SWO3	Übung zu Softwareentwicklung mit klassischen Sprachen und Bibliotheken 3	WS 2021/22, ÜZ 5

☐ Gruppe M. Hava

☒ Gruppe J. Heinzelreiter Name: Angelos Angelis Aufwand [h]: 21

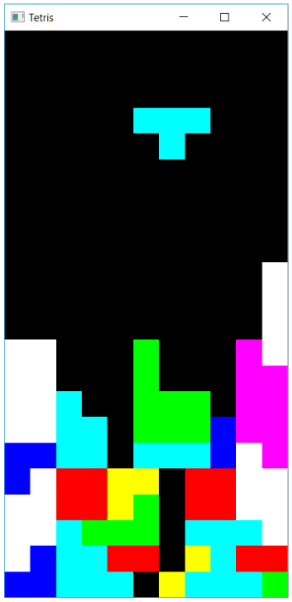
☐ Gruppe P. Kulczycki Peer-Review von: _____

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (100 P)	100	80	100

Beispiel 1: GLFW-Applikation „Tetris“ (`src/tetris/`)

Vervollständigen Sie die in der Übung begonnene GLFW-Applikation „Tetris“. Beachten Sie dabei die folgenden Anforderungen und Hinweise:

1. Es müssen prinzipiell beliebig geformte Spielsteine (Tetriminos) unterstützt werden. Ein Spielstein ist dabei eine Matrix aus eingefärbten „Pixeln“. Natürlich gibt es in Ihrer Applikation einen vordefinierten Satz mit den sieben bekannten Tetriminos I, J, L, O, S, T und Z.
2. Tetriminos müssen die Bewegungen „links“, „rechts“, „drehen“ und „fallen“ durchführen können.
3. Führen Sie eine entsprechende Kollisionsbehandlung durch. Nur die Berücksichtigung eines „minimal umgebenden Rechtecks“ ist nicht ausreichend.
4. Komplett gefüllte Reihen verschwinden vom Bildschirm.
5. Die Fallgeschwindigkeit der Tetriminos erhöht sich mit Fortgang eines Spiels.
6. Ein Spiel endet, sobald die nicht abgebauten Reihen den Bildschirm füllen.
7. Strukturieren Sie sauber, indem Sie Module und (Hilfs-)Funktionen schreiben. Auch die in der Übungsstunde vorgezeigten und implementierten Funktionen können noch besser strukturiert werden.
8. Siehe die Quelle https://en.wikipedia.org/wiki/Tetris und vor allem die Quelle http://tetris.wikia.com/wiki/Tetris_Guideline.



Lösungsidee:

ES FEHLEN DAS DREHEN DER TETRIMINOS UND DAS SCHNELLERE FALLEN DER TETRIMINOS IM FORTGANG EINES SPIELS.

Steuerung:

- Arrowdown: Schneller fallen
- Arrowleft: nach links bewegen
- Arrowright: nach rechts bewegen
- R: Neu Starten

Tetriminos:

Tetriminos bestehen aus einem 2-dimensionalem Array aus Blöcken und einem enum shapes. Die Form(shape) und die Farbe der Tetriminos werden zufällig ausgewählt. Es gibt für jede Form eine eigene methode die den Tetrimino mit dieser Form initialisiert.

Game Board:

Das gameboard ist ein 2-dimensionales Block array mit der Größe [GB_ROWS][GB_COLS]. Wenn erkannt wird dass tetriminos ganz unten angekommen ist wird das dann auf das gameboard gerendert und das game board array wird aktualisiert sodass jetzt erkannt werden kann ob ein tetrimino auf ein tetrimino landet. Ebenfalls ist das gameboard dafür zuständig zu erkennen falls eine Reihe gefüllt ist dass diese gelöscht wird und die tetriminos über der reihe eine Reihe nach unten rücken. Das gameboard merkt sich auch wie viele Reihen man gelöscht hat.

Game Engine:

Ist dafür zuständig die Logik auszuführen nach jedem „move“ vom User.

