

Rapport du projet 2 : ULBMP

LI Min-Tchun

12/05/2024



Contents

1	Introduction	3
2	Méthodes	3
2.1	Pixel et Image	3
2.2	Encoder	4
2.3	Decoder	6
3	Résultats	6
4	ULBMP 3.0	6
5	ULBMP 4.0	6
6	Conclusion	6

1 Introduction

Pour le deuxième projet du cours INFO-F-106, nous avons dû implémenter un compresseur d'image appelé ULBMP, basé sur le format BMP, développé par Microsoft dans les années 80.

Afin de pouvoir venir à bout de ce projet, il était demandé d'importer le module *Pyside6* afin de pouvoir créer une interface graphique pour l'utilisateur. Il était donc important de lire la documentation de ce module afin de pouvoir l'utiliser correctement.

De plus, il est fondamental de maîtriser la manipulation de fichier, de bits (notamment le shifting et le masking), ainsi que la compréhension de la représentation binaire et hexadécimale.

En effet, comme nous devons utiliser des pixels, il est important de comprendre que ces derniers sont composées de 3 valeurs : Rouge, Vert, Bleu (RVB ou RGB en anglais). Chacune de ces valeurs est codée sur 8 bits, ce qui signifie que chaque valeur peut prendre $2^8 = 256$ valeurs différentes.

Il était donc utile d'utiliser le programme **hexdump** afin de pouvoir visualiser les données hexadécimale.

L'un des objectifs de ce projet est de pouvoir compresser une image. En effet, ce n'est pas efficace de stocker une image de manière brut car elle prendrait trop de place dans le stockage. Prenons l'exemple d'une image de dimension 360×480 . Si chaque pixel est codé sur 3 bytes (un byte pour chaque couleur), cela signifie que l'image prendrait $360 \times 480 \times 3 = 518400$ bytes, soit 518.4 kilo-octets. Nous pourrions se poser la question suivante : comment pouvons-nous réduire la taille de l'image tout en gardant une qualité acceptable ?

Afin de résoudre ce problème, on va procéder comme suit, nous allons créer une classe *Decoder* et *Encoder*.

Le premier permettra, à partir d'un chemin d'accès du fichier, de lire chaque bytes et de retourner une *image* (nous définirons plus tard plus précisément ce que c'est).

Le deuxième permettra, à partir d'une *image*, d'écrire dans un fichier les bytes correspondants aux valeurs RGB de chaque pixel.

Il est important de notifier qu'il y a plusieurs version du compresseur d'image. Cela signifie que le header (ce qui se trouve à chaque début de fichier pour reconnaître le format du fichier) doit être adapté à chaque version.

2 Méthodes

Comme cité dans l'introduction, nous allons devoir implémenter deux classes : *Decoder* et *Encoder*. Mais avant ça, il va falloir définir les classes *Pixel* et *Image*.

2.1 Pixel et Image

La classe *Pixel* possède les attributs suivants : r, g, b (ces derniers correspondent aux valeurs RGB d'un pixel). Le constructeur *Image* prend en paramètre la hauteur, la largeur de l'image, ainsi qu'une liste de taille *hauteur* \times *largeur* comprenant des instances de la classe *Pixel*. Certaines fonctions spécifiques à la construction de classes doivent être créées pour faciliter la manipulation des pixels et des images. Par exemple, la fonction `__getitem__` de la classe *Image* permet de récupérer un pixel à partir de ses coordonnées (x, y).

```
def __getitem__(self, pos: tuple[int, int]):
    x = pos[0]
    y = pos[1]
    if not (x <= self.w and y <= self.h):
        raise IndexError("Not enough pixels.")
    return self.img[y * self.w + x]
```

Ou bien la fonction `__setitem__` qui permet de modifier un pixel à partir de ses coordonnées (x, y).

```
def __setitem__(self, pos: tuple[int, int], pix):
    x = pos[0]
    y = pos[1]
    if x < self.w or y < self.h:
        self.img[x + y * self.w] = pix
    else:
        raise IndexError("Invalid position.")
```

Par ailleurs, rappelons nous que chaque couleur du pixel est encodée sur 8 bits, ce qui signifie qu'il est impossible d'avoir des valeurs négatives ou supérieures à 255. De plus, une image ne peut pas posséder une quantité de pixel supérieure à la dimension de l'image. Par exemple, si une image a une hauteur de 5 pixels et une largeur de 3 pixels, il est impossible d'accéder au pixel d'indice (6, 2) car il n'existe pas.

Voilà pourquoi les constructeurs des classes *Pixel* et *Image* sont définis comme suit :

Pour la classe *Pixel* :

```
def __init__(self, r, g, b):
    self.red = r
    self.green = g
    self.blue = b
    if self.red < 0 or self.green < 0 or self.blue < 0:
        raise ValueError("RGB values must be positive.")
    self.pixel = (self.red, self.green, self.blue)
```

Pour la classe *Image* :

```
def __init__(self, width: int, height: int, pixels):
    self.w = width
    self.h = height
    self.img = pixels
    if self.w * self.h != len(self.img):
        raise ValueError("Quantity of pixels doesn't match the image's dimension.")
    for i in pixels:
        if not isinstance(i, Pixel):
            raise ValueError("List's elements aren't Pixel instances.")
```

2.2 Encoder

La classe *Encoder* possède les attributs suivants : une instance de la classe *Image*, la version du compresseur que nous voulons utiliser (mis par défaut à 1), et `**kwargs`. Ce dernier permet de passer des arguments optionnels, tels que la profondeur et une valeur booléenne déterminant l'utilisation de l'encodage RLE (passer à True si nous souhaitons utiliser la encodage RLE, False sinon).

