Info-F-106 : Projet d'Informatique
Projet 2 – Représentation et compression d'images

Anthony Cnudde Gwenaël Joret Tom Lenaerts Robin Petit
Loan Sens Cédric Simar

version du 6 février 2024

Dans ce second projet, il vous est demandé d'écrire un programme en charge de lire, d'écrire, d'afficher et de compresser des images.

Contexte

Afficher des images ou des vidéos est devenu une partie intégrante de l'utilisation moderne de l'informatique. En effet, que ce soit quand vous regardez une série sur Netflix, quand vous *scrollez* votre feed instagram, ou même quand vous ouvrez cet énoncé, il y a des images de partout !

Rappelons tout de même que ça n'a pas toujours été le cas : il fut un temps (pas si lointain) où les ordinateurs tournaient uniquement en terminal et où il n'y avait pas d'interfaces graphiques. Lors de l'arrivée de ces dernières, il a fallu créer des *standards* pour pouvoir représenter des images dans des fichiers. En effet, définir un standard permet d'utiliser un même fichier sur plusieurs ordinateurs avec des systèmes d'exploitation différents. Un format phare et classique a été développé par Microsoft dans les années 1980 : le format *BMP* (pour *BitMap Picture*) et permettait de représenter une image de dimension arbitraire.

Cependant, un problème est très vite apparu : représenter des images de manière brute est particulièrement inefficace en terme d'utilisation de la mémoire de stockage. En effet, une simple image de résolution 640x480 (donc de 640 pixels de long et 480 de hauteur) où chaque pixel est représenté de manière classique en RGB demande $640 \times 480 \times 3 = 921\,600$ bytes, à savoir près d'un MB ! Une image d'1MB peut ne pas vous paraître surprenant et encore moins problématique, mais rappelons tout de même que les disquettes permettant 1.44MB de stockage n'ont été introduites qu'en 1987...

Il a donc très vite été nécessaire de *compresser* les images (entre autres) afin de ne pas gaspiller de stockage. Il existe plein d'algorithmes de compression différents, et donc il existe plein de formats d'images. Vous connaissez très certainement les formats `GIF`, `PNG`, `JPG` car ce sont les plus connus, et bien que tous standardisés il y a plus de 30 ans, ils restent extrêmement importants et utilisés à ce jour.

Attention tout de même : s'il existe plusieurs formats différents, il faut donner aux programmes manipulant ces images une manière d'identifier le format d'un fichier qui lui serait donné. Nous pouvons utiliser une extension pour cela (et donc avoir `image.bmp` d'un côté et `image.png` d'un autre), mais en réalité les fichiers commencent par un *header*, *i.e.* un groupe de bytes défini par le standard permettant de connaître le format du fichier, la version utilisée, les options nécessaires, etc.

Les formats bien connus mentionnés ci-dessus existent à ce jour dans des dizaines de versions différentes, ont plein de particularités et seraient très pénibles à implémenter dans ce projet. Pour cette raison, nous allons ici définir un nouveau type de fichier : ULBMP (ou ULB-bitMaP), inspiré des premières versions du format BMP.

Remarque : il existe deux familles d'algorithmes de compression : les algorithmes de compression *à perte* (ou *lossy compression* en anglais) et les algorithmes de compression *sans perte* (ou *lossless compression* en anglais). Les premiers reposent sur l'idée que pour diminuer la taille du fichier, il est possible de *retirer* de l'information peu importante pour son intégrité. En particulier le format `JPG` utilise un paramètre Q de qualité (entre 1 et 100) qui détermine la proportion d'*information* qui doit être gardée dans le fichier compressé, et évacue l'information restante. Il est donc impossible de reconstruire *exactement* l'image originale puisqu'une partie de l'information est manquante, mais ces algorithmes cherchent à ce que l'information retirée ne concerne que des *détails* de l'image afin de conserver une qualité bonne voire correcte, même à de haut taux de compression. Les algorithmes de compression sans perte quant à eux fonctionnent, comme leur nom l'indique, sans perdre d'information, et uniquement en écrivant l'information de manière plus intéressante. Par exemple le format `GIF` utilise une variante de l'algorithme LZW qui utilise un *dictionnaire* pour reconnaître des suites de bytes déjà vus et les encoder sur très peu de bits. Une image compressée sans perte peut donc toujours être reconstruite exactement et n'est associée à aucune perte de qualité. En fonction des applications, une compression à perte peut être acceptable (comme dans le cas d'une vidéo en 1920x1080 en 60FPS où la plupart des détails est totalement manquée à l'œil nu) ou non (comme quand vous encodez votre projet dans un fichier `ZIP` avant de le remettre sur l'UV). Nous ne considérerons ici que de la compression sans perte car les outils mathématiques dans la compression à perte ont tendance à être plus compliqués à comprendre.

Phase 1 — Le format ULBMP

Nous allons définir la standardisation du format ULBMP dans cette section. La première phase de ce projet consistera en l'écriture d'un code permettant de lire et d'écrire un fichier respectant ce format, ou tout du moins la première version de ce format.

Le format ULBMP est un format binaire : nous nous intéresserons à la valeur de chaque byte individuellement, le tout sans nous restreindre aux caractères affichables en ASCII. Afin de ne pas confondre les nombres avec les caractères '0', '1', etc. (de code 48-57 en ASCII), nous donnerons toutes les valeurs en hexadécimal, en les préfixant de `0x` pour l'expliciter. Certaines valeurs seront également données en binaire et seront alors préfixées de `0b` pour l'expliciter. Puisqu'un byte peut stocker une valeur entre 0 (à savoir `0x00`) et 255 (à savoir `0xff`), tous les bytes seront donc écrits avec deux caractères hexadécimaux. `0x0c` correspond donc à `0xc`, *i.e.* la valeur 12. Dans ce document, les suites de bytes seront coupées de manière à afficher au plus 16 bytes par ligne. C'est bien une convention choisie ici pour ne pas dépasser la largeur d'une page A4 et ne correspond en rien au caractère `'\n'` (`0x0a` en ASCII).

Un fichier au format ULBMP doit commencer par les bytes suivants : `0x55 0x4c 0x42 0x4d 0x50` correspondant aux lettres composant 'ULBMP' en ASCII. Le 6e byte est un entier (donc entre 0 et $2^8 - 1 = 255$) définissant la version du standard respectée par le fichier en question. Les septième et huitième bytes définissent un entier encodé sur deux bytes (donc entre 0 et $2^{16} - 1 = 65\,535$) qui correspond au nombre de bytes total du header (attention, cela comprend les 8 bytes définis jusqu'ici !)

Les bytes 9 et 10 définissent un entier encodé sur deux bytes correspondant à la *largeur* de

l'image ; et les bytes 11 et 12 définissent un entier sur deux bytes correspondant à la hauteur de l'image.

