Dokumentation
–
Online Snake

A1exHorst, GermanBrandt, Sabeleis, ultra-ms

Stand: 9. Juli 2025

Inhaltsverzeichnis

1 Checkliste Entwicklungsumgebung

Unser Projekt besteht aus zwei separaten Komponenten - einem Frontend (auf React Framework, Verzeichnisname „OnlineSnake“) und einem Backend (auf Elysia Framework, Verzeichnisname „OnlineSnakeServer“). Beide Komponenten werden aus verschiedenen Git-Repositories bezogen und können unabhängig voneinander betrieben werden. Um die Ausführung zu vereinfachen, wird in diesem Dokument nur die Installation des Docker-Images ausgeführt.

1. **Terminalauswahl**
   - Windows: PowerShell verwenden
   - Linux/macOS: Bash verwenden
2. **Docker Installation**
   (a) Prüfen ob Docker installiert ist:

   ```
   docker --version
   ```

   (b) Falls notwendig, Installation für Windows:
   - Docker Desktop für Windows herunterladen: `https://www.docker.com/products/docker-desktop`
   - Installationsassistent ausführen
   - Nach Installation PowerShell neu starten und prüfen:

     ```
     docker --version
     ```

   (c) Falls notwendig, Installation für Linux (distribution-spezifisch):
   - Für Ubuntu/Debian:

     ```
     sudo apt-get update
     sudo apt-get install docker-ce docker-ce-cli
        ↪ containerd.io
     ```

   - Für andere Distributionen siehe: `https://docs.docker.com/engine/install/`

   (d) Falls notwendig, Installation für macOS:
   - Docker Desktop für Mac herunterladen: `https://www.docker.com/products/docker-desktop`
   - Installationsassistent ausführen
   - Nach Installation Terminal neu starten und prüfen:

     ```
     docker --version
     ```

2 Befehl zum Starten des Programms

Im Verzeichnis des kombinierten Repositories folgenden Befehl ausführen:

```
docker-compose up --build
```

Die Anwendung ist danach erreichbar unter:

- Frontend: `http://localhost:5173`
- Backend: `http://localhost:3000`

3 Code-Struktur

3.1 Projektaufbau

Das Projekt wurde in zwei seperaten Repositories erstellt: Das Frontend liegt im öffentlichen GitHub-Repository „OnlineSnake“ von der Gruppe „OnlikeSnakeOrganization“, das Backend lieft im privaten GitHub-Repository „OnlikeSnakeServer“ von GitHub-Nutzer A1exHorst.
Urspünglich waren beide Repositories der Organisation untergestellt, jedoch musste das Repo. auf ein Gruppenmitglied übertragen werden, um es auf Railway hochzuladen und dort kostenfrei zu hosten.
Bei der Auslieferung des Codes wird auch ein Docker-Image bereitgestellt, welches beide Repositories vereint und gleichzeitig ausführen lässt.

3.1.1 Struktur OnlineSnake

Das Frontend besteht aus folgenden wichtigen Ordnern und Dateien:

- `.git, .github`: GitHub-Daten, GitHub-Pages-Pipelines
- `package.json, package-lock.json`: Verweise auf notwendige Bibliotheken
- `index.html`: Webseite, auf der die React App geschaltet wird.
- `src/assets/`: Ordner für Medien
- `src/components/`: Ordner für Dialogfelder
- `src/context/`: Ordner für Spielekontext-Datei
- `src/css/`: Ordner für Stylesheets
- `src/fonts/`: Ordner für Schriftarten
- `src/game/`: Ordner für Spielelogik, unterteilt in Einzel- und Mehrspieler
- `src/pages/`: Ordner für Haupt-/ und Spieleseite
- `src/schema/`: Ordner für Schema (nötig für Typenvalidierung)
- `src/App.tsx`: React-Router-App
- `src/main.tsx`: Haupt-App von React

3.1.2 Struktur OnlineSnakeServer

- `.git`: GitHub-Daten
- `.env`: Datenbank-Link Speicherdatei (geheim)
- `package.json, package-lock.json`: Verweise auf notwendige Bibliotheken
- `drizzle.config.ts`: drizzle-kit-Koniguration zum Aufsetzen der Datenbank
- `src/config/`: Ordner für CORS-Konfigurationsdatei

- `src/controllers/`: Ordner für Highscore-Controller
- `src/db/`: Ordner für Datenbank-Schema und -Verbindungsmethoden
- `src/game/`: Ordner für Mehrspielerlogik
- `src/models/`: Ordner für Schema (nötig für Typenvalidierung)
- `src/routes/`: Ordner für Routing (Abzweigung von GET- und POST-Anfragen)
- `src/schema/`: Ordner für Schema (nötig für Merhspieler)
- `src/services/`: Ordner für Highscore-Methoden (Verknüpft mit `/routes/`)
- `src/index.ts`: Elysia-App

3.1.3 Struktur Docker Image

Das Docker Image besteht aus den beiden oberen Strukturen, wobei das Repository ÖnlineSnakeünd ÖnlikeSnakeServer"darin als Ordner enthalten sind. Zudem findet sich in diesem Archiv das Setup- und Start-Docker-Skript und diese Dokumentation.

