

Département d'Informatique

Informatique 2 SPC

2016-2017

TD4 – Fichiers et stéganographie

1. La stéganographie

La stéganographie est l'art de la dissimulation : son objet est de faire passer inaperçu un message dans un autre message. Elle se distingue de la cryptographie, « art du secret », qui cherche à rendre un message inintelligible à la personne ne possédant pas la clé.

Par exemple l'image de gauche contient un message caché dans l'image de droite (originale).



Ce message est la répétition de plusieurs centaines de fois de :

```
Ceci est un message top secret pour les etudiants.  
-----> POUR REUSSIR, IL FAUT TRAVAILLER ! :) <-----  
Pffff... Franchement c'est nul ce message !  
007, alias, James Bond __@_y
```

Comment cela est-il possible ? Une image est composée de pixels ayant chacun 3 composantes Rouge, Vert et Bleu. Une composante est représentée sur 8 bits par une valeur décimale entre 0 et 255 (ce sont les différentes valeurs possibles sur 8 bits – voir le cours Bonus « Machines Numériques »). Sur la composante rouge, la valeur 0 signifie « ce pixel a peu de rouge » alors que la valeur 255 signifie « ce pixel a le maximum de rouge ». La combinaison des trois couleurs R,V et B donne la couleur finale du pixel.

Nous savons qu'un caractère se représente aussi sur 8 bits selon la table ASCII. L'idée de ce TD va consister à charger une image, à accéder à son tableau de pixels et pour chaque caractère à cacher à dissimuler ses bits dans les pixels de l'image de la façon la plus furtive.

Une approche naïve consisterait à remplacer par exemple la seule composante rouge (qui fait 8 bits) par le codage ASCII du caractère. On voit que le résultat n'est pas concluant (et surtout pas discret) !



Aussi nous allons décomposer les 8 bits de nos caractères ASCII en 2 paquets de 3 bits et en 1 paquets de 2 bits c'est-à-dire que si notre caractère vaut xxxyyyzz_b.

Nous allons ensuite utiliser ces paquets pour faire des opérations sur les bits de poids faibles de chaque composante.

Pourquoi les bits de poids faibles ? Tout simplement car la modification de ceux-ci est ce qui donne le moins d'impact visuel. Prenons un exemple. Si on modifie un pixel dont la composante rouge est 129 qui se représente en binaire par 10000001_b la modification de 2 bits de poids fort aura un impact très fort alors que la même modification de 2 bits de poids faible sera mineure. Par exemple inverser les bits donne 01000001_b soit la valeur décimale 65 contre 10000010_b soit la valeur décimale 130. On imagine bien la différence de perception.

Une solution pourrait consister à tout simplement remplacer les bits de poids faibles de chaque composante. Par exemple les 3 bits de la composante rouge serait xxx_b, les 3 bits de la composante verte serait yyy_b et 2 bits de la composante bleue serait zz_b. Le problème de cette solution est qu'une personne peut retrouver le message simplement en connaissant la méthode utiliser pour cacher nos bits.

Afin d'être plus subtile nous allons les « chiffrer » avec la table de vérité du XOR.

En Processing le xor se note ^

En plus le XOR ayant pour propriété que :

(A XOR B) XOR B = A

nous pourrons « déchiffrer » en possédant l'image originale.

AND			OR			XOR		
A	B	Output	A	B	Output	A	B	Output
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

exclusive-OR

En Processing, un pixel est représenté sur un entier sur 32 bits dont les trois composantes sont codés sur 8 bits (un octet). Si on représente en hexadécimal un pixel on a 0x00RRVVBB avec 0xRR qui est la valeur hexadécimale de la composante rouge, 0xVV la valeur de la composante verte et 0xBB la valeur de la composante bleue. Pour manipuler ces composantes il faudra donc procéder à des décalages de bits à droite >> ou à gauche <<. On utilisera aussi les propriétés du & (ET) et du | (OU) logique pour éliminer ou conserver certains bits.

