Ausarbeitung 06

Niklas Vest

November 23, 2018

1 Tetris

1.1 Lösungsidee

Tetrominos werden durch eine $n \times n$ Matrix von boolschen Werten dargestellt, wobei $n = max(tetrLnge, tetrHhe)$. 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 vorraus, 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 reskursiven 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*

```c
#include <stdio.h>
#include <stdlib.h>
#include "GLFW/glfw3.h"

#include "tetromino.h"
#include "gameboard.h"

#define WIDTH  400
#define HEIGHT WIDTH * (GB_ROWS / GB_COLS)

/**
 * Prints the keyboard controls to the
 * standard output.
 */
static void print_instructions()
{
    printf("Keymap:\n");
    printf("Arrow Up ^ : Rotate tetromino clockwise\n");
    printf("Arrow Down v : Rotate tetromino counter clockwise\n");
    printf("Arrow Right > : Move tetromino to the right\n");
    printf("Arrow Left < : Move tetromino to the left\n");
    printf("Space _ : Drop the tetromino like it’s hot\n");
}

int main()
{
    GLFWwindow* window;
    if (!glfwInit())
        return -1;

    window = glfwCreateWindow(WIDTH, HEIGHT, "Tetris 4.0", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        printf("Failed to create window.\n");
        return EXIT_SUCCESS;
    }

    print_instructions();

    int width, height;
    glfwGetWindowSize(window, &width, &height);
    glfwSetWindowAspectRatio(window, width, height);

    glfwMakeContextCurrent(window);
    glfwSetKeyCallback(window, gameboard_keymap);

    init_game_board();

    double last_time = glfwGetTime();

    while (!glfwWindowShouldClose(window))
    {
        // the board is always rendered!
        // when running == false, the background
        // color fades through a few colors over
        // time!
        render_board(window, width, height);

        if (running) {
            // the update rate changes over time so
            // it’s a variable
            if (glfwGetTime() - last_time > (1.0 / ticks_pers_second)) {
                last_time = glfwGetTime();
                update_board();
            }
        }

        const GLenum error = glGetError();
        if(error != GL_NO_ERROR) fprintf(stderr, "ERROR: %d\n", error);
```

```c
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    destroy_game_board();
    glfwDestroyWindow(window);
    glfwTerminate();
    return EXIT_SUCCESS;
}
```

Listing 2: *gameboard.h*

```c
#ifndef GAMEBOARD_H
#define GAMEBOARD_H

#include <GLFW/glfw3.h>
#include "tetromino.h"

#define GB_ROWS 22
#define GB_COLS 11

/**
 * Every information about the
 * game board and the current state
 * of the game.
 */
typedef struct {
    /**
     * All tetrominoes that are on the game board.
     */
    tetromino_t *tetrominoes;

    /**
     * The number of tetrominoes on the game board.
     */
    int nr_of_tetrominoes;

    /**
     * The current size of the tetromino vector.
     */
    int vec_size;

    /**
     * A matrix of boolean values which state whether
     * a particular tile is occupied.
     */
    bool *tile_matrix;
} tetris_board_t;

/**
 * The single game board instance.
 */
extern tetris_board_t game_board;

/**
 * A global flag which reflects
 * the current state of the game.
 */
```

```
extern bool running;

/**
 * The number of updates that
 * the game receives per second
 * (approximately).
 */
extern int ticks_pers_second;

/**
 * Initializes the game board.
 * (Bet u didn't suspect that huh?)
 */
void init_game_board();

/**
 * Renders the game board to the supplied glfw window.
 * @param window The window to draw to.
 * @param width The width of the viewport.
 * @param height The height of the viewport.
 */
void render_board(GLFWwindow *window, int width, int height);

/**
 * Updates the game state.
 */
void update_board();

/**
 * The keymap for the game. (Using GLFWs setKeyCallback)
 */
void gameboard_keymap(GLFWwindow* window, int key, int scancode, int action,
     int mods);

/**
 * Destroys the game board and cleans up.
