Rapport du TP 6 d'IGI-2001

Rémi Nicole

Chapitre 1

Exercice 1

Listing 1.1 – Programme

```c
#include <stdio.h>
#include <string.h>

int main() {
    char nom[20] = "", prenom[20];
    int age;
    FILE* f = fopen("exo1.f.out", "w");

    if(f == NULL) {
        printf("Could not open file\n");
        return 1;
    }

    while(strcmp(nom, "FIN") != 0) {
        printf("Quel est votre nom ?\n");
        scanf("%s", nom);
        if(strcmp(nom, "FIN") == 0)
            break;
        printf("Quel est votre prénom ?\n");
        scanf("%s", prenom);
        printf("Quel est votre age ?\n");
        scanf("%i", &age);

        fprintf(f, "Nom: %s , Prénom: %s , Age: %i\n", nom, prenom, age);

    }

    fclose(f);

    return 0;
}
```

Listing 1.2 – Résultat

```
Quel est votre nom ?
Skywalker
Quel est votre prénom ?
Luke
```

```
Quel est votre age ?
1000
Quel est votre nom ?
Vader
Quel est votre prénom ?
Darth
Quel est votre age ?
1030
Quel est votre nom ?
Nicole
Quel est votre prénom ?
Rémi
Quel est votre age ?
18
Quel est votre nom ?
FIN
```

Listing 1.3 – Fichier de sortie

```
Nom: Skywalker , Prénom: Luke , Age: 1000
Nom: Vader , Prénom: Darth , Age: 1030
Nom: Nicole , Prénom: Rémi , Age: 18
```

Chapitre 2

Exercice 2

Listing 2.1 – Programme

```c
#include <stdio.h>

int main() {
    FILE* f = fopen("exo1.f.out", "r");

    if(f == NULL) {
        printf("Could not open file\n");
        return 1;
    }

    char nom[20], prenom[20];
    int age;

    while(fscanf(f, "Nom: %s , Prénom: %s , Age: %i\n", nom, prenom, &age) == 3)
        printf("Nom:\t%s\nPrénom:\t%s\nAge:\t%i\n", nom, prenom, age);

    fclose(f);
    return 0;
}
```

Listing 2.2 – Résultat

```
Nom:    Skywalker
Prénom: Luke
Age:    1000
Nom:    Vader
Prénom: Darth
Age:    1030
Nom:    Nicole
Prénom: Rémi
Age:    18
```

Chapitre 3

Exercice 3

Listing 3.1 – Programme

```c
#include <stdio.h>

int main() {
    FILE* f = fopen("exo1.f.out", "r");
    char c = 0;

    if(f == NULL) {
        printf("Could not open file\n");
        return 1;
    }

    do {
        c = fgetc(f);
        printf("%c, %i\n", c, c);
    } while(c != EOF);

    fclose(f);

    return 0;
}
```

Listing 3.2 – Résultat

```
N, 78
o, 111
m, 109
:, 58
 , 32
S, 83
k, 107
y, 121
w, 119
a, 97
l, 108
k, 107
e, 101
r, 114
 , 32
```

```
,, 44
 , 32
P, 80
r, 114
, -61
, -87
n, 110
o, 111
m, 109
:, 58
 , 32
L, 76
u, 117
k, 107
e, 101
 , 32
,, 44
 , 32
A, 65
g, 103
e, 101
:, 58
 , 32
1, 49
0, 48
0, 48
0, 48

, 10
N, 78
o, 111
m, 109
:, 58
 , 32
V, 86
a, 97
d, 100
e, 101
r, 114
 , 32
,, 44
 , 32
P, 80
r, 114
, -61
, -87
n, 110
o, 111
m, 109
:, 58
 , 32
D, 68
a, 97
r, 114
t, 116
h, 104
```

```
 , 32
,, 44
 , 32
A, 65
g, 103
e, 101
:, 58
 , 32
1, 49
0, 48
3, 51
0, 48

, 10
N, 78
o, 111
m, 109
:, 58
 , 32
N, 78
i, 105
c, 99
o, 111
l, 108
e, 101
 , 32
,, 44
 , 32
P, 80
r, 114
, -61
, -87
n, 110
o, 111
m, 109
:, 58
 , 32
R, 82
, -61
, -87
m, 109
i, 105
 , 32
,, 44
 , 32
A, 65
g, 103
e, 101
:, 58
 , 32
1, 49
8, 56

, 10
, -1
```