3.1.4 Verwendete Frameworks/Bibliotheken/Services

1. Frameworks
    - React (Frontend)
    - Elysia (Backend)
2. Bibliotheken bzw. Fremder Code
    - zod (Typenvalidierung Frontend & Backend)
    - react-router-dom (Routing Frontend)
    - @elysiajs/cors (CORS-Handling Backend)
    - @elysiajs/websocket (WebSocket-Unterstützung Backend)
    - elysia-rate-limit (Rate-Limiting Backend)
    - @sinclair/typebox (Schema-Definitionen Backend)
    - @neondatabase/serverless (PostgreSQL-Client Backend)
    - drizzle-orm (Datenbank-ORM Backend)
    - dotenv (Umgebungsvariablen Backend)
3. Services
    - GitHub Pages (Hosting Dienst Frontend)
    - Railway (Hosting Dienst Backend)
    - Neon (Hosting Dienst Datenbank)

4 Web-Design

4.1 Gestaltungsziele und Grundprinzipien

Das Design der Webseiten von OnlineSnake orientiert sich an drei zentralen Eigenschaften: Einfachheit, Funktionalität und visuelle Konsistenz. Die Gestaltung orientiert sich an etablierten Prinzipien der Benutzerführung, wobei auf komplizierte Designs wie Untermenüs und Werbung verzichtet wurde. Das Farbschema beschränkt sich auf Schwarz und Grautöne sowie ein Neon-Grün, um einen hohen Kontrast und damit die Leserlichkeit zu gewährleisten. Die Wahl einer pixeligen Schriftart unterstützt das retro-inspirierte Erscheinungsbild des Spiels.

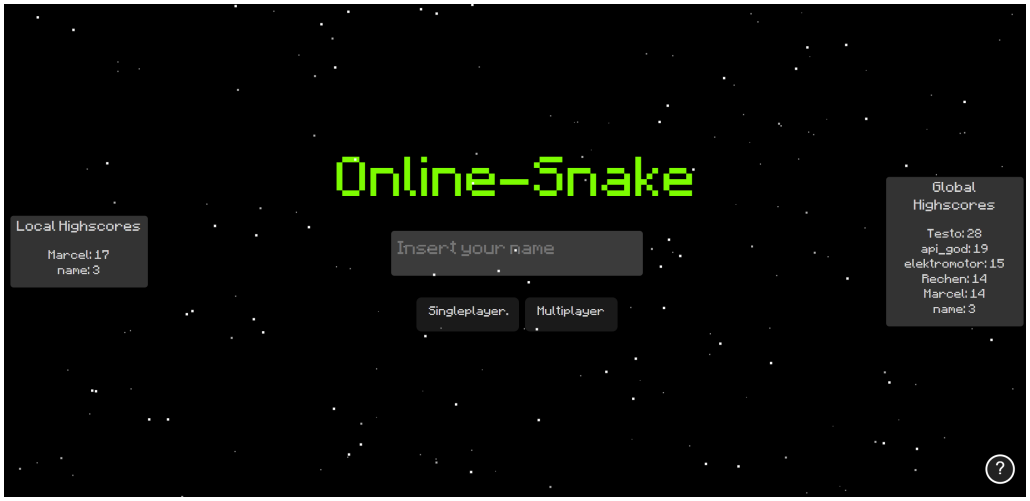
4.2 Analyse: Startseite

Die Startseite ist strukturell in drei Bereiche unterteilt:

Zentrale Interaktionselemente Der Titel, das Eingabefeld für den Spielernamen und die zwei Knöpfe für das Starten eines Einzel- oder Mehrspieler-Spiels sind mittig ausgerichtet und nah beisammen. Dadurch wird vermittelt, dass diese Knöpfe die zentralen Funktionen erfüllen. Der Fragezeichen-/Hilfe-Knopf fällt aus diesem Schema heraus und steht am unteren rechten Rand in den Hintergrund, da es nur als Hilfestellung agiert.

Leaderboards Die lokalen und globalen Bestenlisten sind symmetrisch links und rechts am Rand platziert. Sie verwenden dunkelgraue Container, um sich vom Hintergrund abzuheben. Diese Listen sind nicht interaktiv, weder durch anklicken noch darüberschweben und sollten deshalb von der interaktiven „Insel“ in der Mitte abgeschieden sein.

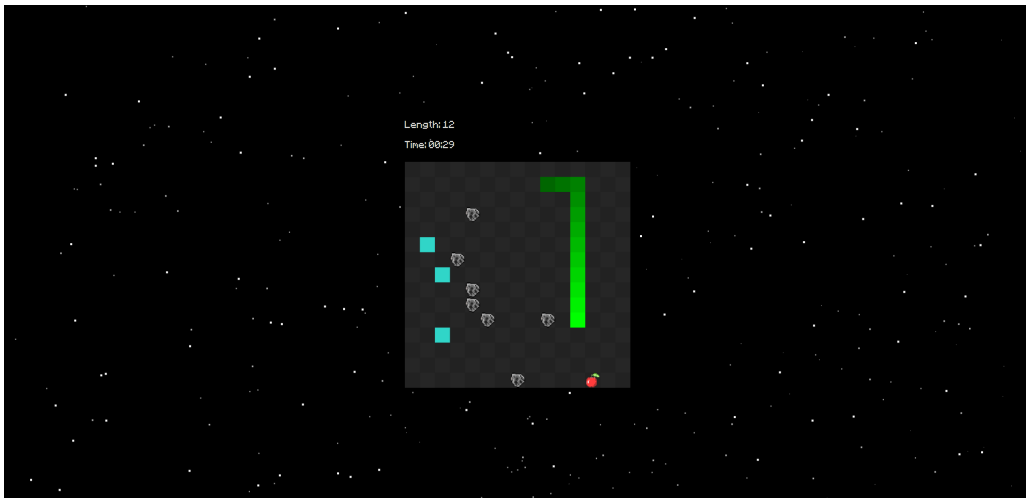
Hintergrund Das Parallax-Scrolling des Stern-Hintergrunds erzeugt räumliche Tiefe, ohne die Bedienelemente zu überlagern. Die Interaktions- und Bestenlisten-Elemente sind durch deren undurchsichtigen Grauton vom Hintergrund abgehoben. Die Knöpfe zum Start des Einzel- und Mehrspielers reagieren auf den Mauszeiger mit einem grünen Aufleuchten, was ihre Funktionalität verdeutlicht.



