TP C#7 : TinyPhotoshop

Assignment

Archive

You must submit a zip file with the following architecture :

```
rendu-tp-firstname.lastname.zip
|-- firstname.lastname/
    |-- 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
```

- You shall obviously replace *firstname.lastname* with your login (the format of which usually is *firstname.lastname*).
- The code **MUST** compile.
- In this practical, you are allowed to implement methods other than those specified. They will be considered bonuses. However, you must keep the archive's size as small as possible.

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline.
Here is an example (where `$` represents the newline and ␣ a blank space):

```
*␣firstname.lastname$
```

Please note that the filename is AUTHORS with **NO** extension. To create your `AUTHORS` file simply, you can type the following command in a terminal:

```
echo "* firstname.lastname" > AUTHORS
```

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

1 Introduction

You recently saw how to use files. Today you are going to learn the manipulation of images in C#

1.1 Objectives

During this TP, we are going to do some applied maths but not only. The goal of this TP is to put into practice the notions you have learned the past weeks to create a program with relatively simple functionalities but which can need some time to see how it works.

2 Course

2.1 L'ordre supérieur

In this topic, we will deal with image. As you will see later, you will have to apply filters to the image. To simplify the matter, we will use a little higher order. Our use is simple and will be content to use it as a function parameter.

In this example, the function **Apply** will return the result of the function **function** on the integer **x**.

```
int Apply(int x, Func<int, int> function)
{
    return function(x);
}
// The word Func makes it clear that this is a function

// The type of input is the type of the parameters that the
// function and is the first int

// The return type is the type that the function returns and is
// the second int

// The function function is a function and is used as such
```

To call our function, we just have to call it with a parameter that corresponds to the input and output types.

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

2.2 Images

The images are stored in different formats (.jpg, .png, .gif, etc.). The C# Library allows us to easily read an image file and directly get useful data about the image inside an object.

Images are divided in pixels (exactly like a matrix). Inside a pixel there 4 elements, red, green, and blue to indicate the real color of the pixel and there is also a last value which is the Alpha field allowing the pixel to have transparency in the image.

2.3 Images Processing

In C# to process images most of the tools are available here:

```
System.Drawing // General Library
System.Drawing.Image //General image object
System.Drawing.Bitmap //The class we use that inherits image
```

I encourage you to read the documentation of the Bitmap object if you need to have more specific details about its behavior (Like how to open/save an image etc.).

It is possible in a Bitmap object to directly access the pixels data by specifying the coordinates (x, y). You can then use this method of Bitmap to get or set the color of a pixel:

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

2.4 Filters

Filters allow the user to easily modify an entire image to give it some effects or upgrade its final look.

There are different kinds of filters.

The first are the simplest, they are filters working on points. It means that the application of the filter does not depend on the whole image but only on the current working pixel. This means that this filter works directly pixel by pixel.

Another kind is the geometrical filters. They are filters which move part of the image somewhere else by copying. The filter copy selected pixels of the image to another part of the image. For example, these filters can be rotation, translation etc.

Finally, the last kind of filter is the one using convolutions.

Convolution is the process by wich a new pixel value is computed by performing a weighed sum of its neighbors and itself. Neighbors are defined using a "window" (3×3, 5×5, . . .) centered on the target pixel. Corresponding coefficients can be found in a convolution matrix, also called mask or kernel. The operation can be written as follows, N being the mask size, P a pixel and M a mask:

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

These calculations will be applied on each component of a pixel, and will be brought back to 0 or 255 if the component exceeds those values. This last operation is called clamping. In our convolution implementation, neighbors that are outside the image will be equivalent to a black pixel, meaning 0 for each component.
The weighed sum of a mask influences the overall intensity of the resulting image. When this sum is 1, the intensity is preserved. Most of the masks used here are given in the source code.

2.5 Steganography

Steganography is the art of dissimulation: its purpose is to make a message go unnoticed in another message. In this lab we will do steganography of text in an image and an image in an image.

