Steganography in computer graphics

InSaNe^WaRl0rD *OgGiZ*

14th February 2006

Contents

Chapter 1

Introduction

Hi everyone and thank you to have chosen this article to lead you trough the world of steganography. You will probably notice that this paper uses heavy mathematic descriptions but don't care if you don't like mathz... Don't care of the code if you don't like programming. Formulas are written for mathematicians, text is written for presentation and code is written for programmers. I hope that everybody will be able to fully understand the article through one of these three languages.

Steganography is the art of hiding data inside other files. If cryptography is the way you encode data, then steganography is the way you would hide them. From author name in a picture to governemental secrets on an audio tape, steganography is everywhere. But including data in a file means modifying the file itself so if steganography isn't done properly it might be trivial for a steganalyst to reverse your job.

Steganography is a complex art requiring an important knowledge of file formats and a great sense of ingeniosity. While this paper does not aim to teach everything about the topic, I hope it will help you to get introduced to steganography's best kept secrets.

I would like to note before your begin to read this article that everything here is the product of my own individual work. You will find theorems that probably do not exist in other scientific publications but they are formulated to make everything nice to understand. Also, since I have no official background in steganography this document might seem far away from the reality, or it might even look clever ; I don't know ; But it works. Anyway, I hope you will enjoy it.

Also, the chapter about JPEG[1] steganography is still theoretical only since I haven't got enought time to test it. Encoding and decoding JPEG files is a very complex process which may explain why there are no reports of any real steganography program that works with JPEG. Those which exists does it using hidden fields in the file such as those used for the author name and such...

[1] *Joint Photographic Experts Group*, see http://www.jpeg.org.

Chapter 2

Bits, bytes and data

Before jumping in the deep concepts of image steganography we will need a few theory about bits, bytes and data alignment.

In the computer world, everything is ruled by *bits*. But what is a bit ? Basicaly it is a 1 or a 0, representing the electric state of some latches inside your computer. Then, is it the same than a byte ? Of course not, a *byte* is eight bits; but you might already knew that. With eight bits you can do several things :

1. You can use it as *flags*. That means, a byte would hold eight open/close, active/unactive, set/not set, ... values. Each value is called a flag.
2. You can use it to represent a *positive integer*. As well as you reprensent numbers in base 10, you can represent them in base 2. For example, 123 in base 10 would be written 01111011 in base 2 because $1.2^0+1.2^1+0.2^2+1.2^3+1.2^4+1.2^5+1.2^6+0.2^7 = 1+2+8+16+32+64 = 123$. So, with eight bits you can represent any positive integer between 0 and 255.
3. Any other things your brain can imagine ! For example, there are tricks to represent positive and negative integers.

There are two ways to represent a byte :

1. *Least significant bit* first (LSB first). We write the number with the left handed bit as the smallest impact on the number.
2. *Most significant bit* first (MSB first). We write the number with the right handed bit as the smallest impact on the number. This is the most used format.

Finaly, *data* is just several bytes. Why bytes and not bits ? Because the computer is based around bytes, it is all about that. Note that there are two ways to store data in memory. Imagine we have three bytes called b_1, b_2 and b_3. There are several ways to represent them but only two are plausibles :

1. *Little endian*. We write b_3 first, then b_2 and finaly b_1.
2. *Big endian*. We write b_1 first, then b_2 and finaly b_1.

Remember that Windows OS works with Little Endian and that LSB means Least Significant Bit. Remember that any things in the computer is made of bits : the programs you execute, the files you read, the pictures you watch. And there is only one flavor of bits no matter if they come from a file, a program or a photo !

Chapter 3

24-bits uncompressed image data

There are several data storage formats but here we will play with the easiest one : true color bitmaps. In **BMP24** the image is represented by a set of colors. It just stores colors, no pixel locations. But then, how does it know where to apply the color ? Actually, it does store a bit of localisation informations in its header : width and height. Here is how to draw a bitmap :

You start with the bottom-left pixel and apply the first color value you read. Then, you move on your right and apply the next color you read, then you move on the right and ... and so on. When you have reached the width, you get on the upper line, position at the left handed border and start again. If you are lucky, when you have done all the pixels there will be nothing left to read in the file.

