TP C#2 : TinyPhotoshop

Consignes de rendu

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

```
|-- csharp-tp2-prenom.nom
    |-- README
    |-- TinyPhotoshop/
        |-- TinyPhotoshop.sln
        |-- TinyPhotoshop/
            |-- Everything except bin/ and obj/
            |-- Auto.cs
            |-- Basics.cs
            |-- Convolution.cs
            |-- Geometry.cs
            |-- Steganography.cs
            |-- Insta/
                |-- Nashville.cs
                |-- Toaster.cs
            |-- reference/
                |-- All provided reference to compare your work
            |-- tests/
                |-- All your test image should be stored here
            |-- Program.cs
```

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

— Remplacez `prenom.nom` par votre propre login.
— Le fichier `README` est obligatoire.
— 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#.

Au cours de ce TP, nous allons faire des mathématiques appliquées 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.

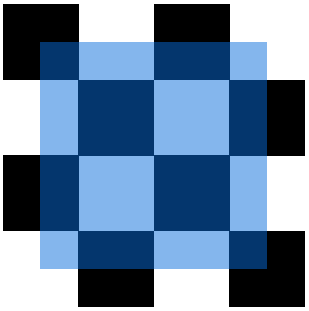
2 Cours

2.1 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.

Une image est une matrice de pixels.
Un pixel peut-être représenté par un ensemble de quatre données, sa quantité de rouge, de vert et de bleu, ainsi que son champ alpha.
Ce dernier indique le niveau de transparence de l'image : à son maximum, le pixel est totalement opaque ; à son minimum (0), le pixel est totalement transparent.

(a=**255**, r=**10**, g=**110**, b=**220**)



(a=**127**, r=**10**, g=**110**, b=**220**)

2.1.1 Manipulation d'images

En C# pour manipuler des images la plupart 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
```

Nous vous invitons à lire la page de documentation de la classe Bitmap pour savoir comment manipuler des images.
Il est notamment possible dans un objet Bitmap d'accéder à n'importe quel de ses pixels en utilisant ses 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);
```

2.2 Filtres

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

2.2.1 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 peut être faite pixel par pixel, indépendamment du reste de l'image.
Un autre type, les filtres géométriques, correspondent aux déplacement (par copie) des pixels sur l'image. Comme par exemple les rotations, les translations, ...

2.2.2 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.

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

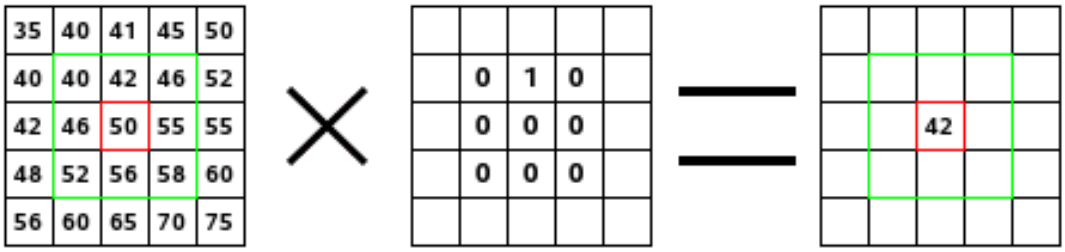


FIGURE 1 – Convolution

Pour aller plus loin

Vous trouverez ici de plus amples détails sur les matrices de convolution
`https://en.wikipedia.org/wiki/Kernel_(image_processing)`

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 donnés vides dans les sources. C'est à vous de faire des recherches sur internet pour trouver quels sont les poids à donner à chaque case.

2.3 Stéganographie

La stéganographie est l'art de la dissimulation : son objet 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 autre image.

Abordons d'abord la dissimulation de texte dans une image. Avant de commencer il vous faut savoir que chaque composante d'un pixel d'un object Bitmap est codée sur un byte (8 bits). Il se trouve qu'un caractère ASCII est également codé sur un byte. On peut alors logiquement se dire que l'on peut remplacer un pixel de l'image originale par un pixel formé par la représentation de trois caractères consécutifs du texte (sans toucher à la composante alpha de l'image originale). Cependant, cette approche s'avère être problèmatique car elle endomage très lourdement l'image originale, en écrasant totalement ses pixels originaux.