4.3 Analyse: Spielseite

Wenn das Spiel gestartet wird, öffnet sich die Spielseite, wo das Spielfeld zu sehen ist.

Das Spielfeld besteht aus einem schachbrettartigen Raster in zwei Graustufen, das als neutraler Untergrund dient und sich von Hintergrund abgrenzt. Ein leuchtender Rand hilft zusätzlich, das Spielfeld von Hintergrund abzuheben. Die Schlange ist in einem Hell-Zu-Dunkel Grünverlauf dargestellt, der ihr Wachstum und die Bewegungsrichtung visuell vermittelt. Nahrung wird durch rote Apfel-Symbole gekennzeichnet, Hindernisse durch Asteroiden- und Raumschiff-Icons. Oberhalb des Spielfelds befinden sich zwei Informationsanzeigen: Die aktuelle Schlangenlänge als Ganzzahl und die Spielzeit, angegeben in Minuten und Sekunden. Die Texte auf dieser Seite sind in der gleichen pixeligen Schriftart wie die Startseite gehalten, was die Konsistenz zwischen den Seiten gewährleistet.



4.4 Durchgängige Designmerkmale

Beide Seiten teilen folgende gestalterische Elemente:

- Farbpalette: Beschränkung auf Schwarz, Graustufen und Akzente in Neon-Grün
- Typografie: Durchgängige Verwendung der pixeligen Schriftart
- Interaktionsmuster: Reaktionen auf Benutzereingaben (z.B. Button-Effekte)
- Räumliche Gestaltung: Gleichbleibender Abstand zwischen Elementen und einheitliche Ausrichtung

Der Verzicht auf übermäßige Dekoration unterstreicht den minimalistischen Aspekt des Spiels. Animationen sind auf das Nötigste beschränkt und dienen ausschließlich der besseren Verständlichkeit der Spielmechaniken.

5 Erläuterung der Funktionen

5.1 Einzelspieler und Mehrspieler auf dem Frontend

Wir hatten uns vorgenommen, ein Einzelspieler und ein Mehrspieler zu bauen.

Als Spieler soll man sich zwischen diesen beiden Modi auf der HomePage entscheiden können. Da unser Snake Spiel sehr viel Logik enthält, haben wir uns dazu entschieden, jeweils eine Klasse zu schreiben (`SinglePlayerLogic` und `MultiPlayerLogic`), um den Code zu abstrahieren. Zudem haben beide Klassen fast dieselben Methoden und Attribute, sie unterscheiden sich innerhalb der Methoden aber deutlich.

Wir schauen uns die *Einzelspielerlogik* an.
Wenn man das Repository Pattern im Hinterkopf behält, dient die Klasse „`SinglePlayerLogic`“ „.src/game/logic/SinglePlayerLogic.ts“ für die *Einzelspielerlogik* hier als „Repository“ und gleichzeitig auch als Logik. Sie ist verantwortlich für:

- Das Spiel starten/resetten (das Spawnen von Essen und statischer bzw. beweglicher Hindernissen)
- Den Input des Spielers integrieren.
- Das Spiel laufen lassen, sodass die Schlange (in die gewünschte Richtung des Spielers) und bewegliche Hindernisse sich ständig bewegen

Die `GamePage` zieht die Daten aus dieser Logic, um auf dem Canvas zu malen. Zum Beispiel die Positionen aller Nahrung oder der Hindernisse. Für das Starten des Spiels wird die Methode ‘start‘ aufgerufen, ein Abschnitt vom Code:

```
// OnlineSnake/src/game/logic/SinglePlayerLogic.ts
public start(): void {
    ...

    // Places the first Snake segment on the bottom left
       ↪ corner.
    // Edit this someday so that the snake cannot die
       ↪ instantly by spawning in front of an obstacle.
    this.snakeSegments.push({x: 0, y: 0, color: ""});

    this.painter.ApplyColorsToSnakeSegments();
    this.displaySnakeLength(this.snakeSegments.length);
    this.entityGenerator.generateObstacles();
    this.entityGenerator.generateMovingObstacles();
    this.entityGenerator.generateFood();

    ...
```

Das `snakeSegments` Attribut ist ein Array, das aus allen Segmenten der Schlange besteht. Wir geben `snakeSegments` einen Anfang oder einen "Kopf", in diesem Projektstand erscheint dieser Kopf immer bei Position `(0, 0)`.

Das `entityGenerator` Attribut wird für das Spawnen von Essen und Hindernissen verwendet. Mehr Informationen im Kapitel 5.1.3.

