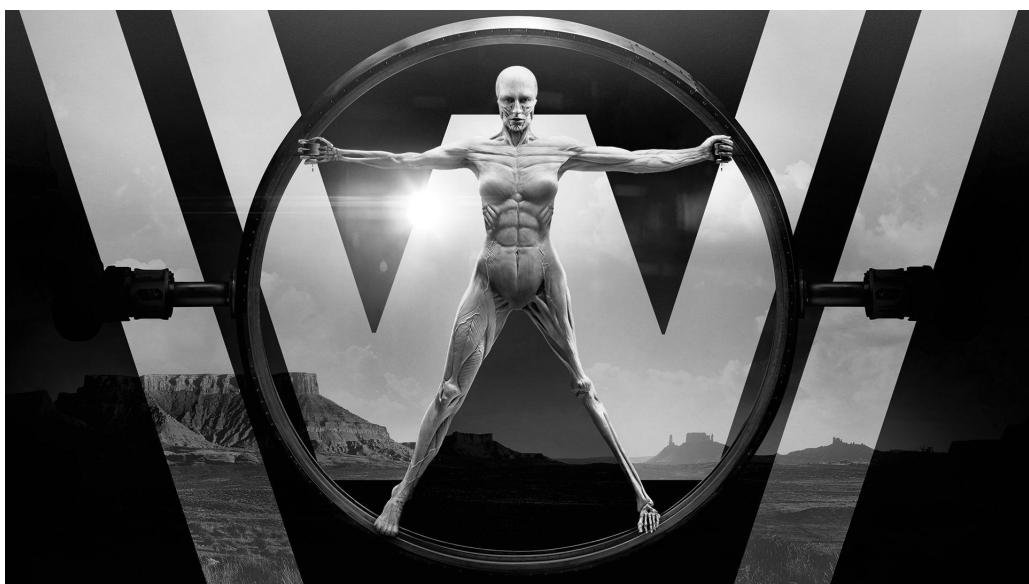


FIGURE 3 – Grey

These violent deadlines have violent ends.



