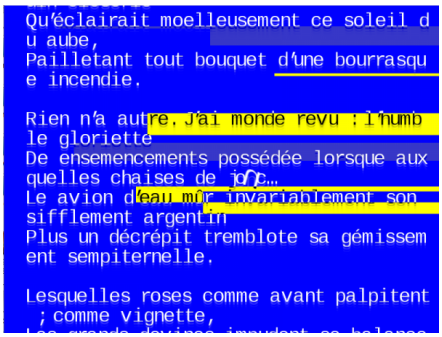
Corruption Littéraire

Lors de cette résidence j'ai voulu expérimenter la corruption sémantique de textes en remplaçant certains mots par des synonymes. Les synonymes sont récupérés dynamiquement à travers l'API du site http://thesaurus.altervista.org. Pourquoi j'ai choisi ce site ? Ben les dictionnaires de synonymes en langue française proposant une API ne courent pas les cyber-rues, le choix a été rapide. Il faut noter que chaque API a ses propres règles de communication. Il devrait donc être possible d'utiliser le programme de corruption de texte avec un autre dictionnaire en ligne à condition d'adapter le programme aux règles de la nouvelle API.
Le programme est composé de deux modules maladroitement intégrés ensemble (ça s'est fait à la va vite sans réelle vision d'ensemble, et puis il y avait une tireuse avec bière à volonté...). Le premier module, écrit en Python, se charge de récupérer les synonymes en lignes et de sauvegarder le texte corrompu dans un nouveau fichier. Le second module, en Processing (Java), fait défiler les textes à l'écran, ajoute des effets de corruption graphique et fait régulièrement appel au premier module.



Le module de corruption de texte

Il peut s'utiliser de façon autonome, à condition d'avoir installé Python3 sur votre machine.
N'oubliez pas de remplacer la clé d'API du site http://thesaurus.altervista.org par votre propre clé (après avoir crée votre compte) dans la variable `key`.

Usage:
`$ python3 textcorrupt.py` *`fichier_original fichier_de_sortie`* [*`nombre_d'itérations`*]
où `nombre_d'itérations` est le nombre (optionnel) de fois où l'algorithme corrompra le texte (sachant que par défaut 2% des mots sont modifiés à chaque itération).

Si le code est plus complexe qu'il ne devrait c'est parce qu'il utilise deux fichiers locaux de cache :
`french_filter_out.txt` pour conserver les mots qui ne seront jamais modifiés (mots pour lesquels le dictionnaire en ligne renvoie une erreur mais vous pouvez également l'éditer manuellement pour rajouter vos mots à protéger).
`syn_dict.txt` pour conserver tous les mots, ainsi que leurs synonymes, pour lesquels une requête a déjà été envoyé au dictionnaire en ligne.
Ces deux fichiers seront créés automatiquement à la première exécution du programme.

textcorrupt.py (cliquer pour afficher le code)