Chapitre 4

Exercice 4

Listing 4.1 – Programme

```
1  #include <X11/Xlib.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <string.h>
6  #include <sysexits.h>
7  #include <math.h>
8  #include <time.h>
9
10 #ifndef DEFAULT_WIDTH
11 //! @brief The width used if "-random" is given in the program parameters
12 #define DEFAULT_WIDTH 200
13 #endif
14
15 #ifndef DEFAULT_HEIGHT
16 //! @brief The height used if "-random" is given in the program parameters
17 #define DEFAULT_HEIGHT 200
18 #endif
19
20 //! @brief 16 millions colors in the RRGGBB color space
21 #define COLOR_COUNT 16777216
22
23 enum programMode {
24     random,
25     bmp,
26     rle
27 };
28
29 typedef unsigned char byte;
30
31 //! @brief A structure containing the colors of a pixel
32 struct pixel {
33     //! The 3 bytes of colors of the pixel (in the order of the BMP file)
34     byte pixelBytes[4];
35 };
36
37 //! @brief A structure containing an allocated color
38 struct colorAllocation {
```

```
39      XColor color;   //!< The allocated color
40      byte allocated; //!< Equals to 1 if the color has been allocated, 0 else
41  };
42
43  //! The allocated X11 colors
44  struct colorAllocation allocatedColors[COLOR_COUNT];
45
46  // Print Usage <<
47  /*! @brief Print the command line arguments for this program
48   *
49   * @param programName the name used to invoke this program
50   */
51  void printUsage(const char* programName) {
52      fprintf(stderr, "Usage:\n");
53      fprintf(stderr, "   %s --random [width height] [--toroidal]\n", programName);
54      fprintf(stderr, "   %s --bmp file.bmp [--toroidal]\n", programName);
55      fprintf(stderr, "   %s --rle file.rle [--toroidal]\n", programName);
56  }
57  // >>
58  // Get Hex Code <<
59  /*! @brief Return the hexadecimal code of a pixel
60   *
61   * It returns the hexadecimal code (RRGGBB) of the given pixel.
62   *
63   * @param p the pixel from which the color is wanted
64   * @param red the position of the red color code
65   * @param green the position of the green color code
66   * @param blue the position of the blue color code
67   * @return The hexadecimal code of the color
68   */
69  char* getHexCode(const struct pixel p, const unsigned short int red,
70                   const unsigned short int green, const unsigned short int blue) {
71      char* colorHexCode = (char*)malloc(8*sizeof(char));
72      char redHexCode[3], greenHexCode[3], blueHexCode[3];
73      if(p.pixelBytes[red] < 0x10)
74          sprintf(redHexCode, "0%X", p.pixelBytes[red]);
75      else
76          sprintf(redHexCode, "%X", p.pixelBytes[red]);
77
78      if(p.pixelBytes[green] < 0x10)
79          sprintf(greenHexCode, "0%X", p.pixelBytes[green]);
80      else
81          sprintf(greenHexCode, "%X", p.pixelBytes[green]);
82
83      if(p.pixelBytes[blue] < 0x10)
84          sprintf(blueHexCode, "0%X", p.pixelBytes[blue]);
85      else
86          sprintf(blueHexCode, "%X", p.pixelBytes[blue]);
87
88      sprintf(colorHexCode, "#%s%s%s", redHexCode, greenHexCode, blueHexCode);
89      return colorHexCode;
90  }
91  // >>
92  // Get Int Code <<
93  /*! @brief Return the int code of a pixel
94   *
```

```
 * It returns the int code (RRGGBB convert from hexadecimal to decimal)
 * of the given pixel.
 *
 * @param p the pixel from which the color is wanted
 * @param red the position of the red color code
 * @param green the position of the green color code
 * @param blue the position of the blue color code
 * @return The int code of the color
 */
unsigned int getIntCode(const struct pixel p, const unsigned short int red,
                        const unsigned short int green,
                        const unsigned short int blue) {
    // intCode = red × 256^2 + green × 256 + blue
    return p.pixelBytes[red]*65536 + p.pixelBytes[green]*256
            + p.pixelBytes[blue];
}
// >>
// Is RLE Data <<
/*! @brief returns 1 if the given character is an RLE data caracter
 *
 * A character is considered a RLE data character if and only if it is an number
 * or a 'b' or 'o' or a '$'.
 *
 * @param c the character to be checked
 * @return 1 if it is a RLE data character, 0 else
 */
byte isRLEdata(const char c) {
    return ('0' <= c && c <= '9') || c == 'b' || c == 'o' || c == '$';
}
// >>
// Is Allocated <<
/*! @brief Return 1 if the color is already allocated, 0 else
 *
 * @param intCode the int code of the color to be checked
 * @return 1 if the color is already allocated, 0 else
 */
byte isAllocated(const unsigned int intCode) {
    return allocatedColors[intCode].allocated == 1;
}
// >>
// Affiche <<
/*! @brief Display an array of pixels into an X11 window
 *
 * @param dpy the display used
 * @param w the window to be drawn over
 * @param gc the graphical context to use
 * @param tab the array to draw
 * @param width the width of the array
 * @param height the height of the array
 * @param red the position of the red color code
 * @param green the position of the green color code
 * @param blue the position of the blue color code
 */
void affiche(Display *dpy, Window w, GC gc, const struct pixel* const tab,
             const unsigned int width, const unsigned int height,
             const unsigned short int red, const unsigned short int green,
```

```
151             const unsigned short int blue) {
152     unsigned int i , j ;
153 
154     for(i = 0 ; i < width ; i++) {
155         for(j = 0 ; j < height ; j++) {
156             const struct pixel p = tab[i + (j*width)];
157 
158             char* const colorHexCode = getHexCode(p, red, green, blue);
159             const int   colorIntCode = getIntCode(p, red, green, blue);
160             XColor color;
161             const Colormap cmap = DefaultColormap(dpy, 0);
162             XParseColor(dpy, cmap, colorHexCode, &color);
163             free(colorHexCode);
164 
165             if(!isAllocated(colorIntCode)) {
166                 XAllocColor(dpy, cmap, &color);
167                 allocatedColors[colorIntCode].color = color;
168                 allocatedColors[colorIntCode].allocated = 1;
169             } else {
170                 color = allocatedColors[colorIntCode].color;
171             }
172 
173             XSetForeground(dpy, gc, color.pixel);
174             XDrawPoint(dpy, w, gc, i, j);
175         }
176     }
177 
178     XFlush(dpy);
179 }
180 // >>
181 // Init Randomly <<
182 /*! @brief Randomly initialize an array of pixels
183  *
184  * A pixel have a probability of $\dfrac{1}{5}$ not to be black.
185  *
186  * @param tab the array of pixels to be filled
187  * @param tabWidth the width of the array
188  * @param tabHeight the height of the array
189  */
190 void initRandomly(struct pixel* const tab, const unsigned int tabWidth,
191                   const unsigned int tabHeight) {
192     unsigned int i,j;
193     for(i = 0 ; i < tabWidth * tabHeight ; ++i) {
194         if(rand() % 5) {
195             for(j = 0 ; j <= 3 ; ++j)
196                 tab[i].pixelBytes[j] = 0;
197         } else {
198             for(j = 0 ; j <= 3 ; ++j)
199                 tab[i].pixelBytes[j] = rand() % 256;
200         }
201     }
202 }
203 // >>
204 // Init From BMP <<
205 /*! @brief Initialize an array of pixel with the pixels of a BMP file
206  *
```

```
 * The file must already be opened. See https://en.wikipedia.org/wiki/BMP_file
 *
 * @param tab the array of pixels to be filled
 * @param f the opened file containing the BMP data
 * @param pixelStart the start of the pixel array inside the BMP file