Une question se pose : Comment pouvons-nous faire pour faire cohabiter au mieux les informations du texte et de l'image ?
Au lieu de remplacer tous les bits d'un pixel dans l'image, nous pouvons n'en remplacer qu'une partie.
Les bits de poids fort d'une composante sont ceux qui modifient le plus sa valeur, nous privilégierons donc la modification de ses bits de poids faible. Par convention pour ce TP, nous choisissons de ne modifier que les quatre bits de poids faible des composantes. Il va donc falloir découper les caractères en deux pour pouvoir les insérer dans l'image.

Prenons un exemple, nous souhaitons coder le texte *"abc"* dans deux pixels gris. Nous allons donc d'abord couper chaque caractère en deux puis les intégrer dans deux pixels de l'image.

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

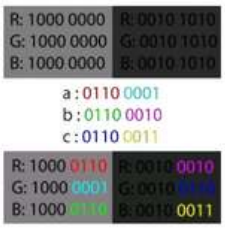


FIGURE 2 – Stéganographie de texte

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.
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 devrons comme pour le texte, dégrader les deux images.
Les poids forts d'une composante sont ceux qui modifient le plus la valeur de celle-ci, les bits de poids forts de chaque composante des deux images sont donc les plus importants. Puisque nous ne pouvons pas supperposer les bits de poids fort des deux images, il va falloir ruser. Le principe est de basculer les bits de poids fort (les plus importants) de l'image à dissimuler dans les bits de poids faible (les moins importants) de l'autre.
Pour l'exemple nous allons reprendre les deux mêmes pixels de l'exemple précédent, dans lequels nous dissimulerons un pixel bleu et un pixel vert.

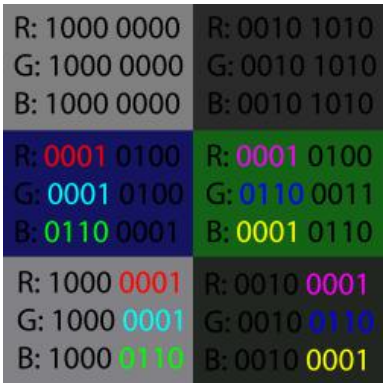


FIGURE 3 – Stéganographie d'image

3 Exercices

3.1 Exercice 1 : 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 les textes et sur les images.

3.1.1 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 continuer sur le pixel en $x + 1$, jusqu'à arriver au bord d'une des deux images et recommencer sur la ligne suivante.
- Les bits de poids fort de la composante rouge de l'image à encoder remplacent les bits de poids faible la composante rouge de l'image originale.
- Les bits de poids fort de la composante verte de l'image à encoder remplacent les bits poids faible de la composante verte de l'image originale.
- Les bits de poids fort de la composante bleu de l'image à encoder remplacent les bits de poids faible de la composante bleu de l'image originale.

EncryptImage

La fonction *EncryptImage* doit suivre ce protocole pour encrypter le bitmap *enc* dans le bitmap *img* et la retourner.
Vous devez répartir 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.

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

DecryptImage

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

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

3.1.2 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. Il 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.

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

DecryptText

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

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

3.2 Exercice 2 : 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 appliquer des filtres sur des images.

3.2.1 Palier 0 : La Manipulation de pixels

Dans ce palier vous allez vous occuper des images en les modifiants pixel 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 5

```
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 le pixel soit plus proche de l'un ou de l'autre. Voir Figure 6

```
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 (*255*), soit au minimum (*0*). Voir Figure 7

```
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 8

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

Tinter

La fonction *Tinter* doit appliquer la couleur *c1* à *c* à 50% et la retourner, c'est-à-dire que la couleur résultante est un mélange des deux couleurs.