Attention : tous les nombres entiers sont encodés en *little-endian*, c'est-à-dire que les bytes de poids faibles sont écrits avant les bytes de poids plus forts. Dès lors la suite de bytes `0x10 0x3a` est à lire comme le nombre `0x3a10` et non comme le nombre `0x103a` !

La version 1.0 de ce format ne permet pas de stocker d'informations supplémentaires dans le header. Dès lors le header d'une image au format ULBMP 1.0 représentant un image 640x480 doit commencer par les 12 bytes suivants :

```
0x55 0x4c 0x42 0x4d 0x50 0x01 0x0c 0x00 0x80 0x02 0xe0 0x01
```

Chaque pixel d'une image est une combinaison de rouge, de vert et de bleu (d'où la dénomination RGB pour *Red, Green and Blue*). Ils sont dès lors représentés par des triplets (r, g, b) d'entiers entre 0 et 255 représentant l'intensité dans chaque canal. La couleur noire correspond au triplet $(0, 0, 0)$ puisqu'on y met ni rouge, ni vert, ni bleu. La couleur blanche correspond au triplet $(255, 255, 255)$ puisque les trois canaux sont saturés. La couleur *rouge* correspond au triplet $(255, 0, 0)$, etc.

Dès lors, après ce header, tous les pixels doivent être encodés (d'abord la première ligne de gauche à droite, ensuite la deuxième ligne de gauche à droite, etc.) sur 3 bytes chacun : d'abord l'intensité de rouge, puis l'intensité de vert, et finalement l'intensité de bleu.

Le fichier suivant correspond donc à une image carrée de 2 pixels sur 2 dont la première diagonale est en noir et la deuxième diagonale est en blanc :

```
0x55 0x4c 0x42 0x4d 0x50 0x01 0x0c 0x00 0x02 0x00 0x02 0x00 0x00 0x00 0x00 0xff
0xff 0xff 0x00 0x00 0x00 0xff 0xff 0xff
```

Ce fichier est disponible sur l'UV et s'appelle `squares.ulbmp`. Son contenu est donné de manière plus explicite dans le tableau 1.

Décalage	Valeur (hex)	interprétation
`0x00`	`0x55 0x4c 0x42 0x4d 0x50`	ULBMP en ASCII
`0x05`	`0x01`	Version 1 du format
`0x06`	`0x0c 0x00`	Le header fait 12 bytes
`0x08`	`0x02 0x00`	L'image fait 2 pixels de large
`0x0a`	`0x02 0x00`	L'image fait 2 pixels de haut
`0x0c`	`0x00 0x00 0x00`	Le premier pixel est noir (R=G=B=0)
`0x0f`	`0xff 0xff 0xff`	Le second pixel est blanc (R=G=B=255)
`0x12`	`0x00 0x00 0x00`	Le troisième pixel est noir
`0x15`	`0xff 0xff 0xff`	Le quatrième pixel est blanc

TABLE 1 – Contenu du fichier `squares.ulbmp`. La délimitation entre le header et le contenu des pixels (au décalage `0x0c`) est marqué par une séparation dans le tableau.

Code à écrire

Il vous est demandé d'écrire les classes suivantes :
— `Pixel` dans un fichier `pixel.py` ;

— `Image` dans un fichier `image.py` ;
— `Encoder` et `Decoder` dans un fichier `encoding.py`.

La classe `Pixel` représente, comme son nom l'indique, un pixel et doit contenir trois entiers entre 0 et 255 : l'intensité des canaux rouge, vert et bleu. Ce type doit être immuable : les valeurs RGB ne peuvent pas être modifiées, mais elles doivent pouvoir être récupérées. Deux instances de `Pixel` doivent pour être comparées par l'opérateur `==`.

La classe `Image` doit contenir les méthodes suivantes :
— `__init__(self, width: int, height: int, pixels: list[Pixel])`, le constructeur qui prend en paramètres `width`, `height` représentant les dimensions de l'image et une liste de taille `width*height` comprenant des instances de la classe `Pixel` et représentant l'image ;
— `__getitem__(self, pos: tuple[int,int]) -> Pixel` pour surcharger l'opérateur `[]` en lecture ;
— `__setitem__(self, pos: tuple[int,int], pix: Pixel) -> None` pour surcharger ce même opérateur en écriture.
— `__eq__(self, other: 'Image') -> bool` pour surcharger l'opérateur d'égalité.

Les méthodes spéciales `__getitem__` et `__setitem__` doivent lancer une exception de type `IndexError` si `pos` n'est pas une position valide dans l'image.

La classe `Encoder` quant à elle doit contenir les méthodes suivantes :
— `__init__(self, img: Image)`, le constructeur qui prend en paramètre l'image à encoder ;
— `save_to(self, path: str) -> None`, la méthode qui va encoder l'image reçue et l'enregistrer au format ULBMP version 1.0 dans le fichier dont le chemin est donné en paramètre.

Finalement, la classe `Decoder` consiste en une unique méthode statique `load_from(path: str) -> Image` qui charge l'image encodée dans le fichier dont le chemin est donné en paramètre et renvoie cette image. S'il y a une erreur dans le format du fichier, cette fonction doit lancer une exception.

Bien sûr, nous vous invitons à écrire autant de fonctions/méthodes/classes que vous voulez pour écrire un code propre. Vous pouvez aussi ajouter des fonctionnalités non-demandées, mais pensez à bien les documenter et à ce qu'elles n'empiètent pas sur le comportement imposé !

Remarque : Vous pouvez utiliser le logiciel `hexdump` pour regarder facilement le contenu d'un fichier binaire. En particulier, si vous ouvrez un terminal dans le dossier qui contient `squares.ulbmp`, vous pouvez l'exécuter comme ceci :

```
$ hexdump -C squares.ulbmp
00000000  55 4c 42 4d 50 01 0c 00  02 00 02 00 00 00 00 ff  |ULBMP...........|
00000010  ff ff ff ff ff 00 00 00                           |........|
00000018
```

Le paramètre `-C` permet de lire un byte à la fois et de les afficher en hexadécimal (il n'y a juste pas le préfixe `0x`). Si vous ne voulez lire que les premiers bytes d'un fichier, vous pouvez le spécifier via le paramètre `-n <nb_bytes>`.

La colonne de gauche est le décalage (en bytes) depuis le début du fichier, donné en hexadécimal. On voit donc ici que chaque ligne affiche 16 bytes. La dernière colonne est une sorte d'affichage textuel : les bytes correspondant à des caractères ASCII affichables sont affichés, les autres sont représentés par des points.