Ok but, what is a color ? So far we have talked about colors but not the way they were actually stored in the file. As we have seen in the chapter title, the format is 24-bits. That means the color information is stored on 24 bits which is 3 bytes. In computer stuff, this means we have three channel coded on one byte : red channel, green channel, and blue channel. You have probably heard about that as the RGB model. Problem is that Windows OS works with Little Endian so you won't read RGB but BGR. If you plan to draw bitmaps yourself, take care of this or it is going to look odd !

So, a bmp file just looks like that :

$$(HEADER)\,(RGB)_1\,(RGB)_2\,(RGB)_3 \ldots (RGB)_{width*height}$$

Where $(RGB)_i$ is the i-th color in the image.

As you have read, anything in the computer is made of bits and there is only one flavor of bit. So, we must find a way to mix picture data with some other file data without demaging the way the picture looks. If we change the color channel bits, the picture will change but we can modify it in so tiny amounts that even the most accurate eye would not see the difference. This is called *LSB Steganography*. If we modify the least significant bits, the difference will be about one 256th on the modified pixels.

3.1 Visibility

Knowing a bit can be either 1 or 0 (probability $p = 0,5$) we can approximate with some statistics and say

$$B(p,3) = \sum_{i=1}^{3} \begin{pmatrix} 3 \\ i \end{pmatrix} p^i (1-p)^{3-i} = 0,875$$

$$R = \frac{\sqrt{3}}{255} = 0,0068$$

which implies that

About 87,5% of the pixels will be changed in an amount of less than 0,7%.

So, even if the modification amount is important, the modification itself is very small. You can modify more than one bit but take care because, for N bits used (all beeing in the LSB order) the change can be up to

$$R = \frac{\sqrt{3}}{255} \sum_{i=0}^{N} 2^i$$

We have seen that for N=1, R=0,7%. For N=2, R=2%. For N=3, R=5%. For N=4, R=10%. And so on... this is exponential ! Note that for N=7 you will find R=173% because of the $\sqrt{3}$. This comes from the definition of the carthesian distance bewteen two pixels :

$$d(A,B) = \sqrt{(A_R - B_R)^2 + (A_G - B_G)^2 + (A_B - B_B)^2}$$

So, if A is the new color and B the original color, there is a not-null probability that we have maximum difference on the three channels. Note that this is not a probability but a normalized distance where 100% = 255.

$$Q \equiv A_R - B_R = A_G - B_G = A_B - B_B$$
$$d(A,B) = \sqrt{Q^2 + Q^2 + Q^2} = \sqrt{3Q^2} = Q.\sqrt{3}$$

Look at the pictures below to see the visibility factor, it is exponential !

3.2 Encoding

So far we haven't discussed how to achieve the LSB steganography. The first thing to do is to program a function that retrieves the Nth bit of a given stream (the file to hide). This can be achieved by an easy algorithm like this one :

```
unsigned char rb (unsigned char *stream, unsigned long n) { return (stream[n/8]
>> (n%8)) & 1; }
```

Then, you just erase the LSB in the picture stream and replace it by the current bit read from the file :

```
bmp_stream[i] &= ~1;
bmp_stream[i] |= rb (file_stream, i);
```

Note that we consider the bitmap as a stream of byte and not a stream of color informations. So, no matter big or little endian, no matter if it is RGB or BGR !

3.3 Decoding

To decode the file, it is as easy. You just read the LSB and append it to the file stream previously filled with zeros, like that :

```
   void sb (unsigned char *stream, unsigned long n, unsigned char b) { stream[n/8]
|= b << (n%8); }

   sb (file_stream, i, bmp_stream[i] & 1);
```

Chapter 4

Continuous data representation

The mathematical definition of continuity says :

$$f : \Re \longmapsto \Re : x \longmapsto f(x)$$

$$\forall x_1, x_2 \mid |x_1 - x_2| \leq \delta \Longrightarrow \exists\, \varepsilon \mid |f(x_1) - f(x_2)| \leq \varepsilon$$

A fonction is continuous if for any given points x_1 and x_2 that are distant of less than a given number δ, the image of theses points are distant of less than another number ε.