FIGURE 4 – Binarize



FIGURE 5 – BinarizeColor

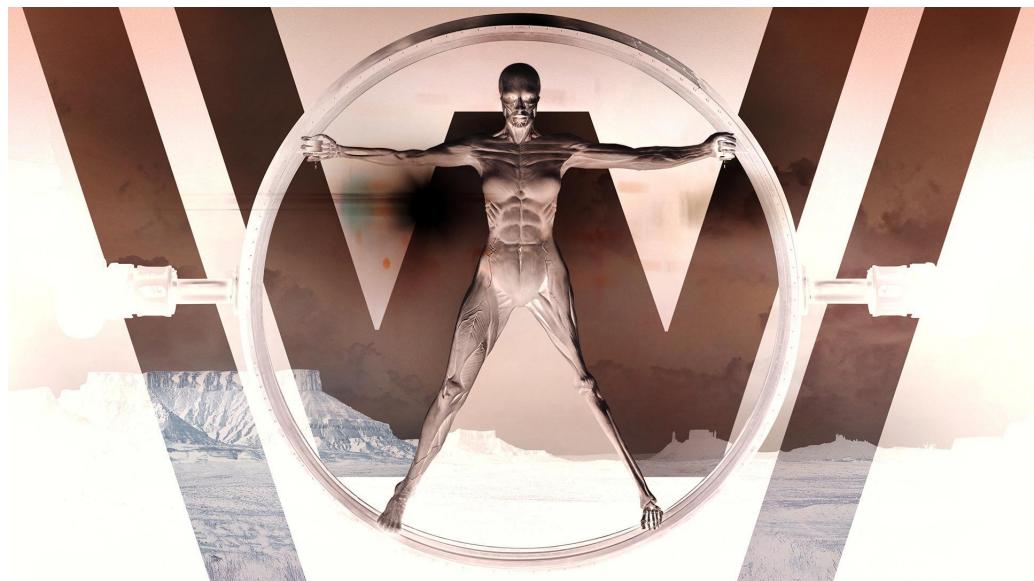


FIGURE 6 – Negative

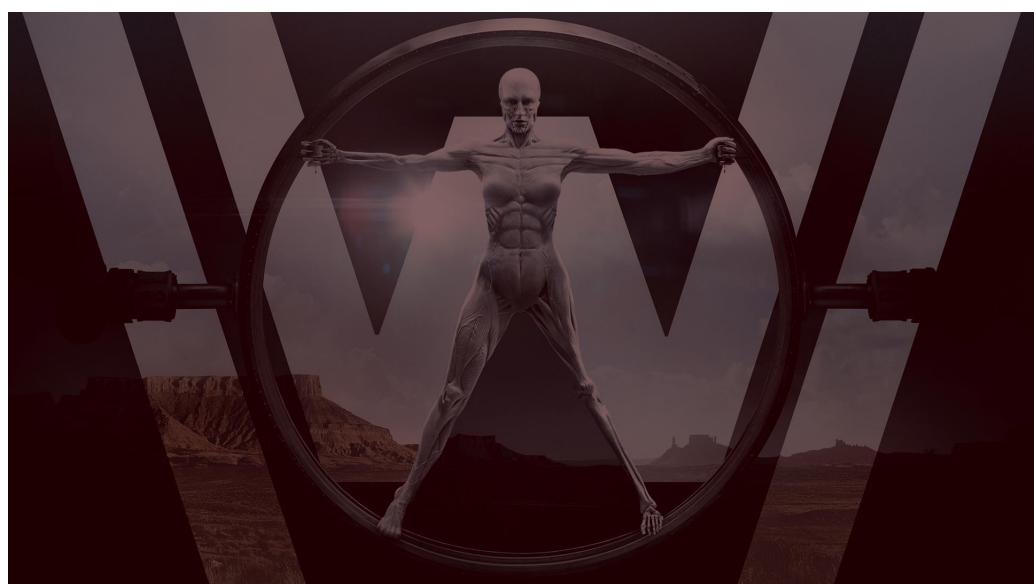


FIGURE 7 – Tinter



FIGURE 8 – Shift 960x540



FIGURE 9 – Horizontal



FIGURE 10 – Vertical



FIGURE 11 – Point 250x250