 * @param pixelEnd the end of the pixel array inside the BMP file
 * @param bpp the number of bytes per pixels in the BMP file
 * @param paddingSize the size of the padding of the BMP file
 * @param tabWidth the width of the array
 * @param tabHeight the height of the array
 */
void initFromBMP(struct pixel* const tab, FILE* const f,
                    const unsigned int pixelStart, const unsigned int pixelEnd,
                    const unsigned short int bpp,
                    const unsigned short int paddingSize,
                    const unsigned int tabWidth, const unsigned int tabHeight) {

    fseek(f, pixelStart, SEEK_SET);

    // BMP starts at bottom left
    unsigned int row = tabHeight - 1, col = 0;
    while (ftell(f) < pixelEnd) {
        // Store the pixel into the tab at the right place
        fread(&tab[col + (row*tabWidth)], bpp, 1, f);

        // Then go until the last column
        col = (col == tabWidth - 1)? 0 : col + 1;
        // And after the last column, go one row up and restart at column 0
        if(col == 0) {
            --row;
            // And skip the padding
            fseek(f, paddingSize, SEEK_CUR);
        }
    }
}
// >>
// Init From RLE <<
/*! @brief Initialize an array of pixel with an RLE file
 *
 * The file must already be opened. See http://www.conwaylife.com/wiki/RLE
 *
 * @param tab the array of pixels to be filled
 * @param f the opened file containing the RLE data
 * @param tabWidth the width of the array
 * @param tabHeight the height of the array
 */
void initFromRLE(struct pixel* const tab, FILE* const f,
                 const unsigned int tabWidth, const unsigned int tabHeight) {
    struct pixel* pattern;
    const struct pixel black = {{0, 0, 0, 0}};
    unsigned int patternWidth = 0, patternHeight = 0,
                 x = 0, y = 0;

    fseek(f, 0, SEEK_END);
    unsigned int end = ftell(f);
    fseek(f, 0, SEEK_SET);
```

```
263
264     do {
265         const unsigned int lineBeginPos = ftell(f);
266         const char firstLineCharacter = fgetc(f);
267         if(firstLineCharacter == '#')
268             while(fgetc(f) != '\n');
269
270         // Get the height and the width of the pattern
271         if(firstLineCharacter == 'x' && patternWidth == 0 && patternHeight == 0) {
272             fscanf(f, "%*[ ]=%*[ ]%u,%*[ ]y%*[ ]=%*[ ]%u",
273                     &patternWidth, &patternHeight);
274             pattern = malloc(patternWidth*patternHeight * sizeof(struct pixel));
275             while(fgetc(f) != '\n');
276         }
277
278         // Beginning of a RLE data line
279         if(isRLEdata(firstLineCharacter)) {
280
281             if(patternWidth == 0 || patternHeight == 0) {
282                 fprintf(stderr, "No non-null pattern width or height given\n");
283                 exit(EX_DATAERR);
284             } else {
285                 fseek(f, lineBeginPos, SEEK_SET);
286                 unsigned int cellCount = 0;
287                 char c = fgetc(f);
288
289                 if(isRLEdata(c)) {
290                     while('0' <= c && c <= '9') {
291                         // Convert c to number
292                         cellCount = (cellCount * 10) + (c - '0');
293                         c = fgetc(f);
294                     }
295                     cellCount = (cellCount == 0)? 1 : cellCount;
296                     // 'b' means dead cell
297                     if(c == 'b') {
298                         for(unsigned int i = 0 ; i < cellCount ; ++i, ++x) {
299                             pattern[x + (y * patternWidth)] = black;
300                         }
301                     // 'o' means alive cell
302                     } else if(c == 'o') {
303                         for(unsigned int i = 0 ; i < cellCount ; ++i, ++x) {
304                             /* pattern[x + (y * patternWidth)] = white; */
305                             for(unsigned int j = 0 ; j <= 3 ; ++j)
306                                 pattern[x + (y * patternWidth)].pixelBytes[j] = rand() % 256;
307                         }
308                     // '$' means end of line (can take a count before)
309                     } else if(c == '$') {
310                         for(unsigned int i = 0 ; i < cellCount ; ++i) {
311                             // Fill the rest of the line
312                             while(x < patternWidth) {
313                                 pattern[x + (y * patternWidth)] = black;
314                                 ++x;
315                             }
316                             ++y;
317                             x = 0;
318                         }
```

```
                    } else {
                        fprintf(stderr, "Error: expected 'b', 'o' or '$' in RLE file\n");
                        exit(EX_DATAERR);
                    }
                // '!' means end of data. Everything else can be ignored
                } else if(c == '!') {
                    // Fill the rest
                    while(y < patternHeight) {
                        while(x < patternWidth) {
                            pattern[x + (y * patternWidth)] = black;
                            ++x;
                        }
                        ++y;
                        x = 0;
                    }
                    break;
                }

            }

        }
    } while(ftell(f) != end);

    // Centering of the pattern in the window
    const unsigned int left = (tabWidth / 2) - (patternWidth / 2);
    const unsigned int top = (tabHeight / 2) - (patternHeight / 2);

    for(x = 0 ; x < tabWidth ; ++x) {
        for(y = 0 ; y < tabHeight ; ++y) {
            tab[x + (y * tabWidth)] = black;
        }
    }

    // Copy the pattern in the array
    for(x = left ; x < left + patternWidth ; ++x) {
        for(y = top ; y < top + patternHeight ; ++y) {
            tab[x + (y * tabWidth)] = pattern[x - left + ((y-top) * patternWidth)];
        }
    }

    free(pattern);
}
// >>
// Color Position <<
/*! @brief Return the position of a color considering its bitmask
 *
 * @param colorMask the bitmask of the color
 * @return The position of the color
 */
unsigned short int colorPosition(const unsigned int colorMask) {
    // Position = \log_{256}\left(\frac{mask}{255}\right)
    return log(colorMask/255.)/log(256);
}
// >>
// Is Alive <<
```

```
374 /*! @brief Returns 1 if the given pixel is considered "alive"
375  *
376  * A pixel is considered alive when its color is different from black
377  *
378  * @param p the pixel to be tested
379  * @param red the position of the red color code
380  * @param green the position of the green color code
381  * @param blue the position of the blue color code
382  * @return 1 if the pixel is considered "alive", 0 else
383  */
384 unsigned short int isAlive(const struct pixel p, const unsigned short int red,
385                            const unsigned short int green,
386                            const unsigned short int blue) {
387     return !(p.pixelBytes[red] == 0 && p.pixelBytes[green] == 0
388             && p.pixelBytes[blue] == 0);
389 }
390 // >>
391 // Neighbour Count <<
392 /*! @brief Count the number of "alive neighbours" surrounding a pixel
393  *
394  * It counts the number of "cells" considered "alive" of a pixel's 8
395  * surrounding locations.
396  *
397  * @param tab the array of pixels
398  * @param x the x location of the pixel
399  * @param y the y location of the pixel
400  * @param tabWidth the width of the array
401  * @param tabHeight the height of the array