Si une suite de 16 bytes se répète dans le fichier demandé, au lieu d'afficher plusieurs fois la même ligne, le logiciel `hexdump` écrit simplement `*` (une astérisque) sur une ligne. Par

exemple, s'il existe un fichier `zeros.bin` de 64 bytes, tous valant `0x00`, alors la commande `hexdump -C zeros.bin` affichera :

```
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000040
```

Phase 2 — Interface graphique et canevas d'affichage

Afin de vous permettre de voir le contenu des images (et puisque le format ULBMP n'est pas standard et donc pas géré par votre viewer d'images préféré), vous devez implémenter une petite interface graphique vous permettant d'ouvrir une image, de l'afficher et de l'enregistrer. Cette dernière fonctionnalité peut vous sembler inutile dans l'immédiat, mais dans les phases qui suivent, nous décrirons de nouvelles version du format ULBMP, dès lors vous pourrez charger une image depuis une certaine version et la sauvegarder dans une autre (en particulier en utilisant diverses formes de compression).

Pour votre interface graphique, vous devez *impérativement* utiliser le module `PySide6` qui est la dernière version du portage officiel de `Qt` (qui est une bibliothèque en C++) pour Python. Vous en trouverez la documentation ici.

Votre interface doit contenir :

— un bouton pour charger une image ;
— un canevas pour dessiner l'image ;
— un bouton pour enregistrer l'image actuellement ouverte.

Une proposition d'interface est donnée dans la figure 1. Le bouton pour enregistrer l'image est désactivé puisqu'aucune image n'est chargée au début.

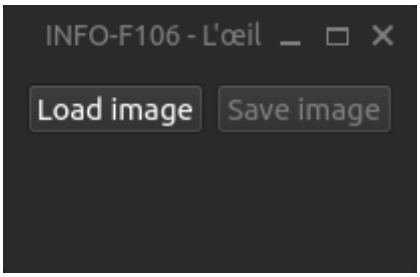

FIGURE 1 – Proposition d'interface au démarrage du programme.

Cliquer sur le bouton de chargement d'image doit ouvrir une boîte de dialogue de fichier (*cf.* `QFileDialog` dans la documentation) permettant le choix du fichier à ouvrir (attention : seuls les fichiers d'extension `.ulbmp` doivent pouvoir être ouverts). Si le fichier ouvert n'est pas valide (donc si `Encoder.load_from` ne peut s'exécuter correctement), votre code doit ouvrir une boîte de dialogue pour mentionner l'erreur (*cf.* `QErrorMessage` dans la documentation). Un exemple de tel problème (où le fichier demandé commence par 'ULBPM' au lieu de 'ULBMP') est donné dans la figure 2.

Si par contre le chargement se fait sans problème, alors l'image doit être affichée, la fenêtre doit être redimensionnée, et le bouton de sauvegarde doit être activé. Un exemple vous est donné dans la figure 3. Le fichier ouvert dans cet exemple est disponible sur l'UV et s'appelle `checkers.ulbmp`. C'est une image de résolution 640x480 encodée en ULBMP v1.0 contenant une alternance de carrés noirs et de carrés blancs (où chaque carré fait exactement 16x16 pixels).

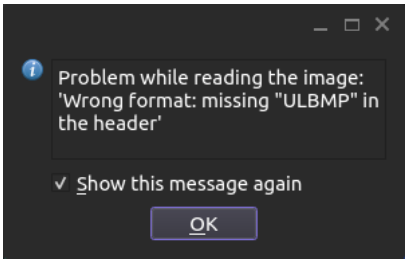

FIGURE 2 – Exemple de problème lors de l'ouverture d'un fichier.

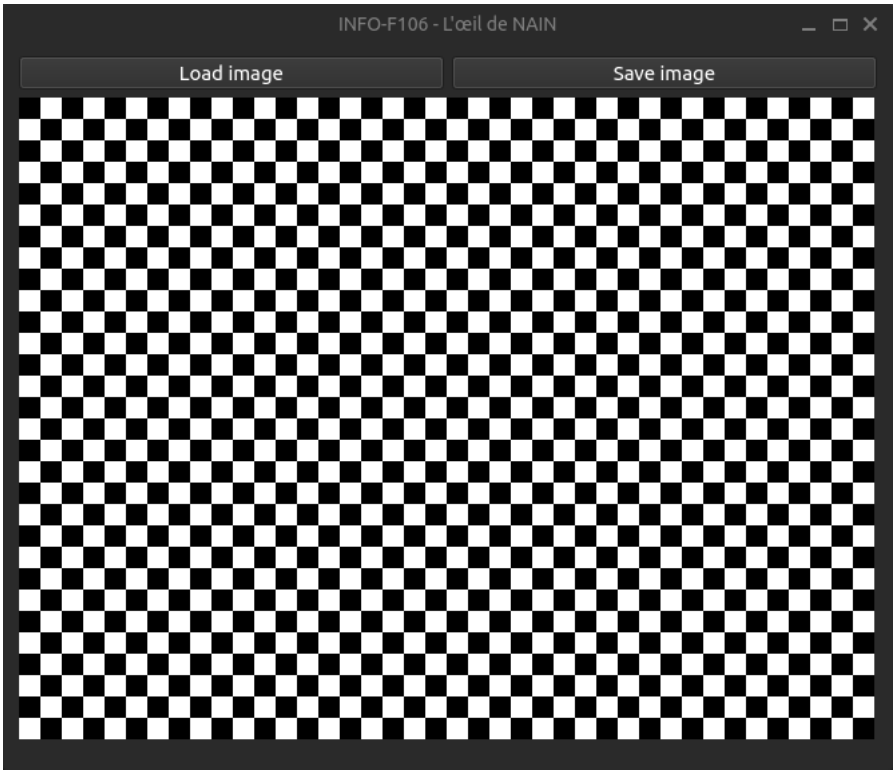

FIGURE 3 – Exemple du programme après l'ouverture de `checkers.ulbmp`.

Pour l'instant, nous laissons le bouton de sauvegarde inutilisé, nous y reviendrons lorsque nous aurons différentes version de ULBMP.

Code à écrire

Dans un fichier `window.py`, écrivez toutes les classes nécessaires pour votre interface graphique. Notez que vous pouvez utiliser `QtCreator` pour construire votre interface graphique (*cf.* la documentation).

Ajoutez un fichier `main.py` permettant de lancer votre programme en mode graphique.

Phase 3 — ULBMP 2.0 : la compression RLE

Repensons au fichier `checkers.ulbmp`. Nous savons que chaque ligne de l'image est décomposée en blocs de 16 pixels identiques. Il est facile de se convaincre qu'il y a donc énormément de redondance et qu'il y a donc moyen de compresser ce fichier. L'encodage RLE (pour *Run Length Encoding*) est une méthode de compression qui utilise précisément le fait que des séquences de bytes apparaissent plusieurs fois à la suite. Au lieu d'encoder 16 fois consécutives le pixel *noir* (*i.e.* `0x00 0x00 0x00`), nous pouvons l'encoder une unique fois et préciser qu'il doit être lu 16 fois lors de la décompression.