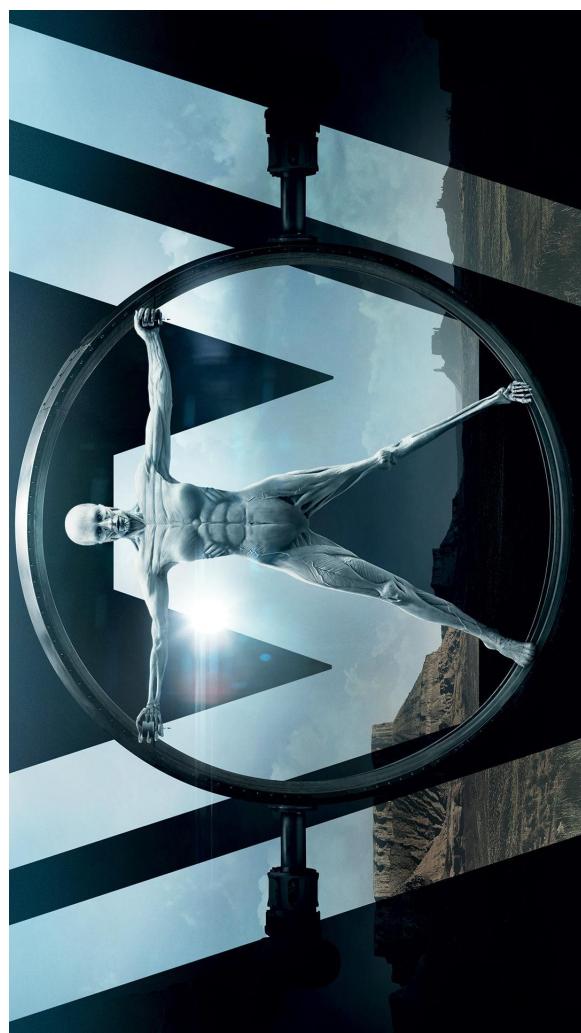


FIGURE 12 – Left



FIGURE 13 – Right



FIGURE 14 – Resize 3840x2160



FIGURE 15 – Gauss



FIGURE 16 – Sharpen



FIGURE 17 – Blur

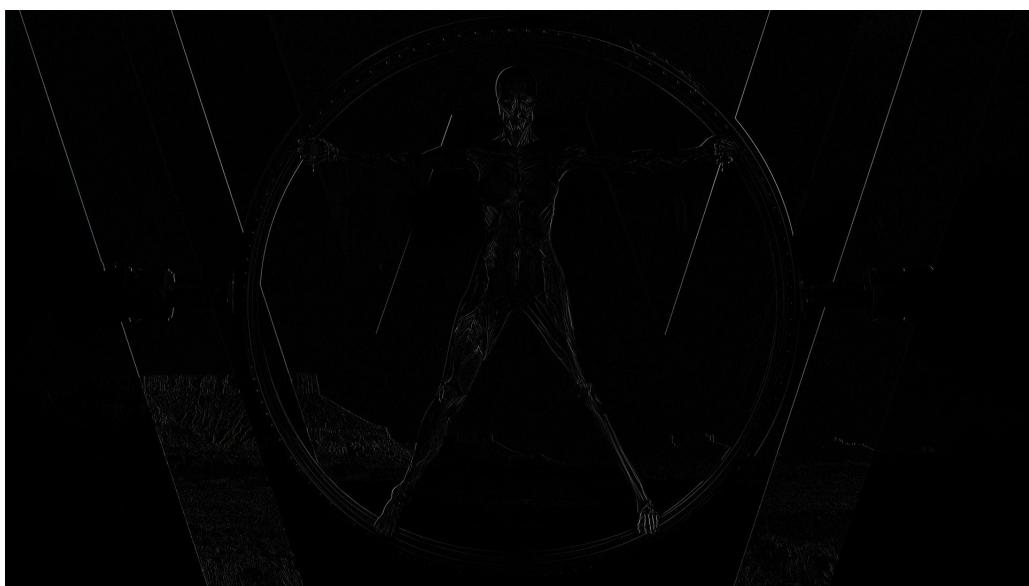


FIGURE 18 – Enhance