402  * @param red the position of the red color code
403  * @param green the position of the green color code
404  * @param blue the position of the blue color code
405  * @return The number of neighbours of the given pixel
406  */
407 unsigned short int neighbourCount(const struct pixel* const tab,
408                                   const unsigned int x, const unsigned int y,
409                                   const unsigned int tabWidth,
410                                   const unsigned int tabHeight,
411                                   const byte toroidal,
412                                   const unsigned short int red,
413                                   const unsigned short int green,
414                                   const unsigned short int blue) {
415     const unsigned int xStart = (x == 0)? 0 : x - 1,
416                        yStart = (y == 0)? 0 : y - 1,
417                        xEnd = (x == tabWidth - 1)? tabWidth - 1 : x + 1,
418                        yEnd = (y == tabHeight - 1)? tabHeight - 1 : y + 1;
419     unsigned short int count = 0;
420     for(unsigned int i = xStart ; i <= xEnd ; ++i)
421         for(unsigned int j = yStart ; j <= yEnd ; ++j)
422             if(isAlive(tab[i + (j * tabWidth)], red, green, blue))
423                 ++count;
424 
425     if(toroidal) {
426         if(x == 0 || x == tabWidth - 1) {
427             const unsigned int wrapCol = (x == 0)? tabWidth - 1 : 0;
428             for(unsigned int i = yStart ; i <= yEnd ; ++i)
429                 if(isAlive(tab[wrapCol + (i * tabWidth)], red, green, blue))
```

```
                    ++count;
        }

        if(y == 0 || y == tabHeight - 1) {
            const unsigned int wrapRow = (y == 0)? tabHeight - 1 : 0;
            for(unsigned int i = xStart ; i <= xEnd ; ++i)
                if(isAlive(tab[i + (wrapRow * tabWidth)], red, green, blue))
                    ++count;
        }

        if((x == 0 || x == tabWidth - 1) && (y == 0 || y == tabHeight - 1)) {
            const unsigned int wrapCol = (x == 0)? tabWidth - 1 : 0;
            const unsigned int wrapRow = (y == 0)? tabHeight - 1 : 0;
            if(isAlive(tab[wrapCol + (wrapRow * tabWidth)], red, green, blue))
                ++count;
        }
    }

    return (isAlive(tab[x+(y*tabWidth)],red,green,blue))? count - 1 : count;
}
// >>
// Mix Neighbours Colors <<
/*! @brief Return a color which is a mix of all the neighbours colors
 * of a given pixel
 *
 * It uses the mean of the red, green and blue channels of the colors of the
 * neighbours.
 *
 * @param tab the array of pixels
 * @param x the x location of the pixel
 * @param y the y location of the pixel
 * @param tabWidth the width of the array
 * @param tabHeight the height of the array
 * @param red the position of the red color code
 * @param green the position of the green color code
 * @param blue the position of the blue color code
 * @return The mix of all the neighbours colors
 */
struct pixel mixNeighboursColors(const struct pixel* const tab,
                                 const unsigned int x, const unsigned int y,
                                 const unsigned int tabWidth,
                                 const unsigned int tabHeight,
                                 const byte toroidal,
                                 const unsigned short int red,
                                 const unsigned short int green,
                                 const unsigned short int blue) {
    const unsigned int xStart = (x == 0)? 0 : x - 1,
                       yStart = (y == 0)? 0 : y - 1,
                         xEnd = (x == tabWidth)? tabWidth : x + 1,
                         yEnd = (y == tabHeight)? tabHeight : y + 1;
    const unsigned short int count = neighbourCount(tab, x, y, tabWidth, tabHeight,
                                                    toroidal, red, green, blue);
    unsigned short int redMean = 0, greenMean = 0, blueMean = 0;

    for(unsigned int i = xStart ; i <= xEnd ; ++i)
        for(unsigned int j = yStart ; j <= yEnd ; ++j)
```

```
            if(isAlive(tab[i + (j * tabWidth)], red, green, blue)) {
                redMean   += tab[i + (j * tabWidth)].pixelBytes[red];
                greenMean += tab[i + (j * tabWidth)].pixelBytes[green];
                blueMean  += tab[i + (j * tabWidth)].pixelBytes[blue];
            }

    if(toroidal) {
        if(x == 0 || x == tabWidth - 1) {
            const unsigned int wrapCol = (x == 0)? tabWidth - 1 : 0;
            for(unsigned int i = yStart ; i <= yEnd ; ++i)
                if(isAlive(tab[wrapCol + (i * tabWidth)], red, green, blue)) {
                    redMean   += tab[wrapCol + (i * tabWidth)].pixelBytes[red];
                    greenMean += tab[wrapCol + (i * tabWidth)].pixelBytes[green];
                    blueMean  += tab[wrapCol + (i * tabWidth)].pixelBytes[blue];
                }
        }

        if(y == 0 || y == tabHeight - 1) {
            const unsigned int wrapRow = (y == 0)? tabHeight - 1 : 0;
            for(unsigned int i = xStart ; i <= xEnd ; ++i)
                if(isAlive(tab[i + (wrapRow * tabWidth)], red, green, blue)) {
                    redMean   += tab[i + (wrapRow * tabWidth)].pixelBytes[red];
                    greenMean += tab[i + (wrapRow * tabWidth)].pixelBytes[green];
                    blueMean  += tab[i + (wrapRow * tabWidth)].pixelBytes[blue];
                }
        }

        if((x == 0 || x == tabWidth - 1) && (y == 0 || y == tabHeight - 1)) {
            const unsigned int wrapCol = (x == 0)? tabWidth - 1 : 0;
            const unsigned int wrapRow = (y == 0)? tabHeight - 1 : 0;
            if(isAlive(tab[wrapCol + (wrapRow * tabWidth)], red, green, blue)) {
                redMean   += tab[wrapCol + (wrapRow * tabWidth)].pixelBytes[red];
                greenMean += tab[wrapCol + (wrapRow * tabWidth)].pixelBytes[green];
                blueMean  += tab[wrapCol + (wrapRow * tabWidth)].pixelBytes[blue];
            }
        }
    }

    struct pixel mean;
    mean.pixelBytes[red] = redMean / count;
    mean.pixelBytes[green] = greenMean / count;
    mean.pixelBytes[blue] = blueMean / count;
    return mean;
}
// >>
// Next Step <<
/*! @brief Compute the next step of the given array of pixel considering
 * the laws of Conway's Game of Life
 *
 * The rules are:
 * - Any live cell with fewer than two live neighbours dies,
 *   as if caused by under-population.
 * - Any live cell with two or three live neighbours lives on
 *   to the next generation.
 * - Any live cell with more than three live neighbours dies,
 *   as if by overcrowding.
```

```
 * - Any dead cell with exactly three live neighbours becomes a live cell,
 *   as if by reproduction.
 *
 * @param tab the array of pixels
 * @param tabWidth the width of the array
 * @param tabHeight the height of the array
 * @param red the position of the red color code
 * @param green the position of the green color code
 * @param blue the position of the blue color code
 */
void nextStep(struct pixel* const tab, const unsigned int tabWidth,
            const unsigned int tabHeight, const byte toroidal,
            const unsigned short int red, const unsigned short int green,
            const unsigned short int blue) {
    struct pixel tabTmp[tabWidth * tabHeight];
    // Tab in which the changes are made before applied
    for(unsigned int i = 0 ; i < tabWidth ; i++) {
        for(unsigned int j = 0 ; j < tabHeight ; j++) {
            const unsigned short int neighboursCount = neighbourCount(tab, i, j,