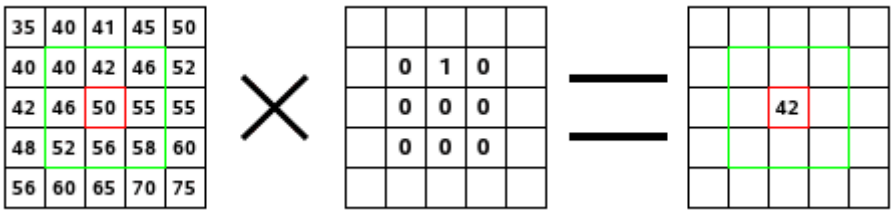

Figure 1: Convolution

Let's start by understanding the principle of coding text in an image. Before you start you need to know that a component is coded on a byte, just like the characters used to build text. That is to say, one pixel is coded on three bytes, we will have one byte for red, one for green and one for blue. We can logically say that we can replace a pixel by three characters, but this is not the case because we would totally lose the original value of the pixel. If we transpose it on large texts, in the end we will destroy a large part of the pixels so there will be no more image.

A question arises: How can we do to lose as little information as possible about the image? Instead of replacing all the bits of the image we will replace only a part. The most significant bits are those that change the values the most, so we will touch the bits of low weight. By convention for this TP one chooses to modify only the four least significant bits of the components. It will thus be necessary to cut the characters in two to be able to insert them in the image.

Let's take an example, we want to code the text "abc" in two gray pixels. We will first cut each character in half and then we will place them in the components of the two pixels.

Letter	a	b	c
ASCII representation	97	98	99
Binary representation	0110 0001	0110 0010	0110 0011

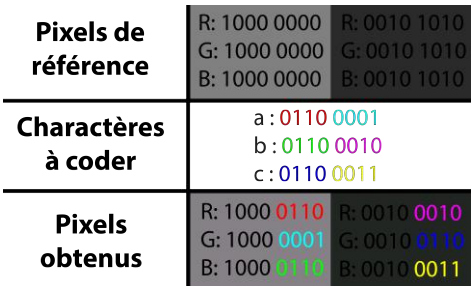

Figure 2: Text Steganography

Let's move on image steganography. This one is easier to understand and to put in place. A component is always coded on a byte, so a pixel always on three bytes. But what prevents us from coding a pixel of an image in a pixel of the other?

Absolutely nothing and that's what you will have to do. Despite the simplicity of this process we will have to, as for the text, degrade the two images. The strongest weights are the

ones that change the value the most, the strongest of the two images are therefore important. Unfortunately we can not stack the two weights so we will be cunning. The principle is to switch the weight of the image to be encoded in the low weight of the other.
For the example we will take the same two pixels as the previous one, with them a blue pixel and a green pixel. In this case it is enough just to take the weight of the pixels to code and put them in the low weight of others.

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

Figure 3: Image Steganography

3 Exercises

3.1 Exercise 1: Image processing

Now you know how to encrypt images to be able to pass along secrets. In spite of that your learning is not finished, you will now learn how to handle images.

3.1.1 Threshold 0: Processing pixels

In this step you will work on the images by modifying them pixel by pixel.
The following functions are in Basic.cs

Grey

The **Grey** function inputs a color and returns the corresponding color in the grey level.
See Figure 5

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

Binarize

The **Binarize** function takes a color and returns either white or black depending on whether the color is closer to one or the other.
See Figure 6

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

BinarizeColor

The **BinarizeColor** function does the same thing as **Binarize**, but it processes component by component, that is, each of the three components (red, green, blue) must be either at the maximum or at the minimum.
See Figure 7

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

Negative

The **Negative** function must inverse each component of color **c** (white becomes black and vice versa) and return it.
See Figure 8

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

Tinter

The **Tinter** function must apply the color **tint** to **c** as factor% and return it. That is, the resulting color is a mixture of the two colors. See Figure 9

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

Apply

Now that we have done all the filters, we will have to apply them to our image. This is where the higher order comes in. See ??.
The **Apply** function must apply the **func** function to each pixel in the **img** image and must return it.