Quellcode:

CURRENT BLOCK

```
#pragma once

#include "types.h"

extern bool cb_try_move_block(block bl, int dx, int dy);
extern void cb_move_block(block* bl, int dx, int dy);
extern block cb_new_block_color(color color);


#include <stdlib.h>
#include "current_block.h"
#include "game_board.h"

bool cb_try_move_block(block bl, int dx, int dy) {
	position pos = bl.pos;
	pos.x += dx;
```

```
	pos.y += dy;
	if (!gb_is_valid_position(pos))
		return false;

	return true;
}

void cb_move_block(block* bl, int dx, int dy) {
	bl->pos.x += dx;
	bl->pos.y += dy;
}

void cb_render(block block) {
	render_block(block);
}

block cb_new_block_color(color color) {
	block b;
	b.pos.x = GB_COLS / 2;
	b.pos.y = GB_ROWS - 1;
	b.color = color;

	return b;
}
```

GAMEBOARD

```
#pragma once

#include <stdbool.h>
#include "types.h"

#define MAX_BLOCKS_COUNT ((GB_ROWS)*(GB_COLS))

static block blocks[GB_ROWS][GB_COLS];

extern bool gb_is_valid_position(const position pos);

extern void gb_add_block(const block _block);
extern void gb_render(void);
extern void gb_init_game_board(void);
extern int gb_get_score(void);


#include <stddef.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <time.h>
#include "game_board.h"

static int score = 0;

bool gb_is_valid_position(const position pos) {
	if (!(pos.x >= 0 && pos.x < GB_COLS && pos.y >= 0 && pos.y < GB_ROWS))
		return false;

	return (blocks[pos.y][pos.x].color == color_black);
```

```
}

bool gb_is_row_full(const int row) {
	for (int i = 0; i < GB_COLS; i++) {
		if (blocks[row][i].color == color_black)
			return false;
	}

	return true;
}

static void gb_remove_row(const int row) {
	for (int i = row + 1; i < GB_ROWS; i++) {
		for (int j = 0; j < GB_COLS; j++) {
			blocks[i - 1][j] = blocks[i][j];
			blocks[i - 1][j].pos.y--;
		}
	}
	score++;
}

void gb_add_block(const block _block) {
	assert(gb_is_valid_position(_block.pos));
	blocks[_block.pos.y][_block.pos.x] = _block;

	if (gb_is_row_full(_block.pos.y))
		gb_remove_row(_block.pos.y);
}

void gb_init_game_board(void) {
	srand(time(0));
	score = 0;
	for (int i = 0; i < GB_ROWS; i++) {
		for (int j = 0; j < GB_COLS; j++) {
			blocks[i][j].pos.x = j;
			blocks[i][j].pos.y = i;
			blocks[i][j].color = color_black;
		}
	}
}

void gb_render(void) {
	for (size_t i = 0; i < GB_ROWS; i++) {
		for (size_t j = 0; j < GB_COLS; j++) {
			render_block(blocks[i][j]);
		}
	}
}

int gb_get_score(void) {
	return score;
}
```

GAME ENGINE

```h
#pragma once
#include <stdbool.h>

extern bool ge_handle_move(int dx, int dy, int turn);
extern bool ge_is_game_over(void);
extern void ge_restart(void);
```

```c
#include "game_engine.h"
#include "current_block.h"
#include "game_board.h"
#include "Tetriminos.h"

static bool end = false; //end of round

bool ge_handle_move(int dx, int dy, int turn) {
	if (end)
		return false;

	if (!tt_try_move(dx, dy) && dy == -1) {
		Tetrimino current = tt_get_current();
		for (int r = 0; r < MAX_SIZE; r++) {
			for (int c = 0; c < MAX_SIZE; c++) {
				if (is_block(current.blocks[r][c]))
					gb_add_block(current.blocks[r][c]);
			}
		}

		tt_new_tetrimino();

		if (!tt_try_move(0, 0))
			end = true;
	}

	return !end;
}

bool ge_is_game_over(void) {
	return end;
}

void ge_restart(void) {
	end = false;
}
```

