Multimedia-Projektarbeit

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

13. Oktober 2021

Besondere Danksagung an

Inhaltsverzeichnis

Abbildungsverzeichnis

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.

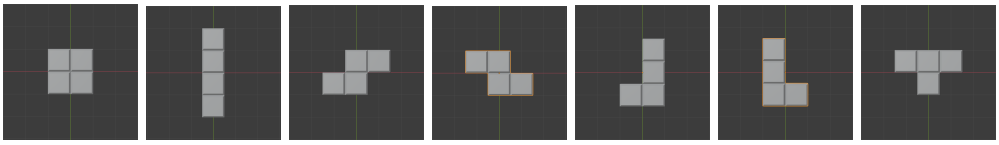
In der folgenden Dokumentation wird erläutert, wie die Implementierung des bekannten Spiels in OpenGL umgesetzt wurde und welche Herausforderungen gelöst werden mussten.

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]



(a) O (b) I (c) S (d) Z (e) J (f) L (g) T

Abbildung 1: Tetrominos

Jedes mal wenn mindestens eine Zeile voll ist, wird diese entfernt und man erhält Punkte. Je mehr Zeilen man gleichzeit 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.

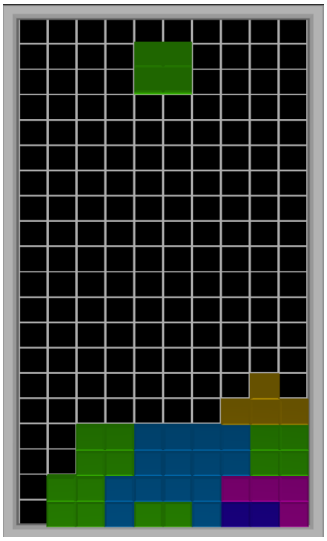


Abbildung 2: Spielfeld (Arena)

Die Tetrominos dürfen während sie bewegt werden, nicht mit anderen Objekten in dem Spielfeld kolliederen und sich auch nicht aus dem Grenzen bewegen. Wenn es eine Kollsion, 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.

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 dimensionalen 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 verdeulicht diesen Zusammenhang noch einmal. [CS2]

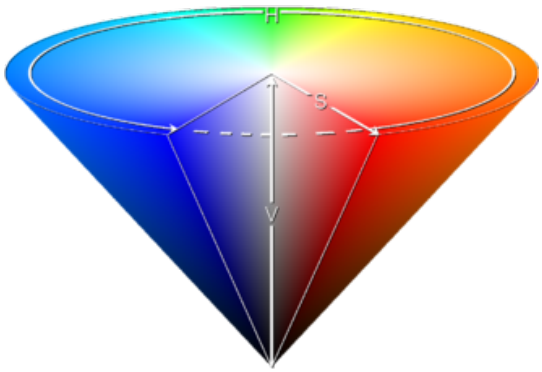


Abbildung 3: Darstellung HSV-Farbraum als Kegel

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 Speichermanagment 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 geschalten. 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 übergagangen. Alle Zustandsinformationen werden resettet.

```
enum GameState {
    PAUSE,
    PLAYING,
    GAME_OVER
};

enum GameState gameState;
```

Abbildung 4 zeigt eine Darstellung des Zustandsautomaten.

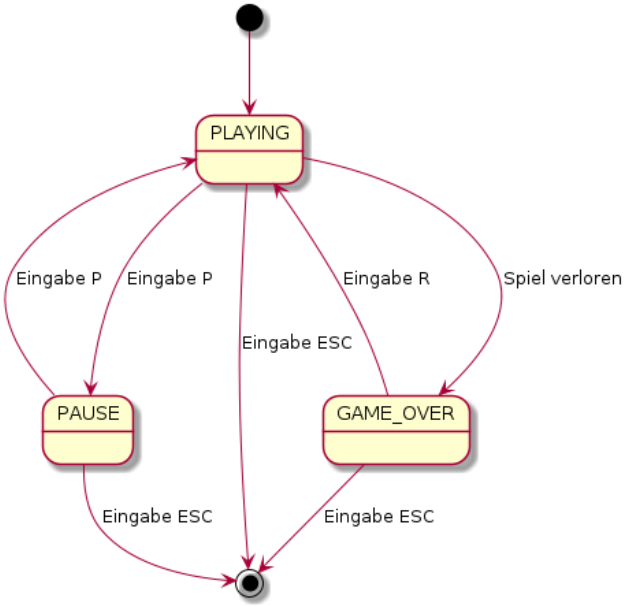


Abbildung 4: Zustandsautomat

Die Tetrominos selbst sind als Matritzen definiert, die mithilfe vom eindimensionale 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.