Ainsi si le caractère à cacher vaut : (c₈c₇c₆c₅c₄c₃c₂c₁)_b et que pixel initial vaut 0x00RRVVBB avec 0xRR=(r₈r₇r₆r₅r₄r₃r₂r₁)_b, avec 0xVV=(v₈v₇v₆v₅v₄v₃v₂v₁)_b et avec 0xBB=(b₈b₇b₆b₅b₄b₃b₂b₁)_b il subira les opérations suivantes

0xRR=(r₈r₇r₆r₅r₄r₃r₂r₁)_b avec r'₃=r₃^c₈, r'₂=r₂^c₇, r'₁=r₁^c₆

0xVV=(v₈v₇v₆v₅v₄v₃v₂v₁)_b avec v'₃=v₃^c₅, v'₂=v₂^c₄, v'₁=v₁^c₃

0xBB=(b₈b₇b₆b₅b₄b₃b₂b₁)_b avec b'₂=b₂^c₂, b'₁=b₁^c₁

Arbitrairement nous décidons de cacher les fins de lignes (retour à la ligne) par le caractère 255 de la table ASCII. Quand nous avons caché tous nos caractères nous arrêtons de faire des xor.

- 1) Définir une fonction **cacheCaract** qui prend en entrée un caractère et un entier constituant le pixel dans lequel il faut cacher le caractère et qui retourne entier qui constitue le nouveau pixel.
Remarque : Pour reconstruire un pixel à partir des trois composantes il est possible d'utiliser `color()`.
- 2) Définir une fonction **retrouverCaract** qui prend en entrée deux entiers constituant le pixel de l'image original et le pixel de l'image « chiffrée » et qui retourne le caractère centier qui constitue le nouveau pixel.
- 3) Écrire la fonction **saisieFichierExistant** qui prend en paramètre un message et demande à l'utilisateur de saisir au clavier un nom de fichier en conformité avec ce message. **Si le fichier existe** la fonction retourne une chaîne contenant le nom du fichier à lire et sinon elle retourne la valeur `null`.
- 4) Écrire la fonction **saisieFichierNonExistant** qui prend en paramètre un message et demande à l'utilisateur de saisir au clavier un nom de fichier en conformité avec ce message. **Si le fichier n'existe pas** la fonction retourne une chaîne contenant le nom du fichier à créer et sinon elle retourne la valeur `null`.
- 5) Écrire la fonction **cacheTexte** qui demande à l'utilisateur le nom du fichier texte à cacher et le nom de l'image original dans laquelle on va cacher et nom du fichier de sortie. Elle retourne la valeur `false` si le fichier clair ou l'image originale n'existent pas ou si le fichier de sortie existe déjà. Elle utilise ensuite la fonction `cacheCaract` pour réaliser l'image contenant le message à cacher. Si tout se passe bien cette fonction retourne `true`. Elle peut retourner `false` si l'image est trop petite pour le nombre de caractères à cacher (il n'y a pas assez de pixels).
- 6) Écrire la fonction **retrouverTexte** qui demande à l'utilisateur le nom de l'image contenant le message, le nom de l'image original et le nom du fichier qui contiendra le texte à retrouver. Elle retourne la valeur `false` si les fichiers d'image n'existent pas ou si le fichier à créer existe déjà. Elle utilise ensuite la fonction `retrouverCaract` pour réaliser l'image contenant le message à cacher. Si tout se passe bien cette fonction retourne `true`.
- 7) Écrire le programme principal qui proposera un menu à l'utilisateur, en faisant appel aux fonctions précédentes, pour lui permettre de faire différentes opérations de stéganographie sur les fichiers qu'il choisira. On pensera à traiter les erreurs des appels aux différentes fonctions

Le menu proposera :

- De cacher un texte
- De retrouver un texte
- De quitter le programme

Pour manipuler les images, nous utiliserons :

- `loadImage` qui donne une variable de type `PImage`
- `loadPixels` pour pouvoir avoir accès au tableau de pixels de l'image
- `image` pour afficher une image
- `updatePixels`
- `save` pour créer une image