                                                        tabWidth, tabHeight,
                                                        toroidal,
                                                        red, green, blue);
            if(isAlive(tab[i + (j * tabWidth)], red, green, blue)) {
                // Loneliness and overcrowding
                if(neighboursCount < 2 || neighboursCount > 3) {
                    tabTmp[i + (j * tabWidth)].pixelBytes[red]   = 0;
                    tabTmp[i + (j * tabWidth)].pixelBytes[green] = 0;
                    tabTmp[i + (j * tabWidth)].pixelBytes[blue]  = 0;
                } else
                    tabTmp[i + (j * tabWidth)] = tab[i + (j * tabWidth)];

            } else {
                // Reproduction
                if(neighboursCount == 3)
                    tabTmp[i + (j * tabWidth)] = mixNeighboursColors(tab, i, j,
                                                        tabWidth, tabHeight,
                                                        toroidal,
                                                        red, green, blue);
                else {
                    tabTmp[i + (j * tabWidth)].pixelBytes[red]   = 0;
                    tabTmp[i + (j * tabWidth)].pixelBytes[green] = 0;
                    tabTmp[i + (j * tabWidth)].pixelBytes[blue]  = 0;
                }
            }
        }
    }
    memcpy(tab, tabTmp, sizeof(struct pixel) * tabWidth * tabHeight);
}
// >>

int main (int argc, char const* argv[]) {

    FILE* f;
    unsigned short int bpp, red = 2, green = 1, blue = 0;
    unsigned int pixelStart, pixelEnd;
    byte toroidal = 0;
```

```
598
599     for(int i = 0 ; i < COLOR_COUNT ; ++i)
600         allocatedColors[i].allocated = 0;
601
602     srand(time(NULL));
603
604     // ====== Check command-line usage ====== <<
605     for(int i = 0 ; i < argc ; ++i) {
606         if(!strcmp(argv[i], "--toroidal")) {
607             toroidal = 1;
608             --argc;
609             break;
610         }
611     }
612
613     if(argc == 1 || argc > 4) {
614         printUsage(argv[0]);
615         return EX_USAGE;
616     } else if(argc == 2 || argc == 4) {
617         if(strcmp(argv[1], "--random")) {
618             printUsage(argv[0]);
619             return EX_USAGE;
620         }
621     } else if(argc == 3) {
622         // Neither "-bmp" nor "-rle"
623         if(strcmp(argv[1], "--bmp") && strcmp(argv[1], "--rle")) {
624             printUsage(argv[0]);
625             return EX_USAGE;
626         }
627     }
628     // >>
629
630     enum programMode mode;
631     if(!strcmp(argv[1], "--random")) {
632         mode = random;
633     } else if(!strcmp(argv[1], "--bmp")) {
634         mode = bmp;
635     } else {
636         mode = rle;
637     }
638
639     unsigned int width, height;
640
641     if(mode == bmp) {
642         // BMP <<
643         f = fopen(argv[2], "rb");
644
645         // ====== Check if readable file ====== <<
646         if(f == NULL) {
647             fprintf(stderr, "Error: Could not load the file \"%s\"\n", argv[2]);
648             return EX_NOINPUT;
649         }
650         // >>
651         // ====== BMP file magic check ====== <<
652         // If it really is a BMP, the first two bytes are the ASCII code of "BM"
653         char magic[3];
```

```
654             // Read the first two bytes one time starting from the magic memory block
655             fread(&magic, 2, 1, f);
656             if (!(magic[0] == 'B' && magic[1] == 'M')) {
657                 fprintf(stderr, "Error: Not a BMP file\n");
658                 return EX_DATAERR;
659             }
660             // >>
661             // ====== Width and Height ====== <<
662             // width and height located at offsets 0x12 and 0x16
663             fseek(f, 0x12, SEEK_SET);
664             // Load the width and height into the corresponding variables
665             // (4 bytes each)
666             fread(&width, 4, 1, f);
667             fread(&height, 4, 1, f);
668 
669             printf("Height = %u, Width = %u\n", height, width);
670             // >>
671             // ====== Pixel array offset ====== <<
672             fseek(f, 0xA, SEEK_SET);
673             fread(&pixelStart, 4, 1, f);
674             // Go to the end
675             fseek(f, 0, SEEK_END);
676             pixelEnd = ftell(f);
677             printf("Pixel array starts at offset: %X and ends at offset: %X\n",
678                     pixelStart, pixelEnd);
679             // >>
680             // ====== Bytes per pixels ====== <<
681             // The number of bytes per pixels located at offset 0x1C
682             fseek(f, 0x1C, SEEK_SET);
683             fread(&bpp, 2, 1, f);
684             // Convert to bytes
685             bpp /= 8;
686 
687             printf("Number of bytes per pixels: %hu\n", bpp);
688             if(bpp == 4) {
689                 fprintf(stderr, "Warning: Alpha channel will not be displayed.\n");
690                 unsigned int redMask, greenMask, blueMask;
691                 fseek(f, 0x36, SEEK_SET);
692                 fread(&redMask, 4, 1, f);
693                 red = colorPosition(redMask);
694                 fread(&greenMask, 4, 1, f);
695                 green = colorPosition(greenMask);
696                 fread(&blueMask, 4, 1, f);
697                 blue = colorPosition(blueMask);
698             } else if(bpp != 3) {
699                 fprintf(stderr, "Error: Only RGB and RGBA file supported.\n");
700                 return EX_DATAERR;
701             }
702             // >>
703             // >>
704         } else if(mode == random) {
705             // Random <<
706             if(argc == 2)
707                 width = DEFAULT_WIDTH, height = DEFAULT_WIDTH;
708             else
709                 width = atoi(argv[2]), height = atoi(argv[3]);
```

```
        // >>
    } else {
        // RLE <<
        f = fopen(argv[2], "r");

        // ====== Check if readable file ======
        if(f == NULL) {
            fprintf(stderr, "Error: Could not load the file \"%s\"\n", argv[2]);
            return EX_NOINPUT;
        }

        width = DEFAULT_WIDTH, height = DEFAULT_WIDTH;
        // >>
    }

    struct pixel pic[width*height];

    if(mode == bmp) {
        // BMP <<
        // ====== Padding Size ====== <<

        // There is a padding column until the row reaches a multiple of 4 bytes
        // bytesPerRow = rowSize × bytesPerPixel
        // paddingSize = { 0                    if bytesPerRow%4 = 0
        //               { 4 − (bytesPerRow%4)  else
        const unsigned short int paddingSize = (4 - ((width * bpp) % 4) ) % 4;
        // >>
        // Read pixel array
        initFromBMP(pic, f, pixelStart, pixelEnd, bpp, paddingSize, width, height);
        fclose(f);
        // >>
    } else if(mode == random) {
        // Random <<
        initRandomly(pic, width, height);
        // >>
    } else {
        // RLE <<
        initFromRLE(pic, f, width, height);
        // >>
    }

    // ====== X11 initialization ====== <<
    XEvent e;
    Display* const dpy = XOpenDisplay(NULL);
    const int noir = BlackPixel(dpy, DefaultScreen(dpy));
    const Window w = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy), 0, 0,
            width, height, 0, noir, noir);
    XMapWindow(dpy, w);
    const GC gc = XCreateGC(dpy,w,0,NULL);
    XSelectInput(dpy, w, StructureNotifyMask);

    while (e.type != MapNotify)
        XNextEvent(dpy, &e);
    // >>