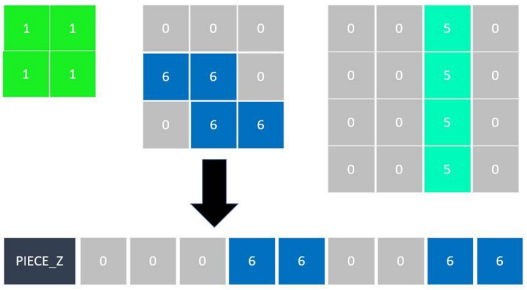


Abbildung 5: Speicherlayout Tetromino

```c
enum Piece { PIECE_O, 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 ($20 \cdot 10$) Elemente groß.

```c
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.

```c
#define COORDS_TO_ARENA_INDEX(x, y) (coords_to_array_index((x), (y), ARENA_WIDTH))

size_t coords_to_array_index(size_t x, size_t y, size_t width)
{ return y * width + x; }
```

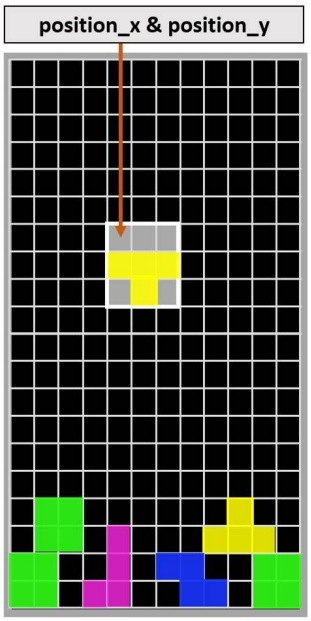


Abbildung 6: Position in der Arena

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

```c
int position_x;
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 "Geschwingkeit"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.

```c
uint32_t score;
uint32_t level;
uint32_t cleared_lines;
int* piece_count;
```

Die Flag `bool fast_drop` speichert, ob der Spieler gerade den Knopf gedrückt hält, damit die Fallgeschwindigkeit seht 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.

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.

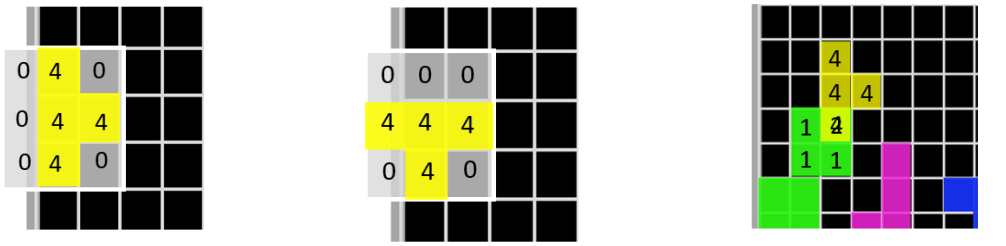
Dazu gehören als ersten die Funktionen, die nach Kollisionen mit den Wänden bzw. der Arena überprüfen. Das wurde mit 2 Helferfunktionen implementiert, die jeweils einer der beiden Fälle übernehmen. Die Funktion `check_collision_arena_wall` evaluiert, ob der Spieler das aktuelle Objekt versucht aus der Arena herauszubewegen. Dafür muss aufgrund der Darstellung als Array, die erste Spalte von links bzw. rechts gefunden, nicht nur nullen enthält. Dann wird zu `postion_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 Enitrag, 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.

```
int size = get_piece_size(game_data->current_piece);
int non_zero_index_left = 0;
for (int x = 0; x < size; x++) {
    for (int y = 0; y < size; y++) {
        if (game_data->current_piece[coords_to_array_index(x, y, size) + 1] != 0) {
            non_zero_index_left = x;
            goto LOOP_END1;
        }
    }
}
LOOP_END1:;

return non_zero_left_index < 0;
```

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

Desweitern 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.



(a) Keine Wandkollision

(b) Wandkollision

(c) Kollision in der Arena

Abbildung 7: Kollisionsfälle

Die oben beschriebenen Helferfunktionen werden nun benötigt, wenn Objekte in der Arena bewegt werden sollen. Dabei gibt es die horizontale Bewegung, die nur durch den Nutzer durchgeführt werden kann und das Fallen, das automatisch durch die Gameengine passiert.

Für die Bewegung durch den Nutzer ist die Funktion `move` zuständig. Die Funktion akzeptiert einen Pointer auf das aktuelle `struct GameData` Objekt und die Richtung, in die das aktuelle Tetromino bewegt werden soll. Dann wird die Bewegung durchgeführt, und falls eine Kollision erkannt wird, wieder rückgängig gemacht. Diese Funktion wird dann durch den Eventhandler aufgerufen.