This means, if you draw a curve and follow it with your hand, you won't have to raise your finger from the graph. Inspired by this definition, we will say that a file format is continuous if a small change in its number representation induce a small change in the graphic's output.

This is the case for true color 24-bits bitmap. A small change in any of the color channel will result in a very small change in the output color. This may seem trivial but it is not. Actually, it is an important property of bitmaps. We can make a generalisation of the previous chapter by formulating the *LSB Theorem* :

If a medium has continuous data representation, it can be steganographied by a modification of its least significant bits.

This is the case for true color bitmaps but also for other true color format (some works in 16 bits per channel or more), for uncompressed wav files, ... This is not the case for plain text because a little modification of the representation of the character ***C*** (ASCII binary 1000011) could yield character ***B*** (ASCII binary 1000010).

A modification need to be done to the coding and decoding algorithms discussed so far :

```
unsigned char rb (unsigned char *stream, unsigned long n, unsigned char Z) { return
(stream[n/Z] >> (n%Z)) & 1; }
void sb (unsigned char *stream, unsigned long n, unsigned char b, unsigned char
Z) { stream[n/Z] |= b << (n%Z); }
```

Where Z is the number of bits used. For BMP-24 it will be 8 but for WAV-16 it will be 16. It depends on the medium used.

Chapter 5

Indexed data

Data does not always follow the continuity rule and so is the indexed data. In computer graphics, indexed data is mainly used through the usage of *palettes*. After the header, and before the data, is placed a part called the palette. We will look at 8 bits palette but the algorithm also work for 16 bits palette or any other type.

If a file uses only a small amount of colors, there is no need to write explicitely every three channels for each pixels. This would take a lot of memory size so we rather prefer define all the colors and give them an ID. Then once we want to call a color, we just write its ID.

$$(HEADER)\,(PALETTE)_0\,(PALETTE)_1 \dots (PALETTE)_{255}\,(ID)_0\,(ID)_1\,(ID)_2 \dots (ID)_{width*height}$$

Where $(PALETTE)_i$ is the color with the ID i, $(ID)_j$ is the j-th pixel represented by an ID coded on 8 bits (8 bits palette means that you can have up to 256 colors in the palette).

Then, there are two cases that we have to examine :

5.1 Pseudo-Continuous data representation

Sometimes, a palette may follow the continuity rule. This is the case for some grayscale palette for example. But take care, a grayscale palette is not necesseraly continuous ! To be continuous, the data must be written so that the following rule is respected :

$$\forall n : 0 \leq n \leq N-1 \qquad d\left((PALETTE)_n, (PALETTE)_{n+1}\right) \leq \varepsilon$$

For any entry in the palette, the carthesian distance between its neightbours must be less than a given ε.