La version 2.0 du format ULBMP fonctionne justement sur ce principe. Le header est le même que pour la version 1.0 (hormis le sixième byte qui doit maintenant être `0x02` puisqu'il encode la version utilisée), mais l'encodage des pixels se fait de manière différente. Il faut maintenant 4 bytes pour encoder une entrée (au lieu de 3), mais le premier byte est précisément le nombre (entre 1 et 255) de fois que le pixel encodé par les 3 bytes suivants doit être répété. En particulier le fichier `checkers2.ulmbp` (qui encode la même image que `checkers.ulbmp`, mais au format ULBMP 2.0) commencera par le header suivant :

```
0x55 0x4c 0x42 0x4d 0x50 0x02 0x0c 0x00 0x80 0x02 0xe0 0x01
```

et ensuite encodera la suite de bytes `0x10 0x00 0x00 0x00` pour désigner l'existence de 16 bytes noirs, puis la suite de bytes `0x10 0xff 0xff 0xff` pour désigner l'existence de 16 bytes blancs, etc. Attention : il y a un nombre pair de blocs par ligne, dès lors le premier bloc et le dernier bloc de chaque ligne sont de couleur différente. Cependant, la première rangée de bloc commence par noir pour finir sur blanc alors que la deuxième rangée commence par blanc pour finir sur noir. Lors de l'encodage RLE, les 16 derniers pixels blancs de la fin de la 16ème ligne (*i.e.* fin de la première rangée de blocs) et les 16 premiers pixels blancs du début de la 17ème ligne (*i.e.* début de la deuxième rangée de blocs) sont consécutifs. L'encodage de ces 32 pixels sera donc `0x20 0xff 0xff 0xff`. Une occurrence de 20 pixels consécutifs de même couleur arrive donc à la fin de chaque rangée, *i.e.* à la fin de toutes les lignes correspondant à un multiple de 16. La première est donc au byte 2 568 (`0xa08` en héxadécimal). Chaque bloc de 16x16 pixels (à part le premier de chaque ligne) demande donc $16 \times 4 = 64$ bytes pour être encodé. Le premier bloc de chaque ligne (sauf celui de la première ligne) demande quant à lui $15 \times 4 = 60$ bytes pour être encodé. L'image est de dimension 640x480 et contient donc 40 blocs par ligne et 30 blocs par colonne. La taille totale du fichier est donc 12 bytes pour le header, et $(1 + 30 \times 39) \times 64 + (30 - 1) \times 60 = 76\,684$ bytes pour encoder tous les pixels avec RLE, pour un total de 76 696 bytes.

Chaque bloc de 16x16 pixels demande donc $16 \times 4 = 64$ bytes. L'image est de dimension 640x480 et contient donc 40x30 tels blocs. En comptabilisant les 12 bytes de header et les $40 \times 30 \times 16 \times 4 =$

Une version plus explicite du contenu de ce fichier est donnée dans le tableau 2. Notez bien que 640 (en décimal) correspond à `0x280` (en hexadécimal) puisque $2 \times 16^2 + 8 \times 16^1 + 0 \times 16^0 = 512 + 128 = 640$. De même `0x1e0` (en hexadécimal) correspond à 480 en décimal puisque $1 \times 16^2 + 14 \times 16^1 + 0 \times 16^0 = 256 + 224 = 480$. Le dernier bloc d'encodage RLE du fichier est bien au byte 76 692 (ou `0x12b94` en hexadécimal).

Le *taux de compression* ici est donc ≈ 12 puisque chaque suite de 16 pixels monochromes passe de $16 \times 3 = 48$ bytes dans la version 1 à $1 + 3 = 4$ bytes dans la version 2 (sauf pour les jonctions entre rangées). L'encodage des pixels prend donc de l'ordre de douze fois moins de place ici. Bien sûr ce taux de compression est uniquement dû au fait que les pixels apparaissent toujours par paquets de 16. S'ils apparaissaient systématiquement par paquets de 255, alors le taux de compression serait $\approx \frac{3 \times 255}{4} = 191 + \frac{1}{4}$ (*cf.* les fichiers `lines1.ulbmp` et `lines2.ulbmp`).

Décalage	Valeur (hex)	interprétation
0x00	0x55 0x4c 0x42 0x4d 0x50	ULBMP en ASCII
0x05	0x02	Version 2 du format
0x06	0x0c 0x00	Le header fait 12 bytes
0x08	0x80 0x02	L'image fait 640 pixels de large
0x0a	0xe0 0x01	L'image fait 480 pixels de haut
0x0c	0x10 0x00 0x00 0x00	Les 16 premiers pixels sont noirs
0x10	0x10 0xff 0xff 0xff	Les 16 pixels suivants sont blancs
⋮		
0xa08	0x20 0xff 0xff 0xff	32 pixels blancs (transition lignes 16 et 17)
0xa0c	0x10 0x00 0x00 0x00	16 pixels noirs
⋮		
0x12b90	0x10 0xff 0xff 0xff	Les avant-derniers 16 pixels sont blancs
0x12b94	0x10 0x00 0x00 0x00	Les 16 derniers pixels sont noirs

TABLE 2 – Contenu du fichier `checkers2.ulbmp`, encodé au format ULBMP 2.0.

Par contre notons bien que dans le pire des cas, l'image à encoder n'a jamais deux pixels identiques l'un à la suite de l'autre. Dans ce cas, au lieu de nécessiter 3 bytes par pixels, nous aurons besoin de 4 bytes par pixel, et donc le taux de compression sera $\approx \frac{3}{4}$, *i.e.* les fichiers seront de l'ordre de 33% plus grand après la compression...

Code à écrire

Adaptez la classe `Encoder` de manière à ce que son constructeur prenne un nouvel entier en paramètre : la version du format ULBMP à utiliser pour enregistrer l'image. Ce nouveau paramètre doit avoir 1 pour valeur par défaut afin de rester compatible avec la première phase. Vous devez également adapter les classes `Encoder` et `Decoder` pour pouvoir gérer la version 2 du format. Par exemple le code suivant doit pouvoir lire le fichier `lines1.ulbmp` (sans connaître sa version au préalable) et l'enregistrer dans le format ULBMP 2.0 :

```
lines = Decoder.load_from('lines1.ulbmp')
Encoder(lines, 2).save_to('lines_rle.ulbmp')
```

Adaptez également votre interface graphique de manière à ce que lorsque le bouton de sauvegarde d'image est cliqué, la version du format dans laquelle encoder et le nom du fichier dans lequel écrire l'image soient demandés. Assurez-vous bien que la version soit sensée (*e.g.* ne permettez pas d'encoder en version ULBMP 126, puisque cette standardisation n'existe pas...)

Phase 4 — ULBMP 3.0 : Profondeur des pixels

Les images que nous avons vues jusqu'à présent étaient toutes en noir et blanc. Il y avait donc uniquement deux choix par couleur, autrement dit chaque pixel ne nécessitait qu'un unique bit d'encodage, et pourtant ils ont tous été encodés sur 3 bytes, à savoir 24 bits. Nous pouvons donc gagner un facteur 24 si on encode directement l'image en noir et blanc.

