

Multimedia-Projektarbeit

Yannik Höll & Christoph Paul Pooch & Marie Luise Clemenz & Viktoryia Talaknjanik

11. Oktober 2021

Inhaltsverzeichnis

1	Einleitung	4
2	Theoretische Grundlagen	4
2.1	Tetris	4
2.2	Räumliche Transformationen	5
2.3	HSV - Farbraum	5
3	Codedokumentation	5
3.1	Implementierung	5
3.1.1	Gameengine	5
3.1.2	Rendern	9
3.1.3	Update	9
3.2	Fehlerbehandlung	9
3.3	Userinterface	9
3.4	Nicht implementierte Features	9
4	Projektdokumentation	9
4.1	Tools & Entwicklungsumgebung	9
4.2	Arbeitsaufteilung	9
5	Benutzerhandbuch	9

Abbildungsverzeichnis

1	Tetrominos	4
2	Spielfeld (Arena)	4
3	Darstellung HSV-Farbraum als Kegel	5
4	Zustandsautomat	6
5	Speicherlayout Tetromino	7
6	Position in der Arena	7
7	Kollisionsfälle	8

1 Einleitung

Eins der wohl bekanntesten Computerspiele ist Tetris. Denkbar einfach, allerdings stecken hinter dem recht trivialen Spiel doch im Nachhinein gesehen sehr viele aufwendige Funktionen.

In der folgenden Dokumentation wird erläutert, wie die Implementierung des bekannten Spiels in OpenGL umgesetzt wurde und welche Herausforderungen gelöst werden mussten.

2 Theoretische Grundlagen

2.1 Tetris

Ziel des Spiels ist es, ein Spielfeld (Arena) mit Tetrominos gefüllt werden muss. Der Spieler sollte das möglichst kompakt machen, damit volle Zeilen ohne Lücken entstehen. Die Implementierung dieses Projektes hält sich an die NES-Version von Tetris. Deswegen ist das Spielfeld 20 Blöcke hoch und 10 breit (siehe Abbildung 2).

Tetrominos gibt es in Tetris genau 7. Alle sind aus 4 zusammenhängenden Blöcken aufgebaut. Man bezeichnet sie mit den Buchstaben, denen ihrer Form ähneln.[Tetb]

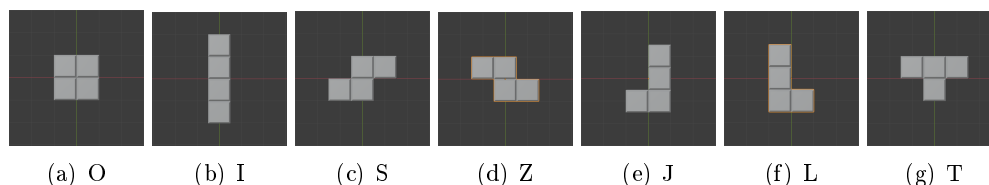


Abbildung 1: Tetrominos

Jedes mal wenn mindestens eine voll ist, wird diese entfernt und man erhält Punkte. Je mehr Zeilen man gleichzeitig füllt, umso mehr Punkte erhält. Jedoch kann man maximal 4 auf einmal voll sein, weil das längste Tetromino nur die Länge 4 hat (siehe Abbildung 1). Jedes mal, wenn 10 Zeilen entfernt wurden, erhöht sich das Level um 1.[Teta]

Der Spieler kontrolliert ein herunterfallendes Tetromino, welches er allerdings nur nach links oder rechts bewegen kann. Nach unten fällt es automatisch. Man kann den Fall beschleunigen aber nicht anhalten. Der Fall ist diskret, d.h. das aktuelle Tetromino immer nur nach einer bestimmten Zeit einen Block nach unten fällt. Diese Zeit hängt vom Level ab und wird immer kürzer, umso höher es ist. Eine Begrenzung des Levels und somit der Fallgeschwindigkeit gibt es dabei nicht, aber irgendwann ist es für Menschen nicht mehr machbar. Die Objekte können auch rotiert werden, allerdings nur um 90 Grad Schritten. Somit haben die manche Tetrominos 4 Positionen, das O nur eine und I, S, Z nur 2.