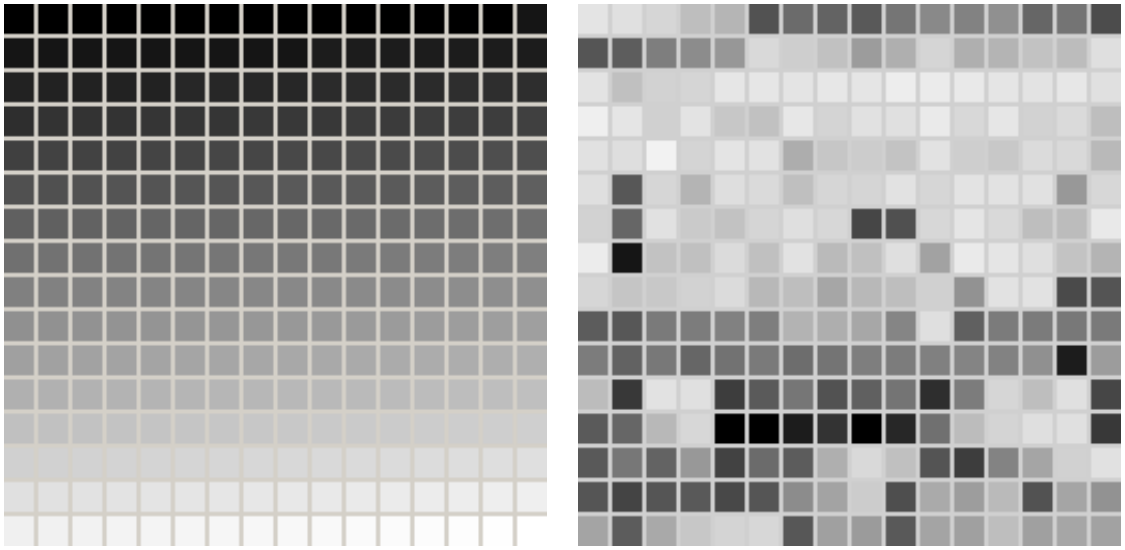
Where N is the number of entries in the palette and d is the carthesian distance discussed in *Chapter 3*. The visibility will depend on the value of ε. If ε is 1 the data will be called continuous, otherelse it will be called pseudo-continuous of class ε. Continuous data might be called as pseudo-continuous of class 1.

If data is pseudo-continuous the LSB Theorem can still be applied. It can even be generalised :

If a medium has pseudo-continuous data representation of class K, it can be steganographied by a modification of its least significant bits. Visibility will depend upon the value of K.

A little modification of the ID representation will yield an ID that gives an output close to the original by the factor ε.

On the figure below you can see an example of continuous grayscale palette and a non-continuous grayscale palette.



5.2 Non-Continuous data representation

Of course, most palette are not pseudo-continuous and an alternative must be found. First, we will append random colors to the palette so that it is full of 256 entries. At least, we will need it to be pair.

You can see on the figure above a common palette. At first, it does not show any interresting pattern.

$$\exists\, i, j \mid 0 \leq i \leq N,\ 0 \leq j \leq N,\ i \neq j : \qquad d\left(\left(PALETTE\right)_i, \left(PALETTE\right)_j\right) \leq \varepsilon$$

Two different entries of the palette may exist such that their distance is less than a given ε.

For a given ε called *depth* on which will depending visibility. With this rule we build $\frac{N}{2}$ couples of palette entries. Each palette entry is used only one time. So, if we invert the colour in all the couples it should have only little change on the output. The usable space will depend on the picture and the chosen ε. In computer terminology we will write it as :

```
struct couple_s
{
   unsigned char id1;
   unsigned char id2;
};

struct couple_s palorder[128];
```

Any colours that is in a couple and so fits the previous condition will be called *compatible* otherelse *incompatible*. There are then two ways to accomplish steganography :

5.2.1 Quick Steganography

The palette will have two channels modified, choose the ones you want to use. The first channel LSB will be set to 1 if the current palette entry is compatible otherelse 0. The second channel will be set to 0 for the first entry of the couple and to 1 for the second entry of the couple. So each palette entries store two informations :

1. Is the couple compatible ?
2. Do I represent a 1 or a 0 ?

Then the data of the picture is read. The couple that owns the palette ID is found and if the couple exists the ID within the stream is replaced by the first ID of the couple if the current bit to code is a zero, or the second ID of the couple if the current bit to code is a one. If the couple does not exist because the color were incompatible then we skip and go to the next byte without incrementing the current bit to code.

Decoding is even easier. Just iterate the picture stream and read the first channel. If the LSB is null, skip and go to the next byte. If the first channel is not null, read the second channel and append the LSB to your file stream. Decoding is the advantage of the *Quick Steganography* issue as you do not have to build the couples list again.

5.2.2 Nice Steganography

With *Nice Steganography* you will have to build the couples list in both encoding and decoding process. If the current color to code or to decode is incompatible we skip to the next byte otherelse we use the relative positions of the entries in the couple.

$$\begin{cases} b_0 = min\,(p_0, p_1) \\ b_1 = max\,(p_0, p_1) \end{cases}$$

Where b_0 is the representation for either 0 or 1, b_1 is the opposite of b_0, p_0 is the position in the palette (the ID) of the first entry found in the couple and p_1 is the position of the second entry found in the couple.