Zurück zur `start()` Methode:

```
// OnlineSnake/src/game/logic/SinglePlayerLogic.ts
    ...
    this.controller?.disable();
    if(this.diagonalMovementAllowed){
        this.controller = new DiagonalController(document,
           ↪ this);
    }else{
        this.controller = new StraightController(document,
           ↪ this);
    }
    this.controller.enable();
    ...
    this.snakeInterval = setInterval(this.snakeLoop, 125);
    this.obstacleInterval = setInterval(this.obstacleLoop,
       ↪ 1000);
}
```

Das Spiel unterstützt im Einzelspieler zwei Arten von Controllern, einen `StraightController` und einen `DiagonalController`. Der `StraightController` unterstützt nur horizontale und vertikale Bewegungen der Schlange, wobei mit dem `DiagonalController` auch diagonale Bewegungen möglich werden.[1]
In dem Konstruktor von `SinglePlayerLogic` fragen wir den Parameter `diagonalMovementAllowed: boolean` ab. Basierend auf diesem Wert, erstellen wir dann einen `Straight-/ oder DiagonalController`. Nach dem Aktivieren des Controllers ist die Initialisierung fertig und wir können zwei Intervalle setzen, die die Methoden `snakeLoop` und `obstacleLoop` aufrufen.
`snakeLoop` mit einer Verzögerung von 125 ms und `obstacleLoop` 5.1.1 mit einer Verzögerung von 1 Sekunde.

```
// OnlineSnake/src/game/logic/SinglePlayerLogic.ts
class SinglePlayerLogic {
    ...
    private snakeLoop = (): void => {
        // Create another Snakesegment
        const head = { ...this.snakeSegments[0] };

        ...

        this.controller.moveHead(head);
        this.snakeSegments.unshift(head);   //Add it to the
           ↪ front of the Snake

        if (this.isGameOver()) {
```

[1] Im Multiplayer ist der diagonale Controller noch nicht fertig umgesetzt worden.

```
            const playerName = localStorage.getItem('
               ↪ playerName') || 'Unknown';

            // Score = snakeSegments.length - 1 (Anzahl
               ↪ gegessener Nahrung)
            const score = Math.max(this.snakeSegments.length -
               ↪  1, 0);
            this.highscores.saveScore(playerName, score);
            this.highscores.uploadScore(playerName, score,
               ↪ this.stopWatch.getTime());
            this.killSnake();
            ...
            if (this.onGameOver) this.onGameOver();
        } else {
            this.painter.pullSnakeColorsToTheHead();    //To
               ↪ keep the colors after each movement.

            //Check if the snake eats food
            let justAteFood: boolean = false;
            for (let i = 0; i < this.food.length; i++) {
                if (head.x === this.food[i].x && head.y ===
                   ↪ this.food[i].y) {
                    justAteFood = true;
                    this.food = this.food.filter(food => !(
                       ↪ food.x === head.x && food.y === head.
                       ↪ y));
                    break;
                }
            }
            if (justAteFood === true) {
                this.entityGenerator.generateFood();
                this.displaySnakeLength(this.snakeSegments.
                   ↪ length);
                this.painter.ApplyColorsToSnakeSegments();
            } else {
                this.snakeSegments.pop(); // Remove the tail
                   ↪ if no food is eaten
            }
        }
    }
    ...
/* (Das ist die gesamte snakeLoop-Methode) */
```

Jedes Aufrufen von snakeLoop führt dazu, dass sich die Schlange um einen Block in die Richtung schleicht, die der Spieler angibt. Um das umzusetzen, erstellen wir eine Kopie des Kopfes, positionieren diesen Kopf dann dort, wo sich die Schlange zubewegt, und entfernen das letzte Segment der Schlange, wenn sie nichts gegessen hat. Diese Vorgehensweise simuliert die Bewegung der Schlange und ist deutlich effizienter, als wenn wir von vorne bis hinten jedes Segment der Schlange nehmen und ihre Position auf

das vorherige Segment setzen. Danach soll überprüft werden, ob sie in etwas gestoßen ist, das tödlich ist, wie zum Beispiel sich selbst oder ein Hindernis. In dem Fall stirbt sie. Wenn das nicht so ist, dann wird überprüft, ob sie etwas Essbares gegessen hat. Wenn sie etwas isst, soll sie um einen Block länger werden.

```
// OnlineSnake/src/game/logic/SinglePlayerLogic.ts
class SinglePlayerLogic {
    ...
    private snakeLoop = (): void => {
        // Create another Snakesegment
        const head = { ...this.snakeSegments[0] };

        ...

        this.controller.moveHead(head);
        this.snakeSegments.unshift(head);   //Add it to the
            ↪ front of the Snake

        ...
```

Die Bewegung der Schlange übernimmt der `Controller`. Dazu erstellen wir eine Kopie des Schlangenkopfes und übergeben sie dem 'Controller' in dem Methodenaufruf `this.controller.move(head);`. Dieser Methodenaufruf repositioniert diese Kopie in die Richtung, die der Spieler angegeben hat und in dem Moment überhaupt erlaubt ist. Keine der beiden Controller erlaubt es dem Spieler, illegale/unlogische Bewegungen auszuführen. Mit illegalen Bewegungen sind Bewegungen gemeint, die keinen Sinn machen. Wenn sich die Schlange nach oben bewegt, darf sie nicht im nächsten Moment plötzlich nach unten schleichen.
(Für genauere Details, siehe Kapitel zum `StraightController`)
Nachdem die Kopie des Kopfes positioniert wurde, fügen wir ihn zum Anfang der Schlange hinzu.

```
// OnlineSnake/src/game/logic/SinglePlayerLogic.ts
class SinglePlayerLogic {
    ...
    private snakeLoop = (): void => {
        ...
        if (this.isGameOver()) {
            const playerName = localStorage.getItem('
                ↪ playerName') || 'Unknown';

            // Score = snakeSegments.length - 1 (Anzahl
                ↪ gegessener Nahrung)
            const score = Math.max(this.snakeSegments.length -
                ↪  1, 0);
            this.highscores.saveScore(playerName, score);
            this.highscores.uploadScore(playerName, score,
                ↪ this.stopWatch.getTime());
            this.killSnake();
```

```
            ...
            if (this.onGameOver) this.onGameOver();
        } else {
            ...
        }
    }
    ...
```

