

# Compression d'images par surfaces unies

Rapport de projet

Lucien Cartier-Tilet

December 3, 2018

# Contents

<b>1</b>	<b>Le problème</b>	<b>3</b>
<b>2</b>	<b>Résolution initiale du problème</b>	<b>3</b>
<b>3</b>	<b>Améliorations possibles</b>	<b>3</b>
3.1	Algorithme	3
3.1.1	Théorie	3
3.1.2	Modifications apportées	4
3.1.3	Résultats	4
3.2	Sortir du concept d'image avec des lignes et colonnes	4
3.2.1	Théorie	4
3.2.2	Modifications apportées	4
3.2.3	Résultats	5
3.3	Passer à de la compression à pertes	5
3.3.1	Théorie	5
3.3.2	Modifications apportées	5
3.3.3	Résultats	8
<b>4</b>	<b>Annexes</b>	<b>10</b>
4.1	Liens	10
4.2	Image de test	11
4.3	Code source	12

# 1 Le problème

Le but de ce projet a été de créer un logiciel permettant la compression d'images via la détection de surfaces d'une même couleur, ainsi que de permettre la décompression du fichier ainsi généré à la compression en une image identique à l'image d'origine. Il est à noter que cet algorithme est orienté vers les images de style *comics* ou avec peu de couleurs, d'où l'image test que vous trouverez en [annexe](#) qui fut mon image de test principale.

Le lien de l'énoncé original se situe également en annexe. Une documentation du projet sera également disponible en suivant le lien donné également en annexe.

## 2 Résolution initiale du problème

Afin de résoudre ce problème, je me suis basé sur un algorithme de M. Jean-Jaques Bourdin et l'ai adapté au problème de la manière suivante : pour chaque pixel de l'image, je teste si le pixel courant a une couleur identique à la couleur d'une surface, ou zone, déjà existante. Si une telle zone n'existe pas déjà, alors je la crée puis pour chaque pixel de la ligne courante je me déplace à l'extrême gauche et à l'extrême droite du segment de la même couleur que le pixel courant. Ainsi on obtient un segment de couleur unie, et il est donc possible de maintenant stocker uniquement les limites de ce segment et à l'ajouter à la collection de segments de la zone, qui elle contient les informations de la couleur de la zone. Ensuite, pour chaque pixel en dessus puis en dessous de ce segment, on répète la même procédure. Cela permet de lister toutes les zones contiguës et de les ajouter ainsi à la zone courante.

Actuellement, avec l'image de test que l'on peut trouver sur mon dépôt git (lien en annexes) d'une taille de 3582016 octets, j'obtiens un taux de compression de 67% environ avec une image compressée à 2403581 octets.

Pour ce qui est du temps d'exécution, j'exécute le script suivant avec la commande `./run-time.sh 10` (10 étant le nombre d'exécutions que je souhaite effectuer) afin d'obtenir le temps d'exécution à répétition du programme :

```
#!/bin/bash
if [ "$#" -ne 1 ]; then
    echo "Bad usage of run-time.sh"
    echo "./run-time <number of runs>"
    exit 1
fi
echo "Starting $1 runs of surfaces-unies"
start=$(date +%s.%N)
for i in `seq 1 $1`; do
    ./bin/surfaces-unies -c -i ./img/asterix.ppm
    ./bin/surfaces-unies -u -i output.su
done
end=$(date +%s.%N)
runtime=$(python -c "print((${end} - ${start}) / ${1})")
echo "Average runtime was $runtime"
```

Ainsi, j'obtiens la moyenne sur dix exécutions d'un temps d'exécution de 7.06 secondes sur un CPU Intel i7-6700HQ (3.500GHz).

## 3 Améliorations possibles

### 3.1 Algorithme

#### 3.1.1 Théorie

Il est possible d'améliorer l'algorithme par rapport à celui utilisé initialement en ignorant l'étape de test des pixels supérieurs et inférieurs à un segment testé. En effet, cette étape est un résidu de l'algorithme d'origine qui a pour but de ne détecter qu'une zone dont tous les segments sont contigus à au moins un autre segment de la même forme, le tout ne constituant qu'une seule et unique forme continue. Hors ici ce

dernier point ne nous intéresse pas, et deux formes de la même couleur n'ont pas à être considérées comme étant deux entités différentes. Ainsi, en ignorant cette étape cela permet de passer de manière itérative sur tous les pixels, simplifiant ainsi l'algorithme.

### 3.1.2 Modifications apportées

La modification apportée au code source est relativement simple étant donné qu'il suffit de commenter les deux dernières boucles for de la fonction `addPixelToSelectedZone` du fichier `compress.c`. Cela bloque l'exécution de la fonction créant les segments à partir des pixels au dessus et en dessous du segment courant.

```
    }
    darrayPushBack(t_zone->segments,
                  newSegment((uint32_t)right_limit, (uint32_t)left_limit));
-   for (; left_limit <= right_limit; ++left_limit) {
-       addPixelToSelectedZone(t_img, t_idx + t_img->sizeX, t_zone);
-   }
-   for (; left_limit <= right_limit; ++left_limit) {
-       addPixelToSelectedZone(t_img, t_idx - t_img->sizeX, t_zone);
-   }
+   /* for (; left_limit <= right_limit; ++left_limit) { */
+   /*     addPixelToSelectedZone(t_img, t_idx + t_img->sizeX, t_zone); */
+   /* } */
+   /* for (; left_limit <= right_limit; ++left_limit) { */
+   /*     addPixelToSelectedZone(t_img, t_idx - t_img->sizeX, t_zone); */
+   /* } */
    }
```

### 3.1.3 Résultats

Avec dix exécutions successives, sur le même processeur avec les mêmes arguments et le programme également compilé avec les options d'optimisation '-O3' également, j'obtiens une vitesse d'exécution moyenne à la compression/décompression successives de 7.23s, soit une perte de 2.35% du temps d'exécution d'origine. J'explique cela par le fait que pour chaque nouveau pixel non-traité, il est nécessaire de trouver la zone correspondant à la couleur du nouveau pixel, ce qui en soit prend du temps alors qu'avec la méthode d'origine, on connaît déjà la zone et on n'a qu'à explorer les pixels de couleur unie, sachant que là où on a le plus de chance de trouver des pixels de couleur identique sur des images de type comics est au niveau des pixels adjacents au pixel actuel. Ainsi, cette piste s'est révélée non concluante, et je ne continuerai pas dans cette direction.

## 3.2 Sortir du concept d'image avec des lignes et colonnes

### 3.2.1 Théorie

L'algorithme actuel considère le fichier d'entrée comme étant une image en deux dimensions, limitant ainsi les segments au début d'une ligne de pixels même si le pixel précédent dans le vecteur dans lequel ces derniers sont stockés est de la même couleur, divisant ainsi des segments en plusieurs segments inutilement. En ignorant ces fins de lignes pour ne procéder qu'à l'analyse des pixels comme ne faisant partie que d'une ligne unique permettrait d'éviter ces ajouts inutiles de segments, et ainsi économiser un peu d'espace avec le fichier compressé. Cependant, je ne pense pas que cette solution soit réellement efficace du fait du peu de cas où ce problème se pose.

### 3.2.2 Modifications apportées

Les deux boucles de la fonction `addPixelToSelectedZone` cherchant les segments ont été modifiées ainsi :

```
for (right_limit = t_idx; right_limit < img_size; ++right_limit) {
    current_pixel = darrayGet(t_img->pixels, (size_t)right_limit);
    if (!sameColor(current_pixel, t_zone)) {
```

```

        break;
    }
    (*current_pixel).visited = 1;
}

for (left_limit = t_idx; left_limit > 0; --left_limit) {
    current_pixel = darrayGet(t_img->pixels, (size_t)left_limit);
    if (current_pixel->visited || !sameColor(current_pixel, t_zone)) {
        break;
    }
    (*current_pixel).visited = 1;
}

```

Cette modification permet d'ignorer les retours à la ligne et ainsi potentiellement économiser de l'espace lors de l'écriture de fichier compressés.

### 3.2.3 Résultats

Hélas le gain d'espace n'est pas flagrant, et cela s'explique par le faible nombre de lignes de pixels de l'image, 1500 dans ce cas, et du fait qu'économiser deux `uint32_t` par ligne est une amélioration mineure, permettant dans le cas de l'image `asterix.ppm` d'économiser uniquement 12 kilo octets. La différence sur le fichier compressé de 2.3 méga octets d'origine est donc négligeable. Je ne continuerai donc pas de travailler sur cette piste.

## 3.3 Passer à de la compression à pertes

### 3.3.1 Théorie

Il existe toujours la possibilité de ne plus faire que de la compression sans pertes mais de faire également de la compression avec pertes, en acceptant en argument un taux de tolérance quant à la similarité des couleurs entre elles. Ainsi, cela éliminera certaines couleurs de l'image et économisera ainsi des zones disposant d'un petit nombre de zones ou de segments en les assimilant aux couleurs qui leur sont proches. L'inconvénient est qu'avec ce paramètre, il sera impossible de restituer l'image d'origine à l'identique.