 */
void destroy_game_board();

#endif
```

Listing 3: *gameboard.c*

```
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdbool.h>
#include "gameboard.h"

/**
 * @return A reference to the currently falling tetromino.
 */
static tetromino_t *currently_falling_tetromino();

/**
 * Applies a translation to the supplied tetromino.
```

```c
 * @param tetromino The tetromino to move.
 * @param xDelta The distance to move it on the x-axis.
 * @param yDelta The distance to move it on the y-axis.
 */
static void move_tetromino(tetromino_t *tetromino, int xDelta, int yDelta);

/**
 * Generates a new tetromino and adds it to the game board.
 */
static void spawn_tetromino();

/**
 * Writes the supplied tetromino to the game board matrix
 * with regards to its current position (on the board).
 * @param tetromino The tetromino to write.
 */
static void write_tetromino_to_matrix(const tetromino_t *tetromino);

/**
 * Removes any rows that are filled with tiles .
 */
static void remove_first_filled_row();

/**
 * @param row The row to check.
 * @return True if the specified row is filled with tiles ,
 *         False otherwise.
 */
static bool row_can_be_deleted(int row);

/**
 * @param tetromino The tetromino to check.
 * @return True if the supplied tetromino can still
 *         fall further down on the game board without
 *         intersecting with other tetrominoes.
 */
static bool can_fall(const tetromino_t *tetromino);

/**
 * Attempts to rotate the tetromino. If the rotation
 * would lead to the tetromino goin out of bounds
 * the rotation is prevented.
 * @param tetromino The tetromino to rotate.
 * @param rotation The rotation direction.
 */
static void rotate_within_board(tetromino_t *tetromino, rotation_t rotation);

/**
 * @param tetromino The tetromino to check.
 * @return True if the tetromino itself is within
 *        the board boundaries, False otherwise.
 */
static bool is_within_board(const tetromino_t *tetromino);

/**
 * @return True if at least one tile of the uppermost row
 *         is not empty, False otherwise.
 */
static bool is_board_full();

// global (extern) variables
tetris_board_t game_board;
bool running = true;
int ticks_pers_second = 2;

void init_game_board()
{
    // seed RNG with system time
    time_t t;
    srand((unsigned) time(&t));

    // the average player won't be able to reconcile more
    // than 20 tetrominoes on the board :P
    game_board.tetrominoes = malloc(sizeof(tetromino_t) * 50);
    game_board.vec_size = 50;

    // create a matrix representing the game boards occupied tiles
    game_board.tile_matrix = malloc(sizeof(bool) * GB_ROWS * GB_COLS);
    for (size_t i = 0; i < GB_ROWS * GB_COLS; ++i) {
        game_board.tile_matrix[i] = false;
    }

    game_board.nr_of_tetrominoes = 0;
}

void render_board(GLFWwindow *window, int width, int height)
{
    glfwGetFramebufferSize(window, &width, &height);
    glViewport(0, 0, width, height);
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // basically flip this boi so that the coordinate system
    // has its origin in the upper left hand corner. that makes
    // it easier to work with the matrices!
    glOrtho(0, width, height, 0, 0, 1);

    glScalef((float) width / (float) GB_COLS, (float) height / (float) GB_ROWS, 1);

    // let all tetrominoes render themselves
    if (running) {
        for (int i = 0; i < game_board.nr_of_tetrominoes; ++i) {
            render_tetromino(&game_board.tetrominoes[i]);
        }
    } else {
        // hehe cheeky macro!