```

```
764     affiche(dpy, w, gc, pic, width, height, red, green, blue);
765     for(;;) {
766         nextStep(pic, width, height, toroidal, red, green, blue);
767         affiche(dpy, w, gc, pic, width, height, red, green, blue);
768     }
769     sleep(5);
770 
771     return 0;
772 }
773 // vim: fdm=marker:fmr=«<,»>
```

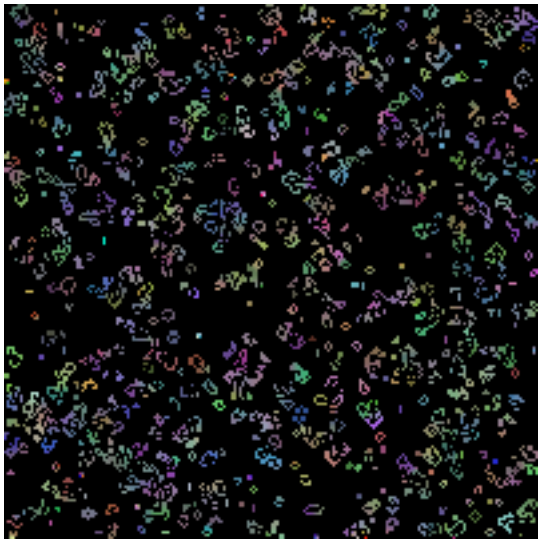

FIGURE 4.1 – Capture d'écran du résultat de l'exercice 4 avec initialisation aléatoire

4.1 Génération aléatoire

Pour la génération aléatoire, il a fallu utiliser une probabilité pondérée car il est considéré que un pixel non noir est "vivant". Ainsi, un pixel a une probabilité de $\frac{1}{5}$ d'être "vivant" et chaque canal correspondant à soit la couleur rouge, verte ou bleue est initialisée aléatoirement (entre `0x0` et `0xFF`, soit dans l'intervalle $[\![0; 256[\![$). On utilise donc la fonction `srand(time(NULL))` pour initialiser la séquence de nombre pseudo aléatoires avec le nombre de secondes écoulées depuis 1970 et on utilise `rand() % 256` pour obtenir un nombre aléatoire dans l'intervalle $[\![0; 256[\![$.

4.2 Calcul du lendemain

Pour le calcul du "jour d'après", il a fallu créer un tableau temporaire contenant le résultat avant de le copier entièrement (avec `memcopy`) dans le tableau original pour éviter que les "cellules" qui étaient vivante le jour d'avant et sont devenue mortes ne compte pas comme voisin et inversement.

De plus, en raison de la gestion de couleur qui a été ajouté dans le programme, il a fallu déterminer la couleur des "cellules naissantes". Ainsi, une fonction `mixNeighbousColors` a été crée afin de faire un "mélange" des couleurs des voisins de la cellule naissante. Pour ce faire, il a suffit de récupérer chaque canaux de couleurs (autre que l'alpha), de faire une moyenne de ces canaux des 3 "voisins" et de les rassembler faisant ainsi une nouvelle couleur qui est un mélange de ses "parents".