### 3.3.2 Modifications apportées

Seuls trois fichiers ont eu à être modifiés : `main.c`, `compress.h` et `compress.c`. Voici ci-dessous les modifications apportées à chacun de ces fichiers (sortie de la commande `git diff`) :

```

diff --git a/src/compress.c b/src/compress.c
index 191265a..25e43aa 100644
--- a/src/compress.c
+++ b/src/compress.c
@@ -5,6 +5,8 @@

#include "compress.h"

+int tolerance;
+
/**
 * Cette fonction permet d'évaluer si le pixel passé en argument est éligible à
 * la zone passée également en argument.
@@ -14,10 +16,17 @@
 * \return Valeur booléenne, `1` si le pixel est éligible, `0` sinon
 */
int32_t sameColor(Pixel *t_pixel, Zone *t_zone) {
- return (t_pixel->red == t_zone->red && t_pixel->green == t_zone->green &&
-         t_pixel->blue == t_zone->blue)

```

```

-         ? 1
-         : 0;
+ int diff_red, diff_green, diff_blue;
+ if(tolerance == 0) {
+     return (t_pixel->red == t_zone->red && t_pixel->green == t_zone->green &&
+             t_pixel->blue == t_zone->blue)
+             ? 1
+             : 0;
+ }
+ diff_red = (abs((int32_t)t_zone->red - (int32_t)t_pixel->red) * 100) / 255;
+ diff_green = (abs((int32_t)t_zone->green - (int32_t)t_pixel->green) * 100) / 255;
+ diff_blue = (abs((int32_t)t_zone->blue - (int32_t)t_pixel->blue) * 100) / 255;
+ return ((diff_red + diff_green + diff_blue) / 3) <= tolerance;
}

/**
@@ -173,13 +182,15 @@ void write_compressed_file(Image *t_img, FILE *t_output, darray *t_zones) {
 * \param[in] t_input_file Nom/chemin du fichier `.ppm` d'entrée
 * \param[in] t_output_file Nom/chemin du fichier `.su` de sortie
 */
-void compress(const char *t_input_file, const char *t_output_file) {
+void compress(const char *t_input_file, const char *t_output_file,
+              int32_t t_tolerance) {
    Image *img;
    darray *zones;
    FILE *output_file;
    if (!t_output_file) {
        t_output_file = DEFAULT_COMPRESSED_NAME;
    }
+ tolerance = t_tolerance;
    img = newImage();
    imageLoadPPM(t_input_file, img);
    output_file = get_file(t_output_file, "wb");
diff --git a/src/compress.h b/src/compress.h
index 9b2f832..50f7c25 100644
--- a/src/compress.h
+++ b/src/compress.h
@@ -23,6 +23,7 @@ void write_segments(FILE *t_output, darray *t_segments);
 /// Écrit les données compressées dans le fichier de sortie
 void write_compressed_file(Image *t_img, FILE *t_output, darray *t_zones);
 /// Comprime l'image d'entrée
-void compress(const char *t_input_file, const char *t_output_file);
+void compress(const char *t_input_file, const char *t_output_file,
+              int32_t tolerance);

 #endif /* SRC_COMPRESS_H_ */
diff --git a/src/main.c b/src/main.c
index 8ba5bee..4676ca7 100644
--- a/src/main.c
+++ b/src/main.c
@@ -24,18 +24,21 @@
 */
 void help(int t_exit_code) {
     puts("Usage:\n"
-        "surfaces-unies -i path [-o path] [-options]\n\n"
+        "surfaces-unies -i path [-o path] [--options] [-t 0-100]\n\n"
         "The default action is to compress the mandatory input image to a .su\n"

```

```

        "file saved in the current directory.\n"
        "The input image MUST be saved in the ppm format.\n"
        "Options available:\n"
-       "-h --help\n\tdisplay the current message\n"
-       "-i --input\n\tPath to the input file (MANDATORY)\n"
+       "-h --help\n\tDisplay the current message\n"
+       "-i --input\n\tPath to the input file (MANDATORY)\n"
        "-o --output\n"
-       "\tPath to the output file (if the file already exists, it will be\n"
+       "\tPath to the output file (if the file already exists, it will be\n"
        "\toverwritten)\n"
-       "-c --compress\n\tcompress the input file\n"
-       "-u --uncompress\n\tuncompresses the input file to the output file.");
+       "-t --tolerance\n"
+       "\tColor tolerance for lossy compression. By default at 0 for lossless\n"
+       "\tcompression, at 100 will consider every color to be the same.\n"
+       "-c --compress\n\tCompress the input file\n"
+       "-u --uncompress\n\tUncompresses the input file to the output file.");
    exit(t_exit_code);
}

```

```

@@ -49,9 +52,10 @@ void help(int t_exit_code) {
    * supplémentaires aux fonctions.
    */
    struct Argres {
-   char *input;    /*!< Nom du fichier d'entrée */
-   char *output;  /*!< Nom du fichier de sortie */
-   char compress; /*!< Le fichier d'entrée doit-il être compressé ? */
+   char *input;    /*!< Nom du fichier d'entrée */
+   char *output;  /*!< Nom du fichier de sortie */
+   char compress; /*!< Le fichier d'entrée doit-il être compressé ? */
+   int32_t tolerance; /*!< Tolérance en pourcentage des différences de couleur */
    };
    typedef struct Argres Argres;

```

```

@@ -72,6 +76,12 @@ void get_args(Argres *t_args, const int *const t_c) {
    case 'o': (*t_args).output = optarg; break;
    case 'c': (*t_args).compress = 1; break;
    case 'u': (*t_args).compress = 0; break;
+   case 't':
+       (*t_args).tolerance = atoi(optarg);
+       if(t_args->tolerance < 0 || t_args->tolerance > 100) {
+           help(ARGERROR);
+       }
+       break;
    case '?':
    default: help(ARGERROR);
    }
@@ -95,16 +105,18 @@ Argres process_args(const int t_argc, char *t_argv[]) {
    res.input = NULL;
    res.compress = 1;
    res.output = NULL;
+   res.tolerance = 0;
    while (1) {
        int option_index = 0;
        static struct option long_options[] = {
            {"help", no_argument, NULL, 'h'},

```

```

        {"input", required_argument, NULL, 'i'},
        {"output", required_argument, NULL, 'o'},
+       {"tolerance", required_argument, NULL, 't'},
        {"compress", no_argument, NULL, 'c'},
        {"uncompress", no_argument, NULL, 'u'},
        {NULL, 0, NULL, 0}};
-     int c = getopt_long(t_argc, t_argv, "hi:o:cu", long_options, &option_index);
+     int c = getopt_long(t_argc, t_argv, "hi:o:t:cu", long_options, &option_index);
        if (c == -1)
            break;
        get_args(&res, &c);
@@ -129,8 +141,8 @@ int main(int argc, char **argv) {
        fprintf(stderr, "ERROR: no input file.");
        help(ARGERROR);
    }
-   if(argresults.compress) {
-       compress(argresults.input, argresults.output);
+   if (argresults.compress) {
+       compress(argresults.input, argresults.output, argresults.tolerance);
    } else {
        uncompress(argresults.input, argresults.output);
    }
}

```

### 3.3.3 Résultats

La nouvelle option `-t` est une option très sensible qui peut profondément changer la qualité de l'image dès des valeurs basses, aux alentours de 15 à 25. Cependant, une nette réduction de la taille de l'image compressée peut être constatée, ainsi qu'une baisse notable du temps de compression de l'image dû aux recherches des zones correspondant aux pixels moins fréquentes, et lorsqu'elles se produisent, la recherche se fait parmi moins de zones. Ainsi, pour différentes valeurs de tolérance, on obtient ces résultats :

tolérance	vitesse d'exécution	taille de l'image compressée
0	7.37s	2.3Mo
5	0.85s	2.0Mo
10	0.82s	2.0Mo
15	0.86s	2.0Mo
20	0.84s	1.9Mo
25	0.85s	1.9Mo
30	0.92s	1.8Mo
40	1.18s	2.1Mo
50	0.90s	1.5Mo
60	1.41s	1.9Mo
70	1.34s	1.8Mo
80	1.40s	1.9Mo
90	1.42s	1.9Mo
100	0.17s	12Ko

On remarque cependant qu'après environ 30% de tolérance de couleur, les temps ont tendance à remonter et les fichiers ont tendance à devenir plus lourds, excepté pour la tolérance de 50%. Je suppose qu'il s'agit du fait d'un grand nombre de segments contigus de couleur différente s'alternant rapidement. En tous cas, bien qu'on aie en effet quelques améliorations concernant la taille de l'image compressée, je ne considère personnellement pas cette technique comme en valant la peine du fait d'artéfacts facilement visibles dès les premières valeurs proches de 0, comme on peut le voir sur l'image ci-dessous, compressée avec une tolérance de 5% :





Dans le cas où cela ne serait pas bien visible en version papier, une version digitale de ce document est disponible en ligne, voir le lien donnée en [annexes](#).