textcorrupt.py

```
import requests
import random
import os.path
import sys

syn_dict = "syn_dict.txt"
filter_out = "french_filter_out.txt"

endpoint = "http://thesaurus.altervista.org/thesaurus/v1"
key = "à remplacer par votre clé personnnelle"
language = "fr_FR"
output = "json"
```

```
def loadFilterOut(filename):
    filterOut = []
    if not os.path.isfile(filename):
        pass
    else:
        with open(filename, 'r') as file:
            for line in file.readlines():
                if line:
                    filterOut.append(line.strip())
    return filterOut

def saveFilterOut(filename, filterOut):
    with open(filename, 'w') as file:
        for word in filterOut:
            file.write(word+'\n')


def loadSynDict(filename):
    synDict = dict()
    if not os.path.isfile(filename):
        pass
    else:
        with open(filename, 'r') as file:
            for line in file.readlines():
                if line:
                    words = line.split('\t')
                    synDict[words[0]] = list(map(cleanWord, words[1:]))
    return synDict

def saveSynDict(filename, synDict):
    with open(filename, 'w') as file:
        for word in synDict:
            file.write(word)
            for syn in synDict[word]:
                file.write('\t' + syn)
            file.write('\n')


def getSynonyms(word, synDict, filterOut):
    word = cleanWord(word)
    if word in synDict:
        return synDict[word]

    sys.stderr.write("Fetching synonym of '{}'...\n".format(word))
    url = f"{endpoint}?word={word}&key={key}&language={language}&output={output}"
    r = requests.get(url)
    try:
        r.raise_for_status()
        json = r.json()
        synList = json["response"][0]["list"]["synonyms"].split('|')
        synList = list(map(cleanWord, synList))
        synDict[word.lower()] = synList
        return synList
    except:
        sys.stderr.write("Synonym not found\n")
        filterOut.append(word)
        return None
    finally:
        sys.stderr.flush()


def chooseSynonym(word, synDict, filterOut):
    syns = getSynonyms(word, synDict, filterOut)
    if syns:
        return random.choice(syns)
    else:
        return word


def validate(word, filterOut):
    return len(word)>1 and word.isalpha() and not word.casefold() in filterOut


def cleanWord(word):
    return word.casefold().strip()


def decomposeWord(word):
    iStart = 0
    iEnd = len(word)
    while(iEnd>0 and not word[iEnd-1].isalpha()): iEnd -= 1
    if len(word)>2 and word[1] == "'": iStart=2

    return word[:iStart], word[iStart:iEnd], word[iEnd:]


def parseText(text, synDict, filterOut):
    lines = text.split('\n')
    corruptedLines = []
    for line in lines:
        words = line.split()
        corruptedWords = []
        for word in words:
```

```
                prefix, radix, suffix = decomposeWord(word)
                # The corruption probability is 0.2 for each word
                if random.random()<0.2 and validate(radix, filterOut):
                    corruptedWord = chooseSynonym(radix, synDict, filterOut)
                    # Keep capital letters
                    if word.istitle():
                        corruptedWord = corruptedWord[0].upper() + corruptedWord[1:]
                    corruptedWords.append(prefix+corruptedWord+suffix)
                else:
                    # Word is kept unchanged
                    corruptedWords.append(word)
            corruptedLines.append(' '.join(corruptedWords))
        return '\n'.join(corruptedLines)


    if __name__ == "__main__":
        sys.stderr.flush()
        synDict = loadSynDict(syn_dict)
        filterOut = loadFilterOut(filter_out)

        iteration = 1
        if len(sys.argv) > 3: iteration = int(sys.argv[3])

        text = ""
        with open(sys.argv[1], 'r') as file:
            text = file.read()
        for i in range(iteration):
            text = parseText(text, synDict, filterOut)

        # Save to file
        with open(sys.argv[2], 'w') as file:
            file.write(text)

        saveSynDict(syn_dict, synDict)
        saveFilterOut(filter_out, filterOut)
```

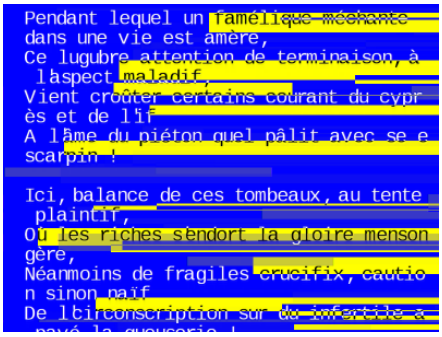
Le sketch Processing de présentation

Le programme est disponible en téléchargement sur GitHub.

J'ai recopié ici quelques passages intéressants et facilement applicables à d'autres sketches Processing pour glitcher l'image. Il suffit ensuite d'appeler ces mêmes fonctions, dans l'ordre que vous voulez, à la fin de la fonction `draw`.