```
void move(struct GameData* game_data, enum Direction dir)
{
    game_data->position_x += (dir == LEFT) ? -1 : 1;
    if (check_collision_side(game_data)) game_data->position_x -= (dir == LEFT) ? -1
     : 1;
}
```

Die Funktion, die das Fallen implementiert, heißt `drop`. Sie übernimmt nicht nur die Bewegung, sondern muss auch, falls eine Kollision passiert, das aktuelle Tetromino in die Arena schreiben, überprüfen, ob Zeilen gefüllt sind, die Punkte berechnen und dann gegebenenfalls ein neues Tetromino erstellen.

Grundsätzlich wird nur die y-Koordinate um eins erhöht und dann für eine Kollision geprüft. Falls es eine gab, dann wird die y-Koordinate wieder dekrementiert und dann die oben beschriebenen Prozeduren durchgeführt, die alle als eigene Funktionen implementiert sind.

```
size_t drop(struct GameData* game_data)
{
    game_data->position_y++;
    size_t rows = 0;

    if (check_collision_arena_pieces(game_data)) {
        game_data->position_y--;

        write_piece_to_arena(game_data);
        rows = check_filled_rows(game_data);
        game_data->cleared_lines += rows;
        spawn_new_piece(game_data);
        level_up(game_data);
    }

    return rows;
}
```

`write_piece_to_arena`, `spawn_new_piece` und `level_up` sind in ihrer Implementierung trivial und werden deswegen hier nicht genauer beschrieben.

Interessanter ist hier die Funktion `check_filled_rows`, welche überprüft, ob es in der Arena volle Zeilen gibt, diese entfernt und dann die Punktzahl berechnet, welche der Spieler erhält.

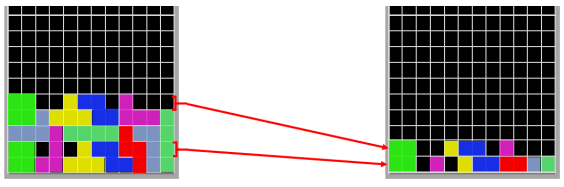


Abbildung 8: Schematische Darstellung: Entfernen von vollen Zeilen

Sie iteriert von unten nach oben durch alle Zeilen der `arena` und speichert diese dann in `row_buffer` zwischen. Danach werden die Punkte, die der Spieler erhält abhängig von der Anzahl der vollen Zeilen und dem aktuellen Level berechnet und direkt die

Punktzahl in `game_data` erhöht.

Danach wird eine neue Arena generiert, in die die Zeilen aus der alten Arena kopiert werden. Dabei werden die Indizes beim Kopieren überprungen, die im `row_buffer`. Das hat den Effekt, dass die vollen entfernt Zeilen werden und allen anderen Zeilen entsprechend nach unten bewegt werden. Am Ende wird der Pointer auf die Arena im `game_data` auf diese neue Arena gesetzt und der alte Speicher wird freigegeben.

```
int* new_arena = (int*)calloc(sizeof(int), ARENA_WIDTH * ARENA_HEIGHT);
size_t current_row_index = ARENA_HEIGHT - 1;
buffer_index = 0;
for (int row = ARENA_HEIGHT - 1; row >= 0; row--) {
    if (row == row_buffer[buffer_index]) {
        buffer_index++;
        continue;
    }

    memcpy(new_arena + ARENA_WIDTH * current_row_index, game_data->arena +
    ARENA_WIDTH * row, sizeof(int) * ARENA_WIDTH);
    current_row_index--;
}

