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]\n", programName);
54      fprintf(stderr, "   %s --bmp file.bmp\n", programName);
55      fprintf(stderr, "   %s --rle file.rle\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(struct pixel p, unsigned short int red,
70                   unsigned short int green, 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(struct pixel p, unsigned short int red,
                        unsigned short int green, 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 '$'.
 *
 * @param c the character to be checked
 * @return 1 if it is a RLE data character, 0 else
 */
byte isRLEdata(char c) {
    return ('0' <= c && c <= '9') || c == 'b' || c == 'o' || c == '$';
}
// >>
// 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(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, struct pixel* tab,
             unsigned int width, unsigned int height, unsigned short int red,
             unsigned short int green, unsigned short int blue) {
    unsigned int i , j ;
```

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

```
 * @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* tab, FILE* f,
                    unsigned int pixelStart, unsigned int pixelEnd,
                    unsigned short int bpp, unsigned short int paddingSize,
                    unsigned int tabWidth, 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* tab, FILE* f,
                 unsigned int tabWidth, unsigned int tabHeight) {
    struct pixel* pattern;
    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);

    do {
        unsigned int lineBeginPos = ftell(f);
```

```
263             char firstLineCharacter = fgetc(f);
264             if(firstLineCharacter == '#')
265                 while(fgetc(f) != '\n');
266 
267             // Get the height and the width of the pattern
268             if(firstLineCharacter == 'x' && patternWidth == 0 && patternHeight == 0) {
269                 fscanf(f, "%*[ ]=%*[ ]%u,%*[ ]y%*[ ]=%*[ ]%u",
270                         &patternWidth, &patternHeight);
271                 pattern = malloc(patternWidth*patternHeight * sizeof(struct pixel));
272                 while(fgetc(f) != '\n');
273             }
274 
275             // Beginning of a RLE data line
276             if(isRLEdata(firstLineCharacter)) {
277 
278                 if(patternWidth == 0 || patternHeight == 0) {
279                     fprintf(stderr, "No non-null pattern width or height given\n");
280                     exit(EX_DATAERR);
281                 } else {
282                     fseek(f, lineBeginPos, SEEK_SET);
283                     unsigned int cellCount = 0;
284                     char c = fgetc(f);
285 
286                     if(isRLEdata(c)) {
287                         while('0' <= c && c <= '9') {
288                             // Convert c to number
289                             cellCount = (cellCount * 10) + (c - '0');
290                             c = fgetc(f);
291                         }
292                         cellCount = (cellCount == 0)? 1 : cellCount;
293                         // 'b' means dead cell
294                         if(c == 'b') {
295                             for(unsigned int i = 0 ; i < cellCount ; ++i, ++x) {
296                                 pattern[x + (y * patternWidth)] = black;
297                             }
298                         // 'o' means alive cell
299                         } else if(c == 'o') {
300                             for(unsigned int i = 0 ; i < cellCount ; ++i, ++x) {
301                                 /* pattern[x + (y * patternWidth)] = white; */
302                                 for(unsigned int j = 0 ; j <= 3 ; ++j)
303                                     pattern[x + (y * patternWidth)].pixelBytes[j] = rand() % 256;
304                             }
305                         // '$' means end of line (can take a count before)
306                         } else if(c == '$') {
307                             for(unsigned int i = 0 ; i < cellCount ; ++i) {
308                                 // Fill the rest of the line
309                                 while(x < patternWidth) {
310                                     pattern[x + (y * patternWidth)] = black;
311                                     ++x;
312                                 }
313                                 ++y;
314                                 x = 0;
315                             }
316                         } else {
317                             fprintf(stderr, "Error: expected 'b', 'o' or '$' in RLE file\n");
318                             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
    unsigned int left = (tabWidth / 2) - (patternWidth / 2);
    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(unsigned int colorMask) {
    // Position = \log_{256}\left(\frac{mask}{255}\right)
    return log(colorMask/255.)/log(256);
}
// >>
// Is Alive <<
/*! @brief Returns 1 if the given pixel is considered "alive"
 *
 * A pixel is considered alive when its color is different from black
```