## 4 Annexes

### 4.1 Liens

**Énoncé** <http://www.ai.univ-paris8.fr/~jj/Cours/Algo/AA18.html#projets> (projet n°22)

**Dépôt** <https://labs.phundrak.fr/phundrak/surfaces-unies/>

**Image de test** <https://labs.phundrak.fr/phundrak/surfaces-unies/blob/master/img/asterix.ppm>

**Image compressée avec pertes** <https://labs.phundrak.fr/phundrak/surfaces-unies/blob/master/img/asterix5p.png>

**Version digitale de ce document** <https://phundrak.fr/surfaces-unies/rapport.pdf>

**Documentation du code source** <https://phundrak.fr/surfaces-unies/>

**Version PDF de la documentation du code source** <https://phundrak.fr/surfaces-unies/refman.pdf>

## 4.2 Image de test



## 4.3 Code source

main.c

```
1  /**
2  *   \file main.c
3  *   \brief Fichier principal du programme
4  *
5  *   Ce fichier contient les fonctions principales du logiciel qui ne sont pas
6  *   directement liées à la logique et au traitement des données du logiciel,
7  *   mais plutôt au traitement des arguments passés au processus et au lancement
8  *   des fonctions cœurs du programme.
9  */
10
11 #include "compress.h"
12 #include "uncompress.h"
13 #include <getopt.h>
14 #include <string.h>
15
16 /**
17 *   \brief Affiche un message d'aide
18 *
19 *   Affiche un message d'aide pour le logiciel ainsi que son utilisation, puis
20 *   termine le processus avec le code de sortie indiqué par l'argument de la
21 *   fonction.
22 *
23 *   \param[in] t_exit_code Code de sortie du processus
24 */
25 void help(int t_exit_code) {
26     puts("Usage:\n"
27         "surfaces-unies -i path [-o path] [--options] [-t 0-100]\n\n"
28         "The default action is to compress the mandatory input image to a .su\n"
29         "file saved in the current directory.\n"
30         "The input image MUST be saved in the ppm format.\n"
31         "Options available:\n"
32         "-h --help\n\tDisplay the current message\n"
33         "-i --input\n\tPath to the input file (MANDATORY)\n"
34         "-o --output\n"
35         "\tPath to the output file (if the file already exists, it will be\n"
36         "\toverwritten)\n"
37         "-t --tolerance\n"
38         "\tColor tolerance for lossy compression. By default at 0 for lossless\n"
39         "\tcompression, at 100 will consider every color to be the same.\n"
40         "-c --compress\n\tCompress the input file\n"
41         "-u --uncompress\n\tUncompresses the input file to the output file.");
42     exit(t_exit_code);
43 }
44
45 /**
46 *   \struct Argres
47 *   \brief Résultats du traitement des arguments du processus
48 *
49 *   Cette structure est utilisée pour consolider ensemble les résultats du
50 *   traitement des arguments et les renvoyer en une fois au lieu d'avoir à
51 *   utiliser des variables globales ou des pointeurs en arguments
52 *   supplémentaires aux fonctions.
53 */
54 struct Argres {
```

```

55  char *input;          /*!< Nom du fichier d'entrée */
56  char *output;        /*!< Nom du fichier de sortie */
57  char compress;       /*!< Le fichier d'entrée doit-il être compressé ? */
58  int32_t tolerance;   /*!< Tolérance en pourcentage des différences de couleur */
59  };
60  typedef struct Argres Argres;
61
62  /**
63   * \brief Processes independently the arguments of the process
64   *
65   * Each option and switch will be processed here and will modify appropriately
66   * the parameter `args`
67   *
68   * \param[out] t_args Result of the arguments processing
69   * \param[in] t_c Switch or option passed
70   */
71  void get_args(Argres *t_args, const int *const t_c) {
72      switch (*t_c) {
73          case 0: break;
74          case 'h': help(NOERROR); break;
75          case 'i': (*t_args).input = optarg; break;
76          case 'o': (*t_args).output = optarg; break;
77          case 'c': (*t_args).compress = 1; break;
78          case 'u': (*t_args).compress = 0; break;
79          case 't':
80              (*t_args).tolerance = atoi(optarg);
81              if(t_args->tolerance < 0 || t_args->tolerance > 100) {
82                  help(ARGERROR);
83              }
84              break;
85          case '?':
86          default: help(ARGERROR);
87      }
88  }
89
90  /**
91   * \brief Traite les arguments passés au processus
92   *
93   * Les arguments passés au processus seront traités ici. Les arguments passés
94   * dans cette fonction ne subiront aucune modification. La fonction renvoie une
95   * structure \ref Argres contenant le nom de fichier d'entrée et de sortie
96   * ainsi qu'un booléen indiquant si le fichier d'entrée doit être compressé ou
97   * décompressé.
98   *
99   * \param[in] t_argc Nombre d'arguments reçus
100  * \param[in] t_argv Arguments reçus par le processus
101  * \return structure \ref Argres
102  */
103  Argres process_args(const int t_argc, char *t_argv[]) {
104      Argres res;
105      res.input = NULL;
106      res.compress = 1;
107      res.output = NULL;
108      res.tolerance = 0;
109      while (1) {
110          int option_index = 0;
111          static struct option long_options[] = {

```

```

112     {"help", no_argument, NULL, 'h'},
113     {"input", required_argument, NULL, 'i'},
114     {"output", required_argument, NULL, 'o'},
115     {"tolerance", required_argument, NULL, 't'},
116     {"compress", no_argument, NULL, 'c'},
117     {"uncompress", no_argument, NULL, 'u'},
118     {NULL, 0, NULL, 0}};
119     int c = getopt_long(t_argc, t_argv, "hi:o:t:cu", long_options, &option_index);
120     if (c == -1)
121         break;
122     get_args(&res, &c);
123 }
124 return res;
125 }
126
127 /**
128  * \brief Fonction `main` lancée avec le processus
129  *
130  * This function is launched with the process. It will analyze the arguments it
131  * received, and depending on them will either compress or uncompress the input
132  * file, or will throw an error and stop in case of incorrect arguments.
133  *
134  * \param[in] argc Nombre d'arguments reçus par le processus
135  * \param[in] argv Tableau des arguments reçus par le processus
136  * \return Code de status du processus
137  */
138 int main(int argc, char **argv) {
139     Argres argresults = process_args(argc, argv);
140     if (NULL == argresults.input) {
141         fprintf(stderr, "ERROR: no input file.");
142         help(ARGERROR);
143     }
144     if (argresults.compress) {
145         compress(argresults.input, argresults.output, argresults.tolerance);
146     } else {
147         uncompress(argresults.input, argresults.output);
148     }
149     return 0;
150 }

```

compress.h

```

1 /**
2  * \file compress.h
3  * \brief Déclaration pour la (dé)compression d'images
4  */
5
6 #ifndef SRC_COMPRESS_H_
7 #define SRC_COMPRESS_H_
8
9 #include "ppm.h"
10
11 #define DEFAULT_COMPRESSED_NAME "output.su"
12
13 /// Teste l'éligibilité d'un pixel à une zone
14 int32_t sameColor(Pixel *t_pixel, Zone *t_zone);
15 /// Ajoute un pixel et ses pixels connexes à une zone
16 void addPixelToSelectedZone(Image *t_img, int64_t t_idx, Zone *t_zone);

```

```

17 // Sélectionne la zone correspondant à la couleur d'un pixel
18 void chooseZoneForPixel(Image *t_img, int64_t t_idx, darray *zones);
19 // Créé les zones d'une image
20 darray *imgToZones(Image *t_img);
21 // Écrit tous les \ref Segment d'une zone dans le fichier de sortie
22 void write_segments(FILE *t_output, darray *t_segments);
23 // Écrit les données compressées dans le fichier de sortie
24 void write_compressed_file(Image *t_img, FILE *t_output, darray *t_zones);
25 // Comprime l'image d'entrée
26 void compress(const char *t_input_file, const char *t_output_file,
27              int32_t tolerance);
28
29 #endif /* SRC_COMPRESS_H */