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

3.1.2 Threshold 1: Image Processing

In this stage you will work on modifying the shapes of the images.
The following functions are found in **Geometry.cs**.

Shift

The **Shift** function shifts the image **img** relative to its position of **x**, **y** where the resulting image is. If part of the image is cut by the edges, it must be placed at the far opposite side.
See Figure 10

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

SymmetryHorizontal

The **SymmetryHorizontal** function horizontally mirrors the image **img** at its center and returns the resulting image.
See Figure 11

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

SymmetryVertical

The **SymmetryVertical** function vertically mirrors the image **img** at its center and returns the resulting image.
See Figure 12

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

SymmetryPoint

The **SymmetryPoint** function makes a point symmetry at the **x**, **y** coordinates of the **img** image and returns the resulting image.
See Figure 13

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

RotationLeft

The **RotationLeft** function makes a simple + 90 ° rotation on the **img** image and returns the resulting image.
See Figure 14

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

RotationRight

The **RotationRight** function simply rotates the image by -90 ° and returns the resulting image. See Figure 15

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

Resize

The Resize function uses linear interpolation to resize the **img** image to **x**, **y**.
For that you must already create the resulting image of size **x**, **y**.
First, you have to go through all the pixels of the resulting image. On each pixel you have to look for the corresponding pixel on the original image (using the magnification ratio on the dimensions of the images, the resulting image divided by the original image). If it falls on a pixel we apply directly its value. Otherwise we do the weighted average of the value of the neighboring pixels and we recover the result to apply it to our pixel.
See `https://fr.wikipedia.org/wiki/Interpolation_numerique`
Voir Figure 16

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

3.1.3 Threshold 2: Convolution

In this step you will code the convolution on images. The following functions are in **Convolution.cs**. You can see several examples with different masks (Figure 17 - Figure 22). Only one mask is provided, for others you must find the implement and find the right settings to match the reference.

Clamp

The **Clamp** function takes in any float and must return the closest integer between 0 and 255.

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

```
1  Clamp(-4563232.1254) // 0
2  Clamp(25.1249) // 25
3  Clamp(20201997.4169) // 255
```

IsValid

The **IsValid** function checks whether **x** and **y** are in the range defined by Size. The function must return **true** if it is, **false** otherwise.

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

Convolute

The **Convolute** function computes the **mask** filter on the **img** image and returns the result. For this purpose the function will scan each pixel of the image and for each pixel will make a product of it with the **mask**. To succeed in this function you have at your disposal the two functions that you have just written; you have to use them.
Be careful, do not do the calculations and read the pixels on the same image!

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

3.2 Exercise 2: Steganography

The goal of this exercise is to do steganography on images. For that you will implement two different types of steganography: on texts and on images.

3.2.1 Image section: That's a beautiful picture

Let's go to the pictures! All explanations are in the course section but you will have to respect the following rules:
- You must start by writing the image in the pixel ($\mathbf{y} = 0$, $\mathbf{x} = 0$), then you continue on the pixel in $\mathbf{x} + 1$ until you reach the edge of one of the two images and start again on the next line . - The weak bits of the red component replace the low-order bits of the red. - The strong bits of the green component replace the low-order bits of the green. - The weak bits of the blue character component replace the low-order bits of the blue.

EncryptImage

The textbf EncryptImage function must follow this protocol to encrypt the bitmap **enc** in the bitmap **img** and return it. You must allocate each component of each pixel of the image to be encoded on the components of each pixel of the resulting image. If the image to encode is larger than the image where it is encoded, the function should do nothing.
See Figure 23

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

DecryptImage

The **DecryptImage** function must follow the reverse protocol to decrypt the image **img** and return the corresponding image.