To encode data, you replace the current ID by the entry that represent the bit you want to code. To decode data, you just determinate if the ID is b_0 or b_1 and append a 0 or a 1 depending on the results.

The big advantage is that no harm is done to the palette as it is not modified. The disadvantage is that you have to recompute the couples list for the decoding process.

Chapter 6

Lossless compression

It is possible to compress data such that the whole picture stream takes less place. You probably know ZIP[1] encoding but there is also LZW[2] encoding. LZW is used in GIF[3] image processing.

The big advantage with gif files is that your data is compressed but if you decompress the file, nothing is lost ! The information is kept in its integrity. This is known as *lossless compression*.

This also means that any modifications on the LSB would survive the compression/decompression algorithm. Plus, you might know that GIF uses indexed data. So, we have our very first popular picture file format that can be steganographied. While a bmp file would be suspect on the Internet, a GIF will not.

GIF steganography mainly deals with non-continuous data representation and the encoding or decoding process as already been seen in the previous chapter.

[1] Originaly designed for the program *PKZIP* by *Phil Katz*.
[2] *LZW*, Created by *Abraham Lempel*, *Jacob Ziv* and published by *Terry Welch* in 1984. **US Patent 4,558,302**.
[3] *Graphics Interchange Format* introduced in 1987 by *CompuServe*.

Chapter 7

Lossy compression

While there are file formats that store graphics without destruction of the data, some does not. This is the case of the very famous JPEG file format. You may have all experienced it yourself, JPEG induce loss in the compression. But, JPEG is also the major graphics storage medium used on the Internet so it would be very interesting to find a way to apply steganography to it.

As you will see, JPEG is much more mathematic than the other file formats and this makes the steganographic process very difficult.

7.1 Encoding

Encoding is done in several steps. Imagines you have a true color image stream, you will have to apply these techniques :

7.1.1 Color Switch

JPEG does not deal with RGB color but with another format used in TV and known as YCbCr, sometimes refered as YCC. Y is the luminence (or a factor very close to it) and varies from zero to one. Cb and Cr are the color difference and color chroma and varry from -0,5 to +0,5.

Here is the transformation matrix :

$$A = \begin{pmatrix} +0,299 & +0,587 & +0,114 \\ -0,168736 & -0,331264 & +0,5 \\ +0,5 & -0,418688 & -0,081312 \end{pmatrix}$$

and is applied on a **normalized** RGB vector to yield a YCC vector. The previous transformation can also be read as :

$$\begin{cases} Y = 0,299.R + 0,587.G + 0,114.B \\ Cb = -0,168736.R - 0,331264.G + 0,5.B \\ Cr = 0,5.R - 0,418688.G - 0,081312.B \end{cases}$$

Where R, G and B varries from zero (dark) to one (bright).

7.1.2 Fourier Transform

JPEG uses a special type of Fourier Transform called the Discrete Cosine Transform. This uses much heavier mathematics than we can afford, roughly it consists of using the set defined as :

$$f_j = \sum_{n=0}^{N-1} x_n cos\left[\frac{\pi}{N} j\left(n + \frac{1}{2}\right)\right]$$

The picture is decomposed in a set of 8x8 pixels blocks and the Fourier Transform is applied to them. But prior the transform is done the color of the image is shifted so that it varries from -128 to +127. The transform is applied on Y, Cb and Cr separately. As we deal with integer, we round them to the nearest one.

7.1.3 Quantization

Now, each component of the 8x8 block is divided by a particular scalar. Every A_{ij} will be scaled by the same factor but this factor will change depending on the value of i and j.

This step will introduce some small values and even some zeros.

7.1.4 Encoding

Finaly, the picture is compressed. A typical compression algorithm for JPEG is the Huffman[1] coding which is lossless compression.