```

compress.c

```

1 /**
2  * \file compress.c
3  * \brief Implémentation de la (dé)compression d'images
4  */
5
6 #include "compress.h"
7
8 /**
9  * \var uint32_t tolerance
10 * \brief Color tolerance
11 *
12 * Cette variable est la valeur du pourcentage de tolérance couleur lors de la
13 * création de nouvelles zones. Cette variable contient une valeur située entre
14 * 0 et 100 inclus.
15 */
16 int32_t tolerance;
17
18 /**
19 * Cette fonction permet d'évaluer si le pixel passé en argument est éligible à
20 * la zone passée également en argument. Si la \ref tolerance a pour valeur 0,
21 * alors les couleurs doivent être strictement identiques. Sinon, leur
22 * différence doit être inférieure à la tolérance de couleur.
23 *
24 * \param[in] t_pixel Pointeur vers le pixel dont l'éligibilité est testée
25 * \param[in] t_zone Zone à laquelle le pixel est éligible ou non
26 * \return Valeur booléenne, `1` si le pixel est éligible, `0` sinon
27 */
28 int32_t sameColor(Pixel *t_pixel, Zone *t_zone) {
29     int diff_red, diff_green, diff_blue;
30     if (tolerance == 0) {
31         return (t_pixel->red == t_zone->red && t_pixel->green == t_zone->green &&
32                t_pixel->blue == t_zone->blue)
33             ? 1
34             : 0;
35     }
36     diff_red = (abs((int32_t)t_zone->red - (int32_t)t_pixel->red) * 100) / 255;
37     diff_green =
38         (abs((int32_t)t_zone->green - (int32_t)t_pixel->green) * 100) / 255;
39     diff_blue = (abs((int32_t)t_zone->blue - (int32_t)t_pixel->blue) * 100) / 255;
40     return ((diff_red + diff_green + diff_blue) / 3) <= tolerance;
41 }
42

```

```

43 /**
44  *^^IAjoute un pixel à la zone passé en argument si le pixel à l'index passé en
45  *^^Iargument est éligible à la zone. Si un pixel n'a pas encore été visité, cela
46  *^^Iveut dire également qu'il ne fait partie d'aucun segment, il sera donc
47  *^^Iajouté à un nouveau segment auquel seront rajoutés tous les pixels connexes
48  *^^Iéligibles à la zone. Ensuite, le segment est ajouté à la zone, et la
49  *^^Ifonction actuelle est appelée sur tous les pixels supérieurs et inférieurs
50  *^^ITaux pixels du segment.
51  *
52  *^^I\param[in] t_img Image contenant les pixels explorés
53  *^^I\param[in] t_idx Index du pixel actuel dans l'image `t_img`
54  *^^I\param[out] t_zone Zone à laquelle sera potentiellement ajouté le pixel
55  */
56 void addPixelToSelectedZone(Image *t_img, int64_t t_idx, Zone *t_zone) {
57     const size_t img_size = darraySize(t_img->pixels);
58     Pixel *current_pixel;
59     const uint32_t y = (uint32_t)(t_idx / t_img->sizeX);
60     int64_t left_limit, right_limit;
61     const int64_t xd_limit = (int64_t)t_img->sizeX * (y + 1);
62     if (t_idx >= (int64_t)img_size || t_idx < 0) {
63         return;
64     }
65     current_pixel = darrayGet(t_img->pixels, (size_t)t_idx);
66     if (current_pixel->visited || !sameColor(current_pixel, t_zone)) {
67         return;
68     }
69     (*current_pixel).visited = 1;
70
71     for (right_limit = t_idx; right_limit < xd_limit; ++right_limit) {
72         current_pixel = darrayGet(t_img->pixels, (size_t)right_limit);
73         if (!sameColor(current_pixel, t_zone)) {
74             break;
75         }
76         current_pixel->visited = 1;
77     }
78     for (left_limit = t_idx; left_limit - (y - 1) * (int64_t)t_img->sizeX >= 0;
79         --left_limit) {
80         current_pixel = darrayGet(t_img->pixels, (size_t)left_limit);
81         if (current_pixel->visited || !sameColor(current_pixel, t_zone)) {
82             break;
83         }
84         (*current_pixel).visited = 1;
85     }
86
87     darrayPushBack(t_zone->segments,
88         newSegment((uint32_t)right_limit, (uint32_t)left_limit));
89     for (; left_limit <= right_limit; ++left_limit) {
90         addPixelToSelectedZone(t_img, t_idx + t_img->sizeX, t_zone);
91     }
92     for (; left_limit <= right_limit; ++left_limit) {
93         addPixelToSelectedZone(t_img, t_idx - t_img->sizeX, t_zone);
94     }
95 }
96
97 /**
98  *^^ISélectionne la zone correspondant à la couleur du pixel. Si aucune zone
99  *^^Iexistante ne correspond, une nouvelle est créée et est ajoutée à l'image.

```



```

100  ^^IChaque pixel est itéré, et ignoré si le pixel a déjà été visité auparavant.
101  *
102  ^^I\param[out] t_img L'image contenant les pixels à tester
103  ^^I\param[in] t_idx Index du pixel à tester
104  ^^I\param[out] t_zones Liste des zones de l'image
105  */
106  void chooseZoneForPixel(Image *t_img, int64_t t_idx, darray *t_zones) {
107      Zone *current_zone;
108      Pixel *pixel;
109      size_t i;
110      pixel = darrayGet(t_img->pixels, (size_t)t_idx);
111      if (pixel->visited) {
112          return;
113      }
114      for (i = 0; i < darraySize(t_zones); ++i) {
115          current_zone = darrayGet(t_zones, i);
116          if (sameColor(pixel, current_zone)) {
117              addPixelToSelectedZone(t_img, t_idx, current_zone);
118              return;
119          }
120      }
121      current_zone = newZone(pixel->red, pixel->green, pixel->blue);
122      darrayPushBack(t_zones, current_zone);
123      addPixelToSelectedZone(t_img, t_idx, current_zone);
124  }
125
126  /**
127   ^^IGénère les zones de l'image en titérant chaque pixel de l'image.
128   *
129   ^^I\param t_img Image à convertir en zones
130   ^^I\return Pointeur vers un \ref darray de structures \ref Zone
131   */
132  darray *imgToZones(Image *t_img) {
133      darray *zones;
134      const size_t nb_pixels = darraySize(t_img->pixels);
135      int64_t i;
136      zones = darrayNew(sizeof(Zone));
137      for (i = 0; i < (int64_t)nb_pixels; ++i) {
138          chooseZoneForPixel(t_img, i, zones);
139      }
140      return zones;
141  }
142
143  /**
144   ^^ICette fonction écrit dans \p t_output la taille en `uint64_t` de la zone,
145   ^^Ic'est à dire le nombre de segment qu'elle contient, puis écrit
146   ^^Iindividuellement chaque segment dans \p t_output.
147   *
148   ^^I\param[out] t_output Fichier de sortie
149   ^^I\param[in] t_segments Segments à écrire dans \p t_output
150   */
151  void write_segments(FILE *t_output, darray *t_segments) {
152      uint64_t nb_segments, j;
153      Segment *segment;
154      nb_segments = darraySize(t_segments);
155      fwrite(&nb_segments, sizeof(nb_segments), 1, t_output);
156      for (j = 0; j < darraySize(t_segments); ++j) {

```

```

157     segment = darrayGet(t_segments, j);
158     fwrite(&segment->left_limit, sizeof(Segment), 1, t_output);
159 }
160 }
161
162 /**
163  *Écrit la taille de l'image en abscisse et ordonnées, les deux sous forme de
164  *uint64_t puis le nombre de zones sous forme de uint64_t. Puis, pour
165  *chaque zone son code couleur composé de trois uint8_t successifs
166  *représentant ses couleurs rouge, vert et bleu sont écrit dans le fichier de
167  *sortie \p t_output. Après chaque écriture de zone, l'ensemble des segments
168  *de la zone est libéré de la mémoire. Une fois toutes les zones écrites dans
169  *le fichier de sortie, \p t_zones et libéré de la mémoire.
170  *
171  *\param[in] t_img \ref Image contenant les dimensions du fichier d'origine
172  *\param[out] t_output Fichier où sont écrites les données compressées
173  *\param[in] t_zones Tableau des \ref Zone à écrire puis libérer
174  */
175 void write_compressed_file(Image *t_img, FILE *t_output, darray *t_zones) {
176     uint64_t i, nb_zones = darraySize(t_zones);
177     Zone *current_zone;
178     fwrite(&t_img->sizeX, sizeof(t_img->sizeX), 2, t_output);
179     fwrite(&nb_zones, sizeof(nb_zones), 1, t_output);
180     for (i = 0; i < darraySize(t_zones); ++i) {
181         current_zone = darrayGet(t_zones, i);
182         fwrite(&current_zone->red, sizeof(current_zone->red) * 3, 1, t_output);
183         write_segments(t_output, current_zone->segments);
184         darrayDelete(current_zone->segments);
185     }
186     darrayDelete(t_zones);
187 }
188
189 /**
190  *Convertit une image en zones puis écrit ces zones dans un fichier,
191  *compressant ainsi l'image passée en argument.
192  *
193  *\param[in] t_input_file Nom/chemin du fichier .ppm d'entrée
194  *\param[in] t_output_file Nom/chemin du fichier .su de sortie
195  *\param[in] t_tolerance Pourcentage de tolérance de couleur
196  */
197 void compress(const char *t_input_file, const char *t_output_file,
198              int32_t t_tolerance) {
199     Image *img;
200     darray *zones;
201     FILE *output_file;
202     if (!t_output_file) {
203         t_output_file = DEFAULT_COMPRESSED_NAME;
204     }
205     tolerance = t_tolerance;
206     img = newImage();
207     imageLoadPPM(t_input_file, img);
208     output_file = get_file(t_output_file, "wb");
209     zones = imgToZones(img);
210     write_compressed_file(img, output_file, zones);
211     deleteImage(img);
212     fclose(output_file);
213 }