Cette notion peut se généraliser : si notre image contient au plus 2^k couleurs différentes, nous n'avons besoin que de k bits pour identifier ces couleurs de manière unique (et $k = 1$ dans le

cas des images en noir et blanc). On pourrait se dire naïvement qu'on n'a donc presque jamais besoin de 24 bits par pixel puisqu'une image de résolution 1920x1080 contient *uniquement* 2 073 600 pixels, ce qui reste ≈ 8 fois plus petit que 2^{24}. Faisons tout de même attention à une chose : si on n'utilise que 8 bits par pixel, on a droit à 256 couleurs, mais il faut une manière d'identifier quelles sont ces couleurs. En effet, il faut savoir à quelles couleurs RGB correspondent l'encodage 0, l'encodage 1, etc.. Pour cela, il faut ajouter cet encodage dans le header de l'image. Pour une profondeur de k bits par pixel, il faut donc $k \times WH$ bits pour encoder une image de résolution WxH, mais il faut également $2^k \times 24$ bits pour encoder l'encodage des couleurs. Cet encodage s'appelle la *palette* de l'image.

Pour cette raison, nous ne considérerons ici que les profondeurs suivantes : 1, 2, 4, 8, 24. Une profondeur de 24 correspond, comme précédemment, à un encodage classique RGB pour chaque pixel. Dans les autres cas, on aura besoin de moins de bits pour encoder chaque pixel, mais il faudra encoder le mapping dans le header. De plus, afin de ne pas avoir de problème d'alignement, nous ne considérons que les diviseurs de 8 pour les potentielles profondeurs. En procédant de la sorte, nous savons qu'en lisant un byte (donc 8 bits), nous avons toujours un nombre fixe de pixels lus.

La version 3 du format ULBMP doit gérer les profondeurs d'encodage mentionnées ci-dessus, et doit supporter l'encodage RLE sur les profondeurs 8 et 24 (RLE sur une profondeur < 8 correspondrait à répéter des séquences de pixel, ce qui aurait potentiellement un sens, mais que nous ne ferons pas ici afin de conserver une longueur raisonnable pour ce projet). Le header aura donc la forme suivante :

- la constante magique `0x55 0x4c 0x42 0x4d 0x50` (donc ULBMP en ASCII) sur 5 bytes ;
- la version (donc `0x03`) sur un byte ;
- la taille du header sur 2 bytes ;
- la largeur et la hauteur de l'image, chacune sur 2 bytes ;
- la profondeur d'encodage sur un byte (encodé en bbp, ou encore *bits per pixel*) ;
- 1 byte valant soit `0x01` si l'image est compressée avec RLE soit `0x00` si ce n'est pas le cas (attention : ce byte peut uniquement valoir `0x01` si la profondeur est 8 ou 24) ;
- la palette de couleurs.

Après ce header, les pixels sont encodés selon leur profondeur l'un après l'autre, potentiellement avec encodage RLE (pour les profondeurs 8 ou 24 bbp uniquement). Si la profondeur est 1, 2 ou 4, plusieurs pixels seront encodés sur chaque byte. L'ordre des bits dans chacun de ces bytes est le suivant : les bits de poids fort correspondent aux pixels *le plus à gauche*. Par exemple si la profondeur est 4 bbp et que les deux premiers pixels correspondent aux couleurs associées aux indices respectivement 7 et 3 de la palette, le byte à écrire pour écrire ces deux pixels consécutifs est `0x73`. Si $W \times H \times d$ (où W est la largeur de l'image, H est la hauteur de l'image et d est la profondeur (en bbp) de l'image) n'est pas un multiple de 8, le dernier byte doit obligatoirement être aligné à gauche. Par exemple si l'image ne contient que 3 pixels (tous différents), alors elle peut être encodée avec une profondeur de 2 bbp (puisque $2^2 = 4 > 3$). En supposant que le premier pixel correspond à l'indice 0 de la palette, que le second pixel correspond à la couleur d'indice 1, et que le troisième pixel correspond à la couleur d'indice 2, le byte à écrire est `0x14`. En effet ce dernier correspond à l'écriture binaire `0b00011000`, que l'on peut décomposer en `0b00 0b01 0b10 0b00` pour signaler que le premier pixel correspond à l'indice `0b00`, le second à la couleur `0b01`, et le troisième à la couleur `0b10`. Les deux derniers bits (`0b00`) servent uniquement de *pading* pour l'alignement, mais peuvent en réalité valoir n'importe quoi puisqu'ils ne seront jamais lus.

Notez que la palette n'existe que si la profondeur est ≤ 8. De plus, si l'image est encodée avec profondeur d et contient exactement C couleurs distinctes, il faut bien que $C \leq 2^d$. Dans ce cas, la palette sera encodée sur exactement $3 \times C$ bytes.

Nous voyons maintenant pourquoi la taille du header doit être encodée sur 2 bytes : dans le cas où la profondeur est 8, alors le header fait au plus 14+768 = 782 bytes, hors la plus grande valeur qu'il est possible d'encoder sur 1 byte est 256. Le fichier `checkers3.ulbmp` (qui encode la même image que `checkers.ulbmp` mais au format ULBMP 3) aura pour header :

```
0x55 0x4c 0x42 0x4d 0x50 0x03 0x14 0x00 0x80 0x02 0xe0 0x01 0x01 0x00 0x00 0x00
0x00 0xff 0xff 0xff
```

et pour contenu à la suite du header :

```
                    0x00 0x00 0xff 0xff 0x00 0x00 0xff 0xff 0x00 0x00 0xff 0xff
0x00 0x00 0xff 0xff ...
```

Une version plus explicite du contenu de ce fichier est donnée dans le tableau 3.

Décalage	Valeur (hex)	interprétation
0x00	0x55 0x4c 0x42 0x4d 0x50	ULBMP en ASCII
0x05	0x03	Version 3 du format
0x06	0x14 0x00	Le header fait 20 bytes
0x08	0x80 0x02	L'image fait 640 pixels de large
0x0a	0xe0 0x01	L'image fait 480 pixels de haut
0x0c	0x01	La profondeur est 1 bpp
0x0d	0x00	Pas d'encodage RLE
0x0e	0x00 0x00 0x00	La première couleur de la palette est *noir*
0x11	0xff 0xff 0xff	La deuxième couleur de la palette est *blanc*
0x14	0x00	8 pixels correspondent à la couleur 0 (donc *noir*)
0x15	0x00	8 pixels correspondent à la couleur 0 (donc *noir*)
0x16	0xff	8 pixels correspondent à la couleur 1 (donc *blanc*)
0x17	0xff	8 pixels correspondent à la couleur 1 (donc *blanc*)
⋮		
0x9612	0x00	8 pixels correspondent à la couleur 0 (donc *noir*)
0x9613	0x00	8 pixels correspondent à la couleur 0 (donc *noir*)

TABLE 3 – Contenu du fichier `checkers3.ulbmp`, encodé au format ULBMP 3.0.