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

3.2.2 Text section: Are you talking to me?

The algorithm can be divided into two parts. For the first part, the goal is to create a table that will serve as a buffer to contain all the characters that we want to encode. You will have to divide each character of the character string into two: the four most significant bits and the four least significant bits and put them in the array starting with the strong bits. The end of a string is encoded by the null byte. For us it is enough for us to add in our buffer two 0 in the continuation. For text **abc** which is encoded in ASCII by **97**, **98** and **99** respectively, we expect to have in our buffer **6, 1, 6 , 2, 6, 3, 0, 0**.
For the second part, we must put each element of our buffer in a component of pixels. Nevertheless, the following rule must be respected: - You must start by writing the image in the pixel ($\mathbf{y} = 0$, $\mathbf{x} = 0$), then you continue on the pixel in $\mathbf{x} + 1$ until you reach the edge and start again on the next line.

EncryptText

The **EncryptText** function must follow this protocol to encrypt the string **text** in the bitmap **img** and must return the image. The function should not do anything if the image is too small to accommodate the text.
See Figure 24

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

DecryptText

The function **DecryptText** must follow the reverse protocol to decrypt the text contained in the image **img** and must return the corresponding string.

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

3.2.3 Exercise 3: ContrastStretch (Bonus)

In this stage you will do an image self-correction. The functions are in **Auto.cs**.

Preamble

The histogram of a component of an image represents the distribution of the intensity of the image, that is to say the number of pixels present for each intensity (from 0 to 255). The histogram of an image informs us of its contrast. Indeed, a dark grayscale image will have a larger number of pixels close to 0, the histogram will be shifted to the left.

A histogram whose values are very close together is synonymous with poor contrast. To correct this, expand the histogram. We will apply the same treatment to each of the components of a color image, as if we were processing several gray levels, to simplify the exercise.

To do this, it is necessary to find, for each component of the pixel, the minimum and maximum intensities of its histogram, that is to say the smallest and the greatest intensity existing in the histogram (in abscissa), and not not the smallest or largest number of pixels present in the histogram (ordinate). Then, we must apply the following formula to each pixel, where x is an intensity between 0 and 255, where c is the component we are working on:

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

Instead of calculating each time the corresponding value for each of its components, it is possible to pre-calculate the results of the function. These results will be stored in look-up tables (LUTs). To know the new values, it will be enough to reach the corresponding table by using the original value like index:

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

Implementation

In order to correct the contrast of our images, it will be necessary to calculate histograms for each of the components, as well as their minimum and maximum intensities. To facilitate the storage of this information, we will use dictionaries.

It is simply a set of key-value couples. In our case, the key will be a representative character of the component (R, G, B). The value will be either a table integers to represent a histogram, a simple integer for intensities lower or maximum. Here's how to handle them:

```
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

Complete this function which returns a dictionary containing the different histograms of the red, green and blue components of an image.

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

Find Low/High

Complete these functions, which return the minimum and maximum intensities of a given histogram, respectively. By default, we will consider that they are worth 0 and 255.

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

The textbfFindHigh function should find the maximum value of the **hist** array of the line **rgb** component and return it.