```
1  public static Color Tinter(Color c, Color c1)
```

3.2.2 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 9

```
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 10

```
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 11

```
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 12

```
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 13

```
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 14

```
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 divisée 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`

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

3.2.3 Palier 2 : La Convolution

Dans ce palier vous allez mettre en pratique la convolution sur des images. Les fonctions suivantes se trouvent dans *Convolution.cs*

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

Convolute

La fonction *Convolute* calcule 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)
```

3.2.4 Palier 3 : L'autocorrection (Bonus)

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

BuildHistogram

La fonction *BuildHistogram* créée un histogramme à partir de l'image *img* et la retourne.

```
1  private static Color[] BuildHistogram(Bitmap img)
```

Find Low/High

Dans cette partie, *rgb* représente soit la composante rouge ($rgb = 1$), soit la composante verte ($rgb = 2$), soit la composante bleue ($rgb = 3$).
La fonction *FindLow* doit trouver la valeur minimum du tableau *hist* de la composante *rgb* et doit la retourner.

```
1  private static int FindLow(Color[] hist, int rgb)
```

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

```
1  private static int FindHigh(Color[] hist, int rgb)
```

Compute

La fonction Compute calcule la valeur suivante :
— Si $val < min$, retourne *0*
— Si $val > max$, retourne *255*
— Sinon retourne *255* * (*val* - *min*)/(*max* - *min*)

```
1  public static int Compute(int val, int min, int max)
```

ContrastStretching

La fonction *ContrastStretching* doit appliquer la fonction *Compute* à chaque composante de chaque pixel de l'image *img*.

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

3.2.5 Palier 4 : Instagram (Bonus)

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

Nashville

Pour réaliser ce filtre vous aurez d'abord besoin 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/Colori_balance`
- *ColorMatrix*
- *ImageAttributes*
- *Graphics*

Ces outils sont très pratiques pour modifier une image et vous pouvez retrouver la documentation les concernant 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 !

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 21

MyCorrection

Vous vous souvenez de la classe *auto* ? Oui celle qui ne fait que le *ContrastStretching*. Et bien maintenant il va peut-être falloir en rajouter. Dans *Auto.cs* rajoutez autant de fonctions que vous voulez, le but est qu'elles embellissent les images. Une fois cela fait appelez-les dans la fonction *MyCorrection* avec l'image *img*.