Dann prüfen wir mithilfe der privaten `isGameOver` Methode, ob der Kopf sich in etwas Tödlichem befindet => Allgemein bedeutet das, dass die Schlange sich in etwas Tödliches bewegt hat und demnach stirbt. Beim Tod der Schlange speichern wir einen lokalen Highscore und laden den Highscore auch mithilfe von `highscores.uploadScore(playerName, score, this.stopWatch.getTime())` auf das Backend hoch, mit einem einfachen POST-Request.

```
// OnlineSnake/src/game/logic/SinglePlayerLogic.ts
class SinglePlayerLogic {
    ...
    private snakeLoop = (): void => {
    ...
        if (this.isGameOver()) {
            ...
        } else {
            this.painter.pullSnakeColorsToTheHead();    //To
               ↪ keep the colors after each movement.

            //Check if the snake eats food
            let justAteFood: boolean = false;
            for (let i = 0; i < this.food.length; i++) {
                if (head.x === this.food[i].x && head.y ===
                   ↪ this.food[i].y) {
                    justAteFood = true;
                    this.food = this.food.filter(food => !(
                       ↪ food.x === head.x && food.y === head.
                       ↪ y));
                    break;
                }
            }
            if (justAteFood === true) {
                this.entityGenerator.generateFood();
                this.displaySnakeLength(this.snakeSegments.
                   ↪ length);
                this.painter.ApplyColorsToSnakeSegments();
            } else {
                this.snakeSegments.pop(); // Remove the tail
                   ↪ if no food is eaten
            }
        }
    }
```

```
    ...
```

Hier geht es weiter, falls die Schlange noch am leben ist. Da wir den Kopf geklont haben und ihn vorne an die Schlange gehängt haben und wir einen gleichmäßigen Farbübergang von vorne bis hinten behalten wollen, ziehen wir die Farben der Schlange von hinten bis vorne um eine Position mit `this.painter.pullSnakeColorsToTheHead()`. Dadurch erhalten wir von vorne bis zum zweitletzten Segment einen gleichmäßigen Übergang. Die letzten beiden Segmente haben jetzt dieselbe Farbe, aber das ist uns im Moment egal, weil wir im nächsten Schritt entscheiden, ob wir das letzte Segment jetzt entfernen oder behalten werden. Wenn sich der neue Kopf auf etwas essbaren befindet, entfernen wir diese Nahrung aus der Karte und setzen `justAteFood = true`.
Falls `(justAteFood === true)`, werden wir das gegessene Essen neu auf der Karte generieren, wieder mit `entityGenerator.generateFood()`. Da wir in dem Fall auch das letzte Segment behalten wollen, malen wir die Farben einfach neu mit `painter.ApplyColorsToSnakeSegm` sodass der Farbübergang wieder stimmt. Zudem rufen wir die von der **GamePage** übergebenen Methode `displaySnakeLength` mit der Schlangenlänge auf, sodass die Schlangenlänge als Zahl neben dem **Canvas** angezeigt wird. Wenn `(justAteFood === false)`, entfernen wir einfach das letzte Schlangensegment. Durch das entfernen des letzten Schlangensegmentes wird automatisch für die Erhaltung des Farbübergangs gesorgt.

5.1.1 Statische und Bewegliche Hindernisse

```
// OnlineSnake/src/game/logic/SinglePlayerLogic.ts
...
import MovingObstacle from "./src/game/MovingObstacle.ts";
...
class SinglePlayerLogic {
    ...
    private movingObstacles: MovingObstacle[];
    ...
    //This function moves all movable obstacles
    private movingObstacleLoop = (): void => {
        for(const movingObstacle of this.movingObstacles){
            movingObstacle.moveObstacle();
        }
    }
    ...
```

Jedes Aufrufen von `obstacleLoop` führt dazu, dass sich die **beweglichen Hindernisse** um einen Block bewegen. Sie bewegen sich in die Richtung, in die diese Hindernisse jeweils fliegen. Wenn ein **bewegliches Hindernis** mit etwas kollidierbarem kollidiert (einen Teil einer Schlange oder andere Hindernisse), dann, statt dass es weiter nach vorne fliegt, rotiert sich dieses Hindernis um 90° im Uhrzeigersinn. Diese Rotation wiederholt sich pro Methodenaufruf für jedes bewegliches Hindernis so oft, bis das Hindernis weiterfliegen kann.

5.1.2 Klasse ColorCalculator

```
// OnlineSnake/src/game/SnakeColorCalculator.ts
class SnakeColorCalculator{
    private startColor: number[];
    private endColor: number[];

    constructor(startColor: string, endColor: string){
        this.startColor = [];
        this.startColor.push(parseInt(startColor.substring(0,
            ↪ 2), 16)); // Red
        this.startColor.push(parseInt(startColor.substring(2,
            ↪ 4), 16)); // Green
        this.startColor.push(parseInt(startColor.substring(4,
            ↪ 6), 16)); // Blue

        this.endColor = [];
        this.endColor.push(parseInt(endColor.substring(0, 2),
            ↪ 16)); // Red
        this.endColor.push(parseInt(endColor.substring(2, 4),
            ↪ 16)); // Green
        this.endColor.push(parseInt(endColor.substring(4, 6),
            ↪ 16)); // Blue
    }

    //Returns the color for the given SnakeSegment
    getColor(snakeSegment: number, snakeLength: number):
        ↪ string{
        let segmentColor: string = "#";
        const a: number[] = this.startColor;
        const b: number[] = this.endColor;
        snakeLength = snakeLength - 1;
        if(snakeLength === 0) snakeLength = 1;
        for(let i = 0; i < 3; i++) {
            segmentColor = segmentColor + Math.floor(a[i]-((a[
                ↪ i]-b[i])/snakeLength)*snakeSegment)
                .toString(16)
                .padStart(2, "0");
            //f(x) = a-((a-b)/(l-1))x
            //and l-1 != 0
        }

        return segmentColor;
    }
}
```

Der `SnakeColorCalculator` berechnet die Farben für jedes Schlangensegment, bei angabe des indexes und der gesamt Schlangenlänge, sodass ein Farbübergang über der Schlange entstehen kann.