#define color(f) (float) (f(glfwGetTime()))
        glClearColor(color(0.7*sin), color(cos), color(0.3*sin), 1.0f);
    }
}

void update_board()
```

```c
{
    tetromino_t *curr = currently_falling_tetromino();
    if (curr != NULL) {
        if (can_fall(curr)) {
            move_tetromino(curr, 0, 1);
        } else {
            // fix tetromino on the board so
            // new falling tetrominoes can check
            // for intersections
            write_tetromino_to_matrix(curr);
            remove_first_filled_row();
            if (!is_board_full()) {
                spawn_tetromino();
            } else {
                running = false;
            }
        }
    } else { // curr == NULL -> no tetrominoes on the board
        spawn_tetromino();
    }
}

void gameboard_keymap(GLFWwindow *window, int key, int scancode, int action, int mods)
{
    (void) window; (void) scancode; (void) mods;

    tetromino_t *curr = currently_falling_tetromino();
    if (action == GLFW_PRESS) {
        switch (key) {
            case GLFW_KEY_RIGHT:
                move_tetromino(curr, 1, 0);
                break;
            case GLFW_KEY_LEFT:
                move_tetromino(curr, -1, 0);
                break;
            case GLFW_KEY_UP:
                rotate_within_board(curr, CLOCKWISE);
                break;
            case GLFW_KEY_DOWN:
                rotate_within_board(curr, COUNTER_CLOCKWISE);
                break;
            case  GLFW_KEY_SPACE:
                while (can_fall(currently_falling_tetromino())) {
                    update_board();
                }
                break;
            default:
                // do nothing
                break;
        }
    }
}

void destroy_game_board()
{
    assert(game_board.tile_matrix);
    assert(game_board.tetrominoes);

    free(game_board.tile_matrix);
    game_board.tile_matrix = NULL;

    for (int i = 0; i < game_board.nr_of_tetrominoes; ++i) {
        destroy_tetromino(&game_board.tetrominoes[i]);
    }
    free(game_board.tetrominoes);
    game_board.tetrominoes = NULL;

    game_board.nr_of_tetrominoes = 0;
}

// ------------ TRANSLATION-UNIT-LOCAL FUNCTIONS ------------ //

void remove_first_filled_row()
{
    // find the first row that can be deleted
    int row = GB_ROWS - 1;
    while (row > 0 && !row_can_be_deleted(row)) {
        --row;
    }

    if (row > 0) { // a full row exists

        // adjust the masks of tetrominoes that overlap with that row
        int overlap = 0;
        while (overlap < game_board.nr_of_tetrominoes) {
            tetromino_t *curr = &game_board.tetrominoes[overlap];
            if (curr->pos.y <= row && curr->pos.y + curr->side_len - 1 >= row) {
                remove_tetromino_row(curr, row - curr->pos.y);
            }
            if (curr->pos.y <= row) {
                curr->pos.y += 1;
            }
            ++overlap;
        }

        // rewrite matrix with ”new tetromino configuration”
        for (int i = 0; i < GB_COLS * GB_ROWS; ++i) {
            game_board.tile_matrix[i] = false;
        }
        for (int i = 0; i < game_board.nr_of_tetrominoes; ++i) {
            write_tetromino_to_matrix(&game_board.tetrominoes[i]);
        }

        // recurse if any rows have been removed
        // (in case row right above is filled as well)
        remove_first_filled_row();
    }
}

bool row_can_be_deleted(int row)
{
    int i = 0;
    while (i < GB_COLS && game_board.tile_matrix[row * GB_COLS + i]) {
        ++i;
    }
    return i == GB_COLS;
}

void write_tetromino_to_matrix(const tetromino_t *tetromino)
{
    assert(tetromino);
    int row_offset = tetromino->pos.y;
    int col_offset = tetromino->pos.x;

    for (int row = 0; row < tetromino->side_len; ++row) {
        for (int col = 0; col < tetromino->side_len; ++col) {
            // calculate the game board index to write to using
            // the tetrominoes position.