Die Tetrominos dürfen während sie bewegt werden, nicht mit anderen Objekten in dem Spielfeld kollidieren und sich auch nicht aus dem Grenzen bewegen. Wenn es eine Kollision, nachdem das spielerkontrollierte Objekt gefallen ist, gibt, dann wird es in das Spielfeld platziert und fixiert. Es wird ein neues Tetromino zufällig generiert, welches am oberen Rand des Spielfelds startet.

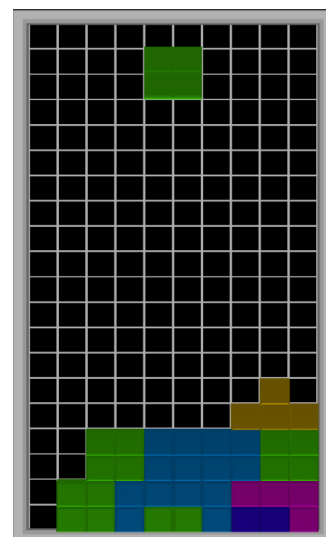


Abbildung 2: Spielfeld (Arena)

Das Spiel endet, wenn ein Turm entstanden ist, der bis oben das Spielfeld füllt, sodass ein neues Tetromino sofort mit der Arena kollidiert. Nachdem man das Spiel verloren hat, muss man von vorn beginnen, das Level und die Punkte werden zurück auf 0 gesetzt.

2.2 Räumliche Transformationen

2.3 HSV - Farbraum

Farbräume sind Modelle, mit denen man Farben durch Attribute definieren. Beispielsweise kann man eine Farbe durch ihren Grün-, Rot- und Blauanteil beschreiben oder durch die Sättigung, Helligkeit und Farbigkeit. Man kann solche Räume auch darstellen, indem man die Attribute als Koordinaten auffasst und dann z.B. in einem 3 dimensional Koordinatensystem abbildet. [CS1]

In diesem Projekt kommt der HSV Farbraum zum Einsatz, welcher Helligkeit(V), Sättigung(S) und Farbigkeit(H). Die ersten beiden werden linear interpretiert und die Farbigkeit ist ein Winkel auf dem, der per Definition auf eine Farbe abgebildet wird. Die untere Abbildung verdeutlicht diesen Zusammenhang noch einmal. [CS2]

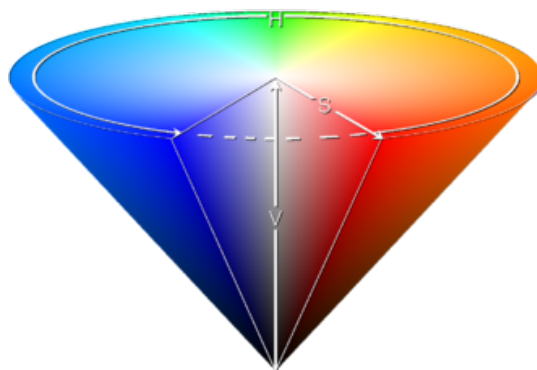


Abbildung 3: Darstellung HSV-Farbraum als Kegel

3 Codedokumentation

3.1 Implementierung

3.1.1 Gameengine

Das Herz des Programms ist die Gameengine, welche die Logik der Objekte im Spiel kontrolliert. Sie überprüft, ob Kollisionen stattfinden und berechnet z.B. die Punkte. Außerdem sorgt unsere Implementierung für das Speichermanagement der Spielobjekte. Ihr gesamter Code befindet sich in `engine.c`.

Alle wichtigen Informationen werden zentral im `struct GameData` gespeichert.