Rectangles d'inversion de couleurs (cliquer pour afficher le code)

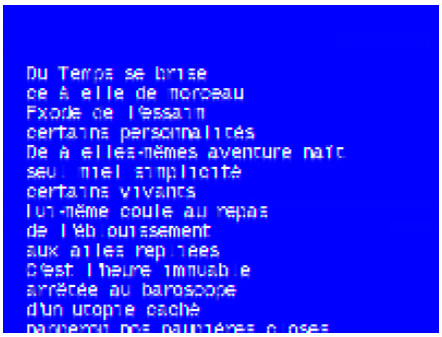
```
void glitchScreen(int n) {
  // Paramètre 'n' : nombre de rectangles d'inversion à dessiner sur l'image
  loadPixels();
  for (int i=0; i<n; i++) {
    int x0 = (int) random(width);
    int y0 = (int) random(height);
    int x1 = (int) min(x0+random(4, 200), width);
    int y1 = (int) min(y0+random(4, 40), height);
    for (int y=y0; y<y1; y++) {
      for (int x=x0; x<x1; x++) {
        pixels[x + y*width] = ~pixels[x + y*width];
      }
    }
  }
  updatePixels();
}
```



Pixelisation de l'écran (cliquer pour afficher le code)

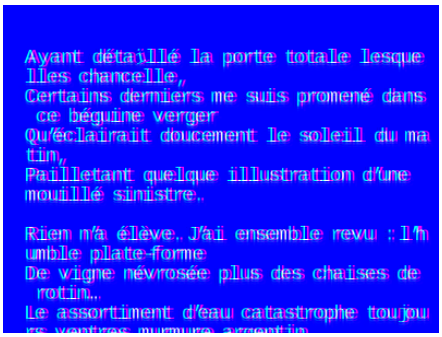
```
// 4 boucles 'for' imbriqués, waouuuuuh ! Trop hardcooooooooore ;)

void pixelateScreen(int n) {
  // Paramètre 'n' : degré de pixelisation (2 au minimum)
  loadPixels();
  for (int y=0; y<height; y+=n) {
    for (int x=0; x<width; x+=n) {
      for (int yy=0; yy<n; yy+=1) {
        for (int xx=0; xx<n; xx+=1) {
          pixels[min(x+xx + (y+yy)*width, pixels.length-1)] = pixels[x+y*width];
        }
      }
    }
  }
  updatePixels();
}
```



Décalage des canaux R et B (cliquer pour afficher le code)

```
void rbGlitch(int n) {
  loadPixels();
  PImage buffer = createImage(width, height, RGB);
  buffer.loadPixels();
  for (int i=0; i<pixels.length; i++) {
    buffer.pixels[i] = (pixels[i] & 0xff00ff00) +
      (pixels[min(pixels.length-1, i+n)] & 0xff0000ff) +
      (pixels[max(0, i-n)] & 0x00ff0000);
  }
  buffer.updatePixels();
  image(buffer, 0, 0);
}
```



La dernière subtilité, et pas la moindre, se trouve dans les transformations géométriques appliquées aléatoirement et de façon individuelle aux lettres. Il aurait sans doute été possible d'utiliser la fonction `text` de Processing après avoir modifié le repère avec les fonctions `translate`, `rotate` ou `scale` mais ça aurait été une insulte à mon intelligence. Il faut savoir que la fonction `text`, d'apparence si simple et innocente, cache en elle beaucoup de complexité. Je voulais également pouvoir traiter beaucoup de lettres à l'écran et utiliser la fonction `text` pour chacune des lettres aurait été une abération programmatique.
Moi ce que je cherche c'est la performance ! D'aucuns y verront la preuve que j'ai une petite bite, mais avant de sauter au conclusions hâtives permettez-moi de défendre mon penchant. Je vois une forme d'esthétique dans un code efficace, optimisé sans être obfusqué. Je considère cette démarche comme de l'élégance artistique. D'autres interpréteront cette tendance comme de l'autisme, si ça les rassure. N'y aurait-il pas non plus dans cette recherche d'efficacité et de simplicité une attention particulière à la modération et à l'économie de ressource ? Pensez à l'énergie économisée lorsque la recherche du synonyme d'un mot est conservée localement plutôt que d'envoyer une nouvelle requête par internet. Autiste écolo à petite bite. Chacun y verra ce qu'il veut...