Der Calculator erwartet bei der Initialisierung zwei Farben: Die Start-/ und die Endfarbe, wobei beide als einen Hex String wie zum Beispiel `00ff00`(pures Grün) und `006600`(pures, aber dunkleres Grün) angegeben werden müssen. Um die Farbe für jedes einzelne Segment auszurechnen, wird **auf jede Grundfarbe** Rot, Grün und Blau (RGB) diese Formel angewendet (Deshalb auch die for schleife):

$$f(x) = a - \left(\frac{a-b}{l-1}\right) x$$

Mit $0 \leq x \leq l-1$ ist $f(x)$ eine lineare Interpolation von a zu b.
Wobei $l-1 \neq 0$, a ist eine Grundfarbe von der Startfarbe und b ist eine Grundfarbe der Endfarbe, x ist der index des zu berechnenden Segments.
Beispiel:
Sei die Grundfarbe Rot der Startfarbe $a = 8$ und $b = 4$.
(das sind minimale Werte aber zum rechnen reicht es aus) Die gesamte Schlangenlänge $l = 7$.
Dann gilt $f(0) = a$
(macht ja Sinn, das Startsegment soll die Startfarbe haben.)
Und $f(l-1) = f(7-1) = f(6) = 8 - \left(\frac{8-4}{7-1}\right) 6 = 8 - \left(\frac{4}{6}\right) 6 = 8 - 4 = 4 = b$
(Macht auch sinn, da das letzte Segment die Endfarbe haben soll.)

5.1.3 Klasse EntityGenerator

Mit dem `EntityGenerator` können wir Essen und Hindernisse auf der Karte verteilt generieren. Durch diese Aufrufe generieren wir:

- `this.entityGenerator.generateObstacles();`
  ... die statischen Hindernisse
- `this.entityGenerator.generateMovingObstacles();`
  ... die bewegenden Hindernisse
- `this.entityGenerator.generateFood();`
  ... die Statischen Hindernisse

Alle drei Methoden achten darauf, dass sie nur so viele Entitäten produzieren, wie es durch diese Attribute erlaubt ist:

```
// OnlineSnake/src/game/logic/SinglePlayerLogic.ts
class SinglePlayerLogic {
    private maxAmountOfFood: number;
    private amountOfStaticObstacles: number;
    private amountOfMovingObstacles: number;
    ...
```

Dabei wird darauf geachtet, dass keine Entitäten auf andere platziert werden (Ausnahme, bewegende Hindernisse auf Essen, da sie sowieso auch über Essen fliegen dürfen). Wenn das Spielfeld zu einem Zeitpunkt bereits voll ist, wird einfach nichts mehr generiert. Die Algorithmen dazu kann man sich in der Klasse `EntityGenerator` in

„OnlineSnake/src/game/EntityGenerator.ts“ ansehen. Der Code ist zu komplex, um sie kompakt in Worte zu fassen, weshalb hier nicht darauf eingegangen wird.

5.1.4 Klasse StraightController

```
// OnlineSnake/src/game/controllers/singleplayer/
   ↪ StraightController.ts
class StraightController implements ControllerInterface {
    ...
    public moveHead(head: SnakeSegment) {
        this.setSnakeDirectionToLatestInputIfAllowed();
        console.log(this.inputsHeld + " | "+ this.logic.
           ↪ snakeDirection)
        if (this.logic.snakeDirection === 'UP') head.y -= 1;
        if (this.logic.snakeDirection === 'DOWN') head.y += 1;
        if (this.logic.snakeDirection === 'LEFT') head.x -= 1;
        if (this.logic.snakeDirection === 'RIGHT') head.x +=
           ↪ 1;

        if(!this.logic.getWallsAreDeadly()){
            //This makes the head appear on the other side of
               ↪ the map if it creeps into a wall.
            const columns: number = this.logic.getColumns();
            const rows: number = this.logic.getRows();
            head.x = (head.x + columns) % columns;
            head.y = (head.y + rows) % rows;
        }
    }
    ...
```

Hier ist die `moveHead(head: SnakeSegment)` Methode, aus dem `StraightController`, die das head auf die gewünschte position setzt. Dabei werden illegale Bewegungen verhindert. Das bedeutet, wenn die Schlange sich im Moment nach oben bewegt, dann soll sie im nächsten Moment nicht plötzlich nach unten gehen können.

5.1.5 Klasse DiagonalController

Die Klasse `DiagonalController` enthällt dieselben Funktionen wie der StraightController, zusätzlich mit diagonalen Bewegungen. Hier ist es sehr wichtig, dass sich die Steuerung aus der Seite des Spielers angenehm anfühlt. Daraus wurde diese Klasse sehr komplex. Viel zu omplex, um in diesem Dokument erklärt zu werden.