Attention : rien ici ne nous oblige à définir noir comme étant la couleur 0 et blanc comme étant la couleur 1, le contraire aurait également été valide. Le seul élément important est que les valeurs encodées pour les pixels correspondent à l'ordre dans lequel la palette est définie.

Dans ce cas-ci, chaque ligne de 16 pixels monochromatiques nécessite uniquement 2 bytes, contre 4 dans le cas de ULBMP 2 (avec RLE et profondeur 24). C'est pourquoi le fichier `checkers3.ulbmp` est ≈ 2 fois plus petit que `checkers2.ulbmp`, et donc ≈ 24 fois plus petit que `checkers.ulbmp`.

Notez également que dans l'exemple donné ci-dessus, chaque byte encode 8 pixels consécutifs mais qui ont tous la même couleur, donc les bytes qui encodent l'image sont soit `0x00`, soit `0xff`. Cette situation apparaît du fait que les pixels existent par blocs dans l'image, mais tout autre motif est possible également.

Code à écrire

Vous devez adapter les classes `Encoder` et `Decoder` afin qu'elles gèrent le format ULBMP en version 3. Faites bien attention à factoriser votre code proprement et à éviter les ré-

pétitions. Le constructeur de la classe `Encoder` doit maintenant prendre `**kwargs` en paramètre pour pouvoir lui donner des paramètres supplémentaires. Sa signature est donc `__init__(self, image: Image, version: int=1, **kwargs)`. Plus précisément, il faut pouvoir préciser la profondeur et le fait de compresser avec RLE ou non.

Le code suivant doit donc lire une image (en noir et blanc) depuis `checkers.ulbmp` et l'enregistrer au format ULBMP 3 dans une image de profondeur 1 et sans RLE (de toute façon puisque RLE n'est disponible que pour les profondeurs 8 et 24) :

```
img = Decoder.load_from('checkers.ulbmp')
Encoder(img, 3, depth=1, rle=False).save_to('checkers3.ulbmp')
```

Dans le cas d'un encodage au format ULBMP 3, les paramètres `rle` et `depth` doivent *obligatoirement* être spécifiés, sans quoi une exception de type `ValueError` doit être lancée.

De plus, il vous faut adapter votre interface graphique de manière à pouvoir choisir la version 3 comme version d'encodage. Dans ce cas, il faut également que votre interface demande la profondeur et si l'encodage RLE est utilisé ou non (faites bien attention à ne pas permettre de donner des valeurs aberrantes, par exemple une profondeur de 12).

Sur votre interface, ajoutez également le nombre de couleurs présentes dans l'image chargée.

Notez que pour pouvoir identifier certains bits d'un byte, vous pouvez utiliser les opérateurs logiques `&`, `|`, `<<`, `>>`, etc. En particulier, identifier le bit de poids faible de l'entier `x` se fait via l'instruction `x&1` qui effectue un *bitwise AND* (ou un ET bit-à-bit). Si ce bit de poids faible est à 0, alors `x&1` sera évalué à 0 et si ce bit est à 1, alors `x&1` sera évalué à 1. Pour identifier le $k$ème bit de `x`, vous pouvez utiliser l'expression `((x & (1 << k))) >> k` (faites toujours bien attention à la précédence des différents opérateurs, vous pouvez en retrouver la description dans la documentation).

Phase 5 — ULBMP 4.0 : différence avec les pixels précédents

La version 4 du format ULBMP est inspirée du format QOI (*Quite Ok Image*). L'idée derrière cette version est la suivante : en moyenne, des pixels *proches* dans l'image sont également *proches* en terme de couleur RGB. En effet, une image contiendra certainement plusieurs transitions d'un élément à un autre qui peuvent s'accompagner de changements brusques dans les valeurs des pixels, mais une photo de nuage sera *globalement* blanche, et le ciel autour sera *globalement* bleu.

Dès lors nous voulons pouvoir encoder soit un pixel arbitraire, soit la différence par rapport au dernier pixel. Évidemment, puisque nous espérons que la majorité des pixels soient proches du pixel précédent, nous voulons encoder ces différences sur moins de 3 bytes afin d'avoir une compression efficace.

De plus, une compression RLE est rarement utile en pratique sur des images avec une profondeur de 24 bits par pixel puisque les pixels voisins sont rarement identiques. Le SIPI (*Signal and Image Processing Institute*) de l'Université de Caroline du Sud propose des images à utiliser pour tester des systèmes de traitement d'image (dont la compression fait partie). Nous utiliserons ici deux de ces images : *jelly beans (4.1.07)* et *house* (*cf.* la figure 4).

Le header de la version 4 est similaire à celui des versions 1 et 2, si ce n'est que la version encodée doit maintenant être `0x04`. Attention contrairement à la version 3 du format, il n'y a pas ici de notion de profondeur, donc pas de palette à encoder.

L'encodage des pixels se fera maintenant par *blocs*, et nous définissons trois types de blocs :

1. nouveau pixel (`ULBMP_NEW_PIXEL`) ;


FIGURE 4 – Les images standard *jelly beans (4.1.07)* (à gauche) et *house* (à droite).

2. pixel très proche du précédent (`ULBMP_SMALL_DIFF`) ;
3. pixel assez proche du précédent (`ULBMP_INTERMEDIATE_DIFF`) ;
4. pixel assez loin du précédent (`ULBMP_BIG_DIFF`) ;

Afin de savoir dans quel cas on se trouve lors de la compression, supposons que nous ayons deux pixels consécutifs $P_1 = (r_1, g_1, b_1)$ et $P_2 = (r_2, g_2, b_2)$ (où chacune de ces 6 valeurs est entre 0 et 255, bornes comprises). Nous notons :

$$\begin{aligned}\Delta_R &\coloneqq r_2 - r_1,\\ \Delta_G &\coloneqq g_2 - g_1,\\ \Delta_B &\coloneqq b_2 - b_1.\end{aligned}$$

Nous notons également :

$$\begin{aligned}\Delta_{R,G} &\coloneqq \Delta_R - \Delta_G & \Delta_{G,R} &\coloneqq \Delta_G - \Delta_R,\\ \Delta_{R,B} &\coloneqq \Delta_R - \Delta_B & \Delta_{B,R} &\coloneqq \Delta_B - \Delta_R,\\ \Delta_{G,B} &\coloneqq \Delta_G - \Delta_B & \Delta_{B,G} &\coloneqq \Delta_B - \Delta_G.\end{aligned}$$

Afin de savoir, lors de la compression, quel type de bloc doit être créé pour encoder le pixel P_2, nous utiliserons les règles suivantes.