7.2 Decoding

Decoding is done in a similar way than encoding but in the reverse order. Note that the inverted Fourier Transform is the Invert Discrete Cosine Transform which is defined as :

$$f_j = \frac{1}{2}x_0 + \sum_{n=1}^{N-1} x_n cos\left[\frac{\pi}{N} n\left(j + \frac{1}{2}\right)\right]$$

7.3 Possible Issue

Possible issue for steganography lies in the quantization process. We have seen that there will be values less than a known ε. We could then say $\varepsilon = 1$ and set to zero every values that is less or equal than 1. Then, every null values would be filled with a one or a zero depending on the current bit to code. The reading process is even easier since one would just have to read the file and note any one or zero values to the bit stream.

As we strike against the little numbers this should have only small report in the RGB version of the JPEG file.

[1] Developped at the MIT by *David A. Huffman* in 1952.

Chapter 8

Diffusion and confusion

Preventing steganalysis is impossible but we can make their job really painful. First of all, we can encode the data stream with some cryptographic algorithm such as *BLOWFISH*[1] or *IDEA*[2]. We can reorder the data so it would be easy to retrieve the stream but hard to have it in the right order. We can also make it hard to catch the stream by modifying the LSB position. Rather than using the standard definition of LSB we might want to use the bit forward too in a chaotic way. Here is a four step method to be nearly steganalysis safe :

8.1 Blur the file

First of all, you will have to add some noise to the picture. If you do not, then a part will look more noisy than the rest. This is an easy way to discover steganographied file. So, be sure that there is no more noise in the part where a file is hidden and the part where there is no hidden data. To accomplish this, just apply the bit ciphering function on the whole bitmap stream using randoms as bits to hide. While we are talking about noise, take care to pure color such as plain black or plain white because a small modification of these could make the steganalysis job easier. Be careful with straight lines that should be uniform in color ; but that is way harder to prevent !

8.2 Encode the data

When the file is loaded, it is first encoded with a key that depends on the graphic itself. For example, you can generate a key by summing all the MSB, then xoring them with the bit backward the MSB. It is up to you but remember that you cannot use the bits that are subjected to modifications !

8.3 Shuffle the data

Rather than encoding the data directly in a linear way in the graphic, design a function that uses the number of bits already written, the number of pixels in the picture and the number of bits to encode. But remember that the function shuffle must be bijective. This can be written with maths as :

$$\forall x_1, x_2 : \qquad f(x_1) = f(x_2) \Leftrightarrow x_1 = x_2$$

Every point has only one image and every image correspond to only one point.

There are two ways to design such a function :

[1] Designed in 1993 by *Bruce Schneier*.

[2] *International Data Encryption Algorithm* designed by *Xuejia Lai* and *James L. Massey* in 1991.

8.3.1 Numerical permutation

The first way might be the best for the programmers as it involve discrete data. The basic idea is to build a small permutation table like this one :

$$P = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 5 & 3 & 0 & 1 & 7 & 4 & 2 & 6 \end{bmatrix}$$

$$P^{-1} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 2 & 3 & 6 & 1 & 5 & 0 & 7 & 4 \end{bmatrix}$$

Note that :

$$P \circ P^{-1} = P^{-1} \circ P = I$$

Where I is the identity operator : $I(x) = x$.

The advantage of a 8 bits permutation box is that it can be applied on a single byte. Imagine a set of permutation called $\langle P \rangle_n$ where there are permutations P_0, P_1, ... P_n. Imagine now a field of 512 bits. This can be devided in a box of 64 bytes which can itself be divided in 8x8 boxes of 8 bits. Here is an example with 64 bits. First you apply P_0 on every eight bits.

$$\begin{cases} A_0 = P_0(b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7) \\ \vdots \\ A_7 = P_0(b_{56} b_{57} b_{58} b_{59} b_{60} b_{61} b_{62} b_{63}) \end{cases}$$

Then you apply P_1 on the resulting eight boxes $\langle A \rangle_n$.