```

## uncompress.h

```
1  /**
2  *   \file uncompress.h
3  *   \brief Décompression de fichiers
4  *
5  *   Ce fichier contient les déclarations des fonctions nécessaires à
6  *   décompresser un fichier en `.su` généré par ce programme vers un fichier
7  *   `.ppm` identique à l'original.
8  *
9  */
10
11 #ifndef SRC_UNCOMPRESS_H_
12 #define SRC_UNCOMPRESS_H_
13
14 #include "ppm.h"
15
16 #define DEFAULT_UNCOMPRESSED_FILE "output.ppm"
17
18 /// Lit les segments compressés dans une zone
19 void read_compressed_zones(FILE *t_file, darray *t_zones);
20 /// Lit les zones compressées dans le fichier d'entrée
21 void read_compressed_file_data(FILE *t_file, darray *zones);
22 /// Lit les premières données du fichier compressé
23 void read_compressed_file_meta(FILE *t_file, Image *t_img);
24 /// Décompresse le fichier d'entrée dans le fichier de sortie
25 void uncompress(const char *t_input_file, const char *t_output_file);
26
27 #endif /* SRC_UNCOMPRESS_H_ */
```

## uncompress.c

```
1  /**
2  *   \file uncompress.c
3  *   \brief Décompression de fichiers
4  *
5  *   Ce fichier contient l'implémentation des fonctions nécessaires à la
6  *   décompression d'un fichier en `.su` généré par ce programme vers un fichier
7  *   en `.ppm` identique à l'original.
8  *
9  */
10
11 #include "uncompress.h"
12
13 /**
14 *   Lit le nombre de segments en `uint32_t` contenus dans la zone compressée,
15 *   puis pour chaque \ref Segment ajoute ses données décompressées au tableau
16 *   dynamique de \p t_zone.
17 *
18 *   \param[in] t_file Fichier d'entrée contenant les segments à lire
19 *   \param[out] t_zone La zone dans laquelle stocker les \ref Segment
20 */
21 void read_segments(FILE *t_file, Zone *t_zone) {
22     uint64_t nb_segments, i;
23     Segment *segment;
24     fread(&nb_segments, sizeof(nb_segments), 1, t_file);
25     for(i = 0; i < nb_segments; ++i) {
26         segment = newSegment(0, 0);
```

```

27     fread(&segment->left_limit, sizeof(Segment), 1, t_file);
28     darrayPushBack(t_zone->segments, segment);
29 }
30 }
31
32 /**
33  * Lit les données compressées du fichier d'entrée, tout d'abord le nombre de
34  * zones dans un `uint42_t`, puis pour chaque zone lit les segments. Une fois
35  * une zone lue, elle est ajoutée à \p t_zones.
36  *
37  * \param[in] t_file Fichier d'entrée contenant les données compressées
38  * \param[out] t_zones Tableau dynamique contenant les zones lues
39  */
40 void read_compressed_zones(FILE *t_file, darray *t_zones) {
41     uint64_t nb_zones, i;
42     Zone *zone;
43     /* read number of zones */
44     fread(&nb_zones, sizeof(nb_zones), 1, t_file);
45     for(i = 0; i < nb_zones; ++i) {
46         zone = newZone(0, 0, 0);
47         /* read RGB into the zone */
48         fread(&zone->red, sizeof(zone->red), 3, t_file);
49         /* read each segments of the zone and add them to their vector */
50         read_segments(t_file, zone);
51         /* add the zone to the zones of the image */
52         darrayPushBack(t_zones, zone);
53     }
54 }
55
56 /**
57  * Lit les données basiques du fichier compressé, à savoir la taille en
58  * abscisse et en ordonnée du fichier ppm d'origine et les inscrit dans \p
59  * t_img.
60  *
61  * \param[in] t_file Fichier d'entrée à lire
62  * \param[out] t_img Structure \ref Image stockant les données lues
63  */
64 void read_compressed_file_meta(FILE *t_file, Image *t_img) {
65     /* read sizeX and sizeY at once */
66     fread(&t_img->sizeX, sizeof(t_img->sizeX), 2, t_file);
67 }
68
69 uint8_t *zones_to_data(darray *t_zones, Image *t_img) {
70     uint64_t nb_zones, nb_segments, zoneID, segmentID, k, left_limit, right_limit;
71     uint8_t *data, red, green, blue;
72     Zone *current_zone;
73     Segment *current_segment;
74     data = (uint8_t *)malloc(sizeof(uint8_t) * t_img->sizeX * t_img->sizeY * 3);
75     nb_zones = darraySize(t_zones);
76     for (zoneID = 0; zoneID < nb_zones; ++zoneID) {
77         current_zone = darrayGet(t_zones, zoneID);
78         red = current_zone->red;
79         green = current_zone->green;
80         blue = current_zone->blue;
81         nb_segments = darraySize(current_zone->segments);
82         for (segmentID = 0; segmentID < nb_segments; ++segmentID) {
83             current_segment = darrayGet(current_zone->segments, segmentID);

```

```

84     left_limit = current_segment->left_limit;
85     right_limit = current_segment->right_limit;
86     for (k = left_limit; k < right_limit; ++k) {
87         data[k * 3] = red;
88         data[k * 3 + 1] = green;
89         data[k * 3 + 2] = blue;
90     }
91 }
92 }
93 return data;
94 }
95
96 /**
97  * Décompresse le fichier d'entrée et écrit son équivalent décompressé au
98  * format ppm dans le fichier de sortie.
99  *
100  * \param[in] t_input_file Nom/chemin du fichier d'entrée
101  * \param[in] t_output_file Nom/chemin du fichier de sortie
102  */
103 void uncompress(const char *t_input_file, const char *t_output_file) {
104     Image *img;
105     darray *zones = darrayNew(sizeof(Zone));
106     uint8_t *data;
107     FILE *input_file;
108     uint64_t i;
109     if (!t_output_file) {
110         t_output_file = DEFAULT_UNCOMPRESSED_FILE;
111     }
112     img = newImage();
113     input_file = get_file(t_input_file, "rb");
114     read_compressed_file_meta(input_file, img);
115     read_compressed_zones(input_file, zones);
116     data = zones_to_data(zones, img);
117     imageSavePPM(t_output_file, img, data);
118     fclose(input_file);
119     /* free memory */
120     for(i = 0; i < darraySize(zones); ++i) {
121         Zone *zone = darrayGet(zones, i);
122         darrayDelete(zone->segments);
123     }
124     darrayDelete(zones);
125 }

```

ppm.h

```

1 /**
2  * \file ppm.h
3  * \brief Fichier d'en-tête pour les fonctions de manipulation d'images ppm
4  *
5  * En-tête contenant la déclaration de fonctions de lecture et d'écriture de
6  * fichiers au format ppm. La définition des fonction se trouve dans \ref ppm.c
7  *
8  */
9
10 #ifndef SRC_PPM_H_
11 #define SRC_PPM_H_
12
13 #include "utilities.h"

```

```

14
15 /// \brief Ouvre un fichier avec les autorisations demandées
16 FILE *get_file(const char *t_filename, const char *t_mode);
17 /// \brief Lit le format d'un fichier ppm ouvert
18 void read_file_format(FILE *t_fp, const char *t_filename);
19 /// \brief Vérifie et ignore d'éventuels commentaires du header d'un fichier
20 void check_for_comments(FILE *t_fp);
21 /// \brief Lit les dimensions du fichier ppm ouvert
22 void read_file_size(FILE *t_fp, Image *t_img, const char *t_filename);
23 /// \brief Lit et vérifie le format RGB du fichier ppm
24 void read_rgb(FILE *t_fp, const char *t_filename);
25 /// \brief Lit dans le conteneur les données images du fichier ppm
26 void read_data(FILE *t_fp, uint64_t t_size, uint8_t **t_data,
27               const char *t_filename);
28 /// \brief Convertit les données brutes de fichier vers des conteneurs de pixels
29 void dataToImage(Image *t_img, uint8_t *t_data, uint64_t t_size);
30 /// \brief Convertit les pixels d'une image en tableau natif OpenGL
31 unsigned char *imageToData(Image *t_img);
32 /// \brief Ouverture et lecture de l'image d'entrée
33 int imageLoadPPM(const char *t_filename, Image *t_img);
34 /// \brief Ouverture et écriture de l'image de sortie
35 void imageSavePPM(const char *t_filename, Image *t_img, uint8_t *data);
36
37 #endif /* SRC_PPM_H */