5.1.6 Mehrspieler

Beim Mehrspieler ist die Logik auf dem Backend aufgelagert, was bedeutet, dass das Backend auch eine „Mehrspielerlogik“ „.src/game/logic/MultiPlayerLogic.ts“ enthält. Die *Mehrspielerlogik* auf dem Frontend dient nur dafür, die Daten von der Logik des Servers einzulesen und dann die eigenen Daten zu ersetzen. So ähnlich wie beim Repository Pattern ist das Frontend dann das Repository und die Logik sitzt auf dem Backend. Die

GamePage verwendet auch wieder diese gesammelten Daten von diesem „Repository“, um auf dem Canvas den jetzigen Zustand des Spiels zu malen. Die Kommunikation beim Spielen zwischen Backend und Client ist vollständig mithilfe von WebSockets realisiert worden. Das Backend, in dem wir Elysia verwenden, öffnet einen WebSocket Server, die die eingehenden und ausgehenden Verbindungen managed.
Im Moment gibt es auf dem Backend kein richtiges Lobbysystem, weshalb nur eine lobby gestarted wird. Alle Clients, die sich mit dem WebSocketServer verbinden, werden in der Logik dieser einen existierenden Lobby als Spieler gespeichert. Wenn ein Client sich erfolgreich verbunden hat, wird es zuerst alle Daten abfragen. Dazu gehören:

- Alle Spieler, die Schlangensegmente, die Farben..
- Alles Essbare auf der Karte
- Alle Hindernisse

Der Client wird jetzt diese WebSocket Verbindung nutzen, um bei jeder Änderung des Controllerinputs, dem Server diese Änderungen mitzuteilen. (Das ist Ressourcenweise sehr teuer, aber wir wollten so vorgehen, um Cheaten unmöglich zu machen) Der Server drückt die erhaltenen Inputs in den für den Spieler individuellen „fiktiven“ Controller ein. Bei jedem Aufruf von `snakeLoop` und `obstacleLoop`, werden die neuen Daten über die WebSocket Verbindung an den Client gesendet.
Nach dem Aufruf von `snakeLoop` sind die Schlangensegmente und die Positionen vom Essen relevant.
Nach dem Aufruf von `obstacleLoop` interessiert dem Client nur die neue Positionen der bewegenden Hindernissen.

!! Zum Stand der Projektabgabe gibt es im Multiplayer noch einen Bug, der auftritt, wenn das Backend auf einem externen Anbieter wie Railway gehostet wird. Da wir die „ws.raw.remoteAdress“ verwenden, um mehrfache Verbindungen von einem Client zu vermeiden, kann es zum Beispiel bei Railway sein, dass wenn ein Spieler die Verbindung trennt, alle übrigen Spieler auch rausgeworfen werden. Das hat höchstwahrscheinlich damit zu tun, wie Railway mit den IP-Adressen umgeht (ich habe beim debugging erfahren, dass alle remoteAdressen fast gleich sind und Railway diese irgendwie managed, weshalb die Adressen nicht konstant sind und nicht für die Identifikation verwendet werden können.) Um diesen Fehler zu beheben hätte ich die Verwendung UUIDs Implementiert.

5.1.7 Geschrieben von A1exHorst

GitHub Nutzername: A1exHorst
Matrikelnummer: 1126225

5.2 Canvas

5.2.1 Was ist ein Canvas?

In Web Development, meint ein Canvas in der Regel das HTML-Element <Canvas>. Konkret ist der Canvas ein Zeichenbereich, auf dem mit JavaScript 2D/3D Grafiken gezeichnet werden kann.

5.2.2 Warum der Wechsel auf einen Canvas?

Die Entscheidung für den Wechsel war recht eindeutig durch einige Probleme die wir beim Implementieren von Png-Dateien als Designelement für unsere Essenspunkte und unsere Hindernisse hatten. Unteranderem hatten wir die Befürchtung, dass die Rechenleistung Probleme machen könnte bei den größeren Karten des Multiplayers. Diese Sorge war hauptsächlich dadurch begründet das jedes Feld ein Div war. Und somit seperat geladen wird bei jedem Datenupdate. Eine nette Nebenwirkung war, dass es eine elegantere Lösung war die auch näher an den Best Practices lag, als die Div-Lösung.

5.2.3 Implementierung

Um einen Canvas zu nutzen in seinem Projekt muss man erstmal eine Canvas Komponente erstellen. Das sieht dann so aus. ->

```
//Standard rows and columns for singleplayer
let rows = 15;
let columns = 15;
const blockWidth = 30; //setzt breite fest durch Pixel
const blockHeight = 30;//setzt hoehe durch Pixel fest

const GamePage: React.FC = () => {
  const navigate = useNavigate();
  const { inGame, gameMode, endGame, ws, setWsObject } =
    ↪ useGame();
  const audioRef = useRef<HTMLAudioElement | null>(null);

  const canvasRef = useRef<HTMLCanvasElement>(null); // [1]
}
```

[1] Dieser Ausdruck erstellt eine Referenz canvasRef auf ein <canvas>-Element. HTMLCanvasElement ist der Typ des Elements für TypeScript und Null ist der Anfangswert. Das Element exestiert beim ersten Rendern noch nicht. Das wird verwendet um in React mit ref direkt auf ein DOM-Element (DOM = Document Object Model 1) zuzugreifen. Um zum Beispiel den Zeichenkontext (ctx) eines Canvas zu holen. Dies ist sehr wichtig da wir ctx häufiger im Code verwenden um auf dem Canvas zu malen oder Farbwerte zusetzen.