            if (col_offset + col >= 0) {
                // tetrominoes can have a blank mask column / row;
                // that row is allowed to go outside the bounds of
                // the game board so that the actual tetromino can
                // align to the edge, thats why I test for x >= 0

                // EDIT: I think there should also be tests for
                // x > GB_COLS and y > GB_ROWS but it works and
                // I don't want to break anything

                int index = (row_offset + row) * GB_COLS + (col_offset + col);
                game_board.tile_matrix[index] |= get_tetromino_mask_at(tetromino, row, col);
            }
        }
    }
}

void spawn_tetromino()
{
    // resize the vector if its full
    if (game_board.nr_of_tetrominoes == game_board.vec_size) {
        // for some reason realloc removed like the 4th element
        // from the original vector ??? And I thought I understood
        // pointers...
        tetromino_t *new_vec = malloc(sizeof(tetromino_t) * (size_t) game_board.vec_size * 2);
        for (int i = 0; i < game_board.vec_size; ++i) {
            new_vec[i] = game_board.tetrominoes[i];
        }
        free(game_board.tetrominoes);
        game_board.tetrominoes = new_vec;
        game_board.vec_size *= 2;
    }

    // exploit the nature of enums: the underlying data type is numeric
    // so a tetromino can be generated using RNG
    tetromino_shape shape = rand() % 7;
    tetromino_t tetromino = create_tetromino(shape);
    tetromino.pos.x = GB_COLS / 2 - tetromino.side_len / 2;
    game_board.tetrominoes[game_board.nr_of_tetrominoes] = tetromino;
    game_board.nr_of_tetrominoes += 1;

    // also increase speed after every 5th tetromino spawn
    if (game_board.nr_of_tetrominoes % 10 == 0) {
        ticks_pers_second += 1;
    }
}

void move_tetromino(tetromino_t *tetromino, int xDelta, int yDelta)
{
    assert(tetromino);
    tetromino_t cpy = *tetromino;
    cpy.pos.x+=xDelta;
    cpy.pos.y+=yDelta;
    if (is_within_board(&cpy)) {
        // apply changes if they are valid
        tetromino->pos.x += xDelta;
        tetromino->pos.y += yDelta;
    }
}

void rotate_within_board(tetromino_t *tetromino, rotation_t rotation)
{
    assert(tetromino);
    tetromino_t cpy = *tetromino;
    rotate_tetromino(&cpy, rotation);
    if (is_within_board(&cpy)) {
        // apply changes if they are valid
        rotate_tetromino(tetromino, rotation);
    }
}

bool can_fall(const tetromino_t *tetromino)
{
    assert(tetromino);
    tetromino_t tetr_cpy = *tetromino;
    tetr_cpy.pos.y += 1;

    int row_offset = tetr_cpy.pos.y;
    int col_offset = tetr_cpy.pos.x;

    // if the tetromino hit the bottom already,
    // it can't fall any further
    bool can_fall = row_offset + real_height(&tetr_cpy) <= GB_ROWS;
    // if it's still above ground level, check if it would
    // intersect with other tetrominoes if it were to fall
    // one tile further
    int row = 0;
    while (row < tetr_cpy.side_len && can_fall) {
        int col = 0;
        while (col < tetr_cpy.side_len && can_fall) {
            // get the matrix index relative to the tetrominoes position
            int gb_index = (row_offset + row) * GB_COLS + (col_offset + col);
            // either one of the tile flags must be False, otherwise they intersect
            can_fall = game_board.tile_matrix[gb_index] == false ||
                    get_tetromino_mask_at(&tetr_cpy, row, col) == false;
            ++col;
        }
        ++row;
    }
    return can_fall;
}

tetromino_t *currently_falling_tetromino()
{
    tetromino_t *last = NULL;
    if (game_board.nr_of_tetrominoes > 0) {
        last = &game_board.tetrominoes[game_board.nr_of_tetrominoes - 1];
    }
    return last;
}

bool is_within_board(const tetromino_t *tetromino)
{
    assert(tetromino);

    // the x position might be negative with the full tetromino
    // still being completely within bounds, hence the complex check