```

ppm.c

```

1 /**
2  * \file ppm.c
3  * \brief Fichier de déclaration des fonctions de manipulation d'images ppm
4  *
5  * Déclaration du corps des fonctions déclarées dans \ref ppm.h
6  */
7
8 #include "ppm.h"
9 #include <assert.h>
10
11 #define RGB_COMPONENT_COLOR 255
12 #define CREATOR "CL"
13
14 /**
15  * \brief fonction description
16  *
17  * Fonction d'ouverture de fichier selon le mode demandé. Si la fonction ne
18  * peut pas ouvrir le fichier, elle arrête le processus qui renverra la valeur
19  * `1`. En cas de succès, la fonction renverra un pointeur de fichier vers le
20  * fichier ouvert.
21  *
22  * \param[in] t_filename Nom du fichier à ouvrir
23  * \param[in] t_mode Mode du fichier à ouvrir
24  * \return Pointeur de fichier
25  */
26 FILE *get_file(const char *t_filename, const char *t_mode) {
27     FILE *fp = fopen(t_filename, t_mode);
28     if (!fp) {
29         fprintf(stderr, "Unable to open file '%s'\n", t_filename);
30         exit(FILE_IO_ERROR);
31     }

```

```

32     return fp;
33 }
34
35 /**
36  * Lit et vérifie le format du fichier passé en argument. Si le format n'est
37  * pas correct, la fonction arrête le processus qui renverra la valeur `1`.
38  *
39  * \param[in] t_fp Fichier ppm où lire les données
40  * \param[in] t_filename Nom du fichier ouvert
41  */
42 void read_file_format(FILE *t_fp, const char *t_filename) {
43     char buff[16];
44     if (!fgets(buff, sizeof(buff), t_fp)) {
45         perror(t_filename);
46         exit(FILE_IO_ERROR);
47     }
48     /* check file format */
49     if (buff[0] != 'P' || buff[1] != '6') {
50         fprintf(stderr, "Invalid image format (must be 'P6')\n");
51         exit(FILE_FORMAT_ERROR);
52     }
53 }
54
55 /**
56  * Vérifie si le header contient des commentaires et les ignore le cas échéant.
57  *
58  * \param[in] t_fp Fichier ppm où lire les données
59  */
60 void check_for_comments(FILE *t_fp) {
61     char c;
62     c = (char)getc(t_fp);
63     while (c == '#') {
64         while (getc(t_fp) != '\n') {
65             }
66         c = (char)getc(t_fp);
67     }
68     ungetc(c, t_fp);
69 }
70
71 /**
72  * Lit la taille des données image et les écrit dans le conteneur d'images
73  * passé en argument.
74  *
75  * \param[in] t_fp Fichier ppm où lire les données
76  * \param[out] t_img Conteneur d'image où écrire les résultats
77  * \param[in] t_filename Nom du fichier ouvert
78  */
79 void read_file_size(FILE *t_fp, Image *t_img, const char *t_filename) {
80     if (fscanf(t_fp, "%lu %lu", &t_img->sizeX, &t_img->sizeY) != 2) {
81         fprintf(stderr, "Invalid image size (error loading '%s')\n", t_filename);
82         exit(FILE_FORMAT_ERROR);
83     }
84 }
85
86 /**
87  * Vérifie le format RGB de l'image ppm. Si le format n'est pas correct, la
88  * fonction arrête le processus qui renverra la valeur `1`.

```

```

89  *
90  *  \param[in] t_fp Fichier ppm où lire les données
91  *  \param[in] t_filename Nom du fichier ouvert
92  */
93  void read_rgb(FILE *t_fp, const char *t_filename) {
94      char d;
95      int rgb_comp_color;
96      /* read rgb component */
97      if (fscanf(t_fp, "%d", &rgb_comp_color) != 1) {
98          fprintf(stderr, "Invalid rgb component (error loading '%s')\n", t_filename);
99          exit(FILE_FORMAT_ERROR);
100     }
101     fscanf(t_fp, "%c ", &d);
102     /* check rgb component depth */
103     if (rgb_comp_color != RGB_COMPONENT_COLOR) {
104         fprintf(stderr, "'%s' does not have 8-bits components\n", t_filename);
105         exit(FILE_FORMAT_ERROR);
106     }
107 }
108
109 /**
110  *  \brief function description
111  *
112  *  Litles données images brutes du fichier ppm ouvert et les stocke dans \p
113  *  t_data.
114  *
115  *  \param[in] t_fp Fichier ppm ouvert source
116  *  \param[in] t_size Taille des données brutes
117  *  \param[out] t_data Pointeur vers le tableau de sortie des données brutes
118  *  \param[in] t_filename Nom du fichier d'entrée
119  *  \return Taille du tableau de données obtenu
120  */
121  void read_data(FILE *t_fp, uint64_t t_size, unsigned char **t_data,
122                const char *t_filename) {
123      *t_data = (unsigned char *)malloc(t_size * sizeof(unsigned char));
124      assert(*t_data);
125      /* read pixel data from file */
126      if (!fread(*t_data, (size_t)1, t_size, t_fp)) {
127          fprintf(stderr, "Error loading image '%s'\n", t_filename);
128          free(*t_data);
129          exit(FILE_IO_ERROR);
130      }
131  }
132
133  /**
134  *  Convertit vers un tableau de `unsigned char` les pixels contenus dans un
135  *  conteneur d'image. La taille du tableau de `unsigned char` est la taille du
136  *  tableau de pixels multipliée par trois du fait des trois emplacements séparés
137  *  par couleur.
138  *
139  *  \param[out] t_img Image dont les pixels doivent être convertis
140  *  \param[in] t_data Données à convertir en structures \ref Pixel
141  *  \param[in] t_size Taille du tableau de `unsigned char`
142  */
143  void dataToImage(Image *t_img, uint8_t *t_data, uint64_t t_size) {
144      uint64_t i;
145      t_img->pixels = darrayNew(sizeof(Pixel));

```



```

146     for (i = 0; i < t_size; i += 3) {
147         darrayPushBack(t_img->pixels,
148             newPixel(t_data[i], t_data[i + 1], t_data[i + 2]));
149     }
150 }
151
152 /**
153  * Convertit le vecteur de pixels d'un conteneur d'image en un tableau de
154  * valeurs de type `uint8_t` afin de permettre l'écriture d'une image dans un
155  * fichier.
156  *
157  * \param[in] t_img Conteneur d'image contenant les pixels à convertir
158  * \return Tableau de pointeurs de `uint8_t`
159  */
160 uint8_t *imageToData(Image *t_img) {
161     Pixel *pixel;
162     uint8_t *data, size;
163     uint64_t i;
164     size = (uint8_t)darraySize(t_img->pixels);
165     data = (uint8_t *)malloc(3 * sizeof(uint8_t) * size);
166     for (i = 0; i < size; i += 3) {
167         pixel = darrayGet(t_img->pixels, i / 3);
168         data[i] = pixel->red;
169         data[i + 1] = pixel->green;
170         data[i + 2] = pixel->blue;
171     }
172     return data;
173 }
174
175 /**
176  * Ouvre le fichier image avec son nom de fichier passé par le paramètre
177  * `filename` et charge ses informations et données dans l'objet `img` dans
178  * lequel les données et l'image seront manipulables. Retourne la valeur 1 en
179  * cas de succès.
180  *
181  * \param[in] t_filename Nom du fichier image à ouvrir
182  * \param[out] t_img Objet \ref Image manipulable
183  * \return Retourne 1 en cas de succès
184  */
185 int imageLoadPPM(const char *t_filename, Image *t_img) {
186     FILE *fp;
187     uint64_t size;
188     unsigned char *data = NULL;
189     fp = get_file(t_filename, "rb");           /* open PPM file for reading */
190     read_file_format(fp, t_filename);         /* read image format */
191     check_for_comments(fp);                   /* check for comments */
192     read_file_size(fp, t_img, t_filename);    /* read image size information */
193     read_rgb(fp, t_filename);                 /* read rgb component */
194     size = t_img->sizeX * t_img->sizeY * 3;
195     read_data(fp, size, &data, t_filename);  /* read data from file */
196     dataToImage(t_img, data, size);
197     free(data);
198     fclose(fp);
199     return 1;
200 }
201
202 /**

```

```

203  *  Ouvre le fichier image avec son nom de fichier passé par le paramètre
204  *  `filename` et y écrit les informations trouvées dans l'objet `img`.
205  *
206  *  \param[in] t_filename Nom du fichier image à ouvrir
207  *  \param[in] t_img  Objet \ref Image à écrire
208  *  \param[in] t_data Données décompressées de l'image au format natif ppm
209  */
210 void imageSavePPM(const char *t_filename, Image *t_img, uint8_t *t_data) {
211     FILE *fp;
212     fp = get_file(t_filename, "wb"); /* open file for output */
213     /* write the header file */
214     fprintf(fp, "P6\n"); /* image format */
215     fprintf(fp, "# Created by %s\n", CREATOR); /* comments */
216     fprintf(fp, "%lu %lu\n", t_img->sizeX, t_img->sizeY); /* image size */
217     fprintf(fp, "%d\n", RGB_COMPONENT_COLOR); /* rgb component depth */
218     fwrite(t_data, (size_t)1, (size_t)(3 * t_img->sizeX * t_img->sizeY), fp);
219     free(t_data);
220     fclose(fp);
221 }