TETRIMINOS

```h
#pragma once
#include "types.h"
#include "current_block.h"
#include <stddef.h>
#include <assert.h>

extern bool tt_try_move(int dx, int dy);
```

```
extern Tetrimino tt_new_tetrimino(void);
extern Tetrimino tt_get_current(void);
extern void tt_render(void);
extern bool is_block(block bl);


#include "Tetriminos.h"

static Tetrimino current;

static shapes random_shape(void);
static void init_shape(void);

static color random_color() {
	static color colors[] = { color_red, color_blue,
color_green,color_yellow,color_magenta,color_cyan };
	int n_color = sizeof(colors) / sizeof(colors[0]);
	return colors[rand() % n_color];
}

bool tt_try_move(int dx, int dy) {
	for (int r = 0; r < MAX_SIZE; r++) {
		for (int c = 0; c < MAX_SIZE; c++) {
			if (is_block(current.blocks[r][c])) {
				if (!cb_try_move_block(current.blocks[r][c], dx, dy))
					return false;
			}
		}
	}

	for (int r = 0; r < MAX_SIZE; r++) {
		for (int c = 0; c < MAX_SIZE; c++) {
			if (is_block(current.blocks[r][c]))
				cb_move_block(&(current.blocks[r][c]), dx, dy);
		}
	}
	return true;
}




void tt_render(void) {
	render_Tetrimino(current);
}

Tetrimino tt_new_tetrimino(void) {
	current.shape = random_shape();
	init_shape();
}

void init_i(void) {
	color randcol = random_color();
	current.blocks[0][1] = cb_new_block_color(randcol);
	current.blocks[1][1] = cb_new_block_color(randcol);
	current.blocks[1][1].pos.y -= 1;
	current.blocks[2][1] = cb_new_block_color(randcol);
	current.blocks[2][1].pos.y -= 2;
```

```
	current.blocks[3][1] = cb_new_block_color(randcol);
	current.blocks[3][1].pos.y -= 3;
}

void init_o(void) {
	color randcol = random_color();
	current.blocks[2][1] = cb_new_block_color(randcol);
	current.blocks[2][2] = cb_new_block_color(randcol);
	current.blocks[2][2].pos.x += 1;
	current.blocks[3][1] = cb_new_block_color(randcol);
	current.blocks[3][1].pos.y -= 1;
	current.blocks[3][2] = cb_new_block_color(randcol);
	current.blocks[3][2].pos.x += 1;
	current.blocks[3][2].pos.y -= 1;
}

void init_t(void) {
	color randcol = random_color();
	current.blocks[0][1] = cb_new_block_color(randcol);
	current.blocks[1][0] = cb_new_block_color(randcol);
	current.blocks[1][0].pos.x -= 1;
	current.blocks[1][0].pos.y -= 1;
	current.blocks[1][1] = cb_new_block_color(randcol);
	current.blocks[1][1].pos.y -= 1;
	current.blocks[1][2] = cb_new_block_color(randcol);
	current.blocks[1][2].pos.x += 1;
	current.blocks[1][2].pos.y -= 1;
}

void init_j(void) {
	color randcol = random_color();
	current.blocks[0][1] = cb_new_block_color(randcol);
	current.blocks[1][1] = cb_new_block_color(randcol);
	current.blocks[1][1].pos.y -= 1;
	current.blocks[2][1] = cb_new_block_color(randcol);
	current.blocks[2][1].pos.y -= 2;
	current.blocks[2][0] = cb_new_block_color(randcol);
	current.blocks[2][0].pos.x -= 1;
	current.blocks[2][0].pos.y -= 2;
}

void init_l(void) {
	color randcol = random_color();
	current.blocks[0][1] = cb_new_block_color(randcol);
	current.blocks[1][1] = cb_new_block_color(randcol);
	current.blocks[1][1].pos.y -= 1;
	current.blocks[2][1] = cb_new_block_color(randcol);
	current.blocks[2][1].pos.y -= 2;
	current.blocks[2][2] = cb_new_block_color(randcol);
	current.blocks[2][2].pos.x += 1;
	current.blocks[2][2].pos.y -= 2;
}

void init_s(void) {
	color randcol = random_color();
	current.blocks[0][1] = cb_new_block_color(randcol);
	current.blocks[0][2] = cb_new_block_color(randcol);
	current.blocks[0][2].pos.x += 1;
```

```
	current.blocks[1][1] = cb_new_block_color(randcol);
	current.blocks[1][1].pos.y -= 1;
	current.blocks[1][0] = cb_new_block_color(randcol);
	current.blocks[1][0].pos.x -= 1;
	current.blocks[1][0].pos.y -= 1;
}

void init_z(void) {
	color randcol = random_color();
	current.blocks[0][0] = cb_new_block_color(randcol);
	current.blocks[0][0].pos.x -= 1;
	current.blocks[0][1] = cb_new_block_color(randcol);
	current.blocks[1][1] = cb_new_block_color(randcol);
	current.blocks[1][1].pos.y -= 1;
	current.blocks[1][2] = cb_new_block_color(randcol);
	current.blocks[1][2].pos.x += 1;
	current.blocks[1][2].pos.y -= 1;

}

void reset_blocks(void) {
	for (int r = 0; r < MAX_SIZE; r++) {
		for (int c = 0; c < MAX_SIZE; c++) {
			current.blocks[r][c].pos.x = -1;
			current.blocks[r][c].pos.y = -1;
		}
	}
}

void init_shape(void) {

	reset_blocks();

	switch (current.shape) {
	case I:
		init_i();
		break;
	case O:
		init_o();
		break;
	case T:
		init_t();
		break;
	case J:
		init_j();
		break;
	case L:
		init_l();
		break;
	case S:
		init_s();
		break;
	case Z:
		init_z();
		break;
	}
}
```

```
bool is_block(block bl) {
	return (bl.pos.x >= 0) || (bl.pos.y >= 0);
}

Tetrimino tt_get_current(void) {
	return current;
}

static shapes random_shape(void) {
	static shapes shapes[] = {I, O, T, J, L, S, Z};
	int n_shapes = sizeof(shapes) / sizeof(shapes[0]);
	return shapes[rand() % n_shapes];
}
```