```
1  public static int FindHigh(int[] hist)
```

FindBound

Complete this function which returns a dictionary containing the values of each component associated with the result of the function f applied to its histogram. It can be used later with the previous functions to retrieve the dictionaries of the minimum and maximum intensities.

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

ComputeLUT

Complete this function which returns the correspondence table of the contrast adjustment function, described in the course part.

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

GetLUT

This function returns the dictionary associating each component to its correspondence table. It will reuse the previous functions.

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

ContrastStretching

This function returns the resulting image of the histogram extension. For this, it will be necessary to recover the correspondence tables. For each pixel, you can then retrieve the new components via the relation expressed at the end of the course part. See Figure 25

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

3.2.4 Exercise 4: Instagram (Bonus)

In this last part you will try to redo some Instagram Filters.

Nashville

To make this filter you will first need to create this function:

In the file **InstaFilter.cs**:

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

This function is about changing the color balance in the image.

To achieve this function, here are some clues:
- `https://en.wikipedia.org/wiki/Color_balance`
- **ColorMatrix**
- **ImageAttributes**
- **Graphics**

These tools are very useful for editing an image and you can find the documentation on MSDN.
Using this function, fill in the **InstaNashville.cs** file.
The filter consists of two colortone calls with the right values for you to search!
See Figure 26

Toaster

Finally toaster is one of the most complicated filters on instagram. To make this one you will have to implement this function:

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

This function allows the user to make a gradient of the **color1** (color of the center) towards the color2 (on the edge) in a circle form.
To help you:
- `https://en.wikipedia.org/wiki/Vignetting`
- **GraphicsPath**
- **PathGradientBrush**

This filter uses several different color balances as well as two different vignettes. Once again, it's up to you to find the right colors!
See Figure 27

MyCorrection

Do you remember the **auto** class? Yes, the one that only contains **ContrastStretching**. Well, now it may be necessary to add more. In **Auto.cs** add as many functions as you want, Their purpose is to embellish the images. Once that is done, call them in the MyCorrection function with the image **img**.

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

4 Exemples

Figure 4: Originale

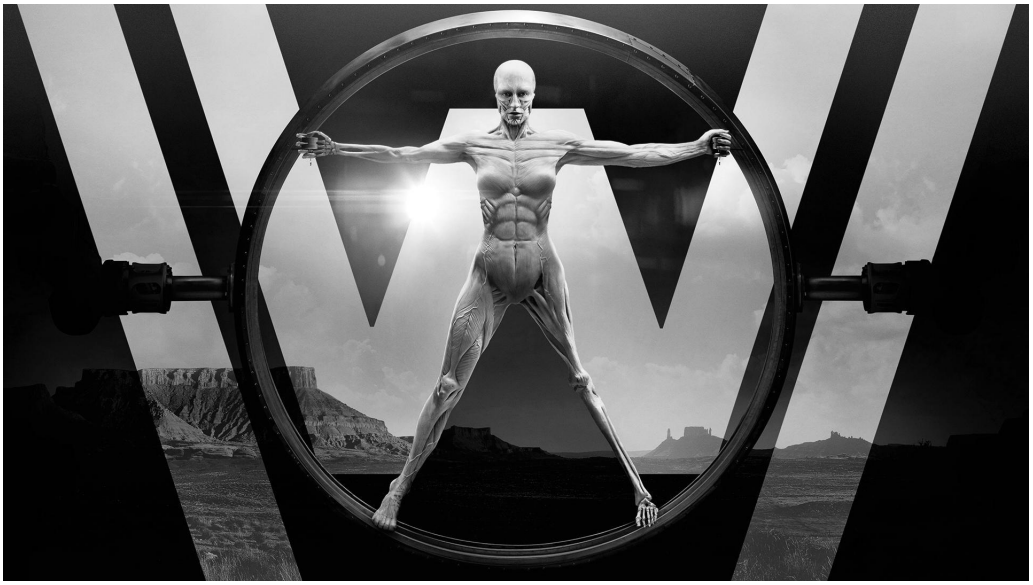

Figure 5: Grey

These violent deadlines have violent ends.

Figure 6: Binarize

Figure 7: BinarizeColor

Figure 8: Negative

Figure 9: Tinter

Figure 10: Shift 960x540

Figure 11: Horizontal

Figure 12: Vertical

Figure 13: Point 250x250

Figure 14: Left

Figure 15: Right

Figure 16: Resize 3840x2160

Figure 17: Gauss

Figure 18: Sharpen

Figure 19: Blur

Figure 20: Enhance

Figure 21: Detect

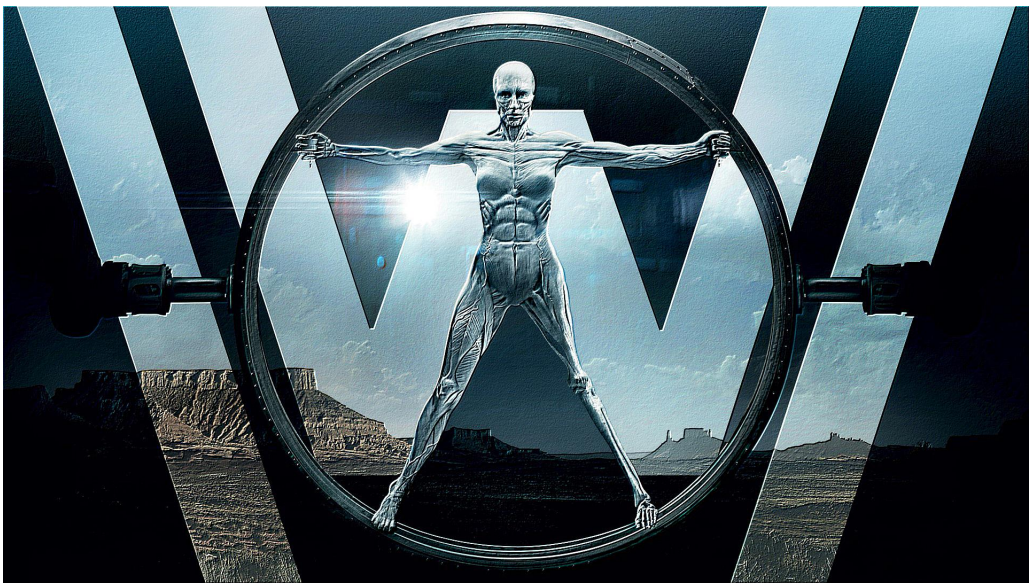

Figure 22: Emboss

Figure 23: Encrypt Image

Figure 24: Encrypt Quote

Figure 25: Auto

Figure 26: Nashville

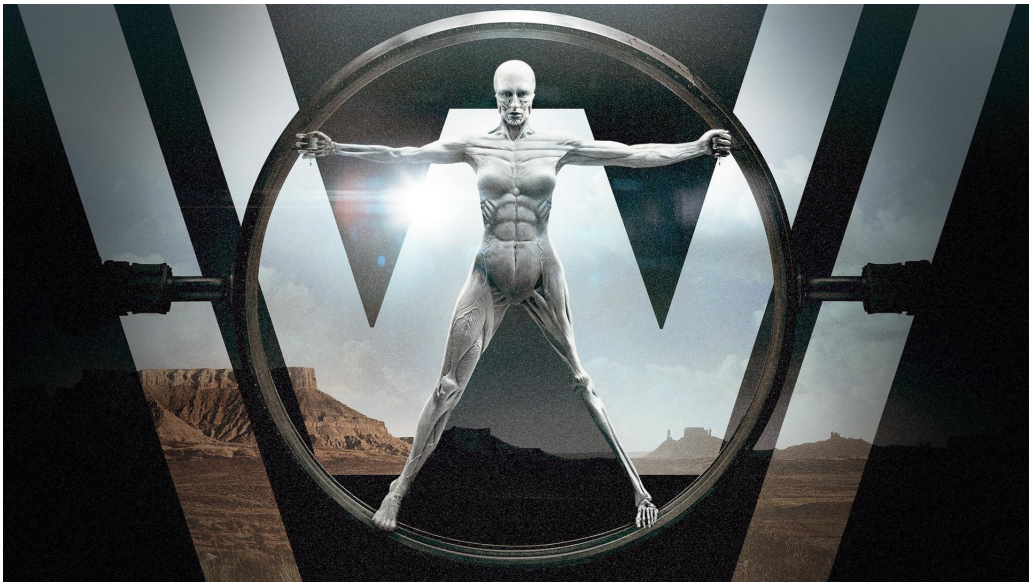

Figure 27: Toaster