4.3 Améliorations

4.3.1 Utilisation de la ligne de commande

Afin d'avoir le contrôle de l'exécution du programme sans avoir à le recompiler, la ligne de commande a été utilisée. Ainsi, le programme prends un paramètre : soit l'option `--random` qui génèrera une image à couleurs aléatoire de taille définie par les `#define` dans le code source à moins que 2 chiffres correspondant respectivement à la largeur et à la hauteur de la fenêtre ne soient donnés après l'option `--random`, soit

l'option `--bmp` suivit d'un chemin (relatif ou absolu) d'une image sous le format BMP qui sera analysée et utilisée comme configuration initiale pour le Jeu de la Vie ou encore l'option `--rle` suivit d'un chemin d'un fichier `.rle` contenant des données RLE d'un motif du Jeu de la Vie (cf. 4.3.3). Il est aussi possible d'activer le monde toroïdal (cf. 4.3.5) en ajoutant l'option `--toroidal` à la fin de la ligne de commande.

4.3.2 Analyse d'un fichier BMP

Afin d'analyser un fichier BMP, il a fallu passer par certaines étapes, notamment la récupération de la taille, la détection du nombre d'octets par pixel, et autres. [1]

On notera tout de même que le premier pixel du tableau de pixel de l'image est le pixel en bas à gauche de la représentation graphique de l'image.

Vérification magique du fichier BMP

La première chose faite après vérification de la lisibilité du fichier lorsqu'un paramètre autre que l'option `--random` est passé au programme est vérifier si le nom du fichier correspond bien à un fichier BMP, non pas au niveau de l'extension du fichier, mais au niveau des nombres magiques. En effet, les deux premiers octets d'un fichier correspondant à une image BMP sont `0x42` et `0x4D`, ce qui correspond au code ASCII des lettres "B" et "M". C'est ce que font les lignes 399–408 du programme.

Récupération de la largeur de hauteur de l'image

Afin de récupérer la largeur et la hauteur de l'image, les 4×2 octets aux offsets `0x12` et `0x16` contenant respectivement la largeur et la hauteur de l'image sont récupérés. Ainsi, si les l'octet d'offset de `0x12` à `0x15` inclus il est contenu `80 02 00 00` soit `00 00 02 80` en big-endian, ce la signifie que l'image est de largeur `0x280` soit 640.

Début et fin du tableau de pixel

Le début du tableau de pixel dans le fichier binaire du BMP est à un offset qui est stocké l'offset `0xA`. Il est donc récupéré dans la variable `unsigned int pixelStart`. Afin de récupérer la fin du tableau de pixel, la fonction `fseek` est utilisée pour aller à la fin du fichier et grâce à la fonction `ftell`, l'offset de fin de fichier est récupéré et stocké dans la variabel `unsigned int pixelEnd`.

Nombres d'octets par pixel

Le nombre de bits par pixel peut varier en fonction de l'image (et notamment à cause du canal alpha). Pour palier à cela, cette valeur est stockée à l'offset `0x1C`. Elle est ensuite convertie en octets pour être exploité plus directement.

Dans le cas où il y a 3 octets par pixel, il n'y a pas de canal alpha et l'ordre des canaux de couleurs est bleu, vert et ensuite rouge, de par le fait que les données sont stockées en little-endian.

Dans le cas où il y a 4 octets par pixel, le canal alpha est présent et l'ordre des canaux change en fonction du logiciel ou de la personne qui a créé le bitmap. Afin de connaître l'ordre des canaux, des masques sont fournis de l'offset `0x36` (masque du canal rouge) à l'offset `0x42` (masque du canal alpha), bien sûr de taille 4 octets et stocké en little-endian.

1. voir l'article Wikipedia

Afin d'utiliser le même code pour les images possédant 3 octets par pixel et 4 octets par pixel, les variables `unsigned short int red` `unsigned short int green` et `unsigned short int blue` sont utilisées et contiennent un chiffre correspondant à leur place dans les 3 ou 4 octets du pixel. Pour calculer leur place dans les octets du pixel, il faut récupérer la place du `0xFF` dans le masque parmi les autres `0x00`. On divise donc le masque par `0xFF` ou 255 et il n'y a plus qu'a obtenir le nombre de fois que cette valeur a été multiplié par `0x100` soit $16 \times 16 = 256$. On fera donc $\log_{256}\left(\frac{masque}{255}\right) = \frac{\log\left(\frac{masque}{255}\right)}{\log(256)}$

Calcul du padding du tableau de pixel

La partie tableau de pixels du fichier de l'image BMP possède un "padding" qui fait en sorte que à chaque fin de ligne de l'image, des données sont ajoutées afin que le nombre d'octets de chaque ligne soient un multiple de 4. On a donc :

$$padding = \begin{cases} 0 & \text{si } nbOctetsParLigne \equiv 0[4] \\ 4 - (nbOctetsParLigne \mod 4) & \text{sinon} \end{cases} \tag{4.1}$$

Ce qui est équivalent à :

$$padding = \big(4 - (tailleLigne * nbOctetsParPixels \mod 4)\big) \mod 4 \tag{4.2}$$

Ainsi, à chaque fin de ligne, il faut "sauter" le padding pour pouvoir accéder à la ligne suivante grâce à la ligne `fseek(fichier, padding, SEEK_CUR)`.

4.3.3 Analyse d'un fichier RLE

Afin de pouvoir rentrer des motifs de manière plus simple que de rentrer chaque pixel un par un ou d'essayer le mode aléatoire de manière continue, un fichier `.rle` peut être fournis et va être analysé par le programme. [2]

Commentaires

Il y a 5 types de commentaires dans les fichiers `.rle` et sont donc ignorés :

Commentaire	Description
#C ou #c	Commentaire classique
#N	Nom du motif
#O	Auteur du motif
#P ou #R	Coordonnées du coin haut gauche (ignorées car le motif est placé au centre de la fenêtre)
#r	Fournis les règles du Jeu de la Vie pour ce motif (ignorées)

TABLE 4.1 – Formes de commentaires d'un fichier `.rle`

Largeur et hauteur du motif

La largeur et la hauteur du motif est généralement stocké après les commentaires dans un ligne de type :
`x = m, y = n`
Cette ligne est donc lue par le programme et il en est déduit après la position du motif dans la fenêtre en calculant la coordonnées de la cellule haut gauche grâce à la formule :