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

3.3 Exemples

FIGURE 4 – Originale

FIGURE 5 – Grey

FIGURE 6 – Binarize

FIGURE 7 – BinarizeColor

FIGURE 8 – Negative

FIGURE 9 – Shift 300x400

FIGURE 10 – Horizontal

FIGURE 11 – Vertical

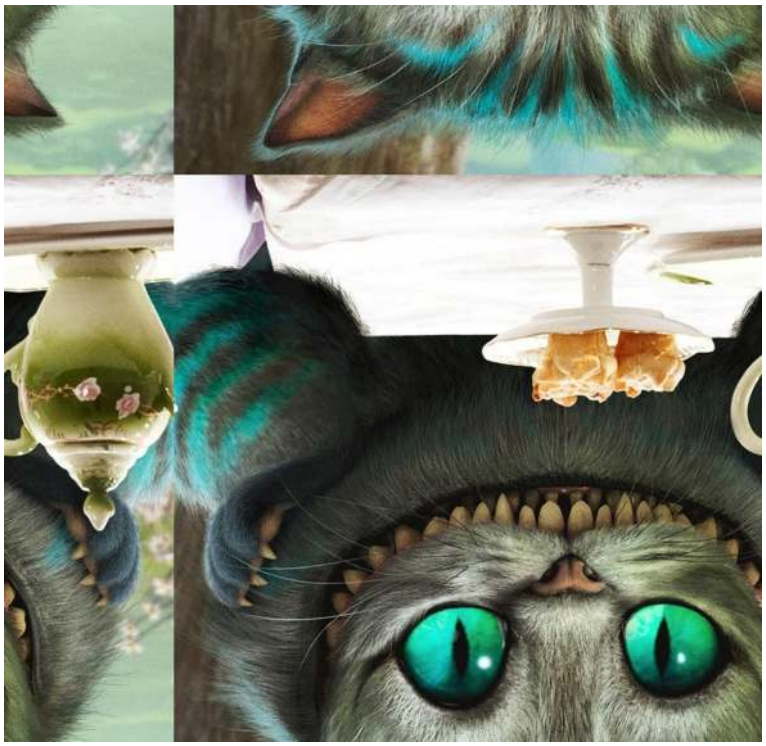


FIGURE 12 – Point 100x100

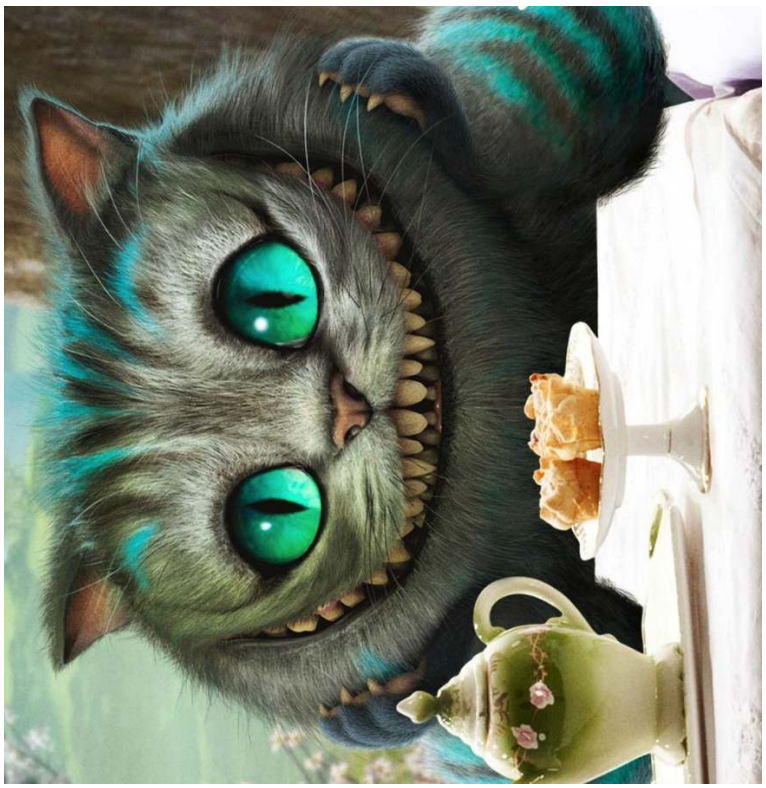


FIGURE 13 – Left

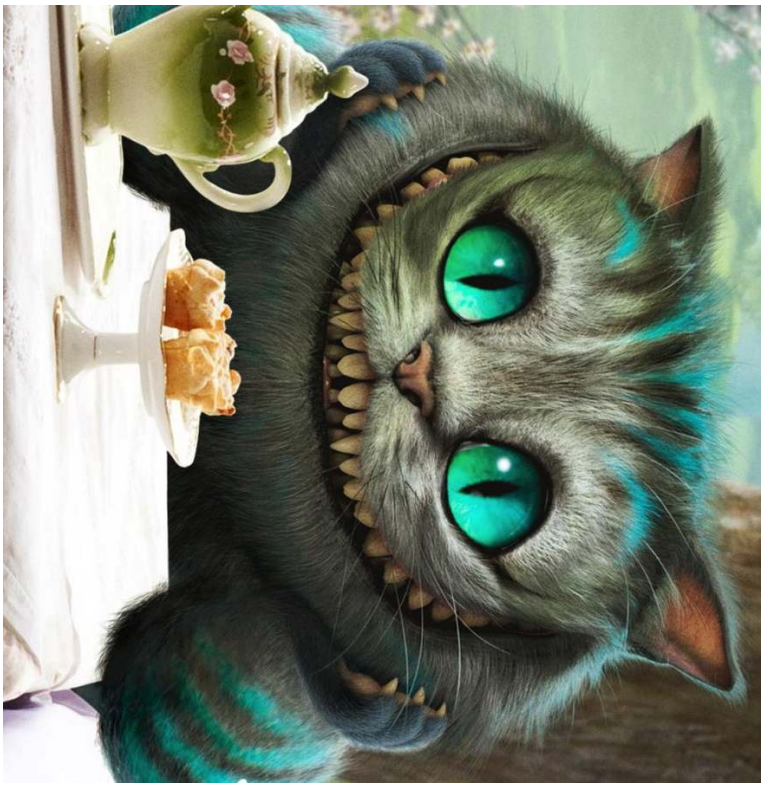


FIGURE 14 – Right

FIGURE 15 – Gauss

FIGURE 16 – Sharpen

FIGURE 17 – Blur

FIGURE 18 – Enhance

FIGURE 19 – Detect

FIGURE 20 – Emboss

FIGURE 21 – Toaster

If you knew time as well as I do, You wouldn't talk about wasting it