$$B = P_1(A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7)$$

You can use a similar pattern for every set of 8^n. The major problem is that the file size *MUST* be a multiple of 8^n. At least, it will work with $n = 1$ because data size is always a multiple of eight bits. To decode the string, just use the inverse set $\langle P^{-1} \rangle_n$ with care of the order. Be sure your permutations are commutables :

$$P_0 \circ P_1 \circ \ldots P_n = P_n \ldots \circ P_1 \circ P_0$$

8.3.2 Polynomial movement

An interresting field of research is the polar coordinate shuffle. This uses more mathematic and have bad sides but can be really great if properly applied. The main idea is to use polar coordinate which can be written as :

$$\begin{cases} x = R.cos(\theta) \\ y = R.sin(\theta) \end{cases}$$

R and θ beeing functions. For example, R can be expressed as a polynom using the number of bits already written :

$$R_N(x) = \sum_{i=0}^{N} a_i x^i$$

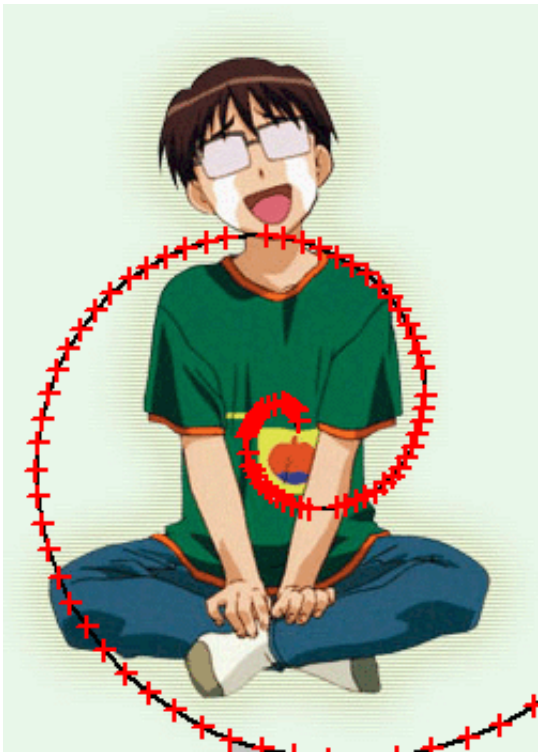
This is the general formula for a polynom of Nth degree with coefficient $\langle a \rangle_N$. Example :

$$R_2(x) = 1 + 3x + x^2$$

There is only one condition that must be respected :

$$\forall x \geq 0 \qquad R_N(x) \geq 1$$

Because a null or negative radius would get us into troubles... There is only one difficulty with the polar coordinates and it lies in the mathematical form of the equation. Maths deals with real numbers that can take any value while we need integer values. A good fix would be to define a function θ that takes a argument k that is incremented as long as we do not have a new value. Also, keep a trace of the points that are already written. As you can see on the figure below, there are many unconvential graphs to achieve. Sample positions for bits to encode are shown with red crosses, the curve itself is in black.



$R(\theta) = 1 + 3\theta + \theta^2$

$R(\theta) = 10.\theta.tg(\theta) + 1$

8.4 Invoke chaos

There is nothing worse than having to put order into chaos or what looks like it. So far we have only used the LSB but imagine we could use one of the two LSB on a 24 bits bitmaps. Here are all the possibilities per color :

$$C = \left\{ \begin{bmatrix} xb \\ xb \\ xb \end{bmatrix} \begin{bmatrix} xb \\ xb \\ bx \end{bmatrix} \begin{bmatrix} xb \\ bx \\ xb \end{bmatrix} \begin{bmatrix} xb \\ bx \\ bx \end{bmatrix} \begin{bmatrix} bx \\ xb \\ xb \end{bmatrix} \begin{bmatrix} bx \\ xb \\ bx \end{bmatrix} \begin{bmatrix} bx \\ bx \\ xb \end{bmatrix} \begin{bmatrix} bx \\ bx \\ bx \end{bmatrix} \right\}$$

For only one color there are eight ways to use the two least significant bits. Then, there is an infinite amount of ways theses combinaisons can be linked. For example, one could use the folowing pattern :

$$C_0C_4C_3C_0C_6C_4C_2C_1C_6C_4C_2\ldots$$

While another one could do :

$$C_1C_2C_1C_0C_0C_3C_1C_0C_0C_2C_7\ldots$$

Any combinaisons of the eight bricks can be used. The more chaotic the better. For N colors, there are 8^N possible ways to rearrange the data. The bad side is that two LSB will be more visible than one LSB, resulting in poorer quality.

