

```

1  /**
2  * @file colorsBack.c
3  * Command handling
4  */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <time.h>
10 #include <ctype.h>
11 #include "error.h"
12 #include "utils.h"
13 #include "defines.h"
14 #include "playGame.h"
15 #include "colorsBack.h"
16
17 bool movePiece ( game_t * game, int argc, char ** argv, char * msg );
18 bool save      ( game_t * game, int argc, char ** argv, char * msg );
19 bool undo      ( game_t * game, int argc, char ** argv, char * msg );
20 bool quit      ( game_t * game, int argc, char ** argv, char * msg );
21 bool help      ( game_t * game, int argc, char ** argv, char * msg );
22 bool roflcopter( game_t * game, int argc, char ** argv, char * msg );
23
24 typedef struct{
25     char com[ MAX_COM_LEN ];
26     bool (* func )( game_t *, int, char **, char * );
27 } command_t;
28
29 const command_t commands[] = {
30     {"[", movePiece},
31     {"save", save},
32     {"undo", undo},
33     {"quit", quit},
34     {"help", help},
35     {"ROFLcopter", roflcopter}
36 };
37
38
39 /**
40 * Parses the command and calls the corresponding function.
41 *
42 * @param game    contains all information about current game
43 * @param s        contains the command line about to be processed
44 * @param[out] msg its an output containing the type of error
45 *
46 * @return false if there is an error, otherwise true
47 *
48 * @see editDistance()
49 * @see movePiece()
50 * @see save()
51 * @see undo()
52 * @see quit()
53 * @see help()
54 * @see ROFLcopter()
55 */
56
57 bool
58 newCommand( game_t * game, const char * s, char * msg )
59 {
60     int i;
61     bool sol;
62     int argc;
63     char ** argv;

```

```

64
65     if( errorCode() != NOERROR ){
66         sprintf( msg, "%s", errorMessage( errorCode() ) );
67         return false;
68     }
69
70     argv = newMatrix( MAX_ARGS, MAX_COM_LEN );
71
72     if( errorCode() != NOERROR ){
73         sprintf( msg, "%s", errorMessage( errorCode() ) );
74         return false;
75     }
76
77     msg[0] = 0;
78     // parse the command
79     argc = sscanf( s, "%s %s %s %s %s %s %s %s %s %s",
80                 argv[0], argv[1], argv[2], argv[3], argv[4],
81                 argv[5], argv[6], argv[7], argv[8], argv[9] );
82
83     if( argc < 1 ){
84         freeMatrix( argv, MAX_ARGS );
85         return true;
86     }
87
88     sol = false;
89     sprintf( msg, "Unknown command" );
90     double auxsim, maxsim = 0;
91     int maxi = 0;
92
93     //call the appropriate function
94     for( i = 0 ; i < sizeof(commands)/ sizeof(command_t) ; i++ ){
95         if( strcmp( argv[0], commands[i].com, strlen(commands[i].com) ) == 0
96             && !isalpha( argv[0][strlen(commands[i].com)] ) ){
97             maxsim = 0;
98             msg[0] = 0;
99             sol = commands[i].func( game, argc, argv, msg );
100             if( errorCode() != NOERROR ){
101                 sprintf( msg, "%s", errorMessage( errorCode() ) );
102                 sol = false;
103             }
104             break;
105         }
106         // check for command similarity
107         if( ( auxsim = editDistance( argv[0], commands[i].com ) ) > maxsim ){
108             maxsim = auxsim;
109             maxi = i;
110         }
111     }
112     // if not succesful and there was a command similar enough
113     if( maxsim >= MIN_SIMILARITY )
114         sprintf( msg+15, "\nDid you mean: \"%s\"", commands[maxi].com );
115
116     freeMatrix( argv, MAX_ARGS );
117
118     if( errorCode() != NOERROR ){
119         sprintf( msg, "%s", errorMessage( errorCode() ) );
120         sol = false;
121     }
122
123     return sol;
124 }
125
126

```

```

127 /**
128  * Checks if there is a valid path between (@a x1,@a y1) and (@a x2,@a y2)
129  * in the board.
130  *
131  * @param game    contains all information about current game
132  * @param x1      initial x coordinate
133  * @param y1      initial y coordinate
134  * @param x2      final x coordinate
135  * @param y2      final y coordinate
136  *
137  * @return true if there is a valid path, otherwise false
138  */
139
140 bool
141 areConnected( game_t * game, int x1, int y1, int x2, int y2 )
142 {
143     // BFS to find minimum path
144     struct coord{
145         int x,y;
146     } move[4] = { {-1,0}, {0,1}, {1,0}, {0,-1} };
147
148     struct node{
149         int x, y;
150     } queue[ game->players[ game->state.next ].board.emptySpots + 1 ];
151
152     int read = -1, write = 0, x, y, i;
153
154     bool touched[ game->options.height ][ game->options.width ];
155
156     memset( &touched[0][0], false, sizeof(touched) * sizeof(bool) );
157     queue[write++] = (struct node){x1,y1};
158
159     while( ++read < write ){
160         if( queue[read].x == x2 && queue[read].y == y2 )
161             break;
162         for( i = 0 ; i < sizeof(move)/sizeof(struct coord) ; i++ ){
163             x = queue[read].x + move[i].x;
164             y = queue[read].y + move[i].y;
165             if( entre( 0, x, game->options.width )
166                 && entre( 0, y, game->options.height )
167                 && game->players[ game->state.next ].board.matrix[y][x] == 0
168                 && !touched[y][x] ){
169                 touched[y][x] = true;
170                 queue[write++] = (struct node){x,y};
171             }
172         }
173     }
174     return read < write;
175 }
176
177
178 /**
179  * Moves token from a position to another checking for winning plays (lines).
180  *
181  * @param game    contains all information about current game
182  * @param argc    size of @a argv
183  * @param argv    parameters followinf the command
184  * @param[out] msg output with message to write in the panel
185  *
186  * @return false if some message is to be written
187  *
188  * @see areConected()
189  * @see randFill()

```

```

190 * @see winningPlay()
191 */
192
193 bool
194 movePiece( game_t * game, int argc, char ** argv, char * msg )
195 {
196     int i;
197     char s[ argc * MAX_ARGS ];
198
199     s[0] = 0;
200     for( i = 0 ; i < argc ; i++ )
201         strcat( s, argv[i] );
202
203     int x1, y1, x2, y2;
204     i = sscanf( s, "[%d,%d][%d,%d]", &y1, &x1, &y2, &x2 );
205     if( i < 4 ){
206         sprintf( msg, "Format error:\n"
207                 "Must be: [ row_1, column_1 ][ row_2, column_2 ]" );
208         return false;
209     }
210     if( ! entre( 0, y1, game->options.height ) ){
211         sprintf( msg, "Rank error:\nThe first row must belong to the "
212                 "interval [0,%d]", game->options.height - 1 );
213         return false;
214     }
215     if( ! entre( 0, x1, game->options.width ) ){
216         sprintf( msg, "Rank error:\nThe first column must belong to the "
217                 "interval [0,%d]", game->options.width - 1 );
218         return false;
219     }
220     if( ! entre( 0, y2, game->options.height ) ){
221         sprintf( msg, "Rank error:\nThe second row must belong to the "
222                 "interval [0,%d]", game->options.height - 1 );
223         return false;
224     }
225     if( ! entre( 0, x2, game->options.width ) ){
226         sprintf( msg, "Rank error:\nThe second column must belong to the "
227                 "interval [0,%d]", game->options.width - 1 );
228         return false;
229     }
230     if( game->players[ game->state.next ].board.matrix[y1][x1] == 0 ){
231         sprintf( msg, "The origin position must not be empty" );
232         return false;
233     }
234     if( game->players[ game->state.next ].board.matrix[y2][x2] != 0 ){
235         sprintf( msg, "The target position must not be occupied" );
236         return false;
237     }
238     if( !areConnected( game, x1, y1, x2, y2 ) ){
239         sprintf( msg, "There must be a path of unoccupied spaces from the "
240                 "origin position to the target position" );
241         return false;
242     }
243     // maintain undo
244     game->players[ game->state.next ].canUndo = true;
245
246     copyMatrix( game->players[ game->state.next ].lastBoard. matrix,
247                 game->players[ game->state.next ].board.matrix,
248                 game->options.height, game->options.width );
249
250     game->players[ game->state.next ].lastBoard.points =
251         game->players[ game->state.next ].board.points;
252

```

```

253 game->players[ game->state.next ].lastBoard.emptySpots =
254         game->players[ game->state.next ].board.emptySpots;
255 // maintain board
256 game->players[ game->state.next ].board.matrix[y2][x2] =
257         game->players[ game->state.next ].board.matrix[y1][x1];
258
259 game->players[ game->state.next ].board.matrix[y1][x1] = 0;
260
261 // if made a line, erase it and count points (winning play)
262 if( ! winningPlay( game, game->state.next, x2, y2, true ) ){
263     // else, randFill()
264     randFill( game, game->state.next, game->options.tokensPerTurn, false );
265     // change turn
266     i = 0;
267     do{
268         game->state.next++;
269         game->state.next %= game->numPlayers;
270         i++;
271     }while( game->players[ game->state.next ].board.emptySpots <= 0 &&
272             i <= game->numPlayers );
273 }else{
274     // print number of points just made
275     sprintf( msg, "%d point/s move",
276             game->players[ game->state.next ].board.points -
277             game->players[ game->state.next ].lastBoard.points );
278     // if board is empty
279     if( game->players[ game->state.next ].board.emptySpots >=
280         game->options.width * game->options.height ){
281
282         randFill( game, game->state.next, game->options.tokensPerTurn, true );
283         if( errorCode() != NOERROR ){
284             sprintf( msg, "%s", errorMessage( errorCode() ) );
285             return false;
286         }
287     }
288 }
289
290 return true;
291 }
292
293
294 /**
295  * Saves current game to file.
296  *
297  * @param game      contains all information about current game
298  * @param argc      size of @a argv
299  * @param argv      parameters followinf the command
300  * @param[out] msg  output with message to write in the panel
301  *
302  * @return false if some message is to be written
303  *
304  * @see writeGame()
305  */
306
307 bool
308 save( game_t * game, int argc, char ** argv, char * msg )
309 {
310     if( argc > 2 || strcmp( "save", argv[0] ) != 0 ){
311         sprintf( msg, "Wrong usage\n"
312             "Try 'save --help' for more information");
313         return false;
314     }
315     if( argc == 1 ){