TIMER

```
#pragma once

typedef void (*timer_func)(void);

extern void timer_start(double itvl, timer_func on_tick);
extern void timer_fire();
extern void timer_stop();
```

```
#include <assert.h>
#include <stdbool.h>
#include <GLFW/glfw3.h>
#include "timer.h"

static timer_func callback;
static double interval = 1;
static void reset_time();
bool is_timer_running = false;

void timer_start(double itvl, timer_func on_tick)
{
	assert(on_tick);
	interval = itvl;
	callback = on_tick;
	is_timer_running = true;
	reset_time();
}

void timer_fire()
{
	if (!is_timer_running)
		return;
	if (glfwGetTime() >= interval) {
		callback();
		reset_time();
	}
}

void timer_stop()
{

}
```

```
void reset_time()
{
	glfwSetTime(0);
}
```

TYPES

```
#pragma once

#include <assert.h>
#include <stdbool.h>

#define GB_ROWS 22
#define GB_COLS 11

#define MAX_SIZE 4

#define UNUSED(var) ((void)var)

typedef enum {
	color_black = 0x000000U,
	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_orange = color_red | color_green / 2
} color;

typedef enum {
	I,
	O,
	T,
	L,
	J,
	S,
	Z
} shapes;

typedef struct {
	int x, y;
} position;

typedef struct {
	position pos;
	color color;
} block;

typedef struct {
	block blocks[MAX_SIZE][MAX_SIZE];
	shapes shape;
} Tetrimino;

extern void render_quad(const position pos, const color color);
```

```
extern void render_block(const block block);
extern void render_Tetrimino(const Tetrimino tetr);


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

void render_quad(const position pos, const color 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();
}

void render_block(const block block) {
	render_quad(block.pos, block.color);
}

void render_Tetrimino(const Tetrimino ttr) {
	for (int r = MAX_SIZE - 1; r >= 0; r--) {
		for (int c = MAX_SIZE - 1; c >= 0; c--) {
			render_block(ttr.blocks[r][c]);
		}
	}
}
```