La première version de l'*Encoder* permettait seulement d'analyser les bits un à un et ensuite l'écrire dans un fichier. Cependant, cette méthode n'est pas optimale car elle ne permet pas de compresser l'image. C'est pourquoi, il est important de créer une deuxième version de l'*Encoder* qui permettra de compresser l'image.

Nous avons cité précédemment la notion de encodage RLE. Ce dernier consiste à remplacer une suite de bits identiques par un seul bit qui correspondra au nombre de répétitions du bit. Dans notre cas, si nous avons une suite de 10 pixels identiques, nous utiliserons un byte qui correspondra à 10. Ainsi, au lieu d'encoder les pixels sur 3 bytes, nous les encoderons sur 4 bytes dont le premier représentera le nombre de répétitions des bits suivants.

Quant à la profondeur, elle correspond au nombre de bits utilisés pour encoder chaque couleur d'un pixel. Par exemple, si la profondeur est de 4, cela signifie que chaque couleur d'un pixel est encodée sur 4 bits. Cette méthode d'encodage sera utilisée pour la version 3.0 du compresseur.

Nous avons également dû implémenter une quatrième version de l'*Encoder*. L'idée est de regrouper les pixels en fonction de leur différence de couleur. Si par exemple, soit deux pixels : $P_1 = (R_1, G_1, B_1)$ et $P_2 = (R_2, G_2, B_2)$, la différence de couleur entre ces deux pixels est définie comme suit :

$$\begin{aligned}\Delta_R &= R_2 - R_1 \\ \Delta_G &= G_2 - G_1 \\ \Delta_B &= B_2 - B_1\end{aligned}$$

Ainsi, en fonction des intervalles de valeurs de Δ_R , Δ_G et Δ_B , nous allons regrouper les pixels en fonction de leur différence de couleur. Plus la différence sera grande, plus on aura besoin de byte pour encoder les pixels. En effet, si la différence n'est pas grande (c'est-à-dire entre -2 et 1), la couleur des pixels sont assez proche pour qu'on ne voit pas la différence à l'oeil nu. Ainsi, on peut se permettre de les encoder sur moins de bytes.

Comme cité précédemment, les valeurs d'un pixel ne peuvent pas être négatives. Il va falloir décaler les valeurs de différences car il est possible que ces dernières soient négatives. De plus, le calcul de différence se fera toujours à partir du pixel actif et le pixel précédent.

Comme il fallait implémenter plusieurs versions de l'*Encoder*, il était important de créer une méthode **get_header** qui permet de retourner une suite de bytes correspondant au header contenant la constante ULBMP (c'est ce qui permet d'identifier nos sommes en train d'analyser le format correct du fichier), la hauteur, la largeur, la taille du header (mis par défaut à 12 car elle varie seulement pour la version 3) et la version. En fonction de la valeur du paramètre *version*, nous ajouterons des informations supplémentaires dont la profondeur et l'encodage RLE. À ceux-là s'ajoute la palette, utilisée pour la version 3.0.

La palette est une liste contenant seulement les couleurs se trouvant sur l'image. Ainsi, nous stockerons les couleurs et leurs indices dans un dictionnaire pour pouvoir faire appel à ces derniers plus rapidement.

```

1 list_pixels = []
2 for i in pixels:
3     list_pixels.append((int.to_bytes(pixel.get_red(), byteorder='little'),
4                               int.to_bytes(pixel.get_green(), byteorder='little'),
5                               int.to_bytes(pixel.get_blue(), byteorder='little')))
6 padding = list(set(list_pixels))
7 grouped_pixels = [b''.join(element) for element in list_pixels]
8 result = [b''.join(element) for element in padding]
9 color_to_index = {color: index for index, color in enumerate(result)}
10
11 indices = []
12 for pixel in grouped_pixels:
13     indices.append(color_to_index[pixel])
14
15 bit_string = ''.join(str(bit) for bit in indices)
16
17 if len(bit_string) != 8:
18     shift = 8 - len(bit_string)
19     shifter = '0' * shift
20     shifted_bit_string = bit_string + shifter
21 decimal_value = int(shifted_bit_string, 2)

```

La variable `pixels` est obtenu grace a la methode de la classe `Image` `get_pixels` qui retourne l'attribut `pixels`

À la ligne 6, les fonctions `list` et `set` permettent de supprimer les doublons car c'est possible d'avoir plusieurs fois la meme couleur

Parcourt la liste de pixels et trouver les indices correspondants dans la palette Convertit les bits en int en fonction de la profondeur

des couleurs utilisé des bytes pour encoder les indices des couleurs de la palette. Par exemple, plutôt que d'encoder 2^k . Cela permet de réduire la taille de l'image car on n'a plus besoin de stocker les valeurs RGB de chaque pixel.

2.3 Decoder

La classe *Decoder* se repose sur la méthode statique `load_from` qui prend en paramètre le chemin d'accès du fichier à lire. Cette méthode permet de lire les bytes du fichier et de les convertir en une instance de la classe *Image*. Cette fonction sera utile lorsque l'utilisateur voudra ouvrir une image.

3 Résultats

Naturellement, il est intéressant de comparer les différentes versions du compresseur. Pour ce faire, nous avons utilisé une image de dimension 360×480 pixels. Nous avons ensuite compressé cette image en utilisant les différentes versions du compresseur. Et voici les résultats obtenus :

4 ULBMP 3.0

Ceci est le chapitre 3.

5 ULBMP 4.0

Ceci est le chapitre 4.

6 Conclusion

Ceci est une conclusion.