```
 *
 * @param p the pixel to be tested
 * @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 1 if the pixel is considered "alive", 0 else
 */
unsigned short int isAlive(struct pixel p, unsigned short int red,
                           unsigned short int green, unsigned short int blue) {
    return !(p.pixelBytes[red] == 0 && p.pixelBytes[green] == 0
            && p.pixelBytes[blue] == 0);
}
// >>
// Neighbour Count <<
/*! @brief Count the number of "alive neighbours" surrounding a pixel
 *
 * It counts the number of "cells" considered "alive" of a pixel's 8
 * surrounding locations.
 *
 * @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 number of neighbours of the given pixel
 */
unsigned short int neighbourCount(struct pixel* tab,
                           unsigned int x, unsigned int y,
                           unsigned int tabWidth, unsigned int tabHeight,
                           unsigned short int red, unsigned short int green,
                           unsigned short int blue) {
    unsigned int xStart = (x == 0)? 0 : x - 1,
                 yStart = (y == 0)? 0 : y - 1,
                 xEnd = (x == tabWidth - 1)? tabWidth - 1 : x + 1,
                 yEnd = (y == tabHeight - 1)? tabHeight - 1 : y + 1;
    unsigned short int count = 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)
                && !(i == x && j ==y))
                ++count;
    return 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 mixNeighbousColors(struct pixel* tab,
                                unsigned int x, unsigned int y,
                                unsigned int tabWidth, unsigned int tabHeight,
                                unsigned short int red, unsigned short int green,
                                unsigned short int blue) {
    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,
                      n = 0;
    unsigned short int count = neighbourCount(tab, x, y, tabWidth, tabHeight,
                                              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];
                ++n;
            }

    redMean /= count;
    greenMean /= count;
    blueMean /= count;
    struct pixel mean;
    mean.pixelBytes[red] = redMean;
    mean.pixelBytes[green] = greenMean;
    mean.pixelBytes[blue] = blueMean;
    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* tab, unsigned int tabWidth, unsigned int tabHeight,
              unsigned short int red, unsigned short int green,
              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++) {
            unsigned short int neighboursCount = neighbourCount(tab, i, j,
                                                                tabWidth, tabHeight,
                                                                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)] = mixNeighbousColors(tab, i, j,
                                                                    tabWidth, tabHeight,
                                                                    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;

    for(int i = 0 ; i < COLOR_COUNT ; ++i)
        allocatedColors[i].allocated = 0;

    srand(time(NULL));

    // ====== Check command-line usage ====== <<
    if(argc == 1 || argc > 4) {
```

```
542             printUsage(argv[0]);
543             return EX_USAGE;
544         } else if(argc == 2 || argc == 4) {
545             if(strcmp(argv[1], "--random")) {
546                 printUsage(argv[0]);
547                 return EX_USAGE;
548             }
549         } else if(argc == 3) {
550             // Neither "-bmp" nor "-rle"
551             if(strcmp(argv[1], "--bmp") && strcmp(argv[1], "--rle")) {
552                 printUsage(argv[0]);
553                 return EX_USAGE;
554             }
555         }
556         // >>
557 
558         enum programMode mode;
559         if(!strcmp(argv[1], "--random")) {
560             mode = random;
561         } else if(!strcmp(argv[1], "--bmp")) {
562             mode = bmp;
563         } else {
564             mode = rle;
565         }
566 
567         unsigned int width, height;
568 
569         if(mode == bmp) {
570             // BMP <<
571             f = fopen(argv[2], "rb");
572 
573             // ====== Check if readable file ====== <<
574             if(f == NULL) {
575                 fprintf(stderr, "Error: Could not load the file \"%s\"\n", argv[2]);
576                 return EX_NOINPUT;
577             }
578             // >>
579             // ====== BMP file magic check ====== <<
580             // If it really is a BMP, the first two bytes are the ASCII code of "BM"
581             char magic[3];
582             // Read the first two bytes one time starting from the magic memory block
583             fread(&magic, 2, 1, f);
584             if (!(magic[0] == 'B' && magic[1] == 'M')) {
585                 fprintf(stderr, "Error: Not a BMP file\n");
586                 return EX_DATAERR;
587             }
588             // >>
589             // ====== Width and Height ====== <<
590             // width and height located at offsets 0x12 and 0x16
591             fseek(f, 0x12, SEEK_SET);
592             // Load the width and height into the corresponding variables
593             // (4 bytes each)
594             fread(&width, 4, 1, f);
595             fread(&height, 4, 1, f);
596 
597             printf("Height = %u, Width = %u\n", height, width);
```

```
598             // >>
599             // ====== Pixel array offset ====== <<
600             fseek(f, 0xA, SEEK_SET);
601             fread(&pixelStart, 4, 1, f);
602             // Go to the end
603             fseek(f, 0, SEEK_END);
604             pixelEnd = ftell(f);
605             printf("Pixel array starts at offset: %X and ends at offset: %X\n",
606                     pixelStart, pixelEnd);
607             // >>
608             // ====== Bytes per pixels ====== <<
609             // The number of bytes per pixels located at offset 0x1C
610             fseek(f, 0x1C, SEEK_SET);
611             fread(&bpp, 2, 1, f);
612             // Convert to bytes
613             bpp /= 8;
614 
615             printf("Number of bytes per pixels: %hu\n", bpp);
616             if(bpp == 4) {
617                 fprintf(stderr, "Warning: Alpha channel will not be displayed.\n");
618                 unsigned int redMask, greenMask, blueMask;
619                 fseek(f, 0x36, SEEK_SET);
620                 fread(&redMask, 4, 1, f);
621                 red = colorPosition(redMask);
622                 fread(&greenMask, 4, 1, f);
623                 green = colorPosition(greenMask);
624                 fread(&blueMask, 4, 1, f);
625                 blue = colorPosition(blueMask);
626             } else if(bpp != 3) {
627                 fprintf(stderr, "Error: Only RGB and RGBA file supported.\n");
628                 return EX_DATAERR;
629             }
630             // >>
631             // >>
632         } else if(mode == random) {
633             // Random <<
634             if(argc == 2)
635                 width = DEFAULT_WIDTH, height = DEFAULT_WIDTH;
636             else
637                 width = atoi(argv[2]), height = atoi(argv[3]);
638             // >>
639         } else {
640             // RLE <<
641             f = fopen(argv[2], "r");
642 
643             // ====== Check if readable file ======
644             if(f == NULL) {
645                 fprintf(stderr, "Error: Could not load the file \"%s\"\n", argv[2]);
646                 return EX_NOINPUT;
647             }
648 
649             width = DEFAULT_WIDTH, height = DEFAULT_WIDTH;
650             // >>
651         }
652 
653         struct pixel pic[width*height];
```

```
654
655     if(mode == bmp) {
656         // BMP «<
657         // ====== Padding Size ====== «<
658
659         // There is a padding column until the row reaches a multiple of 4 bytes
660         // bytesPerRow = rowSize × bytesPerPixel
661         // paddingSize = { 0                    if bytesPerRow%4 = 0
                           { 4 − (bytesPerRow%4)  else
662         unsigned short int paddingSize = (4 - ((width * bpp) % 4) ) % 4;
663         // »>
664         // Read pixel array
665         initFromBMP(pic, f, pixelStart, pixelEnd, bpp, paddingSize, width, height);
666         fclose(f);
667         // »>
668     } else if(mode == random) {
669         // Random «<
670         initRandomly(pic, width, height);
671         // »>
672     } else {
673         // RLE «<
674         initFromRLE(pic, f, width, height);
675         // »>
676     }
677
678     // ====== X11 initialization ====== «<
679     XEvent e;
680     Display *dpy = XOpenDisplay(NULL);
681     int noir = BlackPixel(dpy, DefaultScreen(dpy));
682     Window w = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy), 0, 0,
683             width, height, 0, noir, noir);
684     XMapWindow(dpy, w);
685     GC gc = XCreateGC(dpy,w,0,NULL);
686     XSelectInput(dpy, w, StructureNotifyMask);
687
688     while (e.type != MapNotify)
689         XNextEvent(dpy, &e);
690     // »>
691
692     affiche(dpy, w, gc, pic, width, height, red, green, blue);
693     for(;;) {
694         sleep(.5);
695         nextStep(pic, width, height, red, green, blue);
696         affiche(dpy, w, gc, pic, width, height, red, green, blue);
697     }
698     sleep(5);
699
700     return 0;
701 }
702 // vim: fdm=marker:fmr=«<,»>
```

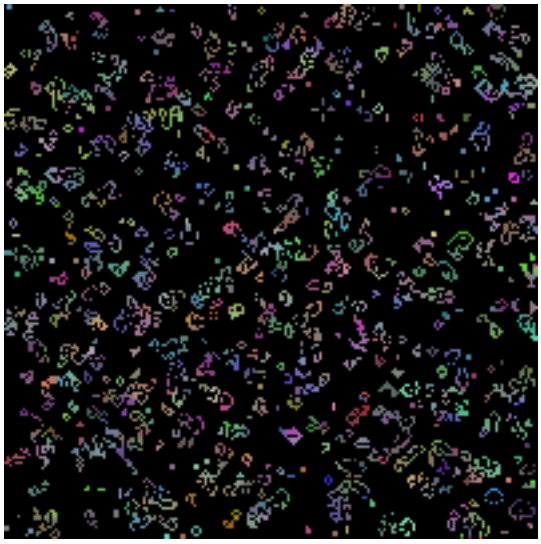

FIGURE 4.1 – Capture d'écran du résultat de l'exercice 4 avec initialisation aléatoire

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).

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.