```

#### utilities.h

```

1  /**
2  *  \file utilities.h
3  *  \brief Déclaration des structures de données et fonctions utilitaires.
4  *
5  *  Dans ce fichier sont déclarées et implémentées les structures qui serviront
6  *  de conteneurs aux données manipulées. Sont également déclarées les fonctions
7  *  utilitaires pour la manipulation de ces structures.
8  */
9
10 #ifndef SRC_UTILITIES_H_
11 #define SRC_UTILITIES_H_
12
13 #include "darray.h"
14 #include <stdio.h>
15
16 /*****
17  *
18  *  DEFINE DIRECTIVES
19  */
20 #ifndef Debug
21 #define DEBUG if (1)
22 #else
23 #define DEBUG if (0)
24 #endif
25
26 /*****
27  *
28  *  STRUCT DECLARATION
29  */
30 struct Image;
31 typedef struct Image Image;
32 struct Pixel;
33 typedef struct Pixel Pixel;
34 struct Zone;
35 typedef struct Zone Zone;
36 struct Segment;

```

```

37 typedef struct Segment Segment;
38
39 *****
40 /* STRUCT IMPLEMENTATION */
41 *****
42
43 /**
44  * \brief Conteneur d'une image
45  *
46  * Une image est une structure définie par ses dimensions verticales et
47  * horizontales x et y, et contenant pour chacune des coordonnées possibles
48  * selon ses dimensions un pixel de type \ref Pixel. Ces pixels sont stockés
49  * dans un tableau dynamique \ref darray.
50  */
51 struct Image {
52     uint64_t sizeX; /*!< Largeur de l'image */
53     uint64_t sizeY; /*!< Hauteur de l'image */
54     darray *pixels; /*!< Vecteur à une dimension de \ref Pixel */
55 };
56
57 /**
58  * \brief Conteneur d'un pixel
59  *
60  * Un pixel est défini par sa couleur représenté en RGB (rouge, vert, bleu).
61  * Il contient également une valeur booléenne afin de savoir si le Pixel fut
62  * visité précédemment par l'algorithme de compression.
63  */
64 struct Pixel {
65     uint8_t red; /*!< Couleur rouge du pixel */
66     uint8_t green; /*!< Couleur verte du pixel */
67     uint8_t blue; /*!< Couleur bleue du pixel */
68     uint8_t visited; /*!< Le pixel a-t-il été visité avant */
69 };
70
71 /**
72  * \brief Conteneur de zone de couleur unie
73  *
74  * Une zone est un ensemble de pixels de même couleur ou de couleur similaire
75  * dont on conserve uniquement les marges dans le tableau dynamique.
76  */
77 struct Zone {
78     uint8_t red; /*!< Couleur rouge de la zone */
79     uint8_t green; /*!< Couleur verte de la zone */
80     uint8_t blue; /*!< Couleur bleue de la zone */
81     darray *segments; /*!< Vecteur de \ref Segment */
82 };
83
84 /**
85  * \brief Conteneur de segment de couleur unie
86  *
87  * Un segment est un ensemble de pixels de même ordonnée et de couleur unie ou
88  * similaire. Il se définit par son ordonnée y et de ses deux pixels de bordure
89  * à son extrême droite et à son extrême gauche.
90  */
91 struct Segment {
92     uint32_t left_limit; /*!< extrême gauche du segment */
93     uint32_t right_limit; /*!< extrême droit du segment */

```

```

94 };
95
96 /*
97  *                               Utility functions declaration                               */
98 /*
99
100 /// \brief Création d'un nouveau pixel
101 Pixel *newPixel(uint8_t t_r, uint8_t t_g, uint8_t t_b);
102 /// \brief Création d'une nouvelle image
103 Image *newImage();
104 /// \brief Destructeur d'une image
105 void deleteImage(Image *t_self);
106 /// \brief Constructeur d'un segment de couleur unie
107 Segment *newSegment(uint32_t t_right_limit, uint32_t t_left_limit);
108 /// \brief Constructeur de conteneur de zone
109 Zone *newZone(uint8_t t_r, uint8_t t_g, uint8_t t_b);
110
111 #endif /* SRC_UTILITIES_H_ */

```

utilities.c

```

1 /**
2  * \file utilities.c
3  * \brief Implémentation des fonctions utilitaires
4  *
5  * Dans ce fichier sont implémentées les fonctions utilitaires pour la
6  * manipulation des structures de données déclarées dans le fichier header
7  * correspondant.
8  */
9
10 #include "utilities.h"
11
12 /**
13  * Créé un nouveau pixel initialisé avec les arguments `r`, `g` et `b` et
14  * renvoie un pointeur vers ce pixel créé.
15  *
16  * \param[in] t_r Valeur rouge du pixel
17  * \param[in] t_g Valeur verte du pixel
18  * \param[in] t_b Valeur bleue du pixel
19  * \return Pointeur sur une structure de type \ref Pixel
20  */
21 Pixel *newPixel(uint8_t t_r, uint8_t t_g, uint8_t t_b) {
22     Pixel *res;
23     res = (Pixel *)malloc(sizeof(Pixel));
24     res->red = t_r;
25     res->green = t_g;
26     res->blue = t_b;
27     res->visited = 0;
28     return res;
29 }
30
31 /**
32  * Constructeur d'un conteneur d'image. Les dimensions sont initialisées à zéro
33  * (0) et son tableau de pixels a été créé et initialisé en tableau vide. Le
34  * constructeur renvoie un pointeur vers la nouvelle structure \ref Image.
35  *
36  * \return Pointeur vers une structure \ref Image initialisée.
37  */

```

```

38 Image *newImage() {
39     Image *res;
40     res = (Image *)malloc(sizeof(Image));
41     res->sizeX = 0;
42     res->sizeY = 0;
43     res->pixels = darrayNew(sizeof(Pixel));
44     return res;
45 }
46
47 /**
48  * Destructeur d'un conteneur d'image. Le destructeur appellera le destructeur
49  * du vecteur de pixels qui sera libéré de la mémoire, puis ultimement le
50  * destructeur libérera la structure \ref Image pointée par le pointeur passé
51  * en argument.
52  *
53  * \param[in] t_self Conteneur d'image à détruire
54  */
55 void deleteImage(Image *t_self) {
56     darrayDelete(t_self->pixels);
57     free(t_self);
58 }
59
60 /**
61  * Constructeur d'un conteneur de segment. Le constructeur va initialiser les
62  * valeurs qu'il contiendra avec les arguments qui lui seront passés lors de
63  * l'appel de la fonction.
64  *
65  * \param[in] t_right_limit Abscisse extrême droite du segment
66  * \param[in] t_left_limit Abscisse extrême gauche du segment
67  * \return Pointeur sur un conteneur de segment
68  */
69 Segment *newSegment(uint32_t t_right_limit, uint32_t t_left_limit) {
70     Segment *res = (Segment *)malloc(sizeof(Segment));
71     res->left_limit = t_left_limit;
72     res->right_limit = t_right_limit;
73     return res;
74 }
75
76 /**
77  * \brief fonction description
78  *
79  * Constructeur de conteneur de zone, initialise grâce aux arguments la couleur
80  * de la zone et initialise un tableau dynamique vide de \ref Segment. Renvoie
81  * un pointeur vers la structure nouvellement créée.
82  *
83  * \param[in] t_r Valeur rouge de la couleur de la zone
84  * \param[in] t_g Valeur verte de la couleur de la zone
85  * \param[in] t_b Valeur bleue de la couleur de la zone
86  * \return Pointeur vers la structure créée
87  */
88 Zone *newZone(uint8_t t_r, uint8_t t_g, uint8_t t_b) {
89     Zone *res = (Zone *)malloc(sizeof(Zone));
90     res->red = t_r;
91     res->green = t_g;
92     res->blue = t_b;
93     res->segments = darrayNew(sizeof(Segment));
94     return res;