2. Voir l'article sur ConwayLife

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{LargeurFentre}{2} - \frac{LargeurMotif}{2} \\ \frac{HauteurFentre}{2} - \frac{HauteurMotif}{2} \end{pmatrix} \tag{4.3}$$

Données des "cellules"

Généralement après les commentaires et obligatoirement après la taille du motif, les données concernant les "cellules" sont stockées et interprétables grâce à trois symboles et des chiffres :

Indicateur	Description
b	Cellule vivante
o	Cellule morte
$	Fin de la ligne (sous entendu le reste des cellules de la ligne sont des cellules mortes)

TABLE 4.2 – Identificateur de vie des cellules dans un fichier `.rle`

Chacun des identificateurs du tableau 4.2 peuvent prendre un compteur comme argument (qui sera placé avant le dit indicateur). Cela impliquera que l'indicateur sera répété autant de fois qu'indiqué par le compteur.

La fin des données du fichier `.rle` est indiqué par le symbole " !" qui sous entend comme le symbole "$" que le reste des "cellules" sont des "cellules mortes".

4.3.4 Allocation des couleurs

Afin de ne pas allouer à nouveau de la mémoire pour des couleurs ayant de la mémoire déjà allouée, la variable globale `allocatedColors` est utilisée. Il s'agit d'un tableau de structure contenant la couleur (de type `XColor`) et un chiffre, 1 si la couleur a déjà de la mémoire allouée, 0 sinon. Le tableau est trié de telle manière que l'élément n°$n \in [\![0; 256^3[\![$ soit la couleur `#XXXXXX` avec `XXXXXX` la représentation de n en base 16.

4.3.5 Monde toroïdal

Puisque le programme ne supporte pas le fait d'avoir un monde infini ou extensible à l'infini, il a été décidé que toutes cellules hors du tableau sont considérées comme "mortes". Cependant, une alternative est possible : il s'agit de considéré le tableau à deux dimension contenant les cellules (ou le monde) comme toroïdal, c'est à dire que les bords du tableau sont connectées. Ainsi, une cellule située à la bordure droite du monde peut influencer une cellule à la bordure gauche. Afin d'arriver à ces fins, des conditions ont été rajoutées dans la fonction du calcul du nombre de voisins et dans la fonctions calculant le mélange des couleurs des cellules voisines. Afin d'activer ce type de monde, l'option ligne de commande `--toroidal` a été rajoutée.

4.4 Bugs / améliorations possibles

4.4.1 Performances

Malgré le fait que les couleurs ne soient allouées qu'une seule fois (cf. 4.3.4) l'allocation des couleurs prends beaucoup de temps, impliquant que le fait d'allouer une couleur uniquement lorsqu'elle est nouvelle au moment de l'affichage fait baisser les performances du programmes.

Une autre manière d'améliorer les performances du programme serait d'utiliser le multi-threading pour le calcul du "lendemain" en divisant le tableau en plusieurs parties. Cependant, cette amélioration pourrait se trouver coûteuse en mémoire : en effet, afin que les tableaux résultant de chaque thread n'influence pas les threads toujours en train de traiter le tableau, il faudrait faire une copie du tableau original pour chaque thread. Un autre manière de contrer ce problème serait de recalculer le lendemain pour les parties critiques. En effet, puisque le traitement tableau est divisé, la "mauvaise influence" des autres threads n'est possible qu'au bordures de ces divisions (d'autant plus si le monde est toroïdal).

4.4.2 Monde toroïdal

Un bug est présent lorsque l'option `--toroidal` est activé, il s'agit du mauvais calcul du mélange des couleurs des voisins lorsqu'une "cellule naissante" naît sur une bordure avec des parents de l'autre côté de la bordure. Biens que le calcul du mélange ne soit pas correct, les couleurs restent cohérentes entre elles (elles ne sont pas pour autant aléatoires).

4.4.3 Format RLE

L'implémentation du format RLE dans le programme supporte la taille du motif. Cependant, lorsque la taille du motif est supérieure à la taille de la fenêtre par défaut (200×200), rien n'est affiché.

Certaines fonctionnalités du format RLE, comme les définitions des règles pour le jeu de la vie, ou encore les coordonnées du coins haut gauche, ne sont pas implémentées.

4.4.4 Ligne de commande

Les options de la lignes de commandes ne sont pas échangeables avec l'option `--toroidal`. Par exemple, la ligne `./exo4.bin --toroidal --random` ne marchera pas.

4.4.5 Boucle infinie

La boucle infinie utilisée n'est pas pseudo-infinie, impliquant que pour quitter le programme, il faut soit lui envoyer un signal `SIGINT`, `SIGTERM` ou `SIGKILL` ou encore fermer la fenêtre affichant le jeu de la vie, ce qui fera quitter le programme avec une erreur. Une manière de palier à cela serait d'écouter des XEvents et d'attendre l'appui d'une touche particulière (par exemple `q`) ou encore de gérer la fermeture de la fenêtre de manière plus orthodoxe.

4.4.6 Affichage

Il arrive que la librairie X11 refuse d'exécuter une requête (ce qui se voit avec le message d'erreur lorsque la fenêtre X11 est fermée), ce qui aura pour conséquence des pixels qui auront leurs état inchangé. Par exemple, une cellule vivante auparavant qui doit mourir sera toujours affichée comme vivante si X11 refuse d'accéder à la requête et inversement. Il faudrait donc trouver un moyen de détecter si X11 a bien exécute la requête en cours et si non, la réitérer.

Au niveau de la fréquence d'images par secondes (fps), elle est déterminée par la performance du programme. Cela implique que la vitesse de changement d'état n'est pas forcément constante et c'est d'autant plus vrai car le nombre de couleurs déjà allouées augmente drastiquement au cours du programme, augmentent par la suite le nombre d'image par seconde au cours du temps. Afin de régler ce problème, l'utilisation de `nanosleep` est possible (bien plus précise que `sleep`) ou encore mieux, faire une difference de timestamp en millisecondes (avec la structure retournée par `gettimeofday`) pour assurer une fréquence d'image constante quand bien même avec des calculs de durée différente.