Als erstes wird der Gamestate gespeichert, welcher den aktuellen Zustand des Spiels repräsentiert. Er ist als `enum GameState` definiert, welcher 3 Einträge hat. `PLAYING` beschreibt den Hauptzustand, in dem der Spieler das fallende Tetromino bewegen kann und die Hintergrundmusik läuft. Der Gamestate `PAUSE` kann durch den Spieler durch Drücken der P Taste erreicht werden. Darin werden alle Eingaben, die das aktuelle Tetromino bewegen ignoriert und die Hintergrundmusik ist stumm geschaltet. Außerdem

fällt kein Objekt. Man kann durch erneutes Drücken von P das Spiel wieder in den Zustand **PLAYING** versetzen. Das Spiel geht dort weiter, wo es pausiert wurde. **GAME_OVER** wird dann erreicht, wenn das Spiel verloren wurde (siehe 2.1). Nun wird nur noch eine Nachricht angezeigt, dass der Spieler verloren und wie viele Punkte er erzielen konnte. Man kann das Spiel nun mit ESC beenden oder mit R neustarten. Bei einem Neustart wird wieder in den **PLAYING**-Zustand übergangen. Alle Zustandsinformationen werden resettet.

```
1 enum GameState {
2     PAUSE,
3     PLAYING,
4     GAME_OVER
5 };
6
7 enum GameState gameState;
```

Abbildung 4 zeigt eine Darstellung des Zustandsautomaten.

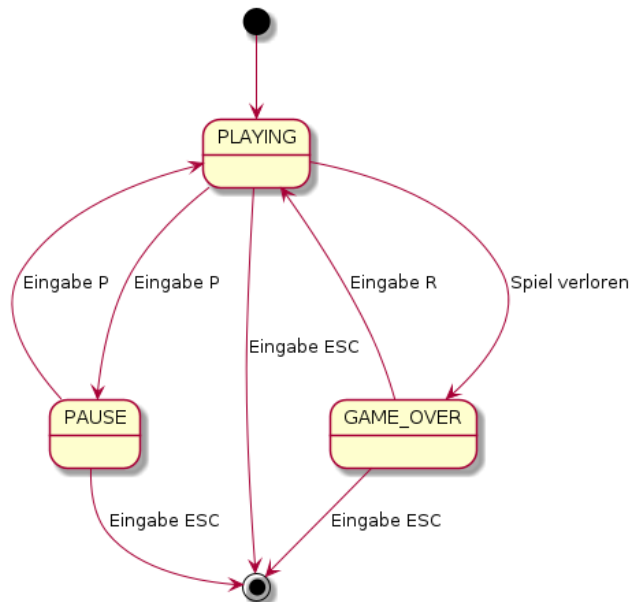


Abbildung 4: Zustandsautomat

Die Tetrominos selbst sind als Matrizen definiert, die mithilfe von eindimensionalen Arrays dargestellt werden. Dabei stellt die 0 kein Block dar und 1-7 einen ausgefüllten Block. Diese Zahl ist bei jedem Block unterschiedlich, weil durch sie später die Farbe des Tetromino bestimmt wird. Sie wird dann als Winkel für die Farbe im HSV-Farbraum verwendet.

Um dem Block an der Stelle (x, y) zu erhalten, muss man diesen 2-dimensionalen Index eindeutig in einen 1-dimensionalen Arrayindex umwandeln. Dafür wurde folgende Formel verwendet:

$$i_{array} = x + y \cdot w$$

Wobei w die Breite der Matrix ist. (Die selbe Formel kann auch später für die Arena verwendet werden.) Dafür muss man allerdings die Breite auch kennen. Deswegen wird am Anfang jeder Tetrominomatrix die Art gespeichert. Diese wird ebenfalls durch **enum** dargestellt.