free(game_data->arena);
game_data->arena = new_arena;
```

Und die letzte wichtige Aufgabe der Gameengine ist das Rotieren der Tetrominos. Das wurde durch simple Matrixoperationen implementiert. Um eine Matrix nach rechts zu rotieren, kann man sie transponieren und dann alle Zeilen umkeheren. Und um dann eine Linksrotation durchzuführen kann man auch stattdessen 3 Rechtsrotationen ausführen.

$$\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix} \xrightarrow{\text{Transponieren}} \begin{bmatrix}1&4&7\\2&5&8\\3&6&9\end{bmatrix} \xrightarrow{\text{Zeilen umkehren}} \begin{bmatrix}7&4&1\\8&5&2\\9&6&3\end{bmatrix}$$

$$\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix} \xrightarrow{\text{Rechtsrotation}} \begin{bmatrix}7&4&1\\8&5&2\\9&6&3\end{bmatrix} \xrightarrow{\text{Rechtsrotation}} \begin{bmatrix}9&8&7\\6&5&4\\3&2&1\end{bmatrix} \xrightarrow{\text{Rechtsrotation}} \begin{bmatrix}3&6&9\\2&5&8\\1&4&7\end{bmatrix}$$

Abbildung 9: Beispiel: Matrixrotationen

Um die Rechtsrotation effizienter zu Implementieren, kann man das Transponieren und Zeilen umkehren in einem Schritt erledigen.

Transponieren $t : a_{i,j} \mapsto a_{j,i}$, Zeilen umkehren $r : a_{ij} \mapsto a_{n-i,j}$ und damit $r \circ t : a_{i,j} \mapsto a_{n-j,i}$ (n ist die Breite der Matrix). Die obere Formel für $r \circ t$ wurde dann so im Code verwendet. Die rotierte Matrix wird dann in einen Buffer gespeichert, der dann am Ende mit der Eingabematrix aus.

```
for (size_t j = 0; j < size; j++) {
  for (size_t i = 0; i < size; i++) {
      size_t from_index = coords_to_array_index(i, j, size) + 1;
      size_t to_index   = coords_to_array_index(size - 1 -j, i, size) + 1;
      buffer[to_index] = (*piece)[from_index];
  }
}
```

3.1.2 Rendern

Das Rendern des Spiels auf dem Bildschirm wurde mithilfe von OpenGL implementiert. Das Fenster wird durch GLFW erzeugt.

Grundsätzlich gibt es 2 verschiedene Arten von Objekten, die gerendert werden müssen. Einmal die 3-dimensionalen Models wie z.B. die Begrenzung der Arena oder die Tetrominos. Andererseits die 2-dimensionalen Grafiken und die Texte.

Für das Rendern der Texte und Bilder wurde ein Model erstellt, welches eine Ebene darstellt. Diese wird dann mithilfe von Shadern im Raum platziert und das Bild wird als Textur verwendet. Die Position des Bildes wird dann als Uniform in den Shadern gegeben. Zusätzlich kann man auch noch die Größe des Bildes angeben. Der Fragmentshader rendert dann einfach die Texture auf die Ebene. Beim Shader für die Schrift kann man allerdings nur die Position angeben.

```
uniform vec3 pos;
uniform vec2 scale;
```

Dann wurden Helferfunktionen definiert, welche die Shader binden und die Funktionsaufrufe von OpenGL durchführen. Ihre Signaturen kann man im unteren Listing sehen.

```
void draw_image(const user_data_t* user_data, GLint texunit, GLfloat* pos, GLfloat
  * scale);
void draw_string(const user_data_t* user_data, const char* string, double pos_x,
  double pos_y);
```

Auf die Funktion `draw_string` sollte dabei nochmal etwas genauer eingegangen werden. Sie kann nur Zahlen rendern, weil alle anderen Schriften durch Texturen dargestellt wurden. Das geht bei den Zahlen leider nicht, weil diese im Userinterface veränderlich sind, weil sich z.B. die Punktzahl oder die Anzahl der gefüllten Zeilen während des Spiels erhöhen.

Das Rendern von Texten wurde nun so implementiert, dass für jede Ziffer von null bis neun eine Textur erstellt wurde. In der Funktion `draw_string` wird dann über die Ziffern der zu zeichnenden Zahl iteriert und es werden nebeneinander die Bilder der entsprechenden Ziffern gerendert. Das kommt dadurch Zustande, dass zur x-Koordinate der Startposition ein Zeichenabstand addiert wird. Dabei ist der Zeichenabstand und die Schriftgröße konstant. Man muss nur die richtige Textur vor dem Aufruf von `glDrawArrays` auswählen.

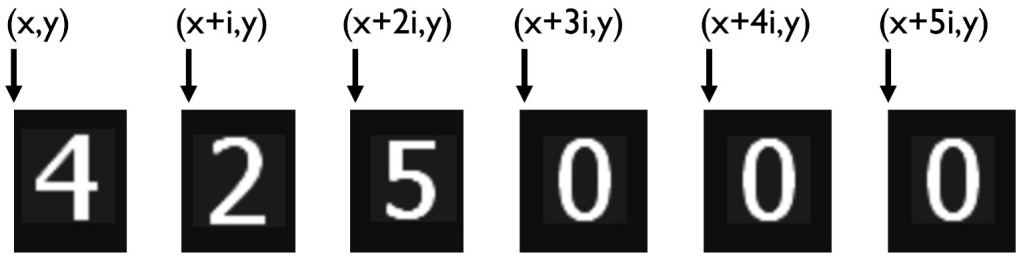


Abbildung 10: Render einer 6 stelligen Zahl (Schema)