Of course, if we put true chaos it would be useless because we would not be able to decode the data after. So we need a function that looks chaotic but is actually reproducible. This will be achieved with only one parameter : the position of the color in the stream. To have pseudo-chaotic function we will deal with prime numbers, for example the indice of C can be found with this function :

$$\begin{cases} \chi(x) = 1 & x = 0 \\ \chi(x) = 0 & x \neq 0 \end{cases}$$

$$f : \aleph \longmapsto \aleph : x \longmapsto \chi(x \; mod \, 2).1 + \chi(x \; mod \, 3).2 + \chi(x \; mod \, 5).4$$

7,0,1,2,1,4,3,0,1,2,5,0,3,0,1,6,1,0,3,0,5,2,1,0,3,4,1,2,1,0,7,0,1,2,1,4,3,0,1,2,5,0,3,0,1,6,
1,0,3,0,5,2,1,0,3,4,1,2,1,0,7,0,1,2,1,4,3,0,1,2,5,0,3,0,1,6,1,0,3,0,5,2,1,0,3,4,1,2,1,0,7,0,
1,2,1,4,3,0,1,2,...

You will see it is a periodic set repeating every 30 numbers. This is because we used prime numbers, $30 = 2 * 3 * 5$!

Chapter 9

From bits to file pack

When you have large pictures and small files to hide, why wasting empty space ? You might have reached a level of steganography where you will want to store several files in only one picture. This is known as a pack-file. There are two common ways to achieve pack-files but they are all based on a header containing the name of the files and their size in bytes and the data itself.

Of course, do not forget to have enough space in the graphic to hold the header too !

9.1 Queue-leu-leu

Queue-leu-leu (*pronounce keu-leu-leu with eu as in "Do you know the answer? ... eeeeuuuuh"*) is a french expression meaning that one thing follows another very closely like a pack of wolves. In this pattern, each header will be followed by the data of the described file then follow by the next file's header, data, and so on.

9.2 Indexation

In the indexation process you first write the total size of the hidden data, the name of each files in the order they appear and finaly the offsets in the data at which appears the beginning of the file. The size can be retrieved by substracting the offsets togheter.

$$\begin{cases} size_{n-1} = offset_n - offset_{n-1} \\ size_n = totalsize - offset_n \end{cases}$$

Chapter 10

A brief history of... motivations

There are many ways steganography can be used. One may think of a spy sending secret informations to a foreign intelligence service while another would think about terrorist groups sending picture containing potential targets. In both cases, steganography is used to transmit hidden and critical informations. But, one could use steganography for other purposes such as sending a cryptographic key. We all know that symetric cryptography has the disadvantage of the trusty way to exchange the key; steganography might be one of these safe ways. But, there are many other way steganography can be used such as the watermarking. Basicaly, watermarking is storing a copyright inside a file so that the author can still prove the file is his job. Watermarking itself is a whole field of research and steganography plays only a small part in its means...

I will conclude with a way that is very clever in its formulation : ***Using steganography to prevent steganography***. It might sound odd but, actually, is very serious. By encoding a known file in a medium through steganography you can know if the file has been corrupted. Imagine you have a BMP24 file where you encode the string "*I'm safe and sane*" repeatidly in the LSB. If someone tries to change the LSB you will not be able to read the original "*I'm safe and sane*" message again. It does not give information about what has been done to the file but tells you that the file has suffered some modifications !

Remember that a strong secret rely on its environement. A suspect file will remain suspect wathever the algorithm you have used while a common picture will still look very common if you take some care (*ie:* not modifying one of the MSBs!).

Index