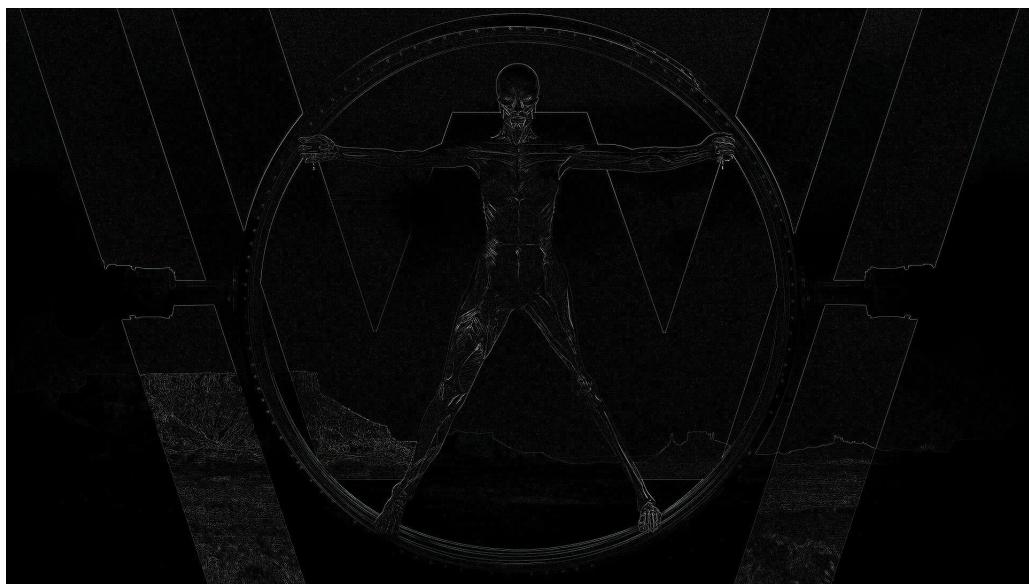


FIGURE 19 – Detect

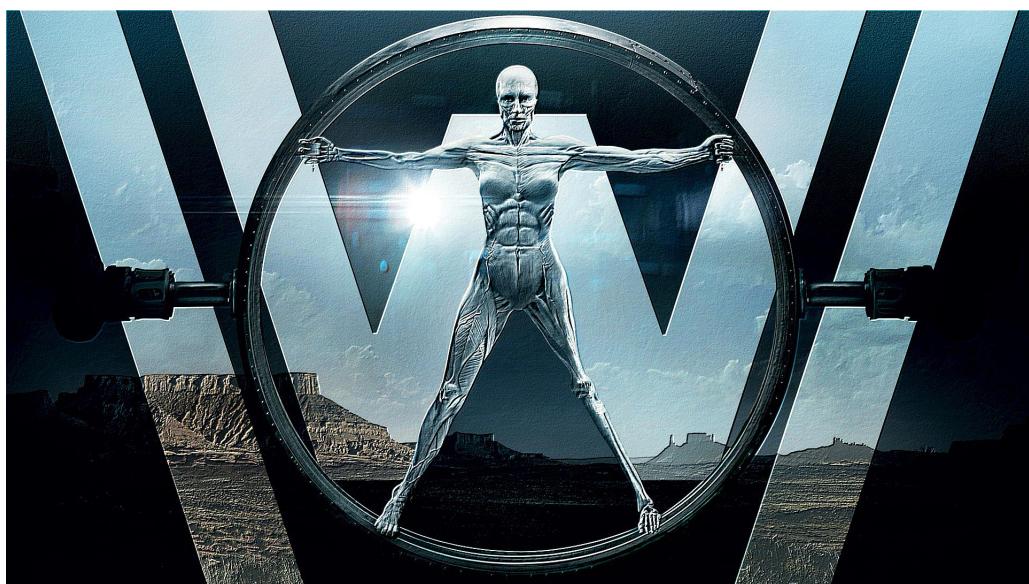


FIGURE 20 – Emboss



FIGURE 21 – Encrypt Image



FIGURE 22 – Encrypt Quote



FIGURE 23 – Auto



FIGURE 24 – Nashville

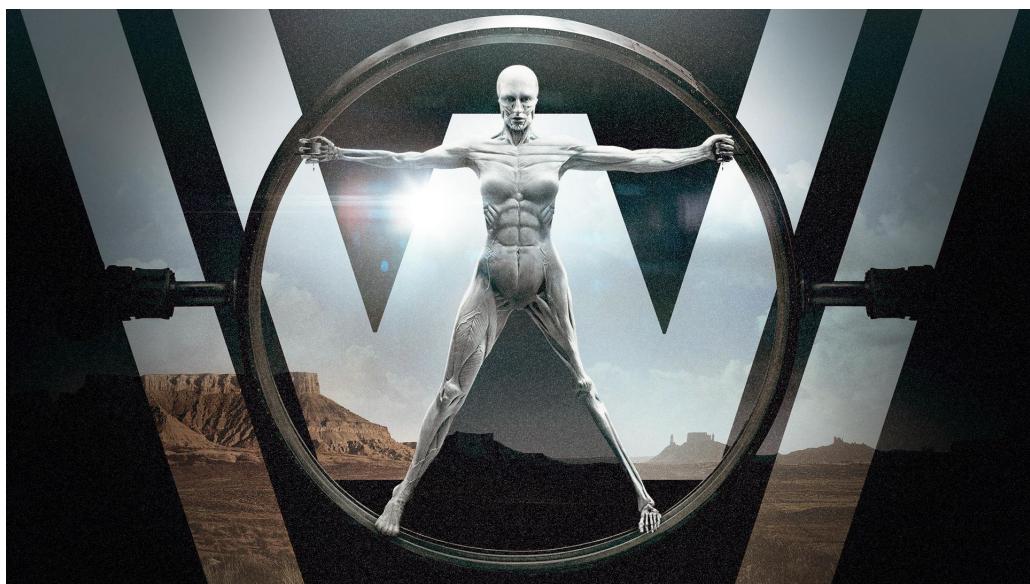


FIGURE 25 – Toaster