

Ausarbeitung 06

Niklas Vest

November 23, 2018

1 Tetris

1.1 Lösungsidee

Tetrominos werden durch eine $n \times n$ Matrix von booleschen Werten dargestellt, wobei $n = \max(\text{tetrLnge}, \text{tetrHhe})$. Jeder Eintrag in dieser Matrix bestimmt, ob das Tetromino an dieser Stelle "ausgeprägt" ist, oder ob es ein leeres Feld ist. So ist zb der Würfel eine 2×2 Matrix, deren Einträge alle den Wert *True* haben, da das Tetromino sozusagen die Matrix füllt.

Rotationen äußern sich nicht durch eine Änderung der Matrix, sondern in der Leserichtung dieser Matrix. Stumpf gesagt wird ein Tetromino nach rechts rotiert, indem dessen Matrix "von unten" betrachtet wird. Dies setzt voraus, dass die Matrix standardmäßig von links betrachtet wird, was der natürlichen Leserichtung des Feldes entspricht welche die Matrix darstellt. Ein Beispiel am T-Tetromino (unter Verwendung von 1 und 0 für jeweils True und False, der Einfachheit halber):

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Das Feld fängt an mit den Werten $[1, 1, 1, 0, 1]$, wenn man eine Row-Major-Order annimmt. liest man diese Matrix jetzt aber von unten links nach oben, so erhält man $[0, 0, 1, 0, 1]$ als erste Werte, was folgender Matrix entspricht:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Anstatt also bei jedem Tastendruck die Matrix umzuschreiben, ändere ich nur die Zugriffsfunktion auf diese Matrix. Diese Zugriffsfunktionen bilden einen Zeilen- und Spaltenindex auf ein Element in der matrix ab, abhängig von der Rotation die sie darstellen sollen. (Siehe `rotate_left`, `rotate_half` etc.)

Das **Spielbrett** wird ebenfalls durch eine Matrix dargestellt, wobei nur bereits fixierte Tetrominos in diese Matrix eingeschrieben werden. Das momentan fallende Tetromino prüft vor jeder weiteren vertikalen Bewegung auf Kollision

mit dem Boden bzw einem anderen Tetromino. Tetromino Kollisionen werden wie folgt verhindert: Bei jedem Update des Spielbretts wird das fallende Tetromino (flach) kopiert und um eins nach unten verschoben, es wird also ein weiterer Fall simuliert. Überschneidet sich diese Kopie mit einem bereits festgeschriebenem Tetromino in der Matrix des Spielbretts, wird die Änderung verworfen, andernfalls übernommen. Derselbe Prozess findet auch für rotationen statt.

Das **Abbauen von Reihen** geschieht rekursiv: Ich suche die erste Reihe die vollkommen mit *True* gefüllt ist und schneide als ersten Schritt die jeweilige Zeile aus den Tetrominos, die von der abgebauten Reihe erfasst werden. Danach werden alle Tetrominos mit einem *Y*-Wert über dieser Reihe um 1 nach unten verschoben. Danach gibt es einen rekursiven Aufruf um die nächste volle Reihe (falls vorhanden) abzubauen. Wird keine gefüllte Reihe gefunden, gibt es auch keinen rekursiven Aufruf und die Funktion terminiert. Anzumerken ist hier, dass in der Implementierung tatsächlich nur diejenigen Tetrominos verschoben werden, deren *Y*-Wert logisch (!) unter der abgebauten Zeile liegen, da der Ursprung im linken oberen Eck des Viewports liegt. Visuell liegen diese Tetrominos dann über der erwähnten Zeile, was der Grund für meine ursprüngliche Formulierung ist.

Das **Fallen eines Tetrominos** wird erreicht, indem der Spielzustand so lange ohne Unterbrechung vorangetrieben wird, bis das fallende Tetromino mit einem anderen Tetr. kollidiert oder den Boden erreicht.

Die **Spielgeschwindigkeit** wird dynamisch geregelt. Das Spielbrett erhält pro Sekunde *x* Updates. Dieses *x* wird bei jedem zehnten Tetromino Spawn um 1 erhöht.

Das Spiel wird **beendet** sobald sich ein Tetromino mit der obersten Zeile der Spielbrett-Matrix überschneidet. Das führt ab und zu zu eigenartigen Fehlern, wo zuerst mehrere Tetrominos spawnen müssen, bis eines erzeugt wird, welches nicht mit einer *False*-Zeile beginnt. (Siehe T- und I-Tetromino!) Um das Ende des Spiels zu signalisieren, werden die Tetrominos nicht mehr gezeichnet und der Hintergrund verändert über Zeit die Farbe.

1.2 Implementierung

Listing 1: *main.c*

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "GLFW/glfw3.h"
4
5 #include "tetromino.h"
6 #include "gameboard.h"
7
8 #define WIDTH 400
9 #define HEIGHT WIDTH * (GB_ROWS / GB_COLS)
10
11 /**
12  * Prints the keyboard controls to the
13  * standard output.
```

```
14  */
15  static void print_instructions()
16  {
17      printf("Keymap:\n");
18      printf("Arrow Up ^ : Rotate tetromino clockwise\n");
19      printf("Arrow Down v : Rotate tetromino counter clockwise\n");
20      printf("Arrow Right > : Move tetromino to the right\n");
21      printf("Arrow Left < : Move tetromino to the left\n");
22      printf("Space _ : Drop the tetromino like it's hot\n");
23  }
24
25  int main()
26  {
27      GLFWwindow* window;
28      if (!glfwInit())
29          return -1;
30
31      window = glfwCreateWindow(WIDTH, HEIGHT, "Tetris 4.0", NULL, NULL);
32      if (!window)
33      {
34          glfwTerminate();
35          printf("Failed to create window.\n");
36          return EXIT_SUCCESS;
37      }
38
39      print_instructions();
40
41      int width, height;
42      glfwGetWindowSize(window, &width, &height);
43      glfwSetWindowAspectRatio(window, width, height);
44
45      glfwMakeContextCurrent(window);
46      glfwSetKeyCallback(window, gameboard_keymap);
47
48      init_game_board();
49
50      double last_time = glfwGetTime();
51
52      while (!glfwWindowShouldClose(window))
53      {
54          // the board is always rendered!
55          // when running == false, the background
56          // color fades through a few colors over
57          // time!
58          render_board(window, width, height);
59
60          if (running) {
61              // the update rate changes over time so
62              // it's a variable
63              if (glfwGetTime() - last_time > (1.0 / ticks_pers_second)) {
64                  last_time = glfwGetTime();
65                  update_board();
66              }
67          }
68
69          const GLenum error = glGetError();
70          if (error != GL_NO_ERROR) fprintf(stderr, "ERROR: %d\n", error);
```

```
71     glfwSwapBuffers(window);
72     glfwPollEvents();
73 }
74
75     destroy_game_board();
76     glfwDestroyWindow(window);
77     glfwTerminate();
78     return EXIT_SUCCESS;
79 }
```

Listing 2: *gameboard.h*

```
1  #ifndef GAMEBOARD_H
2  #define GAMEBOARD_H
3
4  #include <GLFW/glfw3.h>
5  #include "tetromino.h"
6
7  #define GB_ROWS 22
8  #define GB_COLS 11
9
10 /**
11  * Every information about the
12  * game board and the current state
13  * of the game.
14  */
15 typedef struct {
16     /**
17      * All tetrominoes that are on the game board.
18      */
19     tetromino_t *tetrominoes;
20
21     /**
22      * The number of tetrominoes on the game board.
23      */
24     int nr_of_tetrominoes;
25
26     /**
27      * The current size of the tetromino vector.
28      */
29     int vec_size;
30
31     /**
32      * A matrix of boolean values which state whether
33      * a particular tile is occupied.
34      */
35     bool *tile_matrix;
36 } tetris_board_t;
37
38 /**
39  * The single game board instance.
40  */
41 extern tetris_board_t game_board;
42
43 /**
44  * A global flag which reflects
45  * the current state of the game.
46  */
```

```
47 extern bool running;
48
49 /**
50  * The number of updates that
51  * the game receives per second
52  * (approximately).
53  */
54 extern int ticks_pers_second;
55
56 /**
57  * Initializes the game board.
58  * (Bet u didn't suspect that huh?)
59  */
60 void init_game_board();
61
62 /**
63  * Renders the game board to the supplied glfw window.
64  * @param window The window to draw to.
65  * @param width The width of the viewport.
66  * @param height The height of the viewport.
67  */
68 void render_board(GLFWwindow *window, int width, int height);
69
70 /**
71  * Updates the game state.
72  */
73 void update_board();
74
75 /**
76  * The keymap for the game. (Using GLFWs setKeyCallback)
77  */
78 void gameboard_keymap(GLFWwindow* window, int key, int scancode, int action,
79                       int mods);
80
81 /**
82  * Destroys the game board and cleans up.
83  */
84 void destroy_game_board();
85 #endif
```

Listing 3: *gameboard.c*

```
1 #include <stdlib.h>
2 #include <assert.h>
3 #include <stdio.h>
4 #include <math.h>
5 #include <time.h>
6 #include <stdbool.h>
7 #include "gameboard.h"
8
9 /**
10  * @return A reference to the currently falling tetromino.
11  */
12 static tetromino_t *currently_falling_tetromino();
13
14 /**
15  * Applies a translation to the supplied tetromino.
```

```
16  * @param tetromino The tetromino to move.
17  * @param xDelta The distance to move it on the x-axis.
18  * @param yDelta The distance to move it on the y-axis.
19  */
20  static void move_tetromino(tetromino_t *tetromino, int xDelta, int yDelta);
21
22  /**
23   * Generates a new tetromino and adds it to the game board.
24   */
25  static void spawn_tetromino();
26
27  /**
28   * Writes the supplied tetromino to the game board matrix
29   * with regards to its current position (on the board).
30   * @param tetromino The tetromino to write.
31   */
32  static void write_tetromino_to_matrix(const tetromino_t *tetromino);
33
34  /**
35   * Removes any rows that are filled with tiles .
36   */
37  static void remove_first_filled_row();
38
39  /**
40   * @param row The row to check.
41   * @return True if the specified row is filled with tiles ,
42   *         False otherwise.
43   */
44  static bool row_can_be_deleted(int row);
45
46  /**
47   * @param tetromino The tetromino to check.
48   * @return True if the supplied tetromino can still
49   *         fall further down on the game board without
50   *         intersecting with other tetrominoes.
51   */
52  static bool can_fall(const tetromino_t *tetromino);
53
54  /**
55   * Attempts to rotate the tetromino. If the rotation
56   * would lead to the tetromino goin out of bounds
57   * the rotation is prevented.
58   * @param tetromino The tetromino to rotate.
59   * @param rotation The rotation direction.
60   */
61  static void rotate_within_board(tetromino_t *tetromino, rotation_t rotation);
62
63  /**
64   * @param tetromino The tetromino to check.
65   * @return True if the tetromino itself is within
66   *         the board boundaries, False otherwise.
67   */
68  static bool is_within_board(const tetromino_t *tetromino);
69
70  /**
71   * @return True if at least one tile of the uppermost row
72   *         is not empty, False otherwise.
```

```

73  */
74  static bool is_board_full();
75
76  // global (extern) variables
77  tetris_board_t game_board;
78  bool running = true;
79  int ticks_pers_second = 2;
80
81  void init_game_board()
82  {
83      // seed RNG with system time
84      time_t t;
85      srand((unsigned) time(&t));
86
87      // the average player won't be able to reconcile more
88      // than 20 tetrominoes on the board :P
89      game_board.tetrominoes = malloc(sizeof(tetromino_t) * 50);
90      game_board.vec_size = 50;
91
92      // create a matrix representing the game boards occupied tiles
93      game_board.tile_matrix = malloc(sizeof(bool) * GB_ROWS * GB_COLS);
94      for (size_t i = 0; i < GB_ROWS * GB_COLS; ++i) {
95          game_board.tile_matrix[i] = false;
96      }
97
98      game_board.nr_of_tetrominoes = 0;
99  }
100
101  void render_board(GLFWwindow *window, int width, int height)
102  {
103      glfwGetFramebufferSize(window, &width, &height);
104      glViewport(0, 0, width, height);
105      glClear(GL_COLOR_BUFFER_BIT);
106      glMatrixMode(GL_PROJECTION);
107      glLoadIdentity();
108
109      // basically flip this boi so that the coordinate system
110      // has its origin in the upper left hand corner. that makes
111      // it easier to work with the matrices!
112      glOrtho(0, width, height, 0, 0, 1);
113
114      glScalef((float) width / (float) GB_COLS, (float) height / (float)
115              GB_ROWS, 1);
116
117      // let all tetrominoes render themselves
118      if (running) {
119          for (int i = 0; i < game_board.nr_of_tetrominoes; ++i) {
120              render_tetromino(&game_board.tetrominoes[i]);
121          }
122      } else {
123          // hehe cheeky macro!
124          #define color(f) (float) (f(glfwGetTime()))
125          glClearColor(color(0.7*sin), color(cos), color(0.3*sin), 1.0f);
126      }
127
128  void update_board()

```

```

129 {
130     tetromino_t *curr = currently_falling_tetromino();
131     if (curr != NULL) {
132         if (can_fall(curr)) {
133             move_tetromino(curr, 0, 1);
134         } else {
135             // fix tetromino on the board so
136             // new falling tetrominoes can check
137             // for intersections
138             write_tetromino_to_matrix(curr);
139             remove_first_filled_row();
140             if (!is_board_full()) {
141                 spawn_tetromino();
142             } else {
143                 running = false;
144             }
145         }
146     } else { // curr == NULL -> no tetrominoes on the board
147         spawn_tetromino();
148     }
149 }
150
151 void gameboard_keymap(GLFWwindow *window, int key, int scancode, int action,
152                        int mods)
153 {
154     (void) window; (void) scancode; (void) mods;
155
156     tetromino_t *curr = currently_falling_tetromino();
157     if (action == GLFW_PRESS) {
158         switch (key) {
159             case GLFW_KEY_RIGHT:
160                 move_tetromino(curr, 1, 0);
161                 break;
162             case GLFW_KEY_LEFT:
163                 move_tetromino(curr, -1, 0);
164                 break;
165             case GLFW_KEY_UP:
166                 rotate_within_board(curr, CLOCKWISE);
167                 break;
168             case GLFW_KEY_DOWN:
169                 rotate_within_board(curr, COUNTER_CLOCKWISE);
170                 break;
171             case GLFW_KEY_SPACE:
172                 while (can_fall(currently_falling_tetromino())) {
173                     update_board();
174                 }
175                 break;
176             default:
177                 // do nothing
178                 break;
179         }
180     }
181
182 void destroy_game_board()
183 {
184     assert(game_board.tile_matrix);

```



```

185     assert(game_board.tetrominoes);
186
187     free(game_board.tile_matrix);
188     game_board.tile_matrix = NULL;
189
190     for (int i = 0; i < game_board.nr_of_tetrominoes; ++i) {
191         destroy_tetromino(&game_board.tetrominoes[i]);
192     }
193     free(game_board.tetrominoes);
194     game_board.tetrominoes = NULL;
195
196     game_board.nr_of_tetrominoes = 0;
197 }
198
199 // ----- TRANSLATION-UNIT-LOCAL FUNCTIONS
200 // -----
201 void remove_first_filled_row()
202 {
203     // find the first row that can be deleted
204     int row = GB_ROWS - 1;
205     while (row > 0 && !row_can_be_deleted(row)) {
206         --row;
207     }
208
209     if (row > 0) { // a full row exists
210
211         // adjust the masks of tetrominoes that overlap with that row
212         int overlap = 0;
213         while (overlap < game_board.nr_of_tetrominoes) {
214             tetromino_t *curr = &game_board.tetrominoes[overlap];
215             if (curr->pos.y <= row && curr->pos.y + curr->side_len - 1 >= row
216         ) {
217                 remove_tetromino_row(curr, row - curr->pos.y);
218             }
219             if (curr->pos.y <= row) {
220                 curr->pos.y += 1;
221             }
222             ++overlap;
223         }
224
225         // rewrite matrix with "new tetromino configuration"
226         for (int i = 0; i < GB_COLS * GB_ROWS; ++i) {
227             game_board.tile_matrix[i] = false;
228         }
229         for (int i = 0; i < game_board.nr_of_tetrominoes; ++i) {
230             write_tetromino_to_matrix(&game_board.tetrominoes[i]);
231         }
232
233         // recurse if any rows have been removed
234         // (in case row right above is filled as well)
235         remove_first_filled_row();
236     }
237 }
238 bool row_can_be_deleted(int row)
239 {

```

```

240     int i = 0;
241     while (i < GB_COLS && game_board.tile_matrix[row * GB_COLS + i]) {
242         ++i;
243     }
244     return i == GB_COLS;
245 }
246
247 void write_tetromino_to_matrix(const tetromino_t *tetromino)
248 {
249     assert(tetromino);
250     int row_offset = tetromino->pos.y;
251     int col_offset = tetromino->pos.x;
252
253     for (int row = 0; row < tetromino->side_len; ++row) {
254         for (int col = 0; col < tetromino->side_len; ++col) {
255             // calculate the game board index to write to using
256             // the tetrominoes position.
257
258             if (col_offset + col >= 0) {
259                 // tetrominoes can have a blank mask column / row;
260                 // that row is allowed to go outside the bounds of
261                 // the game board so that the actual tetromino can
262                 // align to the edge, thats why I test for x >= 0
263
264                 // EDIT: I think there should also be tests for
265                 // x > GB_COLS and y > GB_ROWS but it works and
266                 // I don't want to break anything
267
268                 int index = (row_offset + row) * GB_COLS + (col_offset + col)
269
270                 ;
271                 game_board.tile_matrix[index] |= get_tetromino_mask_at(
272                     tetromino, row, col);
273             }
274         }
275     }
276 }
277
278 void spawn_tetromino()
279 {
280     // resize the vector if its full
281     if (game_board.nr_of_tetrominoes == game_board.vec_size) {
282         // for some reason realloc removed like the 4th element
283         // from the original vector ??? And I thought I understood
284         // pointers...
285         tetromino_t *new_vec = malloc(sizeof(tetromino_t) * (size_t)
286             game_board.vec_size * 2);
287         for (int i = 0; i < game_board.vec_size; ++i) {
288             new_vec[i] = game_board.tetrominoes[i];
289         }
290         free(game_board.tetrominoes);
291         game_board.tetrominoes = new_vec;
292         game_board.vec_size *= 2;
293     }
294
295     // exploit the nature of enums: the underlying data type is numeric
296     // so a tetromino can be generated using RNG
297     tetromino_shape shape = rand() % 7;

```

```

294     tetromino_t tetromino = create_tetromino(shape);
295     tetromino.pos.x = GB_COLS / 2 - tetromino.side_len / 2;
296     game_board.tetrominoes[game_board.nr_of_tetrominoes] = tetromino;
297     game_board.nr_of_tetrominoes += 1;
298
299     // also increase speed after every 5th tetromino spawn
300     if (game_board.nr_of_tetrominoes % 10 == 0) {
301         ticks_pers_second += 1;
302     }
303 }
304
305 void move_tetromino(tetromino_t *tetromino, int xDelta, int yDelta)
306 {
307     assert(tetromino);
308     tetromino_t cpy = *tetromino;
309     cpy.pos.x+=xDelta;
310     cpy.pos.y+=yDelta;
311     if (is_within_board(&cpy)) {
312         // apply changes if they are valid
313         tetromino->pos.x += xDelta;
314         tetromino->pos.y += yDelta;
315     }
316 }
317
318 void rotate_within_board(tetromino_t *tetromino, rotation_t rotation)
319 {
320     assert(tetromino);
321     tetromino_t cpy = *tetromino;
322     rotate_tetromino(&cpy, rotation);
323     if (is_within_board(&cpy)) {
324         // apply changes if they are valid
325         rotate_tetromino(tetromino, rotation);
326     }
327 }
328
329 bool can_fall(const tetromino_t *tetromino)
330 {
331     assert(tetromino);
332     tetromino_t tetr_cpy = *tetromino;
333     tetr_cpy.pos.y += 1;
334
335     int row_offset = tetr_cpy.pos.y;
336     int col_offset = tetr_cpy.pos.x;
337
338     // if the tetromino hit the bottom already,
339     // it can't fall any further
340     bool can_fall = row_offset + real_height(&tetr_cpy) <= GB_ROWS;
341     // if it's still above ground level, check if it would
342     // intersect with other tetrominoes if it were to fall
343     // one tile further
344     int row = 0;
345     while (row < tetr_cpy.side_len && can_fall) {
346         int col = 0;
347         while (col < tetr_cpy.side_len && can_fall) {
348             // get the matrix index relative to the tetrominoes position
349             int gb_index = (row_offset + row) * GB_COLS + (col_offset + col);
350             // either one of the tile flags must be False, otherwise they intersect

```

```

351         can_fall = game_board.tile_matrix[gb_index] == false ||
352             get_tetromino_mask_at(&tetr_cpy, row, col) == false;
353         ++col;
354     }
355     ++row;
356 }
357 return can_fall;
358 }
359
360 tetromino_t *currently_falling_tetromino()
361 {
362     tetromino_t *last = NULL;
363     if (game_board.nr_of_tetrominoes > 0) {
364         last = &game_board.tetrominoes[game_board.nr_of_tetrominoes - 1];
365     }
366     return last;
367 }
368
369 bool is_within_board(const tetromino_t *tetromino)
370 {
371     assert(tetromino);
372
373     // the x position might be negative with the full tetromino
374     // still being completely within bounds, hence the complex check
375     bool boundary_ok = tetromino->pos.x + left_offset(tetromino) >= 0 &&
376         tetromino->pos.x + (tetromino->side_len - right_offset
377             (tetromino) - 1) < GB_COLS;
378
379     bool no_intersects = true;
380     int row = 0;
381     while (row < tetromino->side_len && no_intersects) {
382         int col = 0;
383         while (col < tetromino->side_len && no_intersects) {
384             int matrix_ind = (tetromino->pos.y + row) * GB_COLS + (tetromino
385                 ->pos.x + col);
386             // if both tiles are set, they intersect
387             no_intersects = get_tetromino_mask_at(tetromino, row, col) ==
388                 false ||
389                 game_board.tile_matrix[matrix_ind] == false;
390             ++col;
391         }
392         ++row;
393     }
394
395     return boundary_ok && no_intersects;
396 }
397
398 bool is_board_full()
399 {
400     int col = 0;
401     while (col < GB_COLS && game_board.tile_matrix[col] == 0) {
402         ++col;
403     }
404     return col != GB_COLS;
405 }

```

Listing 4: *tetromino.h*

```

1  #ifndef UE06_TETROMINO_H
2  #define UE06_TETROMINO_H
3
4  #include <stdbool.h>
5  #include <stdlib.h>
6
7  /**
8   * Handy color typedefs for drawing c:
9   */
10 typedef enum {
11     color_black,
12     color_red    = 0x0000FFU,
13     color_green  = 0x00FF00U,
14     color_blue   = 0xFF0000U,
15     color_yellow = color_red | color_green,
16     color_magenta = color_red | color_blue,
17     color_cyan   = color_green | color_blue,
18     color_white  = color_red | color_green | color_blue,
19 } color_t;
20
21 /**
22  * A 2D position on the game board.
23  */
24 typedef struct {
25     int x, y;
26 } position_t;
27
28 /**
29  * An array holding a mask which represents
30  * a tetromino.
31  */
32 typedef bool* tetromino_mask_t;
33
34 typedef enum {
35     ROTATED_DEFAULT, ROTATED_RIGHT, ROTATED_LEFT, ROTATED_HALF
36 } orientation_t;
37
38 typedef enum {
39     CLOCKWISE, COUNTER_CLOCKWISE
40 } rotation_t;
41
42 /**
43  * A compound representing a tetris object,
44  * called a tetromino.
45  */
46 typedef struct {
47     /**
48      * The current orientation (/rotation)
49      * of the tetromino.
50      */
51     orientation_t orientation;
52
53     /**
54      * The current position of the
55      * tetromino.
56      */

```

```

57     position_t pos;
58
59     /**
60      * The color of the tetromino.
61      */
62     color_t color;
63
64     /**
65      * The side length of the matrix representing
66      * the tetrominos tile-occupation-mask.
67      */
68     int side_len;
69
70     /**
71      * The tetrominos
72      * tile-occupation-mask.
73      */
74     tetromino_mask_t mask;
75 } tetromino_t;
76
77 /**
78  * An enumeration containing all Tetrominoes.
79  */
80 typedef enum {
81     I, J, L, O, S, T, Z
82 } tetromino_shape;
83
84 /**
85  * @param shape The shape of the tetromino to create.
86  * @return A copy of the tetromino prototype with the
87  *         specified shape.
88  */
89 tetromino_t create_tetromino(tetromino_shape shape);
90
91 /**
92  * Renders a tetromino to the current OpenGL Context.
93  * @param tetromino The tetromino to render.
94  */
95 void render_tetromino(const tetromino_t *tetromino);
96
97 /**
98  * Rotates the supplied tetromino according to the
99  * specified rotation direction.
100  * @param tetromino The tetromino to rotate.
101  * @param rotation The direction in which to rotate
102  *                 dat boi.
103  */
104 void rotate_tetromino(tetromino_t *tetromino, rotation_t rotation);
105
106 /**
107  * @param tetromino The tetromino to query.
108  * @param row
109  * @param col
110  * @return The mask value (the occupation) at the
111  *         specified row and column.
112  */
113 bool get_tetromino_mask_at(const tetromino_t *tetromino, int row, int col);

```

```

114
115 /**
116  * @param tetromino The tetromino to query.
117  * @return The height of the tetromino without
118  *         empty mask rows at the bottom.
119  */
120 int real_height(const tetromino_t *tetromino);
121
122 /**
123  * Cuts through the supplied tetromino horizontally
124  * and removes the specified row, shifting all rows
125  * below that one up by 1.
126  * @param tetromino The tetromino to edit.
127  * @param row The row to remove.
128  */
129 void remove_tetromino_row(const tetromino_t *tetromino, int row);
130
131 /**
132  * @param tetromino The tetromino from which to fetch the offset.
133  * @return The left offset of the actual tetromino
134  *         within its tile-occupation-mask.
135  */
136 int left_offset(const tetromino_t *tetromino);
137
138 /**
139  * @param tetromino The tetromino from which to fetch the offset.
140  * @return The right offset of the actual tetromino
141  *         within its tile-occupation-mask.
142  */
143 int right_offset(const tetromino_t *tetromino);
144
145 /**
146  * Destroys all data associated with the supplied tetromino.
147  * @param tetromino The tetromino to destroy.
148  */
149 void destroy_tetromino(tetromino_t *tetromino);
150
151 #endif //!UE06_TETROMINO_H

```

Listing 5: *tetromino.c*

```

1 #include "tetromino.h"
2 #include <assert.h>
3 #include <GLFW/glfw3.h>
4 #include <stdlib.h>
5 #include <stdio.h>
6 #include <memory.h>
7
8 /**
9  * This is a highly inflexible macro
10  * which one should never use in larger
11  * scale projects. But since this is a
12  * small exercise, I use it because I
13  * am lazy and I think it improves
14  * readability.
15  */
16 #define BOOL_MASK(dim, col) { \
17     ROTATED_DEFAULT, \

```

```
18     {0, 0}, \
19         (col), \
20         (dim), \
21         (bool[(dim)*(dim)]) {
22     #define END_MASK }}
23
24     /**
25     * @ @ @ @
26     */
27     static const tetromino_t _I = BOOL_MASK(4, color_red)
28         0, 0, 0, 0,
29         1, 1, 1, 1,
30         0, 0, 0, 0,
31         0, 0, 0, 0
32     END_MASK;
33
34     /**
35     * @
36     * @
37     * @ @
38     */
39     static tetromino_t _J = BOOL_MASK(3, color_green)
40         0, 1, 0,
41         0, 1, 0,
42         1, 1, 0
43     END_MASK;
44
45     /**
46     * @
47     * @
48     * @ @
49     */
50     static tetromino_t _L = BOOL_MASK(3, color_blue)
51         0, 1, 0,
52         0, 1, 0,
53         0, 1, 1
54     END_MASK;
55
56     /**
57     * @ @
58     * @ @
59     */
60     static tetromino_t _O = BOOL_MASK(2, color_yellow)
61         1, 1,
62         1, 1
63     END_MASK;
64
65     /**
66     * @ @
67     * @ @
68     */
69     static tetromino_t _S = BOOL_MASK(3, color_magenta)
70         0, 1, 1,
71         1, 1, 0,
72         0, 0, 0
73     END_MASK;
74
```



```

75  /**
76   * @ @ @
77   *  @
78   */
79  static tetromino_t _T = BOOL_MASK(3, color_cyan)
80      0, 0, 0,
81      1, 1, 1,
82      0, 1, 0
83  END_MASK;
84
85  /**
86   * @ @
87   *  @ @
88   */
89  static tetromino_t _Z = BOOL_MASK(3, color_white)
90      1, 1, 0,
91      0, 1, 1,
92      0, 0, 0
93  END_MASK;
94
95  /**
96   * The tetromino returned for invalid
97   * requests.
98   */
99  static tetromino_t _Invalid = {ROTATED_DEFAULT, {0, 0}, color_black, 0, NULL
100      };
101
102  /**
103   * A function that maps a row and a column to an index in
104   * the matrix. By modifying the mapping those functions
105   * can rotate indices.
106   */
107  typedef int (*rotation_func)(int side_len, int row, int col);
108
109  /**
110   * @param tetromino The tetromino for which to find
111   * a rotation function
112   * @return The function representing a rotation
113   * that corresponds with the tetrominos
114   * orientation.
115   */
116  static rotation_func get_rotation_func(const tetromino_t *tetromino);
117  // todo document
118
119  /**
120   * Renders a tetromino tile to the current OpenGL context.
121   * @param pos The position at which to draw the tile.
122   * @param color The color to use for the tile.
123   */
124  static void render_quad(const position_t * pos, const color_t * color);
125
126  /**
127   * @param side_len The side length of the matrix.
128   * @param row The row of the element to fetch.
129   * @param col The column of the element to fetch.
130   * @return A value from the matrix, reading it

```

```

131 *      from the top left to the bottom right.
132 */
133 static int rotate_none(int side_len, int row, int col);
134
135 /**
136 * @param side_len The side length of the matrix.
137 * @param row The row of the element to fetch.
138 * @param col The column of the element to fetch.
139 * @return A value from the matrix, reading it
140 *      from the bottom left to the top right.
141 */
142 static int rotate_right(int side_len, int row, int col);
143
144 /**
145 * @param side_len The side length of the matrix.
146 * @param row The row of the element to fetch.
147 * @param col The column of the element to fetch.
148 * @return A value from the matrix, reading it
149 *      from the top right to the bottom left.
150 *      (Basically reverse)
151 */
152 static int rotate_half(int side_len, int row, int col);
153
154 /**
155 * @param side_len The side length of the matrix.
156 * @param row The row of the element to fetch.
157 * @param col The column of the element to fetch.
158 * @return A value from the matrix, reading it
159 *      from the top left to the bottom right.
160 */
161 static int rotate_left(int side_len, int row, int col);
162
163 /**
164 * Rotates the supplied tetromino clockwise.
165 * @param tetromino The tetromino to rotate.
166 */
167 static void rotate_cw(tetromino_t *tetromino);
168
169 /**
170 * Rotates the supplied tetromino counter clockwise.
171 * @param tetromino The tetromino to rotate.
172 */
173 static void rotate_ccw(tetromino_t *tetromino);
174
175 /**
176 * @param tetromino The tetromino to clone.
177 * @return A deep clone of the supplied tetromino.
178 */
179 static tetromino_t clone_tetromino(const tetromino_t *tetromino);
180
181 tetromino_t create_tetromino(tetromino_shape shape)
182 {
183     switch (shape) {
184         case I: return clone_tetromino(&_I);
185         case J: return clone_tetromino(&_J);
186         case L: return clone_tetromino(&_L);
187         case O: return clone_tetromino(&_O);

```

```

188     case S: return clone_tetromino(&_S);
189     case T: return clone_tetromino(&_T);
190     case Z: return clone_tetromino(&_Z);
191     default:
192         printf("Unknown tetromino: %d", shape);
193         return _Invalid;
194     }
195 }
196
197 void render_tetromino(const tetromino_t *tetromino)
198 {
199     if (tetromino != NULL) {
200         int len = tetromino->side_len;
201         for (int row = 0; row < len; ++row) {
202             for (int col = 0; col < len; ++col) {
203                 if (get_tetromino_mask_at(tetromino, row, col)) {
204                     position_t relative = {
205                         tetromino->pos.x + col,
206                         tetromino->pos.y + row
207                     };
208                     render_quad(&relative, &tetromino->color);
209                 }
210             }
211         }
212     }
213 }
214
215 void rotate_tetromino(tetromino_t *tetromino, rotation_t rotation)
216 {
217     if (tetromino != NULL) {
218         switch (rotation) {
219             case CLOCKWISE:
220                 rotate_cw(tetromino);
221                 break;
222             case COUNTER_CLOCKWISE:
223                 rotate_ccw(tetromino);
224                 break;
225         }
226     }
227 }
228
229 int real_height(const tetromino_t *tetromino)
230 {
231     int height = 0;
232     if (tetromino != NULL) {
233         int row = 0;
234         while (row < tetromino->side_len) {
235             int col = 0;
236             while (col < tetromino->side_len && !get_tetromino_mask_at(
237                 tetromino, row, col)) {
238                 ++col;
239             }
240             if (col < tetromino->side_len) {
241                 height = row + 1;
242             }
243             ++row;
244         }
245     }

```

```

244     }
245     return height;
246 }
247
248 bool get_tetromino_mask_at(const tetromino_t *tetromino, int row, int col)
249 {
250     bool mask = false;
251     if (tetromino != NULL) {
252         rotation_func rotate = get_rotation_func(tetromino);
253         assert(rotate);
254         mask = tetromino->mask[rotate(tetromino->side_len, row, col)];
255     }
256     return mask;
257 }
258
259 void remove_tetromino_row(const tetromino_t *tetromino, int row)
260 {
261     if (tetromino != NULL && row >= 0 && row < tetromino->side_len) {
262         rotation_func rotate = get_rotation_func(tetromino);
263         assert(rotate);
264         // shift all rows below <row> up by 1
265         while (row < tetromino->side_len - 1) {
266             for (int col = 0; col < tetromino->side_len; ++col) {
267                 tetromino->mask[rotate(tetromino->side_len, row, col)] =
268                     get_tetromino_mask_at(tetromino, row + 1, col);
269             }
270             ++row;
271         }
272
273         // clear last row
274         for (int col = 0; col < tetromino->side_len; ++col) {
275             tetromino->mask[rotate(tetromino->side_len, tetromino->side_len -
276             1, col)] = false;
277         }
278     }
279
280 int left_offset(const tetromino_t *tetromino)
281 {
282     // delegate work to real_height
283     int offset = 0;
284     if (tetromino != NULL) {
285         tetromino_t cpy = *tetromino;
286         rotate_tetromino(&cpy, COUNTER_CLOCKWISE);
287         offset = tetromino->side_len - real_height(&cpy);
288     }
289     return offset;
290 }
291
292 int right_offset(const tetromino_t *tetromino)
293 {
294     // delegate work to real_height
295     int offset = 0;
296     if (tetromino != NULL) {
297         tetromino_t cpy = *tetromino;
298         rotate_tetromino(&cpy, CLOCKWISE);
299         offset = tetromino->side_len - real_height(&cpy);

```

```

300     }
301     return offset;
302 }
303
304 void destroy_tetromino(tetromino_t *tetromino)
305 {
306     if (tetromino != NULL) {
307         free(tetromino->mask);
308         tetromino->mask = NULL;
309         tetromino->side_len = 0;
310     }
311 }
312
313 // ----- TRANSLATION-UNIT-LOCAL FUNCTIONS
314 // -----
315 void render_quad(const position_t * pos, const color_t * color) {
316     static_assert(sizeof(*color) == 4, "detected unexpected size for colors")
317     ;
318     glColor3ubv((unsigned char *)color);
319     glBegin(GL_QUADS); {
320         glVertex2i(pos->x, pos->y);
321         glVertex2i(pos->x, pos->y + 1);
322         glVertex2i(pos->x + 1, pos->y + 1);
323         glVertex2i(pos->x + 1, pos->y);
324     } glEnd();
325 }
326
327 rotation_func get_rotation_func(const tetromino_t *tetromino)
328 {
329     assert(tetromino);
330     rotation_func rotate = NULL;
331     switch (tetromino->orientation) {
332         case ROTATED_DEFAULT:
333             rotate = rotate_none;
334             break;
335         case ROTATED_RIGHT:
336             rotate = rotate_right;
337             break;
338         case ROTATED_LEFT:
339             rotate = rotate_left;
340             break;
341         case ROTATED_HALF:
342             rotate = rotate_half;
343             break;
344     }
345     return rotate;
346 }
347
348 void rotate_cw(tetromino_t *tetromino)
349 {
350     assert(tetromino);
351     switch (tetromino->orientation) {
352         case ROTATED_DEFAULT:
353             tetromino->orientation = ROTATED_RIGHT;
354             break;
355         case ROTATED_RIGHT:

```

```
355         tetromino->orientation = ROTATED_HALF;
356         break;
357     case ROTATED_HALF:
358         tetromino->orientation = ROTATED_LEFT;
359         break;
360     case ROTATED_LEFT:
361         tetromino->orientation = ROTATED_DEFAULT;
362         break;
363     }
364 }
365
366 void rotate_ccw(tetromino_t *tetromino)
367 {
368     assert(tetromino);
369     switch (tetromino->orientation) {
370         case ROTATED_DEFAULT:
371             tetromino->orientation = ROTATED_LEFT;
372             break;
373         case ROTATED_RIGHT:
374             tetromino->orientation = ROTATED_DEFAULT;
375             break;
376         case ROTATED_LEFT:
377             tetromino->orientation = ROTATED_HALF;
378             break;
379         case ROTATED_HALF:
380             tetromino->orientation = ROTATED_RIGHT;
381             break;
382     }
383 }
384
385 // the rotate functions are black magic
386 // in its pures form!
387
388 int rotate_none(int side_len, int row, int col)
389 {
390     return row * side_len + col;
391 }
392
393 int rotate_right(int side_len, int row, int col)
394 {
395     return rotate_none(side_len, side_len - col - 1, row);
396 }
397
398 int rotate_half(int side_len, int row, int col)
399 {
400     return side_len * side_len - rotate_none(side_len, row, col) - 1;
401 }
402
403 int rotate_left(int side_len, int row, int col)
404 {
405     return rotate_none(side_len, col, side_len - row - 1);
406 }
407
408 tetromino_t clone_tetromino(const tetromino_t *tetromino)
409 {
410     assert(tetromino);
411     tetromino_t result;
```

```
412
413     result.side_len = tetromino->side_len;
414     result.pos = tetromino->pos;
415     result.orientation = tetromino->orientation;
416     result.color = tetromino->color;
417
418     size_t arr_length = ((size_t) result.side_len * (size_t) result.side_len)
419         ;
419     result.mask = malloc(sizeof(bool) * arr_length);
420     memcpy(result.mask, tetromino->mask, arr_length);
421
422     return result;
423 }
```

1.3 Tests

Ich wusste bei Gott nicht wie ich sprechende Tests machen soll. Also habe ich einfach "oft" gespielt und nur den Fehler entdeckt, welchen ich im Abschnitt "Beenden des Spiels" angesprochen habe.

Kollisionen werden richtig erkannt, Bewegungen verhindert falls sie Kollisionen verursachen würden. Sowohl "Bodenreihen", als auch beliebige andere Reihen am Spielbrett werden korrekt abgebaut, wenn sie gefüllt sind.

Der Speicher wird artgerecht freigelassen (obwohl ich manchmal SEGFAULTs erhalte, die aber aus dem GLFW-Framework kommen).