```
double offset = 0.12;
for (size_t i = 0; i < strlen(string); i++) {
    GLfloat pos[] = { pos_x + offset * i, pos_y };
    glUseProgram(user_data->shader_program_font);

```

```
        glUniform1i(user_data->digit_tex_uniform, (GLint)(string[i] - '0' + 1));
        glUniform2fv(user_data->digit_pos_uniform, 1, pos);

        glBindVertexArray(user_data->vao[2]);
        glBindBuffer(GL_ARRAY_BUFFER, user_data->vbo[2]);
        glDrawArrays(GL_TRIANGLES, 0, user_data->vertex_data_count[2]);
    }
```

Dann gibt es eine Reihe von Shadern, die die statischen Modelle rendern können. Sie unterscheiden sich hauptächlich dadurch, dass die Position der verschiedenen Objekte in ihnen festgelegt ist. Objekte die auf diese Weise gezeichnet werden sind die Abgrenzung der Arena, die Modelle der Tetrominos im Userinterface. Der Shader, der der die Tetrominos rendert, akzeptiert dabei wieder eine Position, weil sie sich in verschiedenen Positionen im Raum befinden.

$$\begin{bmatrix}
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&4&0\\
0&0&0&0&0&0&0&4&4&4\\
0&0&1&1&2&2&2&2&1&1\\
0&0&1&1&2&2&2&2&1&1\\
0&1&1&2&2&2&2&3&3&3\\
0&1&1&2&1&1&2&6&6&3
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&4&0\\
0&0&0&0&0&0&0&4&4&4\\
0&0&1&1&2&2&2&2&1&1\\
0&0&1&1&2&2&2&2&1&1\\
0&1&1&2&2&2&2&3&3&3\\
0&1&1&2&1&1&2&6&6&3
\end{bmatrix}
\Rightarrow$$

3.1.3 Update

3.1.4 Audio

Für das Abspielen der Hintergrundmusik und den Soundeffekten wurde die Bibliothek SDL2 verwendet. Damit die Musik unabhängig von den Effekten ist, wurden zwei verschiedene `audio_devices` geöffnet, die man sich als Audiokanäle vorstellen kann.

Man kann in diese Kanäle Audiodaten einreihen, weil sie wie eine Warteschlange funktioniert, die die Audiosamples nacheinander abspielt.

Die Musik und Soundeffekte sind in WAV-Datein gespeichert, die durch SDL geladen werden und deren Daten dann im `struct WavData` gespeichert werden.

```
struct WavData {
    SDL_AudioSpec wav_spec;
    Uint32 wav_length;
```

```
    Uint8* wav_buffer;
};
```

Zum Einreihen in die Audiokanäle wurden 2 Helferfunktionen implementiert. `queue_audio` fügt die Samples einfach in die Warteschlange ein, ohne zu überprüfen, ob sie leer ist. Und die Funktion `queue_audio_if_empty` überprüft erst, ob die Warteschlange bereits ist und ruft dann `queue_audio` auf.

Das Abspielen der Hintergrundmusik wurde dann so implementiert, dass einfach nach jedem Update `queue_audio_if_empty` aufgerufen wird, sodass die Musik wiederholt, falls das Lied vorbei ist.

Beim Abspielen von Soundeffekten wird ebenfalls diese Funktion aufgerufen, sodass Soundeffekte verworfen werden, falls sie gleichzeitig abgespielt werden sollen.

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

Abbildung 11: Tastaturlayout

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 ligthness / 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), zuletzt besucht 11.10.2021.

[Tetb] Tetromino. https://tetris.wiki/Tetromino, zuletzt besucht 11.10.2021.