```

```

95 }

    darray.h

1  /**
2   * \file darray.h
3   * \brief Implémentation de \ref darray et déclaration des fonctions pour ce
4   * type
5   */
6
7  #ifndef SRC_DARRAY_H_
8  #define SRC_DARRAY_H_
9
10 #include "errorcodes.h"
11 #include <stdint.h>
12 #include <stdlib.h>
13
14 /**
15  * \struct darray
16  * \brief Tableau dynamique
17  *
18  * Les objets `darray` offrent la possibilité d'avoir des tableaux à taille
19  * variable en C, similairement aux objets `vector` en C++.
20  */
21 typedef struct {
22     void *begin; /*!< Pointeur sur le premier élément du tableau */
23     void *end; /*!< Pointeur sur l'élément situé immédiatement après le dernier
24                élément du tableau */
25     uint64_t element_size; /*!< Taille des éléments stockés dans le tableau */
26     uint64_t capacity; /*!< Capacité maximale du tableau actuel */
27 } darray;
28
29 /// \brief Créé un nouvel objet \ref darray vide
30 darray *darrayNew(uint64_t element_size);
31 /// \brief Augmente la capacité d'un \ref darray
32 void darrayExtend(darray *self);
33 /// \brief Insère un élément à l'endroit pointé dans un \ref darray
34 void darrayInsert(darray *self, void *pos, void *elem);
35 /// \brief Supprime l'élément pointé dans l'objet \ref darray
36 void darrayErase(darray *self, void *pos);
37 /// \brief Retourne l'élément du \ref darray au idx-ème index
38 void *darrayGet(darray *self, uint64_t idx);
39 /// \brief Insère un élément à la fin de l'élément \ref darray
40 void darrayPushBack(darray *self, void *elem);
41 /// \brief Supprime le dernier élément de l'élément \ref darray
42 void darrayPopBack(darray *self);
43 /// \brief Détruit l'élément \ref darray
44 void darrayDelete(darray *self);
45 /// \brief Renvoie la taille de l'élément \ref darray
46 uint64_t darraySize(darray *self);
47 /// \brief Renvoie la taille de l'élément \ref darray
48 uint64_t darrayElemSize(darray *self);
49
50 #endif /* SRC_DARRAY_H_ */

    darray.c

1  /**
2   * \file darray.c

```

```

3  * \brief Implémentation des fonctions pour le type \ref darray
4  */
5
6  #include "darray.h"
7  #include <stdio.h>
8  #include <string.h>
9
10 /**
11  * `darrayNew` permet de créer un nouvel objet de type \ref darray ne
12  * contenant aucun élément. Le seul paramètre, `element_size`, est utilisé afin
13  * de connaître l'espace mémoire à allouer à chacun des éléments dans le
14  * tableau. Cela implique qu'un objet \ref darray ne peut contenir que des
15  * éléments du même type.
16  *
17  * \param[in] t_element_size Taille des objets stockés
18  * \return Pointeur sur le nouvel objet \ref darray
19  */
20 darray *darrayNew(uint64_t t_element_size) {
21     darray *ret;
22     ret = (darray *)malloc(sizeof(darray));
23     ret->begin = NULL;
24     ret->end = ret->begin;
25     ret->element_size = t_element_size;
26     ret->capacity = 0;
27     return ret;
28 }
29
30 /**
31  * `darrayInsert` insère l'élément `elem` avant l'élément pointé par `pos` dans
32  * l'objet \ref darray. Cela décalera tous les éléments stockés dans la `self`
33  * pen d'un cran vers la fin du tableau et insérera à l'endroit pointé le nouvel
34  * élément. Cette fonction modifie les membres `begin` et `end` et
35  * potentiellement `capacity` de `self`.
36  *
37  * \param[in] t_self Objet \ref darray dans lequel on souhaite insérer un nouvel
38  * élément
39  * \param[in] t_pos Position à laquelle on souhaite insérer un nouvel élément
40  * \param[in] t_elem Élément que l'on souhaite insérer
41  */
42 void darrayInsert(darray *t_self, void *t_pos, void *t_elem) {
43     char *itr;
44     int64_t pos_aux;
45     pos_aux = (char *)t_pos - (char *)t_self->begin;
46     if (darraySize(t_self) >= t_self->capacity) {
47         darrayExtend(t_self);
48     }
49     itr = (char *)t_self->begin + pos_aux;
50     memmove(itr + t_self->element_size, itr, ((char *)t_self->end - itr));
51     memcpy(itr, t_elem, t_self->element_size);
52     (*t_self).end = (char *)t_self->end + t_self->element_size;
53 }
54
55 /**
56  * Étend la capacité d'un \ref darray en réallouant sa mémoire, multipliant
57  * sa capacité par deux. Si la réallocation mémoire ne réussit pas, le
58  * programme s'arrête immédiatement, renvoyant la valeur de \ref PTR_ERROR
59  *

```

```

60  * \param[in] t_self L'objet darray à étendre
61  */
62  void darrayExtend(darray *t_self) {
63      void *new_array;
64      uint64_t size;
65      size = darraySize(t_self);
66      new_array =
67          realloc(t_self->begin, (darraySize(t_self) + 1) * t_self->element_size);
68      if (!new_array) {
69          fprintf(stderr, "Failed memory reallocation at %s:%d\nAborting...",
70                  __FILE__, __LINE__ - 2);
71          exit(PTR_ERROR);
72      }
73      (*t_self).begin = new_array;
74      ++(*t_self).capacity;
75      (*t_self).end = (char *)t_self->begin + size * t_self->element_size;
76  }
77
78  /**
79   * `darrayErase` supprime l'élément de objet \ref darray `self` pointé par
80   * `pos`. Cela décalera tous les éléments suivants dans le tableau d'un cran
81   * vers le début du tableau de manière à ce qu'il n'y aie pas d'élément vide
82   * entre les membres `begin` et `end` de `self`. Par ailleurs, le membre `end`
83   * de `self` se retrouve modifié par la fonction.
84   *
85   * \param[out] t_self Objet \ref darray dont on souhaite supprimer un élément
86   * \param[in] t_pos Élément de `self` que l'on souhaite supprimer
87   */
88  void darrayErase(darray *t_self, void *t_pos) {
89      memmove(t_pos, (char *)t_pos + t_self->element_size,
90              (((char *)t_self->end - t_self->element_size) - (char *)t_pos));
91      (*t_self).end = (char *)t_self->end - t_self->element_size;
92  }
93
94  /**
95   * `darrayPushBack` ajoute un nouvel élément `elem` à l'objet `self` à la fin du
96   * tableau de ce dernier. Cette fonction modifie le membre `end` de `self`.
97   *
98   * \param[out] t_self Objet \ref darray à la fin duquel on souhaite ajouter un
99   * nouvel élément
100  * \param[in] t_elem Élément que l'on souhaite ajouter à la fin de `self`
101  */
102  void darrayPushBack(darray *t_self, void *t_elem) {
103      darrayInsert(t_self, t_self->end, t_elem);
104  }
105
106  /**
107   * `darrayPopBack` permet de supprimer le dernier élément de l'objet \ref
108   * darray passé en argument. Cette fonction modifie le membre `end` de ce
109   * dernier objet.
110   *
111   * \param[out] t_self Objet dont on souhaite supprimer le dernier élément
112   */
113  void darrayPopBack(darray *t_self) {
114      darrayErase(t_self, (char *)t_self->end - t_self->element_size);
115  }
116

```



```

117 /**
118  * `darrayDelete` supprime tous les éléments contenus par l'objet \ref darray
119  * passé en argument avant de libérer la mémoire occupée par l'objet lui-même.
120  * L'objet passé en argument ne sera plus utilisable après utilisation de cette
121  * fonction.
122  *
123  * \param[out] t_self Objet \ref darray à supprimer
124  */
125 void darrayDelete(darray *t_self) {
126     free(t_self->begin);
127     free(t_self);
128 }
129
130 /**
131  * `darraySize` renvoie le nombre d'éléments contenu dans le \ref darray
132  * `self` passé en arguments. Cette fonction ne modifie pas l'élément passé en
133  * argument.
134  *
135  * \param[out] t_self Objet \ref darray dont on souhaite connaître le nombre
136  * d'éléments
137  * \return Nombre d'éléments contenus dans `self`
138  */
139 uint64_t darraySize(darray *t_self) {
140     return (uint64_t)((char *)t_self->end - (char *)t_self->begin) /
141         t_self->element_size;
142 }
143
144 /**
145  * `darrayGet` permet de récupérer un élément d'un objet \ref darray grâce à
146  * son index dans le tableau de l'objet `self`. Si l'index est trop grand, alors
147  * le pointeur `NULL` sera renvoyé, sinon un pointeur de type `void*` pointant
148  * sur l'élément correspondant sera renvoyé. Cette fonction ne modifie pas
149  * l'objet `self`.
150  * \param[out] t_self Objet \ref darray duquel on souhaite obtenir un pointeur
151  * sur l'élément à l'index `idx`
152  * \param[in] t_idx Index de l'élément que l'on souhaite récupérer
153  * \return Pointeur de type `void*` pointant sur l'élément si l'index est
154  * valide, sur NULL sinon.
155  */
156 void *darrayGet(darray *t_self, uint64_t t_idx) {
157     if (t_idx >= darraySize(t_self)) {
158         fprintf(stderr, "Error in `darrayGet`, out of bound idx: %zu (max: %zu)\n",
159             t_idx, darraySize(t_self));
160         exit(PTR_ERROR);
161     }
162     void *itr;
163     itr = (char *)t_self->begin + t_idx * t_self->element_size;
164     return itr;
165 }

```