1. Si $-2 \leq \Delta_R, \Delta_G, \Delta_B \leq 1$, alors chaque différence peut être encodée sur 2 bits, et nous sommes dans le cas d'une *petite* différence. Il faudra donc encoder un bloc `ULBMP_SMALL_DIFF`.
2. Si $-32 \leq \Delta_G \leq 31$ et $-8 \leq \Delta_{R,G}, \Delta_{B,G} \leq 7$, alors la différence d'intensité en vert peut être encodée sur 6 bits, et les différences en rouge et en bleu *par rapport* au décalage en vert peuvent chacune être encodée sur 4 bits, et nous sommes dans le cas d'une différence *intermédiaire*. Il faudra donc encoder un bloc `ULBMP_INTERMEDIATE_DIFF`. Attention : le choix du canal vert pour encoder la plus grande différence est arbitraire et nous pouvons considérer qu'il a été choisi ici car il donne de meilleurs résultats en pratique.

3. Le cas d'une *grande* différence va devoir être scindé en fonction de l'ordre de grandeur des valeurs Δ_R, Δ_G et Δ_B.
   (a) Si $-128 \leq \Delta_R \leq 127$ et $-32 \leq \Delta_{G,R}, \Delta_{B,R} \leq 31$, alors nous pouvons encoder Δ_R sur 8 bits et les deux valeurs restantes sur 6 bits chacune, nous serons dans un bloc `ULBMP_BIG_DIFF` de type `R`.
   (b) Si $-128 \leq \Delta_G \leq 127$ et $-32 \leq \Delta_{R,G}, \Delta_{B,G} \leq 31$, alors nous pouvons encoder Δ_G sur 8 bits et les deux valeurs restantes sur 6 bits chacune, nous serons dans un bloc `ULBMP_BIG_DIFF` de type `G`.
   (c) Si $-128 \leq \Delta_B \leq 127$ et $-32 \leq \Delta_{R,B}, \Delta_{G,B} \leq 31$, alors nous pouvons encoder Δ_B sur 8 bits et les deux valeurs restantes sur 6 bits chacune, nous serons dans un bloc `ULBMP_BIG_DIFF` de type `B`.

   Dans les trois cas ci-dessus, nous avons donc besoin de 2 bits pour identifier le type de bloc, de 8 bits pour encoder la différence principale, et puis de 2 fois 6 bits pour encoder les écarts relatifs des deux autres canaux ; ce qui fait un total de $2 + 8 + 2 \times 6 = 22$ bits, alors que nous pourrions en utiliser 24. En réalité, les deux bits restants vont nous servir à distinguer entre les trois types de bloc `ULBMP_BIG_DIFF`.
4. Sinon, alors nous sommes dans le cas où le pixel va devoir être encodé en entier dans un bloc `ULBMP_NEW_PIXEL`

Une représentation schématique des différents blocs possibles (en binaire et non en hexadécimal pour bien voir les bits occupés par l'identification des blocs) est donnée dans la figure 5.

Attention, toutes les valeurs de différences encodées ici doivent être décalées afin d'être obligatoirement ≥ 0. En particulier, dans le cas d'un bloc `ULBMP_SMALL_DIFF`, en supposant que $(\Delta_R, \Delta_B, \Delta_B) = (1, -2, 0)$, nous devons ajouter 2 à chacune de ces valeurs avant de les écrire. Le byte encodé sera donc `0b00110010` (ou encore `0x32`). De la même manière, pour les blocs `ULBMP_INTERMEDIATE_DIFF`, la première valeur (celle de Δ_G) doit être augmentée de 32 afin d'être entre 0 et 63, et les valeurs $\Delta_{R,G}$ et $\Delta_{B,G}$ doivent être augmentées de 8 afin d'être entre 0 et 7. Finalement, dans les blocs `ULBMP_BIG_DIFF`, la première valeur doit être augmentée de 128 afin d'être entre 0 et 255, et les valeurs restantes doivent être augmentées de 32 afin d'être entre 0 et 63.

Notons que cet algorithme de compression a toujours besoin de comparer le pixel actif au pixel précédent. Afin de ne pas avoir de problème lors de la lecture du premier pixel, nous supposons que le pixel avant le premier pixel est toujours un pixel noir (mais qui n'est pas affiché)

L'algorithme d'encodage peut donc être schématisé comme ceci :

```
P' = Pixel noir = (0, 0, 0)
POUR CHAQUE Pixel P = (r, g, b):
    Calculer les différences Dr, Dg, Db
    Identifier le bloc à écrire
    Encoder le bloc
    Modifier P'
```

L'algorithme de décodage procède alors de manière symétrique :

```
P' = Pixel noir = (0, 0, 0)
TANT QUE l'image n'est pas complète:
    Lire un byte
    Identifier le bloc associé
    Charger les potentiels bytes additionnels
```

ULBMP_NEW_PIXEL :

Byte 0 7 6 5 4 3 2 1 0	Byte 1 7 .. 0	Byte 2 7 .. 0	Byte 3 7 .. 0
1 1 1 1 1 1 1 1	R	G	B

ULBMP_SMALL_DIFF :

Byte 0 7 6 5 4 3 2 1 0			
0 0	Dr	Dg	Db

ULBMP_INTERMEDIATE_DIFF :

Byte 0 7 6 5 4 3 2 1 0		Byte 1 7 6 5 4 3 2 1 0	
0 1	Dg	Dr - Dg	Db - Dg

ULBMP_BIG_DIFF_R :

Byte 0 7 6 5 4 3 2 1 0		Byte 1 7 6 5 4 3 2 1 0	Byte 2 7 6 5 4 3 2 1 0
1 0 0 0	Dr	Dg - Dr	Db - Dr

ULBMP_BIG_DIFF_G :

Byte 0 7 6 5 4 3 2 1 0		Byte 1 7 6 5 4 3 2 1 0	Byte 2 7 6 5 4 3 2 1 0
1 0 0 1	Dg	Dr - Dg	Db - Dg

ULBMP_BIG_DIFF_B :

Byte 0 7 6 5 4 3 2 1 0		Byte 1 7 6 5 4 3 2 1 0	Byte 2 7 6 5 4 3 2 1 0
1 0 1 0	Db	Dr - Db	Dg - Db

FIGURE 5 – Vision schématique des blocs du format ULBMP 4.

```
Reconstruire le pixel
Modifier P'
```

Notons que ce format permet de compresser l'image *jelly beans (4.1.07)* avec un taux ≈ 1.91 : l'encodage non-compressé en version 1 prend 193 KB alors que l'encodage en version 4 prend seulement 103 KB (*cf.* les fichiers `jelly_beans1.ulbmp` et `jelly_beans4.ulbmp`).

Le taux de compression pour l'image *house* est plus petit et vaut ≈ 1.48 : 901 KB pour la version 1 et 607 KB pour la version 4 (*cf.* les fichiers `house1.ulbmp` et `house4.ulbmp`).