MAIN

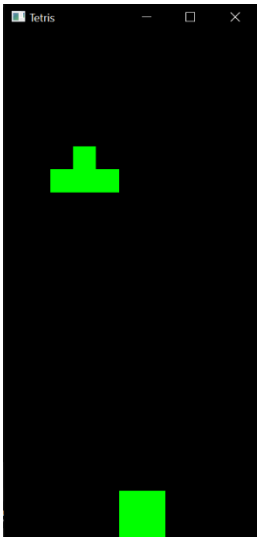
```
#include <stdio.h>
#include <stdlib.h>
#define GLFW_INCLUDE_GLU
#include <GLFW/glfw3.h>

#include "types.h"
#include "current_block.h"
#include "game_board.h"
#include "game_engine.h"
#include "timer.h"
#include "Tetriminos.h"

#define WIDTH  400
#define HEIGHT ((WIDTH * GB_ROWS) / GB_COLS)
#define TIMER_INTERVAL 0.5

static void window_initialized(GLFWwindow* window);
static void render_window(void);
static void on_key(GLFWwindow* window, int key, int scancode, int action, int mods);
static void on_tick(void);


static void window_initialized(GLFWwindow* window) {
```

```
	glfwSetKeyCallback(window, on_key);
	gb_init_game_board();
	tt_new_tetrimino();
	timer_start(TIMER_INTERVAL, on_tick);
}

static void render_window(void) {
	timer_fire();
	gb_render();
	tt_render();
}

static void on_tick(void) {
	on_key(NULL, GLFW_KEY_DOWN, 0, GLFW_PRESS, 0);
}

void on_key(GLFWwindow* window, int key, int scancode, int action, int mods) {
	int dx = 0;
	int dy = 0;
	int turn = 0;
	switch (key) {
	case GLFW_KEY_DOWN:
		dy = -1;
		break;
	case GLFW_KEY_LEFT:
		dx = -1;
		break;
	case GLFW_KEY_RIGHT:
		dx = 1;
		break;
	case GLFW_KEY_UP:
		turn = 1;
		break;
	}

	if (action == GLFW_PRESS || action == GLFW_REPEAT) {
		if (key == GLFW_KEY_R) {
			timer_stop();
			int score = gb_get_score();
			printf("You cleared %d rows!\n", score);
			printf("Restarting game\n");
			ge_restart();
			window_initialized(window);
		}
		else {
			if (!ge_handle_move(dx, dy, turn)) {
				timer_stop();
				int score = gb_get_score();
				printf("You cleared %d rows!", score);

			}
			else {
					timer_stop();
			}
		}
	}
}
```

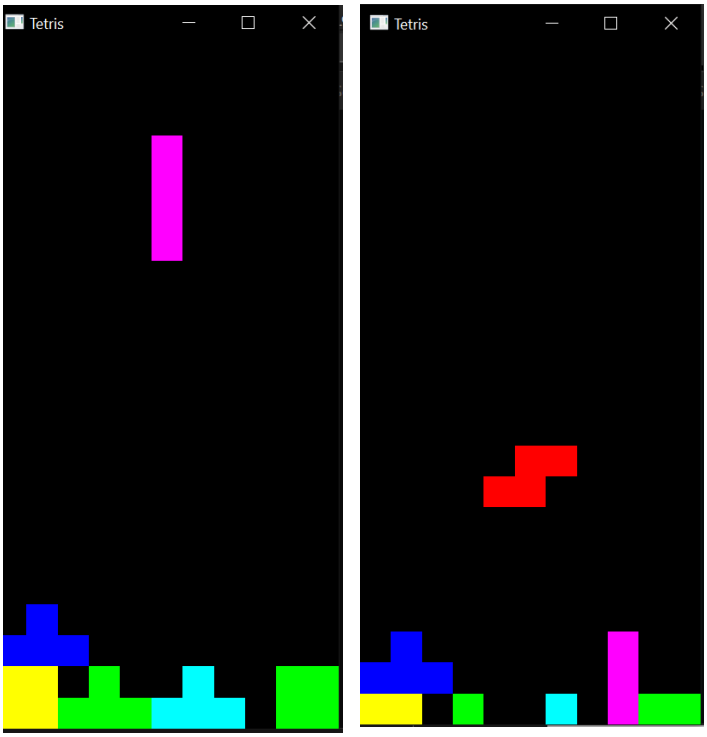
```
int main() {
	if (!glfwInit()) {
		fprintf(stderr, "could not initialize GLFW\n");
		return EXIT_FAILURE;
	}

	GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "Tetris", NULL, NULL);
	if (!window) {
		glfwTerminate();
		fprintf(stderr, "could not open window\n");
		return EXIT_FAILURE;
	}

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

	window_initialized(window);

	while (!glfwWindowShouldClose(window)) {
		glfwGetFramebufferSize(window, &width, &height);
		glViewport(0, 0, width, height);
		glClear(GL_COLOR_BUFFER_BIT); //clear frame buffer
		glMatrixMode(GL_PROJECTION);
		glLoadIdentity();
		gluOrtho2D(0, width, 0, height); //orthogonal projection - origin is in
lower-left corner
		glScalef((float)width / (float)GB_COLS, (float)height / (float)GB_ROWS,
1); //scale logical pixel to screen pixels

		render_window();

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

		glfwSwapBuffers(window);//push image to display
		// glfwPollEvents();
		glfwWaitEventsTimeout(TIMER_INTERVAL / 5); //process all events of the
application
	}

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

Testfälle:

Block platzieren:



Zeile löschen:



Score anzeigen:

Restart: