Rapport PSI

LE YHUELIC Noé

LIPANI BIANCHI Diego

LE NOAN Maël

Concernant la structure de nos données, nous nous sommes aidés de 5 classes que nous avons créées :

* La classe MyImage, qui nous était imposée, pour lire une image et convertir cette image en une instance de cette classe. Nous pouvons grâce à elle également faire le chemin inverse et transformer une instance en fichier binaire au format .bmp ainsi que convertir une séquence d’octets au format little endian en entiers et inversement.

Nous pouvons également effectuer une rotation et agrandissement d’une image. Pour ce faire, nous avons dû déterminer les nouvelles dimensions de la nouvelle image à l’aide de la trigonométrie pour la rotation et en multipliant par le coefficient pour l’agrandissement puis la méthode était de partir de notre nouvelle image, de chercher son “antécédent” dans la matrice initiale puis de regarder si cet antécédent appartenait bien à l’image initiale, si c’était le cas alors nous gardions ce pixel sinon nous le mettions en noir. Puis, nous pouvons appliquer des filtres sur les images comme le repoussage, le renforcement des bords et leur détection, ainsi que rendre l’image floue. Pour cela nous faisons un produit de convolution entre chaque case de notre matrice de pixels et un noyau, qui est une matrice 3x3 déjà définie. Ce produit effectue une sorte de moyenne des valeurs de la matrice 3x3 ayant le pixel en question comme centre. Si jamais une case de cette matrice 3x3 dépasse de la matrice de pixels de base, la valeur 0 est attribué au pixel.

Nous disposons aussi dans cette classe le code permettant de créer des fractales de Julia.

De plus, nous avons un code permettant de coder et décoder une image. Pour cela, il a fallu récupérer les bits de poids fort de l’image “cachante” et ceux de l’image cachée. Il fallait bien évidemment que l’image cachée soit plus petite. Si ce n’était pas le cas, nous agrandissions l’image cachante. Une fois les bits de poids forts récupérées, nous remplissions notre nouvelle matrice de pixel en mettant les bits de poids forts de l’image cachante en poids fort et les bits de poids forts de l’image cachée en poids faible. Pour décoder l’image, il suffisait de récupérer tous les bits puis de mettre les bits de poids forts dans une nouvelle matrice et en mettant les bits de poids faible à 0. On fait de même pour les bits de poids faible en les mettant en poids fort dans une nouvelle matrice. Ainsi, nous pouvons récupérer nos 2 images initiales en “perdant” très peu de couleurs par rapport au début.

* Nous faisons en sorte d’avoir des fichiers composés de pixels, ainsi nous avons créé la classe Pixel : celle-ci permet de travailler plus simplement avec les images, comme ça au lieu de travailler avec des tableaux de bytes, nous travaillons avec des matrices de pixels, plus significatives dans le traitement d’images.
* Pour la création de fractales nous avons dû créer une classe Complex, qui permet de créer facilement des nombres complexes de manière à simplifier les calculs pour les fractales. On peut ainsi aisément manier les modules, multiplications et additions de complexes.
* Classe Huffman : Nous avons dans un premier temps créer une liste de Nœud, un nœud à cette étape étant composé des fréquences de chaque Pixel, et du Pixel en question puis nous trions selon les fréquences cette liste. Ensuite nous créons notre arbre d’Huffman en utilisant le second constructeur de la classe nœud pour créer des nœuds sans Pixel mais avec un nœud gauche et un nœud droit. Nous prenons les 2 nœuds avec la fréquence la plus faible, créons le nouveau nœud et l’insérons en le triant directement dans notre liste. Au final il nous reste seulement un seul nœud qui contiendra tous les autres (sa fréquence sera donc égale aux nombres totaux de Pixel dans la matrice de base). Suite à cela nous avons créé une fonction qui va créer un dictionnaire contenant les pixels et des bits associés. Nous trouvons ces bits en considérant que si nous allons à gauche dans l’arbre le premier bit sera 0 sinon il sera 1. En effet nous avons créé 2 fonctions compressé et décompressé qui vont permettre de finir d’appliquer l’algorithme d’Huffman

Partie innovation :

Nous avons décidé de coder un jeu de couleur dont le but est de repérer parmi 9 pixels de même couleur lequel est légèrement différent des autres. Pour ce faire, nous avons créé une nouvelle fonction dans MyImage. Dans celle-ci nous avons généré de façon aléatoire une matrice RGB de 9 pixels avec 8 pixels de même couleur et un neuvième de couleur légèrement différente qui est placé de manière aléatoire dans la matrice. Puis, nous avons agrandi notre instance à l’aide de la fonction agrandissement que nous avons défini précédemment car il était sinon trop difficile de distinguer 9 pixels sur une image non agrandie.

Par la suite, nous avons utilisé notre fonction from\_image\_to\_file afin de transformer notre instance MyImage en fichier binaire. Nous affichons l’image et l’utilisateur doit trouver le pixel différent en écrivant sur la console sa position dans la matrice en partant du principe que la position 1 correspond au pixel en bas à gauche, puis la position 2 en bas au milieu etc.. jusqu'à la position 9 en haut à droite. Si l’utilisateur trouve le bon pixel, alors un message de réussite s’affiche avec la console colorée en vert et une nouvelle image apparait jusqu’à que le jeu soit fini. En revanche, s’il se trompe, le jeu s’arrête et la console devient rouge durant quelques secondes.