Les fichiers `gradients*.ulbmp` encodent tous la même image (dans les différentes version du format ULBMP) à savoir une image de 3 lignes et 256 colonnes dont :

- la première ligne est un dégradé allant de la couleur *rouge* (*i.e.* R=255, G=B=0) à la couleur *noir* (*i.e.* R=G=B=0) ;
- la deuxième ligne est un dégradé allant de la couleur *vert* (*i.e.* G=255, R=B=0) à la couleur *noir* (*i.e.* R=G=B=0) ;
- la troisième ligne est un dégradé allant de la couleur *rouge* (*i.e.* B=255, R=G=0) à la couleur *noir* (*i.e.* R=G=B=0).

Le tableau 4 explicite le contenu du fichier `gradients4.ulbmp`. Il est clair qu'au sein d'une ligne, le passage d'un pixel au suivant correspond simplement à diminuer l'intensité d'un seul des trois canaux de 1 (et le canal en question dépend de la ligne). Cependant, lors du passage d'une ligne à la suivante (ce qui vaut également pour le premier pixel de l'image), on passe d'un pixel *noir* (R=G=B=0) à un pixel ayant deux de ses trois canaux à 0 et le canal restant à 255. Dans ce cas, deux des valeurs $\Delta_R, \Delta_G, \Delta_B$ vaut 0 et la valeur restante est à 255. Dès lors, la différence est trop grande, même pour être encodée dans un bloc `ULBMP_BIG_DIFF`, le pixel doit donc être encodé en entier dans un bloc `ULBMP_NEW_PIXEL`. Le reste des pixels va donc être dans un bloc `ULBMP_SMALL_DIFF` dans lequel soit (i) $\Delta_R = \Delta_G = 0$ et $\Delta_B = -1$; soit (ii) $\Delta_R = \Delta_B = 0$ et $\Delta_G = -1$; soit (iii) $\Delta_G = \Delta_B = 0$ et $\Delta_R = -1$. Dans le premier cas le bloc sera `0x1a = 0b00 01 10 10` puisque $\Delta_R + 2 = -1 + 2 = 1$ `= 0b01` et $\Delta_G + 2 = \Delta_B + 2 = 0 + 2$ `= 0b10`. Dans le cas (ii), le bloc sera `0x26 = 0b00 10 01 10`, et dans le cas (iii), le bloc sera `0x29 = 0b00 10 10 01`.

Code à écrire

Vous devez adapter vos classes `Encoder` et `Decoder` afin de gérer cette version 4 du format ULBMP. Vous devez également adapter votre interface graphique afin qu'elle permette de charger et d'enregistrer des images au format ULBMP 4.

Consignes de remise

Toutes les consignes d'application pour le projet du premier quadrimestre sont toujours d'application. En particulier le projet vaudra **zéro** sans exception si :

- le projet ne peut être exécuté correctement via les commandes décrites dans l'énoncé ;
- les noms de fonctions (et de vos scripts) sont différents de ceux décrits dans cet énoncé, ou ont des paramètres différents ;
- à l'aide d'outils automatiques spécialisés, nous avons détecté un plagiat manifeste (entre les projets de plusieurs étudiant(e)s, ou avec des éléments trouvés sur Internet). Nous insistons sur ce dernier point car l'expérience montre que chaque année une poignée d'étudiant(e)s pensent qu'un petit copier-coller d'une fonction, suivi d'une réorganisation du code et de quelques renommages de variables passera inaperçu... Ceci sera sanctionné d'une note nulle pour l'entièreté du projet pour toutes les personnes impliquées, ainsi que d'éventuelles

Décalage	Valeur (hex)	interprétation
0x00	0x55 0x4c 0x42 0x4d 0x50	ULBMP en ASCII
0x05	0x04	Version 3 du format
0x06	0x0c 0x00	Le header fait 12 bytes
0x08	0x00 0x01	L'image fait 256 pixels de large
0x0a	0x03 0x00	L'image fait 3 pixels de haut
0x0c	0xff 0xff 0x00 0x00	Nouveau pixel R=255, G=B=0
0x10	0x1a	Pixel suivant a $\Delta_R = -1$, $\Delta_G = \Delta_B = 0$
⋮		
0x10f	0xff 0x00 0xff 0x00	Nouveau pixel G=255, R=B=0
0x113	0x26	Pixel suivant a $\Delta_G = -1$, $\Delta_R = \Delta_B = 0$
⋮		
0x212	0xff 0x00 0x00 0xff	Nouveau pixel B=255, R=G=0
0x216	0x29	Pixel suivant a $\Delta_B = -1$, $\Delta_R = \Delta_G = 0$
⋮		
0x314	0x29	Pixel suivant a $\Delta_B = -1$, $\Delta_R = \Delta_G = 0$

TABLE 4 – Contenu du fichier `gradients4.ulbmp`, encodé au format ULBMP 4.0.

autres sanctions. Afin d'éviter cette situation, veillez en particulier à ne pas partager de bouts de codes sur des forums, Facebook, Discord, etc.

Des séances de questions/réponses seront organisées une à deux fois par semaine à un horaire qui vous sera communiqué via l'UV. Ces dernières auront lieu sur Teams et seront enregistrées. Nous ne répondrons donc à aucune question par mail/message privé Teams/l'UV ou autre.

Le projet devra être rendu dans le devoir associé sur l'UV pour le dimanche 24 mars à 22h00. Notez qu'aucun retard ne sera toléré (même d'une minute) donc pensez à vous y prendre à l'avance. En particulier, n'hésitez pas à uploader régulièrement votre code en l'état sur l'UV pour éviter les surprises de dernière minute. Le fichier à rendre est une unique archive au format `.zip` (Attention : pas de `.rar` ou de `.tar.gz`...) dont le nom doit être votre matricule (*e.g.* `000408282.zip`). Tous vos fichiers doivent se trouver *directement* dans l'archive (à savoir pas dans un sous-dossier). Il est très important que vous respectiez cette règle car nous lançons des scripts pour automatiquement décompresser vos projets et lancer les fichiers de tests automatiques. Si votre projet est mal rendu, il sera simplement ignoré par le script et ne sera pas corrigé !

Tous les fichiers de code Python que vous remettez doivent *impérativement* commencer par le docstring suivant :

```
"""
NOM : <nom>
PRÉNOM : <prénom>
SECTION : <section>
MATRICULE : <matricule>
"""
```

Tests automatiques

Un fichier `tests.py` vous est fourni sur l'UV. Ce dernier contient 23 tests automatiques que votre code doit *impérativement* passer à la remise sans quoi il ne sera pas corrigé.

Notez également qu'une *remise intermédiaire* aura également lieu sur l'UV dans les mêmes conditions que la remise finale le dimanche 10 mars 2024 à 22 :00. Cette remise sera suivie d'une exécution *automatique* des 21 premiers tests de ce même fichier (correspondant aux phases 1 à 4). Si le code que vous remettez ne passe pas ces tests, votre projet final **ne sera pas corrigé !** Veillez donc bien à vous y prendre suffisamment à temps (et n'oubliez pas que vous aurez également des projets dans les autres cours pendant ce quadrimestre).

Bon travail !