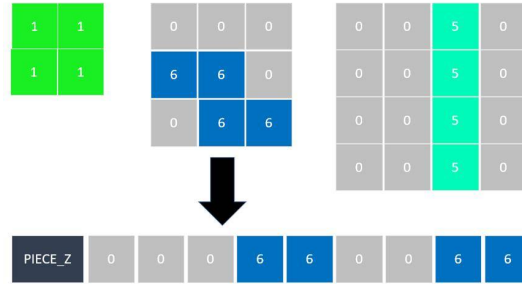


Abbildung 5: Speicherlayout Tetromino

```
1 enum Piece { PIECE_0, PIECE_L, PIECE_J, PIECE_T, PIECE_I, PIECE_Z, PIECE_S };
```

`arena` ist ein Pointer auf ein Array, dass die aktuelle Arena (Spielfeld) speichert. Sie ist 200 (20 · 10) Elemente groß.

```
1 int* arena;
```

Weil das Transformieren zwischen 2D und 1D Indizes werden muss, wurde dafür Helferfunktionen definiert. Außerdem wurde noch eine Macro definiert, die direkt Indices für die Arena ausrechnen kann, ohne ihre Breit explizit anzugeben.

```
1 #define COORDS_TO_ARENA_INDEX(x, y) (coords_to_array_index((x), (y), ARENA_WIDTH))
2
3 size_t coords_to_array_index(size_t x, size_t y, size_t width)
4 { return y * width + x; }
```

Diese beiden Integer speichern die aktuelle Arenaposition des Tetromino, dass durch den Nutzer kontrolliert wird.

```
1 int position_x;
2 int position_y;
```

Das folgenede Listing zeigt wichtige Metadaten über das Spiel. `score` wird ausschließlich für das Nutzerinterface benötigt und speichert die Punkte, die der Spieler erzielt. Das `level` wird dem Spieler auch angezeigt, jedoch auch zur Berechnung der "Geschwindigkeit" verwendet. Zusätzlich wird noch die Anzahl der gefüllten Zeilen in `cleared_lines` gespeichert, was dann zur Berechnung des aktuellen Levels genutzt wird. Und `piece_count` ist ein Pointer auf ein Array, welches speichert, wie oft jedes Tetromino vorgekommen ist. Das wird rein visuell für das Nutzerinterface genutzt.

```
1 uint32_t score;
2 uint32_t level;
3 uint32_t cleared_lines;
4 int* piece_count;
```

Die Flag `bool fast_drop` speichert, ob der Spieler gerade den Knopf gedrückt hält, damit die Fallgeschwindigkeit sehr stark erhöht wird. Und `bool is_defeat` ist eine Flag, die im letzten Update im Zustand `PLAYING` gesetzt wird, um anzuzeigen, dass der Spieler verloren hat und der Gamestate auf `GAME_OVER` gesetzt wird.

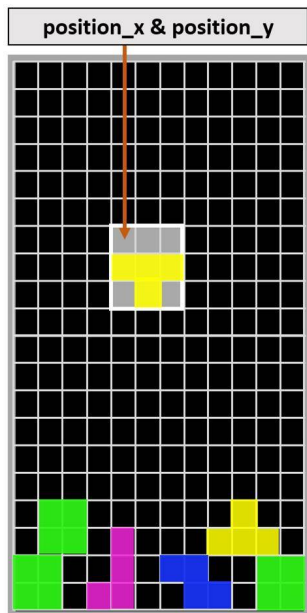


Abbildung 6: Position in der Arena

Es wird auch der `seed`, der zum initialisieren des Zufallsgenerators genutzt wurde, gespeichert. Diese wird allerdings nicht weiter im Program verwendet (siehe 3.4). Das gleiche gilt für `accumulated_time`;

Nun gibt es eine Reihe an Funktionen, die es erlauben, das aktuelle Tetromino bzw. dessen zu manipulieren und überprüfen, dass es nicht in einen unerlaubten Zustand kommt.