    bool boundary_ok = tetromino->pos.x + left_offset(tetromino) >= 0 &&
                       tetromino->pos.x + (tetromino->side_len - right_offset(tetromino) - 1) < GB_COLS;

    bool no_intersects = true;
    int row = 0;
    while (row < tetromino->side_len && no_intersects) {
        int col = 0;
        while (col < tetromino->side_len && no_intersects) {
            int matrix_ind = (tetromino->pos.y + row) * GB_COLS + (tetromino->pos.x + col);
            // if both tiles are set, they intersect
            no_intersects = get_tetromino_mask_at(tetromino, row, col) == false ||
                            game_board.tile_matrix[matrix_ind] == false;
            ++col;
        }
        ++row;
    }

    return boundary_ok && no_intersects;
}

bool is_board_full()
{
    int col = 0;
    while (col < GB_COLS && game_board.tile_matrix[col] == 0) {
        ++col;
    }
    return col != GB_COLS;
}
```

Listing 4: *tetromino.h*

```c
#ifndef UE06_TETROMINO_H
#define UE06_TETROMINO_H

#include <stdbool.h>
#include <stdlib.h>

/**
 * Handy color typedefs for drawing c:
 */
typedef enum {
    color_black,
    color_red     = 0x0000FFU,
    color_green   = 0x00FF00U,
    color_blue    = 0xFF0000U,
    color_yellow  = color_red   | color_green,
    color_magenta = color_red   | color_blue,
    color_cyan    = color_green | color_blue,
    color_white   = color_red   | color_green | color_blue,
} color_t;

/**
 * A 2D position on the game board.
 */
typedef struct {
    int x, y;
} position_t;

/**
 * An array holding a mask which represents
 * a tetromino.
 */
typedef bool* tetromino_mask_t;

typedef enum {
    ROTATED_DEFAULT, ROTATED_RIGHT, ROTATED_LEFT, ROTATED_HALF
} orientation_t;

typedef enum {
    CLOCKWISE, COUNTER_CLOCKWISE
} rotation_t;

/**
 * A compound representing a tetris object,
 * called a tetromino.
 */
typedef struct {
    /**
     * The current orientation (/rotation)
     * of the tetromino.
     */
    orientation_t orientation;

    /**
     * The current position of the
     * tetromino.
     */
    position_t pos;

    /**
     * The color of the tetromino.
     */
    color_t color;

    /**
     * The side length of the matrix representing
     * the tetrominos tile-occupation-mask.
     */
    int side_len;

    /**
     * The tetrominos
     *  tile-occupation-mask.
     */
    tetromino_mask_t mask;
} tetromino_t;

/**
 * An enumeration containing all Tetrominoes.
 */
typedef enum {
    I, J, L, O, S, T, Z
} tetromino_shape;

/**
 * @param shape The shape of the tetromino to create.
 * @return A copy of the tetromino prototype with the
 *         specified shape.
 */
tetromino_t create_tetromino(tetromino_shape shape);

/**
 * Renders a tetromino to the current OpenGL Context.
 * @param tetromino The tetromino to render.
 */
void render_tetromino(const tetromino_t *tetromino);

/**
 * Rotates the supplied tetromino according to the
 *  specified rotation direction.
 * @param tetromino The tetromino to rotate.
 * @param rotation The direction in which to rotate
 *                 dat boi.
 */
void rotate_tetromino(tetromino_t *tetromino, rotation_t rotation);

/**
 * @param tetromino The tetromino to query.
 * @param row
 * @param col
 * @return The mask value (the occupation) at the
 *         specified row and column.
 */
bool get_tetromino_mask_at(const tetromino_t *tetromino, int row, int col);
```

```c

/**
 * @param tetromino The tetromino to query.
 * @return The height of the tetromino without
 *         empty mask rows at the bottom.
 */
int real_height(const tetromino_t *tetromino);

/**
 * Cuts through the supplied tetromino horizontally
 * and removes the specified row, shifting all rows
 * below that one up by 1.
 * @param tetromino The tetromino to edit.
 * @param row The row to remove.
 */
void remove_tetromino_row(const tetromino_t *tetromino, int row);

/**
 * @param tetromino The tetromino from which to fetch the offset.
 * @return The left offset of the actual tetromino
 *         within its tile-occupation-mask.
 */
int left_offset(const tetromino_t *tetromino);

/**
 * @param tetromino The tetromino from which to fetch the offset.
 * @return The right offset of the actual tetromino
 *         within its tile-occupation-mask.
 */
int right_offset(const tetromino_t *tetromino);

/**
 * Destroys all data associated with the supplied tetromino.
 * @param tetromino The tetromino to destroy.
 */
void destroy_tetromino(tetromino_t *tetromino);

#endif //!UE06_TETROMINO_H
```

Listing 5: *tetromino.c*

```c
#include "tetromino.h"
#include <assert.h>
#include <GLFW/glfw3.h>
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>

/**
 * This is a highly inflexible macro
 * which one should never use in larger
 * scale projects. But since this is a
 * small exercise, I use it because I
 * am lazy and I think it improves
 * readability.
 */
#define BOOL_MASK(dim, col) { \
    ROTATED_DEFAULT, \
    {0, 0}, \
        (col), \
        (dim), \
    (bool[(dim)*(dim)]) {
#define END_MASK }}

/**
 * @ @ @ @
 */
static const tetromino_t _I = BOOL_MASK(4, color_red)
    0, 0, 0, 0,
    1, 1, 1, 1,
    0, 0, 0, 0,
    0, 0, 0, 0
END_MASK;

/**
 *   @
 *   @
 * @ @
 */
static tetromino_t _J = BOOL_MASK(3, color_green)
    0, 1, 0,
    0, 1, 0,
    1, 1, 0
END_MASK;

/**
 * @
 * @
 * @ @
 */
static tetromino_t _L = BOOL_MASK(3, color_blue)
    0, 1, 0,
    0, 1, 0,
    0, 1, 1
END_MASK;

/**
 * @ @
 * @ @
 */
static tetromino_t _O = BOOL_MASK(2, color_yellow)
    1, 1,
    1, 1
END_MASK;

/**
 *   @ @
 * @ @
 */
static tetromino_t _S = BOOL_MASK(3, color_magenta)
    0, 1, 1,
    1, 1, 0,
    0, 0, 0
END_MASK;

```

```
/**
 * @ @ @
 *   @
 */
static tetromino_t _T = BOOL_MASK(3, color_cyan)
    0, 0, 0,
    1, 1, 1,
    0, 1, 0
END_MASK;

/**
 * @ @
 *   @ @
 */
static tetromino_t _Z = BOOL_MASK(3, color_white)
    1, 1, 0,
    0, 1, 1,
    0, 0, 0
END_MASK;

/**
 * The tetromino returned for invalid
 * requests.
 */
static tetromino_t _Invalid = {ROTATED_DEFAULT, {0, 0}, color_black, 0, NULL
    };

/**
 * A function that maps a row and a column to an index in
 * the matrix. By modifying the mapping those functions
 * can rotate indices.
 */
typedef int (*rotation_func)(int side_len, int row, int col);

/**
 * @param tetromino The tetromino for which to find
 *           a rotation function
 * @return The function representing a rotation
 *           that corresponds with the tetrominos
 *           orientation.
 */
static rotation_func get_rotation_func(const tetromino_t *tetromino);

// todo document

/**
 * Renders a tetromino tile to the current OpenGL context.
 * @param pos The position at which to draw the tile.
 * @param color The color to use for the tile.
 */
static void render_quad(const position_t * pos, const color_t * color);

/**
 * @param side_len The side length of the matrix.
 * @param row The row of the element to fetch.
 * @param col The column of the element to fetch.
 * @return A value from the matrix, reading it
```

```c
 *           from the top left to the bottom right.
 */
static int rotate_none(int side_len, int row, int col);

/**
 * @param side_len The side length of the matrix.
 * @param row The row of the element to fetch.
 * @param col The column of the element to fetch.
 * @return A value from the matrix, reading it
 *           from the bottom left to the top right.
 */
static int rotate_right(int side_len, int row, int col);

/**
 * @param side_len The side length of the matrix.
 * @param row The row of the element to fetch.
 * @param col The column of the element to fetch.
 * @return A value from the matrix, reading it
 *           from the top right to the bottom left.
 *           (Basically reverse)
 */
static int rotate_half(int side_len, int row, int col);

/**
 * @param side_len The side length of the matrix.
 * @param row The row of the element to fetch.
 * @param col The column of the element to fetch.
 * @return A value from the matrix, reading it
 *           from the top left to the bottom right.
 */
static int rotate_left(int side_len, int row, int col);

/**
 * Rotates the supplied tetromino clockwise.
 * @param tetromino The tetromino to rotate.
 */
static void rotate_cw(tetromino_t *tetromino);

/**
 * Rotates the supplied tetromino counter clockwise.
 * @param tetromino The tetromino to rotate.
 */
static void rotate_ccw(tetromino_t *tetromino);

/**
 * @param tetromino The tetromino to clone.
 * @return A deep clone of the supplied tetromino.
 */
static tetromino_t clone_tetromino(const tetromino_t *tetromino);

tetromino_t create_tetromino(tetromino_shape shape)
{
    switch (shape) {
        case I: return clone_tetromino(&_I);
        case J: return clone_tetromino(&_J);
        case L: return clone_tetromino(&_L);
        case O: return clone_tetromino(&_O);
        case S: return clone_tetromino(&_S);
        case T: return clone_tetromino(&_T);
        case Z: return clone_tetromino(&_Z);
        default:
            printf("Unknown tetromino: %d", shape);
            return _Invalid;
    }
}

void render_tetromino(const tetromino_t *tetromino)
{
    if (tetromino != NULL) {
        int len = tetromino->side_len;
        for (int row = 0; row < len; ++row) {
            for (int col = 0; col < len; ++col) {
                if (get_tetromino_mask_at(tetromino, row, col)) {
                    position_t relative = {
                            tetromino->pos.x + col,
                            tetromino->pos.y + row
                    };
                    render_quad(&relative, &tetromino->color);
                }
            }
        }
    }
}

void rotate_tetromino(tetromino_t *tetromino, rotation_t rotation)
{
    if (tetromino != NULL) {
        switch (rotation) {
            case CLOCKWISE:
                rotate_cw(tetromino);
                break;
            case COUNTER_CLOCKWISE:
                rotate_ccw(tetromino);
                break;
        }
    }
}

int real_height(const tetromino_t *tetromino)
{
    int height = 0;
    if (tetromino != NULL) {
        int row = 0;
        while (row < tetromino->side_len) {
            int col = 0;
            while (col < tetromino->side_len && !get_tetromino_mask_at(
     tetromino, row, col)) {
                ++col;
            }
            if (col < tetromino->side_len) {
                height = row + 1;
            }
            ++row;
        }
```

```c
    }
    return height;
}

bool get_tetromino_mask_at(const tetromino_t *tetromino, int row, int col)
{
    bool mask = false;
    if (tetromino != NULL) {
        rotation_func rotate = get_rotation_func(tetromino);
        assert(rotate);
        mask = tetromino->mask[rotate(tetromino->side_len, row, col)];
    }
    return mask;
}

void remove_tetromino_row(const tetromino_t *tetromino, int row)
{
    if (tetromino != NULL && row >= 0 && row < tetromino->side_len) {
        rotation_func rotate = get_rotation_func(tetromino);
        assert(rotate);
        // shift all rows below <row> up by 1
        while (row < tetromino->side_len - 1) {
            for (int col = 0; col < tetromino->side_len; ++col) {
                tetromino->mask[rotate(tetromino->side_len, row, col)] =
                        get_tetromino_mask_at(tetromino, row + 1, col);
            }
            ++row;
        }

        // clear last row
        for (int col = 0; col < tetromino->side_len; ++col) {
            tetromino->mask[rotate(tetromino->side_len, tetromino->side_len - 1, col)] = false;
        }
    }
}

int left_offset(const tetromino_t *tetromino)
{
    // delegate work to real_height
    int offset = 0;
    if (tetromino != NULL) {
        tetromino_t cpy = *tetromino;
        rotate_tetromino(&cpy, COUNTER_CLOCKWISE);
        offset = tetromino->side_len - real_height(&cpy);
    }
    return offset;
}

int right_offset(const tetromino_t *tetromino)
{
    // delegate work to real_height
    int offset = 0;
    if (tetromino != NULL) {
        tetromino_t cpy = *tetromino;
        rotate_tetromino(&cpy, CLOCKWISE);
        offset = tetromino->side_len - real_height(&cpy);
    }
    return offset;
}

void destroy_tetromino(tetromino_t *tetromino)
{
    if (tetromino != NULL) {
        free(tetromino->mask);
        tetromino->mask = NULL;
        tetromino->side_len = 0;
    }
}

// ------------ TRANSLATION-UNIT-LOCAL FUNCTIONS ------------ //

void render_quad(const position_t * pos, const color_t * color) {
    static_assert(sizeof(*color) == 4, "detected unexpected size for colors");
    glColor3ubv((unsigned char *)color);
    glBegin(GL_QUADS); {
        glVertex2i(pos->x,     pos->y);
        glVertex2i(pos->x,     pos->y + 1);
        glVertex2i(pos->x + 1, pos->y + 1);
        glVertex2i(pos->x + 1, pos->y);
    } glEnd();
}

rotation_func get_rotation_func(const tetromino_t *tetromino)
{
    assert(tetromino);
    rotation_func rotate = NULL;
    switch (tetromino->orientation) {
        case ROTATED_DEFAULT:
            rotate = rotate_none;
            break;
        case ROTATED_RIGHT:
            rotate = rotate_right;
            break;
        case ROTATED_LEFT:
            rotate = rotate_left;
            break;
        case ROTATED_HALF:
            rotate = rotate_half;
            break;
    }
    return rotate;
}

void rotate_cw(tetromino_t *tetromino)
{
    assert(tetromino);
    switch (tetromino->orientation) {
        case ROTATED_DEFAULT:
            tetromino->orientation = ROTATED_RIGHT;
            break;
        case ROTATED_RIGHT:
```

```c
            tetromino->orientation = ROTATED_HALF;
            break;
        case ROTATED_HALF:
            tetromino->orientation = ROTATED_LEFT;
            break;
        case ROTATED_LEFT:
            tetromino->orientation = ROTATED_DEFAULT;
            break;
    }
}

void rotate_ccw(tetromino_t *tetromino)
{
    assert(tetromino);
    switch (tetromino->orientation) {
        case ROTATED_DEFAULT:
            tetromino->orientation = ROTATED_LEFT;
            break;
        case ROTATED_RIGHT:
            tetromino->orientation = ROTATED_DEFAULT;
            break;
        case ROTATED_LEFT:
            tetromino->orientation = ROTATED_HALF;
            break;
        case ROTATED_HALF:
            tetromino->orientation = ROTATED_RIGHT;
            break;
    }
}

// the rotate functions are black magic
// in its pures form!

int rotate_none(int side_len, int row, int col)
{
    return row * side_len + col;
}

int rotate_right(int side_len, int row, int col)
{
    return rotate_none(side_len, side_len - col - 1, row);
}

int rotate_half(int side_len, int row, int col)
{
    return side_len * side_len - rotate_none(side_len, row, col) - 1;
}

int rotate_left(int side_len, int row, int col)
{
    return rotate_none(side_len, col, side_len - row - 1);
}

tetromino_t clone_tetromino(const tetromino_t *tetromino)
{
    assert(tetromino);
    tetromino_t result;

    result.side_len = tetromino->side_len;
    result.pos = tetromino->pos;
    result.orientation = tetromino->orientation;
    result.color = tetromino->color;

    size_t arr_length = ((size_t) result.side_len * (size_t) result.side_len)
     ;
    result.mask = malloc(sizeof(bool) * arr_length);
    memcpy(result.mask, tetromino->mask, arr_length);

    return result;
}
```

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).