```

```

316     sprintf( msg, "Missing file operand\n"
317               "Try 'save --help' for more information");
318     return false;
319 }
320 if( strcmp( argv[1], "--help" ) == 0 ){
321     sprintf( msg, "Usage: save filename\n"
322               "Saves the current game to file 'filename'");
323     return false;
324 }
325 writeGame( game, argv[1] );
326
327 if( errorCode() != NOERROR ){
328     sprintf( msg, "%s", errorMessage( errorCode() ) );
329     return false;
330 }
331
332 return true;
333 }
334
335
336 /**
337  * Undoes the last move made.
338  *
339  * @param game      contains all information about current game
340  * @param argc      size of @a argv
341  * @param argv      parameters followinf the command
342  * @param[out] msg  output with message to write in the panel
343  *
344  * @return false if some message is to be written
345  */
346
347 bool
348 undo( game_t * game, int argc, char ** argv, char * msg )
349 {
350     if( argc == 2 && strcmp( argv[1], "--help" ) == 0 ){
351         sprintf( msg, "Usage: undo\n"
352                   "Undoes the last move\n"
353                   "It can only be used once some move has been done, "
354                   "and it can't be used two consecutive times" );
355         return false;
356     }
357     if( argc > 1 || strcmp( "undo", argv[0] ) != 0 ){
358         sprintf( msg, "Wrong usage\n"
359                   "Try 'undo --help' for more information");
360         return false;
361     }
362     if( game->options.mode != SINGLEMODE ){
363         sprintf( msg, "'undo' command is only available in one player"
364                   ", no time mode" );
365         return false;
366     }
367     if( game->players[ game->state.next ].canUndo == false ){
368         sprintf( msg, "'undo' command cannot be used twice in a row or in "
369                   "the first turn" );
370         return false;
371     }
372     game->players[ game->state.next ].canUndo = false;
373     // swap boards;
374     board_t aux = game->players[ game->state.next ].board;
375
376     game->players[ game->state.next ].board =
377         game->players[ game->state.next ].lastBoard;
378

```

```

379     game->players[ game->state.next ].lastBoard = aux;
380
381     return true;
382 }
383
384
385 /**
386  * Quits the game.
387  *
388  * @param game      contains all information about current game
389  * @param argc      size of @a argv
390  * @param argv      parameters followinf the command
391  * @param[out] msg  output with message to write in the panel
392  *
393  * @return false if some message is to be written
394  */
395
396 bool
397 quit( game_t * game, int argc, char ** argv, char * msg )
398 {
399     if( argc == 2 && strcmp( argv[1], "--help" ) == 0 ){
400         sprintf( msg, "Usage: quit\n"
401                 "Quits the current game" );
402         return false;
403     }
404     if( argc > 1 || strcmp( "quit", argv[0] ) != 0 ){
405         sprintf( msg, "Wrong usage\n"
406                 "Try 'quit --help' for more information");
407         return false;
408     }
409     game->state.quit = true;
410     return true;
411 }
412
413
414 /**
415  * Prints help for the user. Secret commands are not shown.
416  *
417  * @param game      contains all information about current game
418  * @param argc      size of @a argv
419  * @param argv      parameters followinf the command
420  * @param[out] msg  output with message to write in the panel
421  *
422  * @return false if some message is to be written
423  */
424
425 bool
426 help( game_t * game, int argc, char ** argv, char * msg )
427 {
428     if( argc == 2 && strcmp( argv[1], "--help" ) == 0 ){
429         sprintf( msg, "Usage: help\n"
430                 "Shows available commands" );
431         return false;
432     }
433     if( argc > 1 || strcmp( "help", argv[0] ) != 0 ){
434         sprintf( msg, "Wrong usage\n"
435                 "Try 'help --help' for more information");
436         return false;
437     }
438     sprintf( msg,
439             "Type 'name --help' to find out more about the function 'name'"
440             "\n\n"
441             " [row_1,column_1][row_2,column_2]\n"

```

```

442         " save filename\n"
443         " undo\n"
444         " quit\n"
445         " help\n" );
446     return true;
447 }
448
449
450 /**
451  * Secret command.
452  * Prints a ROFLcopter.
453  *
454  * @param game    contains all information about current game
455  * @param argc    size of @a argv
456  * @param argv    parameters followinf the command
457  * @param[out] msg output with message to write in the panel
458  *
459  * @return false if some message is to be written
460  */
461
462 bool
463 roflcopter( game_t * game, int argc, char ** argv, char * msg )
464 {
465     sprintf( msg,
466 #ifdef __unix__
467         "\033[5mROFL:ROFL\033[25m:LOL:\033[5mROFL:ROFL\033[25m\n"
468         "      |\n"
469         " \033[5mL\033[25m  /-----\n"
470         "\033[5mL\033[25m0\033[5mL\033[25m===      []\\\n"
471         " \033[5mL\033[25m  \033[25m  \033[25m  \033[25m  \033[25m\n"
472         "      \033[25m  \033[25m  \033[25m\n"
473         "      |      |\n"
474         "      -----/\n"
475 #else
476         ".....<ROFL ROFL ROFL ROFL>.\n"
477         ".....| |.....\n"
478         "....._|||/_.....\n"
479         ".....|'-|'-|.....\n"
480         ".....|--|--|.....\n"
481         "...../ L \033[25m  /-----\033[25m  \033[25m  \033[25m\n"
482         ".....| L O L|-----0-----,\n"
483         "...../ L /_____,---'-----, /\n"
484         "...../ /.....\033[25m  \033[25m  \033[25m  \033[25m  \033[25m\n"
485         ".....//.....\033[25m  //_____\033[25m  \033[25m  \033[25m  \033[25m\n"
486 #endif
487     );
488
489     return false;
490 }
491
```



```
1 /**
2  * @file colorsBack.h
3  * Command handling
4  */
5
6 #ifndef COLORSBACK_H
7 #define COLORSBACK_H
8
9 #include <stdbool.h>
10 #include "game.h"
11
12
13 bool
14 newCommand( game_t * game, const char * s, char * msg );
15
16
17 #endif // COLORSBACK_H
18
```

```
1 /**
2  * @file colors.c
3  * Contains console color functions depending on the OS.
4  */
5
6 #include "stdbool.h"
7 #include "defines.h"
8 #include "colors.h"
9
10 #if defined(__unix__)
11
12     #include "unixColors.c"
13
14 #elif defined(__win32__) || defined(__WIN32__) || \
15     defined(win32) || defined(WIN32) || \
16     defined(__win32__) || defined(__WIN32__) || \
17     defined(__windows__) || defined(__WINDOWS__)
18
19     #include "winColors.c"
20
21 #else
22
23     #include "noColors.c"
24
25 #endif
26
```

```

1  /**
2  * @file colorsFront.c
3  * Contains the main function with the game loop.
4  */
5
6  #include <stdlib.h>
7  #include <stdbool.h>
8  #include "error.h"
9  #include "defines.h"
10 #include "game.h"
11 #include "colorsBack.h"
12 #include "menu.h"
13 #include "userInterface.h"
14 #include "playGame.h"
15
16
17 /**
18 * This function is actually the hole game loop. It takes care of calling every
19 * other function, including the ones on the frontend, which print everything,
20 * and the ones on the back end, which process everything.
21 *
22 * @return a code to be handled by the operating system depending of the state
23 *         of the program
24 *
25 * @see menu()
26 * @see drawTable()
27 * @see drawText()
28 * @see drawPanel()
29 * @see askCommand()
30 * @see newCommand()
31 */
32
33 int
34 main()
35 {
36     game_t * game;
37     char command[ MAX_COM_LEN ];
38     char message[ MAX_ERR_LEN ];
39
40     while( menu( &game ) ){
41         if( errorCode() != NOERROR ){
42             drawPanel( errorMessage( errorCode() ) );
43             continue;
44         }
45         message[0]=0;
46         while( !game->state.quit && !gameOver( game, game->state.next ) ){
47             clearError();
48             clearScreen();
49             drawTable( game );
50             drawPanel( message );
51             askCommand( command );
52             newCommand( game, command, message );
53         }
54         if( gameOver( game, game->state.next ) ){
55             drawWinner( game );
56             clearScreen();
57             drawTable( game );
58             drawPanel("GAME OVER\nPress ENTER to return to main menu\n");
59             askCommand( command );
60         }
61         freeGame( game );
62     }
63     return EXIT_SUCCESS;

```

```
64 }  
65
```

```
1 /**
2  * @file colors.h
3  * Contains console color functions depending on the OS.
4  */
5
6 #ifndef COLORS_H
7 #define COLORS_H
8
9
10 typedef enum {
11     BLACK          ,
12     RED            ,
13     GREEN          ,
14     BROWN         ,
15     BLUE           ,
16     VIOLET         ,
17     SKY_BLUE       ,
18     LIGHT_GREY     ,
19     GRAY           ,
20     PINK           ,
21     LIGHT_GREEN    ,
22     YELLOW         ,
23     LIGHT_BLUE     ,
24     LIGHT_VIOLET   ,
25     LIGHT_SKY_BLUE ,
26     WHITE
27 } color;
28
29 typedef enum {
30     CLEAR          ,
31     NONE           ,
32
33     BOLD           ,
34     UNDERLINE     ,
35     BLINK         ,
36     INVERTED      ,
37
38     NO_BOLD        ,
39     NO_UNDERLINE   ,
40     NO_BLINK       ,
41     NO_INVERTED
42 } attr;
43
44 void
45 textColor( color c );
46
47 void
48 backColor( color c );
49
50 void
51 textAttr( attr a );
52
53 #endif // COLORS_H
54
55
56
57
```

```
1 /**
2  * @file defines.h
3  * Contains several defines
4  */
5
6 #ifndef DEFINES_H
7 #define DEFINES_H
8
9
10
11 /// maximum error length
12 #define MAX_ERR_LEN 1000
13 /// maximum command length
14 #define MAX_COM_LEN 100
15 /// maximum arguments to a command (counting the command)
16 #define MAX_ARGS 10
17
18
19 /// maximum main text lenght
20 #define MAX_TEXT 5000
21 /// maximum panel lines (a.k.a. commands history)
22 #define MAX_PANEL_LINES 25
23 /// actual number of panel lines @see drawPanel()
24 #define PANEL_LINES 10 // may be a cool variable
25
26
27 /// horizontal board lines enabled @see drawTable()
28 #define HOR_LINES true // may be a cool variable
29
30
31 /// colors enablead @see colors.c
32 #define USE_COLORS true // may be a cool variable
33
34
35 /// input options @see menu.c
36 #define MAX_MINUTES 10000 // Max playing time: 1 week: 7 * 24 * 60 = 10080
37 #define MIN_TAB_DIM 5
38 #define MAX_TAB_DIM 150 // I don't think you have a bigger monitor
39 #define MIN_COLORS 2
40 #define MAX_COLORS 9
41 #define MIN_TOK_PER_LINE 3
42
43
44 /// edit distance options @see editDistance()
45 #define MIN_SIMILARITY 0.65
46 #define MIN_EDIT_LEN 3
47 #define MAX_EDIT_LEN 35
48
49
50 /// maximum processing time for {@link randFill()}
51 #define MAX_WAITING_TIME 15
52
53
54
55 #endif // DEFINES_H
56
```

```
1 /**
2  * @file error.c
3  * Error handling
4  */
5
6 #include "error.h"
7
8 static error nError;
9
10
11 /**
12  * Clears errors.
13  *
14  * @see raiseError()
15  */
16
17 void
18 clearError()
19 {
20     nError = NOERROR;
21 }
22
23
24 /**
25  * Returns the current error code.
26  *
27  * @return error code
28  *
29  * @see raiseError()
30  */
31
32 error
33 errorCode()
34 {
35     return nError;
36 }
37
38
39 /**
40  * Raise an error.
41  *
42  * @param num    error code
43  */
44
45 void
46 raiseError( error num )
47 {
48     nError = num;
49 }
50
51
52 /**
53  * Returns the error message corresponding to the given error code.
54  *
55  * @param error error code
56  *
57  * @return error message
58  *
59  * @see raiseError();
60  */
61
62 char *
63 errorMessage( error num )
```

```
64 {
65     switch(num){
66         case NOERROR:           return "No error";
67         case ARITHMETICERROR:   return "Arithmetic error";
68         case MEMORYERROR:      return "Memory error";
69         case FILEERROR:        return "Error with file";
70         case TIMEERROR:        return "Error while getting time";
71         case INPUTERROR:       return "Error while reading input";
72         case CORRUPTFILE:      return "Input file was corrupted";
73         case COMPUTATIONALERROR: return "Computer power is not enough";
74
75         default:               return "Unknown error";
76     }
77 }
78
```



```

1 /**
2  * @file error.h
3  * Error handling
4  */
5
6 #ifndef ERROR_H
7 #define ERROR_H
8
9 // if DEBUG is defined, raise_error_if prints in screen the error, file,
10 // function & line
11 // #define DEBUG
12
13
14 // Error types
15 typedef enum{
16     NOERROR,
17     ARITHMETICERROR,
18     MEMORYERROR,
19     FILEERROR,
20     TIMEERROR,
21     INPUTERROR,
22     CORRUPTFILE,
23     COMPUTATIONALERROR
24 } error;
25
26 void
27 clearError();
28
29 error
30 errorCode();
31
32 void
33 raiseError( error num );
34
35 char *
36 errorMessage( error num );
37
38 // if "comp" is TRUE raise error "num" and return "ret"
39 #define raiseErrorIf( comp, num, ret ) \
40     do{ if(!(comp)){raiseError(num); return ret;} }while(0)
41
42
43 #ifdef DEBUG
44     #include <stdio.h>
45     #undef raiseErrorIf
46     #define raiseErrorIf( comp, num, ret ) \
47         do{ if (!(comp)){ \
48             raiseError(num); \
49             fprintf( stderr, "\nIn file %s\n%d :: %s => %s\nAssertion failed"\
50                 ": %s\n", __FILE__, __LINE__, __func__, errorMessage(num),#comp);\
51             return ret; \
52         } }while(0)
53 #endif
54
55 #endif // ERROR_H
56

```

```

1  /**
2  * @file game.c
3  * Operations for the struct @c game
4  */
5
6  #include <stdlib.h>
7  #include <stdio.h>
8  #include <time.h>
9  #include "error.h"
10 #include "utils.h"
11 #include "defines.h"
12 #include "playGame.h"
13 #include "game.h"
14
15
16 #define SAFE_FWRITE_INT( x )                                \
17     do{ int_ = (int)(x);                                     \
18         raiseErrorIf( fwrite( &int_, sizeof(int_), 1, out ), FILEERROR,);\
19     }while(0)
20
21 #define SAFE_FWRITE_CHAR( x )                                \
22     do{ char_ = (char)(x);                                   \
23         raiseErrorIf( fwrite( &char_, sizeof(char_), 1, out ), FILEERROR,);\
24     }while(0)
25
26 #define SAFE_FREAD_INT( x )                                  \
27     do{ raiseErrorIf( fread( &(x), sizeof(int), 1, in ),      \
28         feof(in) ? CORRUPTFILE : FILEERROR, NULL ); }while(0)
29
30 #define SAFE_FREAD_CHAR( x )                                  \
31     do{ raiseErrorIf( fread( &(x), sizeof(char), 1, in ),     \
32         feof(in) ? CORRUPTFILE : FILEERROR, NULL ); }while(0)
33
34
35 /**
36 * Creates a new game, based on @a options.
37 *
38 * @throws MEMORYERROR      if there was a problem while allocating memory
39 * @throws COMPUTATIONALERROR if after @b MAX_WAITING_TIME time no solution
40 *                             for {@link randFill()} has been obtained
41 *
42 * @param options    contains options about the game
43 *
44 * @return a pointer to the new game
45 *
46 * @see newMatrix()
47 * @see randfill()
48 */
49
50 game_t *
51 newGame( options_t * options )
52 {
53     int i,j;
54     game_t * sol = malloc( sizeof(game_t) );
55     raiseErrorIf( sol, MEMORYERROR, NULL );
56
57     sol->options = *options;
58     // filling the game mode
59     switch( sol->options.mode ){
60         case SINGLEMODE:
61         case TIMEMODE:
62             sol->numPlayers = 1;
63             break;

```

```

64     case MULTIPLMODE:
65         sol->numPlayers = 2;
66         break;
67     default:
68         sol->numPlayers = 0;
69         break;
70 };
71 // filling players
72 sol->players = malloc( sol->numPlayers * sizeof(player_t) );
73 if( !sol->players ){
74     free( sol );
75     raiseError( MEMORYERROR );
76     return NULL;
77 }
78 // filling the players boards
79 for( i = 0 ; i < sol->numPlayers ; i++ ){
80     sol->players[i].board.matrix=
81         newMatrix( sol->options.height, sol->options.width );
82     sol->players[i].lastBoard.matrix =
83         newMatrix( sol->options.height, sol->options.width );
84
85     if( !sol->players[i].board.matrix || !sol->players[i].lastBoard.matrix ){
86         for( j = 0 ; j <= i ; j++ ){
87             freeMatrix( sol->players[j].lastBoard.matrix, sol->options.height );
88             freeMatrix( sol->players[j].board.matrix, sol->options.height );
89         }
90         free( sol->players );
91         free( sol );
92         raiseError( MEMORYERROR );
93         return NULL;
94     }
95
96     sol->players[i].board.points = 0;
97     sol->players[i].board.emptySpots = sol->options.height * sol->options.width;
98     sol->players[i].canUndo = false;
99
100     randFill( sol, i, sol->options.initialTokens, true );
101
102     if( errorCode() != NOERROR ){
103         for( j = 0 ; j <= i ; j++ ){
104             freeMatrix( sol->players[j].lastBoard.matrix, sol->options.height );
105             freeMatrix( sol->players[j].board.matrix, sol->options.height );
106         }
107         free( sol->players );
108         free( sol );
109         return NULL;
110     }
111 }
112 // filling the starting conditions
113 sol->state.next = 0;
114 sol->state.lastTime = time(NULL);
115 sol->state.timeLeft = sol->options.initialSeconds;
116 sol->state.quit = false;
117
118 return sol;
119 }
120
121
122 /**
123  * Frees the memory reserved by {@link newGame()}.
124  *
125  * @param game the information of the game about to be freed
126  */

```

```

127 * @see newGame()
128 * @see freeMatrix()
129 */
130
131 void
132 freeGame( game_t * game )
133 {
134     int i;
135     for( i = 0 ; i < game->numPlayers ; i++ ){
136         freeMatrix( game->players[i].lastBoard.matrix, game->options.height );
137         freeMatrix( game->players[i].board.matrix, game->options.height );
138     }
139     free( game->players );
140     free( game );
141 }
142
143
144 /**
145  * Saves @a game in a file.
146  *
147  * @throws FILEERROR if there was an problem while opening/writing the file
148  *
149  * @param game contains all the data about the game
150  * @param file contains the name of file about to be saved
151  *
152  * @see readGame()
153  */
154
155 void
156 writeGame( game_t * game, const char * file )
157 {
158     int i,x,y;
159     int int_;
160     char char_;
161     //opening the file
162     FILE * out = fopen(file, "wb");
163     raiseErrorIf(out,FILEERROR,);
164
165     // saving all the content in the file
166     SAFE_FWRITE_INT( game->options.mode );
167     if( game->options.mode == TIMEMODE ){
168         time_t aux = time(NULL);
169         game->state.timeLeft -= aux - game->state.lastTime;
170         game->state.lastTime = aux;
171         SAFE_FWRITE_INT( game->state.timeLeft );
172     }else
173     if( game->options.mode == MULTIPLMODE )
174         SAFE_FWRITE_INT( game->state.next + 1 );
175     SAFE_FWRITE_INT( game->options.height );
176     SAFE_FWRITE_INT( game->options.width );
177     SAFE_FWRITE_INT( game->options.numColors );
178     SAFE_FWRITE_INT( game->options.tokensPerLine );
179     SAFE_FWRITE_INT( game->options.tokensPerTurn );
180     for( i = 0 ; i < game->numPlayers ; i++ ){
181         SAFE_FWRITE_INT( game->players[i].board.points );
182         for( y = 0 ; y < game->options.height ; y++ )
183             for( x = 0 ; x < game->options.width ; x++ )
184                 SAFE_FWRITE_CHAR( game->players[i].board.matrix[y][x] + '0' );
185     }
186     //finishing and closing the file
187     i = fclose(out);
188     raiseErrorIf(i==0,FILEERROR,);
189 }

```

```

190
191
192 /**
193  * Validates @a game, in order to prevent corrupted input files.
194  *
195  * @param game contains all the data about the game
196  *
197  * @return true if the game is valid, otherwise, false
198  *
199  * @see readGame()
200  */
201
202 static bool
203 validateGame( game_t * game )
204 {
205     int i,x,y;
206
207     if( !entre( 0, game->options.mode, 3 ) )
208         return false;
209     if( game->options.mode == TIMEMODE &&
210         !entre( 0, game->state.timeLeft, 60*MAX_MINUTES + 1 ) )
211         return false;
212     if( game->options.mode == MULTIPLMODE && !entre( 0, game->state.next, 2 ) )
213         return false;
214     else
215         if( !entre( 0, game->state.next, 1 ) )
216             return false;
217         if( !entre( MIN_TAB_DIM, game->options.width, MAX_TAB_DIM + 1 ) )
218             return false;
219         if( !entre( MIN_TAB_DIM, game->options.height, MAX_TAB_DIM + 1 ) )
220             return false;
221         if( !entre( MIN_COLORS, game->options.numColors, MAX_COLORS + 1 ) )
222             return false;
223         if( !entre( MIN_TOK_PER_LINE, game->options.tokensPerLine,
224             min( game->options.width, game->options.height ) + 1 ) )
225             return false;
226         if( !entre( 1, game->options.tokensPerTurn,
227             game->options.width * game->options.height + 1 ) )
228             return false;
229         for( i = 0 ; i < game->numPlayers ; i++ ){
230             if( game->players[i].board.points < 0 )
231                 return false;
232             for( y = 0 ; y < game->options.height ; y++ )
233                 for( x = 0 ; x < game->options.width ; x++ )
234                     if( !entre( 0, game->players[i].board.matrix[y][x],
235                         game->options.numColors +1))
236                         return false;
237         }
238         return true;
239 }
240
241
242 /**
243  * Reads a game from a file. Use {@link newGame()} to load the game.
244  *
245  * @throws MEMORYERROR if there was an problem while allocating memory
246  * @throws FILEERROR   if there was an problem while opening/reading from file
247  *
248  * @param file contains the name of the file about to be read
249  *
250  * @return a pointer to a game_t containing all the data about the game
251  *
252  * @see writeGame()

```

```

253 * @see newGame()
254 * @see validateGame()
255 */
256
257 game_t *
258 readGame( const char * file )
259 {
260     int i,x,y;
261     options_t options;
262     state_t state;
263     game_t * sol;
264
265     FILE * in = fopen(file,"rb");
266     raiseErrorIf(in,FILEERROR,NULL);
267
268     SAFE_FREAD_INT( options.mode );
269     state.next = 0;
270     state.quit = false;
271     if( options.mode == TIMEMODE ){
272         SAFE_FREAD_INT( state.timeLeft );
273         state.lastTime = time(NULL);
274     } else
275     if( options.mode == MULTIPLMODE ){
276         SAFE_FREAD_INT( state.next );
277         state.next--;
278     }
279     SAFE_FREAD_INT( options.height );
280     SAFE_FREAD_INT( options.width );
281     SAFE_FREAD_INT( options.numColors );
282     SAFE_FREAD_INT( options.tokensPerLine );
283     SAFE_FREAD_INT( options.tokensPerTurn );
284
285     options.initialTokens = 0;
286     sol = newGame( &options );
287     raiseErrorIf(errorCode()==NOERROR,errorCode(),NULL);
288     sol->state = state;
289
290     for( i = 0 ; i < sol->numPlayers ; i++ ){
291         SAFE_FREAD_INT( sol->players[i].board.points );
292         for( y = 0 ; y < sol->options.height ; y++ )
293             for( x = 0 ; x < sol->options.width ; x++ ){
294                 SAFE_FREAD_CHAR( sol->players[i].board.matrix[y][x] );
295                 sol->players[i].board.matrix[y][x] -= '0';
296                 if( sol->players[i].board.matrix[y][x] )
297                     sol->players[i].board.emptySpots--;
298             }
299     }
300     raiseErrorIf(!fread( &i, sizeof(int), 1, in ),CORRUPTFILE,NULL);
301     i = fclose(in);
302     raiseErrorIf(i==0,FILEERROR,NULL);
303
304     raiseErrorIf(validateGame(sol),CORRUPTFILE,NULL);
305
306     return sol;
307 }
308

```

```
1 /**
2  * @file game.h
3  * Operations for the struct @c game
4  */
5
6 #ifndef GAME_H
7 #define GAME_H
8
9 #include <stdbool.h>
10 #include <time.h>
11
12
13 typedef enum{
14     SINGLEMODE, TIMEMODE, MULTIPLMODE
15 } modus_t; // mode_t is already defined in some compilers libraries
16
17 typedef struct{
18     char ** matrix;
19     int points;
20     int emptySpots;
21 } board_t;
22
23 typedef struct{
24     board_t board;
25     board_t lastBoard;
26     bool canUndo;
27 } player_t;
28
29 typedef struct{
30     time_t lastTime;
31     time_t timeLeft; // seconds left (mode 1)
32     int next; // next player (mode 2)
33     bool quit;
34 } state_t;
35
36 typedef struct{
37     modus_t mode; // 0 normal, 1 time, 2 two-players
38     size_t height;
39     size_t width;
40     int numColors;
41     int tokensPerLine; // minimum number of consecutive pieces to make a line
42     int tokensPerTurn; // new random pieces at each round
43     int initialTokens; // tokens located at random when starting
44     time_t initialSeconds; // seconds before game is over
45 } options_t;
46
47 typedef struct{
48     int numPlayers;
49     options_t options;
50     state_t state;
51     player_t * players;
52 } game_t;
53
54
55 game_t *
56 newGame( options_t * options );
57
58 void
59 freeGame( game_t * game );
60
61 void
62 writeGame( game_t * game, const char * file );
63
```

```
64 game_t *
65 readGame( const char * file );
66
67
68 #endif // GAME_H
69
```



```
1  ### FILES ###
2
3  # executable name
4  TARGET=colorLines
5  # sources names
6  SOURCES=error utils playGame game colorsBack colors menu userInterface colorsFront
7
8  ### DIRECTORIES ###
9
10 # .o files directories
11 OPATH=lib/
12 # .h files directories
13 HPATH=back_end/include front_end/include
14 # .c files directories
15 CPATH=back_end/source front_end/source
16
17 ### COMPILATIO FLAGS ###
18
19 # sources flags
20 CFLAGS=-Wall -pedantic -std=c99 -lm
21 # target flags
22 TFLAGS=-Wall -pedantic -std=c99 -lm
23 # extra debug flags
24 DEBUGFLAGS=-g -O0
25 # extra release flags
26 RELEASEFLAGS=-O3
27
28 ### OTHERS ###
29
30 # compiler
31 CC=gcc
32
33 ### possible targets: debug release clean
34 all: release
35
36 OJS=$(addsuffix .o,$(SOURCES) )
37 OUTPUT_OPTION=-o $(OPATH)$@
38 LDFLAGS += -L $(OPATH)
39 CFLAGS += $(addprefix -I ,$(HPATH))
40 vpath %.c $(CPATH)
41
42 .PHONY: all clean debug release
43
44 $(TARGET): $(OJS)
45     $(CC) $(TFLAGS) -o $(TARGET) $(addprefix $(OPATH),$(OJS))
46
47 debug: override CFLAGS += $(DEBUGFLAGS)
48 debug: override TFLAGS += $(DEBUGFLAGS)
49 debug: $(TARGET)
50
51 release: override CFLAGS += $(RELEASEFLAGS)
52 release: override TFLAGS += $(RELEASEFLAGS)
53 release: $(TARGET)
54
55 clean:
56     rm -f $(OPATH)*.o
57     rm -f $(TARGET)
58     find ./ -name "*~" -delete
```

```

1  ### FILES ###
2
3  # executable name
4  TARGET=colorLines
5  # sources names
6  SOURCES=error utils playGame game colorsBack colors menu userInterface colorsFront
7
8  ### DIRECTORIES ###
9
10 # .o files directories
11 OPATH=lib/
12 # .h files directories
13 HPATH=back_end/include front_end/include
14 # .c files directories
15 CPATH=back_end/source front_end/source
16
17 ### COMPILATIO FLAGS ###
18
19 # sources flags
20 CFLAGS=-Wall -pedantic -std=c99 -lm
21 # target flags
22 TFLAGS=-Wall -pedantic -std=c99 -lm
23 # extra debug flags
24 DEBUGFLAGS=-g -O0
25 # extra release flags
26 RELEASEFLAGS=-O3
27
28 ### OTHERS ###
29
30 # compiler
31 CC=gcc
32
33 ### possible targets: debug release clean
34 all: release
35
36 OBJS=$(addsuffix .o,$(SOURCES) )
37 OUTPUT_OPTION=-o $(OPATH)$@
38 LDFLAGS += -L $(OPATH)
39 CFLAGS += $(addprefix -I ,$(HPATH))
40 vpath %.c $(CPATH)
41
42 .PHONY: all clean debug release
43
44 $(TARGET): $(OBJS)
45     $(CC) $(TFLAGS) -o $(TARGET) $(addprefix $(OPATH),$(OBJS))
46
47 debug: override CFLAGS += $(DEBUGFLAGS)
48 debug: override TFLAGS += $(DEBUGFLAGS)
49 debug: $(TARGET)
50
51 release: override CFLAGS += $(RELEASEFLAGS)
52 release: override TFLAGS += $(RELEASEFLAGS)
53 release: $(TARGET)
54
55 clean:
56     rm -f $(OPATH)*.o
57     rm -f $(TARGET)
58     find ./ -name "*~" -delete
59

```

```

1 /**
2  * @file menu.c
3  * Main menu for the game.
4  */
5
6 #include <stdio.h>
7 #include "error.h"
8 #include "utils.h"
9 #include "defines.h"
10 #include "userInterface.h"
11 #include "menu.h"
12
13 typedef enum{
14     MODE0 = SINGLEMODE,
15     MODE1 = TIMEMODE,
16     MODE2 = MULTIPLMODE,
17     READFROMFILE,
18     QUIT
19 } modeOption_t;
20
21
22 /**
23  * Reads an integer and asserts it's in the interval [a a, @a b].
24  *
25  * @throws INPUTERROR if there was an error while reading from standard input
26  *
27  * @param a lower bound
28  * @param b upper bound
29  *
30  * @return integer between a and b
31  *
32  * @see askCommand()
33  * @see validateInt()
34  */
35
36 static int
37 askInt( int a, int b )
38 {
39     int sol;
40     char c;
41     static char error[MAX_ERR_LEN];
42     static char buffer[MAX_COM_LEN];
43
44     error[0] = 0;
45     do{
46         clearScreen();
47         drawText( NULL );
48         drawPanel( error );
49         askCommand( buffer );
50         raiseErrorIf( errorCode() == NOERROR, errorCode(), -1 );
51         error[0] = 0;
52         if ( ( c = sscanf( buffer, " %d %c ", &sol, &c ) ) != 1 && buffer[0] != '\n' )
53             sprintf( error, "Format error:\nMust be an integer" );
54     }while( c != 1 || !validateInt( a, sol, b+1, error ) );
55
56     return sol;
57 }
58
59
60 /**
61  * Reads two integers and asserts they are in the intervals [a a1, @a b1] and
62  * [a a2, @a b2] respectively.
63  *

```

```

64 * @throws INPUTERROR    if there was an error while reading from standard input
65 *
66 * @param a1              lower bound for @a n1
67 * @param[out] n1         pointer to first element
68 * @param b1              upper bound for @a n1
69 * @param a2              lower bound for @a n2
70 * @param[out] n2         pointer to the second element
71 * @param b2              upper bound for @a n2
72 *
73 * @see askCommand()
74 * @see validateInt()
75 */
76
77 static void
78 ask2Int( int a1, int * n1, int b1, int a2, int * n2, int b2 )
79 {
80     char c;
81     static char error[MAX_ERR_LEN];
82     static char buffer[MAX_COM_LEN];
83
84     error[0] = 0;
85     do{
86         clearScreen();
87         drawText( NULL );
88         drawPanel( error );
89         askCommand( buffer );
90         raiseErrorIf( errorCode() == NOERROR, errorCode(), );
91         error[0] = 0;
92         if( ( c = sscanf( buffer, " %d %d %c", n1, n2, &c ) ) != 2 && buffer[0] != '\n' )
93             sprintf( error, "Format error:\nMust be two integers, "
94                     "space separated" );
95     }while( c != 2 || !validateInt( a1, *n1, b1+1, error ) ||
96           !validateInt( a2, *n2, b2+1, error ) );
97 }
98
99
100 /**
101 * Reads a string and asserts it's not empty.
102 *
103 * @throws INPUTERROR    if there was an error while reading from standard input
104 *
105 * @param[out] str       destination string
106 *
107 * @return string with the filename (@a str)
108 *
109 * @see askCommand()
110 */
111
112 static char *
113 askString( char * str )
114 {
115     char c;
116     static char error[MAX_ERR_LEN];
117     static char buffer[MAX_COM_LEN];
118
119     do{
120         clearScreen();
121         drawText( NULL );
122         drawPanel( error );
123         askCommand( buffer );
124         raiseErrorIf( errorCode() == NOERROR, errorCode(), NULL );
125         error[0] = 0;
126         if( ( c = sscanf( buffer, " %s", str ) ) != 1 && buffer[0] != '\n' )

```

```

127     sprintf( error, "Format error:\nMust be a string" );
128 }while( c != 1 );
129 return str;
130 }
131
132 /**
133  * Displays mode menu and asks game mode.
134  *
135  * @throws INPUTERROR    if there was an error while reading from standard input
136  *
137  * @return game mode
138  *
139  * @see askInt()
140  */
141
142 static modeOption_t
143 chooseMode()
144 {
145     drawText("Enter the game mode [1-5]:\n"
146             " 1. Single player normal mode\n"
147             " 2. Single player time mode\n"
148             " 3. Two players\n"
149             " 4. Recover game from file\n"
150             " 5. Quit\n" );
151
152     switch( askInt(1,5) ){
153     case 1: return MODE0;
154     case 2: return MODE1;
155     case 3: return MODE2;
156     case 4: return READFROMFILE;
157     case 5:
158     default: return QUIT;
159     }
160 }
161
162 /**
163  * Displays game options menu and asks for them.
164  *
165  * @throws INPUTERROR    if there was an error while reading from standard input
166  *
167  * @param mode    game mode
168  *
169  * @return structure containing the options chosen.
170  *
171  * @see askInt()
172  * @see ask2Int()
173  */
174
175 static options_t
176 chooseOptions( modus_t mode )
177 {
178     options_t options;
179     int h, w;
180
181     options.mode = mode;
182     if( options.mode == TIMEMODE ){
183         drawText("Enter the time limit (in minutes):\n");
184         options.initialSeconds = 60 * askInt( 1, MAX_MINUTES );
185         raiseErrorIf( errorCode() == NOERROR, errorCode(), options );
186     }
187
188     drawText("Enter the dimensions of the board "
189             "(rows and columns space separated):\n");

```

```

190     ask2Int( MIN_TAB_DIM, &h, MAX_TAB_DIM, MIN_TAB_DIM, &w, MAX_TAB_DIM );
191     raiseErrorIf( errorCode() == NOERROR, errorCode(), options );
192
193     options.height = h;
194     options.width = w;
195
196     drawText("Enter the number of colors with which you wish to play:\n");
197     options.numColors = askInt( MIN_COLORS, MAX_COLORS );
198     raiseErrorIf( errorCode() == NOERROR, errorCode(), options );
199
200     drawText("Enter the number of pieces that are initially on the board:\n");
201     options.initialTokens = askInt(1, options.width * options.height );
202     raiseErrorIf( errorCode() == NOERROR, errorCode(), options );
203
204     drawText("Enter the number of pieces that make a line:\n");
205     options.tokensPerLine = askInt( MIN_TOK_PER_LINE,
206                                     min( options.width, options.height ) );
207     raiseErrorIf( errorCode() == NOERROR, errorCode(), options );
208
209     drawText("Enter the number of pieces that are added on each turn:\n");
210     options.tokensPerTurn = askInt(1, options.width * options.height );
211     raiseErrorIf( errorCode() == NOERROR, errorCode(), options );
212
213     return options;
214 }
215
216
217 /**
218  * Displays the game menu. Uses {@link chooseMode()},
219  * for @b MODE0, @b MODE1, @b MODE2 calls {@link chooseOptions()}
220  * for @b READFROMFILE calls {@link readGame()}.
221  *
222  * @throws INPUTERROR      if there was an error while reading from
223  *                          standard input
224  * @throws MEMORYERROR     if there was a problem while allocating memory
225  * @throws FILEERROR       if there was an problem while opening/reading
226  *                          from file
227  * @throws COMPUTATIONALERROR if after @b MAX_WAITING_TIME time no solution
228  *                          for {@link randFill()} has been obtained
229  *
230  * @param[out] game pointer to a @c game_t structure
231  *
232  * @return false if user wants to quit game, true otherwise
233  *
234  * @see chooseMode()
235  * @see chooseOptions()
236  * @see askString()
237  * @see readGame()
238  */
239
240 bool
241 menu( game_t ** game )
242 {
243     options_t options;
244     char str[MAX_COM_LEN];
245     modeOption_t modeOption;
246
247     clearError();
248
249     modeOption = chooseMode();
250
251     raiseErrorIf( errorCode() == NOERROR, errorCode(), true );
252

```

```
253     switch( modeOption ){
254         case MODE0:
255         case MODE1:
256         case MODE2:
257             options = chooseOptions( (modus_t)modeOption );
258             raiseErrorIf( errorCode() == NOERROR, errorCode(), true );
259             *game = newGame( &options );
260             return true;
261         case READFROMFILE:
262             drawText("Enter the name of the file:\n");
263             *game = readGame( askString( str ) );
264             return true;
265         case QUIT:
266         default:
267             return false;
268     }
269 }
270
```

```
1 /**
2  * @file menu.h
3  * Main menu for the game.
4  */
5
6 #ifndef MENU_H
7 #define MENU_H
8
9 #include "game.h"
10
11
12 bool
13 menu( game_t ** game );
14
15
16 #endif // MENU_H
17
```



```
1 /**
2  * @file noColors.c
3  * Dummy library, used when not in Unix nor Windows.
4  */
5
6
7 /**
8  * Dummy function, used when not in Unix nor Windows.
9  *
10 * @param c color
11 */
12
13 void
14 textColor( color c )
15 {
16 }
17
18
19 /**
20 * Dummy function, used when not in Unix nor Windows.
21 *
22 * @param c color
23 */
24
25 void
26 backColor( color c )
27 {
28 }
29
30
31 /**
32 * Dummy function, used when not in Unix nor Windows.
33 *
34 * @param a attribute
35 */
36
37 void
38 textAttr( attr a )
39 {
40 }
41
```

```

1  /**
2  * @file playGame.c
3  * Functions for token handling
4  */
5
6  #include <stdlib.h>
7  #include <time.h>
8  #include "error.h"
9  #include "defines.h"
10 #include "utils.h"
11 #include "colors.h"
12 #include "playGame.h"
13
14 typedef struct {
15     int x,y;
16 }direction_t;
17
18
19 /**
20 * Looks for lines of the same color, intersecting (@a x,@a y). If lines are
21 * found, they are erased.
22 *
23 * @param game    game structure
24 * @param x        coordinate
25 * @param y        coordinate
26 * @param dir      line direction to check
27 *
28 * @return new empty spots, tokens extrancted from the board
29 */
30
31 static int
32 lookForLine( game_t * game, int nPlayer, size_t x, size_t y, direction_t dir )
33 {
34     int i, dx, dy, tokens = 1;
35
36     color c = game->players[nPlayer].board.matrix[y][x];
37     // count how many tokens of the same color are aligned
38     dx = x + dir.x; dy = y + dir.y;
39     while ( entre( 0, dx, game->options.width ) &&
40             entre( 0, dy, game->options.height ) &&
41             game->players[nPlayer].board.matrix[dy][dx] == c ){
42         dx += dir.x;
43         dy += dir.y;
44         tokens++;
45     }
46     dx = x - dir.x; dy = y - dir.y;
47     while ( entre( 0, dx, game->options.width ) &&
48             entre( 0, dy, game->options.height ) &&
49             game->players[nPlayer].board.matrix[dy][dx] == c ){
50         dx -= dir.x;
51         dy -= dir.y;
52         tokens++;
53     }
54     // erase the line
55     if( tokens >= game->options.tokensPerLine ){
56         for( i = 0 ; i < tokens ; i++ ){
57             dx += dir.x;
58             dy += dir.y;
59             game->players[nPlayer].board.matrix[dy][dx] = 0;
60         }
61         game->players[nPlayer].board.matrix[y][x] = c;
62         return tokens;
63     }

```

```

64     return 0;
65 }
66
67
68 /**
69  * Checks for lines, erases them, actualizes emptySpots and, if @a countPoints
70  * is true, then it also actualizes points.
71  *
72  * @param game      game structure
73  * @param x          coordinate
74  * @param y          coordinate
75  * @param countPoints if false points won't be taken into account
76  *
77  * @return number of lines deleted
78  *
79  * @see lookForLine()
80  */
81
82 int
83 winningPlay( game_t *game, int nPlayer, size_t x, size_t y, bool countPoints )
84 {
85     int i, aux, emptySpots=0, lines = 0;
86     direction_t directions[]={ {0,1}, {1,0}, {1,1}, {-1,1} };
87
88     for( i = 0 ; i < 4 ; i++){
89         aux = lookForLine( game, nPlayer, x, y, directions[i] );
90         if(aux){
91             lines++;
92             emptySpots += aux - 1;
93         }
94     }
95     if(emptySpots){
96         emptySpots++;
97         game->players[nPlayer].board.matrix[y][x] = 0;
98         game->players[nPlayer].board.emptySpots += emptySpots;
99         if( countPoints ){
100             if( lines > 1 )
101                 game->players[nPlayer].board.points += 8;
102             else
103                 switch( emptySpots - game->options.tokensPerLine ){
104                     case 0:
105                         game->players[nPlayer].board.points += 1;
106                         break;
107                     case 1:
108                         game->players[nPlayer].board.points += 2;
109                         break;
110                     case 2:
111                         game->players[nPlayer].board.points += 4;
112                         break;
113                     case 3:
114                         game->players[nPlayer].board.points += 6;
115                         break;
116                     default:
117                         game->players[nPlayer].board.points += 8;
118                         break;
119                 }
120         }
121     }
122     return emptySpots;
123 }
124
125
126 /**

```

```

127 * Fills the board with a certain number of random tokens.
128 *
129 * @throws COMPUTATIONALERROR    if after @b MAX_WAITING_TIME time no solution
130 *                               has been obtained (just on force mode)
131 *
132 * @param game        contains all the information about the current game
133 * @param nPlayer     player number
134 * @param cant        indicates the number of tokens about to be placed
135 * @param force       indicates if when a line is made, and tokens are erased,
136 *                   tokens should still be filled until @a cant are reached
137 *
138 * @see winningPlay()
139 */
140
141 void
142 randFill( game_t * game, int nPlayer, size_t cant, bool force )
143 {
144     int i, j, pos;
145     time_t initTime = time(NULL);
146
147     struct point{
148         int x,y;
149     } vec[ game->players[nPlayer].board.emptySpots ], aux;
150
151     do{
152         // fill vec with the coordinates of all the empty spots
153         pos = 0;
154         for( i = 0 ; i < game->options.height ; i++ )
155             for( j = 0 ; j < game->options.width ; j++ )
156                 if( !game->players[nPlayer].board.matrix[i][j] )
157                     vec[pos++] = (struct point){j,i};
158         // random shuffle vec
159         for( i = 0 ; i < game->players[nPlayer].board.emptySpots ; i++ ){
160             pos = rand() % game->players[nPlayer].board.emptySpots;
161             aux = vec[i];
162             vec[i] = vec[pos];
163             vec[pos] = aux;
164         }
165         // fill with cant tokens
166         pos = game->players[nPlayer].board.emptySpots;
167         j = 0;
168         for( i = 0 ; i < cant ; i++){
169             if( game->players[nPlayer].board.emptySpots <= 0 || pos <= i ){
170                 break;
171             }
172             game->players[nPlayer].board.emptySpots--;
173             game->players[nPlayer].board.matrix[ vec[i].y ][ vec[i].x ] =
174                 rand() % game->options.numColors + 1;
175
176             j += 1 - winningPlay( game, nPlayer, vec[i].x, vec[i].y, false );
177         }
178         cant -= j;
179
180         raiseErrorIf( time(NULL)-initTime < MAX_WAITING_TIME, COMPUTATIONALERROR,);
181     }while( force && cant > 0 && game->players[nPlayer].board.emptySpots > 0 );
182 }
183
184
185 /**
186 * Checks if game is over for player @a nPlayer
187 *
188 * @param nPlayer player number
189 *

```

```
190 * @returns true if game is over for player @a nPlayer. false otherwise
191 */
192
193 bool
194 gameOver( game_t * game, int nPlayer )
195 {
196     return ( game->options.mode == TIMEMODE
197             && game->state.timeLeft - time(NULL) + game->state.lastTime <= 0 )
198             || game->players[nPlayer].board.emptySpots <= 0;
199 }
200
```

```
1 /**
2  * @file playGame.h
3  * Functions for token handling
4  */
5
6 #ifndef PLAYGAME_H
7 #define PLAYGAME_H
8
9 #include <stdbool.h>
10 #include "colors.h"
11 #include "game.h"
12
13 int
14 winningPlay( game_t *game, int nPlayer, size_t x, size_t y, bool countPoints );
15
16 void
17 randFill( game_t * game, int nPlayer, size_t cant, bool force );
18
19 bool
20 gameOver( game_t * game, int nPlayer );
21
22
23 #endif // PLAYGAME_H
24
```

```
1 /**
2  * @file unixColors.c
3  * Console colors in Unix.
4  */
5
6 #include <stdio.h>
7
8 static const char color2font_color[] =
9 { 30,31,32,33,34,35,36,37,90,91,92,93,94,95,96,97 };
10 static const char color2font_bkcolor[] =
11 { 40,41,42,43,44,45,46,47,100,101,102,103,104,105,106,107 };
12 static const char attr2font_attr[] =
13 { 0,10,1,4,5,7,22,24,25,27 };
14
15
16 /**
17  * Sets the font color.
18  *
19  * @param c color
20  */
21
22 void
23 textColor( color c )
24 {
25     if( !USE_COLORS ) return;
26     printf( "\033[%dm", color2font_color[(int)c] );
27 }
28
29
30 /**
31  * Sets the background color.
32  *
33  * @param c color
34  */
35
36 void
37 backColor( color c )
38 {
39     if( !USE_COLORS ) return;
40     printf( "\033[%dm", color2font_bkcolor[(int)c] );
41 }
42
43
44 /**
45  * Sets text attributes.
46  *
47  * @param a attribute
48  */
49
50 void
51 textAttr( attr a )
52 {
53     if( !USE_COLORS ) return;
54     printf( "\033[%dm", attr2font_attr[(int)a] );
55 }
56
```

```

1  /**
2  * @file userInterface.c
3  * Funtions for human-computer interaction
4  */
5
6  #include <stdio.h>
7  #include <string.h>
8  #include <time.h>
9  #include "error.h"
10 #include "utils.h"
11 #include "defines.h"
12 #include "playGame.h"
13 #include "userInterface.h"
14
15
16 static char commandsBuffer[ MAX_PANEL_LINES ][ MAX_COM_LEN + 1 ];
17
18 static int commandsBufferPos = -1;
19
20
21 /**
22 * Clears the screen printing '\n'
23 *
24 */
25
26 void
27 clearScreen()
28 {
29     int i;
30     backColor(BLACK);
31     for( i = 0 ; i < MAX_TAB_DIM ; i++ )
32         printf("\n");
33     textAttr(CLEAR);
34 }
35
36 /**
37 * Draws the boards of each player in the game, the score and if necessary
38 * the time left, provided that it's not game over, in which case it prints
39 * just the score and "GAME OVER"
40 *
41 * @param game    contains all information about current game
42 */
43
44 void
45 drawTable( game_t * game )
46 {
47     int i, j, player, col;
48     static const int colors[] = {
49         BLACK, RED, LIGHT_BLUE, GREEN, YELLOW,
50         VIOLET, PINK, SKY_BLUE, BROWN, LIGHT_GREEN
51     };
52     backColor(BLACK);
53     textColor(WHITE);
54
55     printf("\n    ");
56     for( player = 0 ; player < game->numPlayers ; player++ ){
57         for( i = 0 ; i < game->options.width ; i++ )
58             printf("    %-2d",i);
59         printf("    ");
60     }
61
62     if( HOR_LINES ){
63         printf("\n    ");

```



```

64     for( player = 0 ; player < game->numPlayers ; player++ ){
65         for( i = 0 ; i < game->options.width ; i++ )
66             printf("+-+");
67         printf("+");
68     }
69 }
70 printf("\n");
71
72 for( i = 0 ; i < game->options.height ; i++ ){
73
74     for( player = 0 ; player < game->numPlayers ; player++ ){
75
76         printf("%2d",i);
77         for( j = 0 ; j < game->options.width; j++ ){
78
79             col = game->players[
80                 ( game->state.next + player ) % game->numPlayers
81             ].board.matrix[i][j];
82
83             printf(" | ");
84             textColor( colors[ (int)col ] );
85             if(col)
86                 printf("%d", (int)col );
87             else
88                 printf(" ");
89             textColor(WHITE);
90         }
91         printf(" | ");
92     }
93     if( HOR_LINES ){
94         printf("\n");
95         for( player = 0 ; player < game->numPlayers ; player++ ){
96             for(j=0; j < game->options.width; j++)
97                 printf("+-+");
98             printf("+");
99         }
100     }
101     printf("\n");
102 }
103 // draw points, time/player/GAME OVER
104 char s[15], t[15];
105 sprintf( s, "%%ds", game->options.width * 4 - 10 );
106
107 for( player = 0 ; player < game->numPlayers ; player++ ){
108
109     i = ( game->state.next + player ) % game->numPlayers;
110
111     printf("    SCORE %-4d", game->players[i].board.points );
112
113     if( gameOver( game, player ) ){
114         printf( s, "GAME OVER" );
115     }else
116     if( game->options.mode == TIMEMODE ){
117         j = game->state.timeLeft - time(NULL) + game->state.lastTime;
118         if( j >= 60 )
119             sprintf( t, "%d:%d", j/60, j%60 );
120         else
121             sprintf( t, "%d", j );
122         printf( s, t );
123     }else
124     if( game->options.mode == MULTIPLMODE ){
125         sprintf( t, "Player %d", i + 1 );
126         printf( s, t );

```

```

127     }
128     printf("  ");
129 }
130
131     textAttr(CLEAR);
132 }
133
134
135 /**
136  * Draws the text main panel. If @a str is provided, then the text is set to be
137  * it, else if @a str is NULL, the last text provided is used.
138  *
139  * @param str    text to print (can be NULL)
140  */
141
142 void
143 drawText( const char * message )
144 {
145     static char buffer[ MAX_TEXT ] = "";
146     if( message && message[0] )
147         strcpy( buffer, message );
148     backColor(BLACK);
149     textColor(WHITE);
150     printf("%s",buffer);
151     textAttr(CLEAR);
152 }
153
154
155 /**
156  * Draws the text panel (secondary). If @a message is provided, then it is
157  * printed. All the previous commands and ' > ' are printed in either case.
158  *
159  * @param str    text to print (can be NULL or empty)
160  */
161
162 void
163 drawPanel( const char * message )
164 {
165     char msg[ MAX_ERR_LEN ];
166     int i;
167
168     if( commandsBufferPos == -1 ){
169         for( i = 0 ; i < MAX_PANEL_LINES ; i++ )
170             sprintf( commandsBuffer[i], "\n" );
171         commandsBufferPos = 0;
172     }
173
174     backColor(BLACK);
175
176     if( message && message[0] ){
177         sprintf( msg, "%s", message );
178         char * aux = strtok( msg, "\n" );
179         while( aux ){
180             sprintf( commandsBuffer[ commandsBufferPos++ ], "%s\n", aux );
181             commandsBufferPos %= MAX_PANEL_LINES;
182             aux = strtok( NULL, "\n" );
183         }
184     }
185
186     printf("\n\n");
187     for( i = PANEL_LINES ; i > 0 ; i-- ){
188         if( commandsBuffer[ (commandsBufferPos-i+MAX_PANEL_LINES)
189             % MAX_PANEL_LINES ][1] == '>' )

```

```

190         textColor(WHITE);
191     else
192         textColor(GRAY);
193     printf( "%s", commandsBuffer[ (commandsBufferPos-i+MAX_PANEL_LINES)
194                                     % MAX_PANEL_LINES ] );
195 }
196
197 textColor(WHITE);
198 printf( " > " );
199 textAttr(CLEAR);
200 }
201
202
203 /**
204  * If in @b MULTIPLMODE, draws in the panel the winner of the game.
205  *
206  * @param game    contains all information about current game
207  *
208  * @see drawPanel()
209  */
210
211 void
212 drawWinner( game_t * game )
213 {
214     char str[15];
215     int i=0,j;
216     if( game->options.mode != MULTIPLMODE )
217         return;
218
219     for( j = 1 ; j < game->numPlayers ; j++ )
220         if( game->players[j].board.points >= game->players[i].board.points )
221             i = j;
222     if( i && game->players[0].board.points == game->players[i].board.points )
223         sprintf( str, "Tie game." );
224     else
225         sprintf( str, "Player %d wins!!!", i+1 );
226
227     drawPanel( str );
228 }
229
230 /**
231  * Reads a command from the standard input.
232  *
233  * @throws INPUTERROR    if there was an error while reading from standard input
234  *
235  * @param[out] str    buffer ( has at least MAX_COM_LEN size allocated )
236  *
237  * @return string with the command (@a str)
238  */
239
240 char *
241 askCommand( char * result )
242 {
243     backColor(BLACK);
244     textColor(WHITE);
245
246     result[ MAX_COM_LEN-5 ] = 0;
247
248     raiseErrorIf( fgets( result, MAX_COM_LEN-3, stdin ), INPUTERROR, NULL);
249
250     if( result[ MAX_COM_LEN-5 ] ){          // we make sure it's \n terminated
251         result[ MAX_COM_LEN-5 ] = '\n';
252         clearBuffer();

```

```
253     }
254     sprintf( commandsBuffer[ commandsBufferPos++ ], " > %s", result );
255     commandsBufferPos %= MAX_PANEL_LINES;
256
257     textAttr(CLEAR);
258     printf("\n");
259
260     return result;
261 }
262
```

```
1 /**
2  * @file userInterface.h
3  * Funtions for human-computer interaction
4  */
5
6 #ifndef UI_H
7 #define UI_H
8
9 #include "game.h"
10 #include "colors.h"
11
12 void
13 clearScreen();
14
15 void
16 drawTable( game_t * game );
17
18 void
19 drawText( const char * message );
20
21 void
22 drawPanel( const char * message );
23
24 void
25 drawWinner( game_t * game );
26
27 char *
28 askCommand( char * result );
29
30
31 #endif // UI_H
32
```

```

1 /**
2  * @file utils.c
3  * Contains useful functions that are used several times across the code.
4  */
5
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <string.h>
9 #include <ctype.h>
10 #include "error.h"
11 #include "defines.h"
12 #include "utils.h"
13
14
15 /**
16  * Creates a new matrix and fills it with 0.
17  *
18  * @throws MEMORYERROR if there was a problem while allocating memory
19  *
20  * @param height height of the new matrix
21  * @param width width of the new matrix
22  *
23  * @return a matrix of characters. every element in the matrix is a 0
24  *
25  * @see freeMatrix()
26  */
27
28 char **
29 newMatrix( size_t height, size_t width )
30 {
31     int i;
32     char ** sol = malloc( height * sizeof(char*) );
33     raiseErrorIf( sol, MEMORYERROR, NULL );
34
35     for( i = 0 ; i < height ; i++ ){
36         sol[i] = calloc( width, sizeof(char) );
37         // free allocated memory in case of error
38         if( !sol ){
39             freeMatrix( sol, i-1 );
40             raiseError( MEMORYERROR );
41             return NULL;
42         }
43     }
44     return sol;
45 }
46
47
48 /**
49  * Frees the reserved memory for a matrix created by {@link newMatrix()}.
50  *
51  * @param mat the matrix of characters about to be freed
52  * @param height the height of the matrix
53  *
54  * @see newMatrix()
55  */
56
57 void
58 freeMatrix( char ** mat, size_t height )
59 {
60     int i;
61     for( i = 0 ; i < height ; i++ )
62         free( mat[i] );
63     free(mat);

```

```

64 }
65
66
67 /**
68 * Copies a matrix to another one.
69 *
70 * @param[out] to    output matrix in wich from was copied
71 * @param from      the matrix to be copied
72 * @param height    the height of the matrix
73 * @param width     the width of the matrix
74 *
75 * @see newMatrix()
76 * @see freeMatrix()
77 */
78
79 void
80 copyMatrix( char ** to, char ** from, size_t height, size_t width )
81 {
82     int i;
83     for( i = 0 ; i < height ; i++ )
84         memcpy( to[i], from[i], width );
85 }
86
87
88 /**
89 * Checks if @a b is in the interval [@a a,@a c).
90 *
91 * @param a  lower limit
92 * @param b  number to be compared
93 * @param c  the upper limit
94 *
95 * @return   true if @a b is in [@a a,@a c), otherwise, false
96 */
97
98 bool
99 entre( int a, int b, int c )
100 {
101     return a<=b && b<c;
102 }
103
104
105 /**
106 * Checks if @a b is in the interval [@a a,@a c). If it's not, returns the
107 * corresponding message.
108 *
109 * @param a      lower limit
110 * @param b      number to be compared
111 * @param c      the upper limit
112 * @param[out] err  buffer to print the @b RANGEERROR message
113 *
114 * @return       true if @a b is in [@a a,@a c), otherwise, false
115 *
116 * @see entre()
117 */
118
119 bool
120 validateInt( int a, int b, int c, char * err )
121 {
122     if( !entre(a,b,c) ){
123         sprintf( err, "Range error:\nIt must belong to the "
124             "interval [%d,%d]", a, c-1 );
125         return false;
126     }

```

```
127     return true;
128 }
129
130
131 /**
132  * Clears the standard input buffer.
133  *
134  */
135
136 void
137 clearBuffer()
138 {
139     while(getchar() != '\n');
140 }
141
142
143 /**
144  * Calculates the minimum between two numbers.
145  *
146  * @param a first number
147  * @param b second number
148  *
149  * @return the minimum between @a a and @a b
150  */
151
152 int
153 min( int a, int b )
154 {
155     return (a<=b)?a:b;
156 }
157
158
159 /**
160  * Calculates the maximum between two numbers.
161  *
162  * @param a first number
163  * @param b second number
164  *
165  * @return the maximum between @a a and @a b
166  */
167
168 int
169 max( int a, int b )
170 {
171     return (a>=b)?a:b;
172 }
173
174
175 /**
176  * Calculates the edit distance between @a str1 and @a str2
177  * (a.k.a Damerau-Levenshtein distance).
178  *
179  * @param str1 first string
180  * @param str2 second string
181  *
182  * @return a number between 0 and 1 that indicates similarity between
183  *         @a str1 and @a str2. greater is better
184  *
185  * @see F.J. Damerau. A technique for computer detection and correction of
186  *       spelling errors. Communications of the ACM, 1964.
187  *
188  * @see V.I. Levenshtein. Binary codes capable of correcting deletions,
189  *       insertions, and reversals. Soviet Physics Doklady, 1966.
```



```

190 */
191
192 double
193 editDistance( const char * str1, const char * str2 )
194 {
195     int i,j,cost=0;
196     int s1len=strlen(str1);
197     int s2len=strlen(str2);
198     int mat[3][ s2len+1 ];
199     int prev2, prev=0, this = 0;
200
201     if( min( s1len, s2len ) < MIN_EDIT_LEN || max( s1len, s2len ) > MAX_EDIT_LEN )
202         return 0;
203
204     for( i = 0 ; i <= s2len ; i++ )
205         mat[0][i] = i;
206
207     for( i = 0 ; i <= s1len ; i++ ){
208
209         prev2 = prev;
210         prev = this;
211         this = (i+1) % 3;
212
213         mat[this][0] = i+1;
214
215         for( j = 0 ; j <= s2len ; j++ ){
216             cost = toupper(str1[i]) != toupper(str2[j]);
217             // in this order: deletions, insertions, substitutions
218             mat[this][j+1] =
219             min( mat[prev][j+1]+1, min( mat[this][j]+1, mat[prev][j]+cost ) );
220             // transposes
221             if( i && j && toupper(str1[i]) == toupper(str2[j-1])
222                 && toupper(str1[i-1]) == toupper(str2[j]) )
223
224                 mat[this][j+1] = min( mat[this][j+1], mat[prev2][j-1] + cost );
225         }
226     }
227
228     return 1 - mat[prev][s2len] / (double)max( s1len, s2len );
229 }
230

```

```
1 /**
2  * @file utils.h
3  * Contains useful functions that are used several times across the code.
4  */
5
6 #ifndef UTILS_H
7 #define UTILS_H
8
9 #include <stdlib.h>
10 #include <stdbool.h>
11
12
13 char **
14 newMatrix( size_t height, size_t width );
15
16 void
17 freeMatrix( char ** mat, size_t height );
18
19 void
20 copyMatrix( char ** to, char ** from, size_t height, size_t width );
21
22 bool
23 entre( int a, int b, int c );
24
25 bool
26 validateInt( int a, int b, int c, char * err );
27
28 void
29 clearBuffer();
30
31 int
32 min( int a, int b );
33
34 int
35 max( int a, int b );
36
37 double
38 editDistance( const char * str1, const char * str2 );
39
40
41 #endif // UTILS_H
42
```

```

1 /**
2  * @file winColors.c
3  * Console colors in Windows.
4  */
5
6 #include <windows.h>
7 #include <wincon.h>
8 #include <stdio.h>
9
10 static const char crazyWinColorsMap[] = {
11     /*NEGRO*/BLACK,/*ROJO*/BLUE,/*VERDE*/GREEN,/*MARRON*/SKY_BLUE,
12     /*AZUL*/RED,/*VIOLETA*/VIOLET,/*CELESTE*/BROWN,/*GRIS_CLARO*/LIGHT_BLUE,
13     /*GRIS*/GRAY,/*ROSA*/LIGHT_BLUE,/*VERDE_CLARO*/LIGHT_GREEN,/*AMARILLO*/LIGHT_BLUE_SKY,
14     /*AZUL_CLARO*/PINK,/*VIOLETA_CLARO*/LIGHT_VIOLET,/*CELESTE_CLARO*/YELLOW,
15     /*BLANCO*/WHITE
16 };
17
18 static const char color2font_color[] =
19 { 0, FOREGROUND_RED | FOREGROUND_INTENSITY, FOREGROUND_GREEN, FOREGROUND_RED, FOREGROUND_BLUE,
20   FOREGROUND_BLUE | FOREGROUND_RED, FOREGROUND_GREEN | FOREGROUND_BLUE,
21   FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE, FOREGROUND_RED | FOREGROUND_GREEN,
22   FOREGROUND_RED | FOREGROUND_INTENSITY, FOREGROUND_GREEN | FOREGROUND_INTENSITY,
23   FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_INTENSITY, FOREGROUND_BLUE | FOREGROUND_INTENSITY,
24   FOREGROUND_BLUE | FOREGROUND_RED | FOREGROUND_INTENSITY,
25   FOREGROUND_GREEN | FOREGROUND_BLUE | FOREGROUND_INTENSITY,
26   FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE | FOREGROUND_INTENSITY
27 };
28
29 static const char color2font_bkcolor[] =
30 { 0, BACKGROUND_RED | BACKGROUND_INTENSITY, BACKGROUND_GREEN, BACKGROUND_RED, BACKGROUND_BLUE,
31   BACKGROUND_BLUE | BACKGROUND_RED, BACKGROUND_GREEN | BACKGROUND_BLUE,
32   BACKGROUND_RED | BACKGROUND_GREEN | BACKGROUND_BLUE, BACKGROUND_RED | BACKGROUND_GREEN,
33   BACKGROUND_RED | BACKGROUND_INTENSITY, BACKGROUND_GREEN | BACKGROUND_INTENSITY,
34   BACKGROUND_RED | BACKGROUND_GREEN | BACKGROUND_INTENSITY, BACKGROUND_BLUE | BACKGROUND_INTENSITY,
35   BACKGROUND_BLUE | BACKGROUND_RED | BACKGROUND_INTENSITY,
36   BACKGROUND_GREEN | BACKGROUND_BLUE | BACKGROUND_INTENSITY,
37   BACKGROUND_RED | BACKGROUND_GREEN | BACKGROUND_BLUE | BACKGROUND_INTENSITY
38 };
39
40
41 /**
42  * Sets the font color.
43  *
44  * @param c color
45  */
46
47 void
48 textColor( color c )
49 {
50     WORD wColor;
51     HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
52     CONSOLE_SCREEN_BUFFER_INFO csbi;
53
54     if( !USE_COLORS ) return;
55     if(GetConsoleScreenBufferInfo(hStdOut, &csbi)){
56         wColor = (csbi.wAttributes & 0xF0) + ( crazyWinColorsMap[c] & 0x0F);
57         SetConsoleTextAttribute(hStdOut, wColor);
58     }
59 }
60
61
62 /**
63  * Sets the background color.

```

```
64 *
65 * @param c color
66 */
67
68 void
69 backColor( color c )
70 {
71     WORD wColor;
72     HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
73     CONSOLE_SCREEN_BUFFER_INFO csbi;
74
75     if( !USE_COLORS ) return;
76     if(GetConsoleScreenBufferInfo(hStdOut, &csbi)){
77         wColor = (csbi.wAttributes & 0x0F) + ( crazyWinColorsMap[c] & 0xF0);
78         SetConsoleTextAttribute(hStdOut, wColor);
79     }
80 }
81
82
83 /**
84 * Sets text attributes.
85 *
86 * @param a attribute
87 */
88
89 void
90 textAttr( attr a )
91 {
92     if( !USE_COLORS ) return;
93     if( a == CLEAR ){
94         backcolor(BLACK);
95         frontcolor(WHITE);
96     }
97 }
98
```