Dazu gehören als ersten die Funktionen, die nach Kollisionen mit den Wänden bzw. der Arena überprüfen. Das wurde mit 2 Helferfunktionen implementiert, die jeweils einer der beiden Fälle übernehmen. Die Funktion `check_collision_arena_wall` evaluiert, ob der Spieler das aktuelle Objekt versucht aus der Arena herauszubewegen. Dafür muss aufgrund der Darstellung als Array, die erste Spalte von links bzw. rechts gefunden, nicht nur nullen enthält. Dann wird zu `position_x` der Index dieser Spalte hinzuaddiert und man erhält `non_zero_index_left`. Diese Zahl muss nun größer-gleich null sein, weil ansonsten die Begrenzung der Arena überschritten wurde. Das gleiche gilt für rechts, nur das dort überprüft wird, das `position_x` addiert mit der ersten Spalte mit einem Eintrag, der nicht null ist, kleiner als die Breite der Arena ist. Das untere Listing zeigt den Algorithmus beispielhaft für die Implementierung für die Kollisionabfrage mit der linken Wand. Für rechts ist so analog.

```

1 int size = get_piece_size(game_data->current_piece);
2 int non_zero_index_left = 0;
3 for (int x = 0; x < size; x++) {
4     for (int y = 0; y < size; y++) {
5         if (game_data->current_piece[coords_to_array_index(x, y, size) + 1] != 0) {
6             non_zero_index_left = x;
7             goto LOOP_END1;
8         }
9     }
10 }
11 LOOP_END1;
12
13 return non_zero_index_left < 0;

```

Die Funktion gibt allerdings `true` aus, wenn es eine Kollision gab, deswegen wird auf kleiner als null überprüft.

Desweiteren gibt es `check_collision_arena_pieces`. Darin wird überprüft, ob das `current_piece` mit irgendeinem Objekt in der Arena kollidiert oder aus der Arena herausfällt. Hier wird einfach über alle Einträge `current_piece` iteriert und geschaut, ob sich an der Position, wo in `current_piece` keine Null steht auch in der Arena keine null ist. Dann würde es zu einer Kollision kommen. Außerdem wird getestet, dass alle Einträge in aktuellen Tetromino, die eine `y`-Koordinate haben, die größer als die Höhe der Arena ist, null sind. Ansonsten würde es aus der Arena fallen.

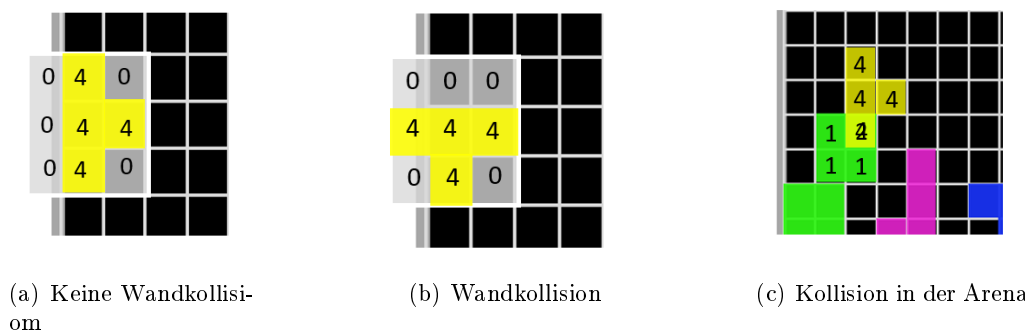


Abbildung 7: Kollisionsfälle

3.1.2 Rendern

3.1.3 Update

3.2 Fehlerbehandlung

3.3 Userinterface

3.4 Nicht implementierte Features

4 Projektdokumentation

4.1 Tools & Entwicklungsumgebung

Github-umgebung Oben GL

4.2 Arbeitsaufteilung

5 Benutzerhandbuch

Literatur

- [CS1] What is a color space? https://www5.in.tum.de/lehre/vorlesungen/graphik/info/csc/COL_4.htm#topic3, zuletzt besucht 11.10.2021.
- [CS2] Hsl and hsv (hue saturation and lighthness / value. https://www5.in.tum.de/lehre/vorlesungen/graphik/info/csc/COL_23.htm#topic22, zuletzt besucht 11.10.2021.
- [Teta] Tetris (nes, nintendo). [https://tetris.wiki/Tetris_\(NES,_Nintendo\)](https://tetris.wiki/Tetris_(NES,_Nintendo)), zuletzt besucht 11.10.2021.
- [Tetb] Tetromino. <https://tetris.wiki/Tetromino>, zuletzt besucht 11.10.2021.