1 Es ist eine strukturierte Darstellung einer HTML- oder XML-Seite, die es Programmen (wie JavaScript) ermöglicht, auf die Inhalte und Struktur der Webseite zuzugreifen und sie zu verändern. (Für genaueres einfach DOM oder Document Object Model suchen)

```
function drawBoard() {                [2]
    const ctx = canvasRef.current?.getContext("2d"); [3]
    if (!ctx || !logic) return; [4]


    rows = logic.getRows();
    columns = logic.getColumns();

    // Background (setzt Background auf schwarz
    ctx.fillStyle = "black";
    ctx.fillRect(0, 0, columns * blockWidth, rows *
       ↪ blockHeight);

    //Chessboard Background [4]
    for (let y = 0; y < rows; y++){
      for (let x = 0; x < columns; x++){
        if ((x + y) % 2 === 0) {
          ctx.fillStyle = "#222"; // darker square color
        }else{
          ctx.fillStyle = "#262626"; //lighter square color
        }
        ctx.fillRect(
          x * blockWidth,
          y * blockHeight,
          blockWidth,
          blockHeight
        );
      }
    }
}
```

[2] In der Drawboard-Funktion wird der ganze Code für die Gestaltung des Canvas geschrieben. [3] Diese Zeile prüft, ob canvasRef.current nicht null oder undefined ist, bevor versucht wird, getContext("2d") aufzurufen. Das verhindert Laufzeitfehler, für den Fall, dass das Canvas-Element noch nicht im DOM vorhanden ist. canvasRef.current greift auf das aktuelle DOM-Element eines <canvas>-Elements zu, das über ein canvasRef (React-Ref) referenziert wird. In unserem Projekt wie auch in normalen React Projekten wird useRef() häufig verwendet, um auf DOM-Elemente direkt zuzugreifen. Der Sinn ist, dass ctx den 2D-Kontext des Canvas in sich trägt, sofern das Canvas-Element existiert. Wenn nicht, dann ist ctx undefined. [4] In diesem Abschnitt wird das Spielfeld gezeichnet. Um diesen Schachbrett-Look zu erhalten prüft das If, ob die Werte erstens gleich 0 sind und ob sie den selben Typen besitzen (deswegen 3x=). Je nach Ausgangspunkt gibt es dann die dunklere Farbe oder die hellere. ctx.fillStyle sorgt dann dafür dass die richtige Farbe genutzt wird und durch ctx.fillRect werden dann die ausgewählten Pixel in diese Farbe geändert.

Nun würde ich gern noch zeigen wie die Implementierung mit PNG-Dateien von außerhalb funktioniert.

```
import appleImg from "../assets/Apple_Online_Snake.png"; //
   ↪ Import the apple image [5]

const GamePage: React.FC = () => {
  const navigate = useNavigate();
  const { inGame, gameMode, endGame, ws, setWsObject } =
     ↪ useGame();
  const audioRef = useRef<HTMLAudioElement | null>(null);

  const canvasRef = useRef<HTMLCanvasElement>(null);

  const [currentSnakeLength, setCurrentSnakeLength] = useState
     ↪ (1);
  const [playTime, setPlayTime] = useState("");
  const [logic, setLogic] = useState<SinglePlayerLogic |
     ↪ MultiplayerLogic | undefined>(undefined);
  const [showGameOverDialog, setShowGameOverDialog] = useState
     ↪ (false);
  const [muted, setMuted] = useState(() => localStorage.
     ↪ getItem("musicMuted") === "true");

  // Food-Bild vorbereiten (ausserhalb von drawBoard, am
     ↪ Anfang der Komponente) [6]
  const foodImageRef = useRef<HTMLImageElement | null>(null);
  useEffect(() => {
    const img = new window.Image();
    img.src = appleImg;
    foodImageRef.current = img;
  }, []);
```

[5] Hier wird die Bild-Datei des Apfels importiert. [6] Als erstes wird eine Konstante (foodImageRef) erstellt. Diese wird später dazu genutzt die Bild-Datei mit dem dazugehörigen Block zu verbinden.

```
// draw food
    logic.getFood().forEach(food => {
      if (foodImageRef.current && foodImageRef.current.
         ↪ complete) {
        ctx.drawImage(
          foodImageRef.current,
          food.x * blockWidth,
          food.y * blockHeight,
          blockWidth,
          blockHeight
        );
      } else {
        // Fallback, if the image is not loaded yet
        ctx.fillStyle = "red";
        ctx.fillRect(
```

```
          food.x * blockWidth,
          food.y * blockHeight,
          blockWidth,
          blockHeight
        );
      }
    });
```

Das If überprüft ob das momentanige Bild komplett ist. Wenn das passt wird per ctx.drawImage die Bild-Datei auf den aktuellen Platz des Essens gelegt und, zu dem Zeitpunkt die dunkel grauen Pixel, durch das Bild ersetzt. Falls dies nicht der Fall sein sollte ist der zweite Teil des If dafür da, die Pixel rot zu färben während das Bild nicht lädt.

5.2.4 Probleme

Kommen wir nun zu den Problemen. Wichtig bei dem Canvas ist, dass es NICHT in einem UseEffect geschrieben wird. Da es passiert ist das man als Person ohne großer Ahnung von TypeScript dachte, dass man das so macht. Und dann refresht es nicht schnell genug und es kommt zu Lags. Dies wird durch die Umstellung zu der Funktion drawBoard geändert und macht es deutlich verständlicher sobald man es einmal verstanden hat. Weitere Probleme sind auch aufgefallen durch die Autovervollständigung. Da diese auf den ersten Blick gleich aussehen. Jedoch durch einen kleinen Unterschied sehr viel Chaos bringen können. Jedoch muss man zu geben, dass Canvas für unseren Anwendungsfall die schnellere, performantere und auch simplere Art war als unser vorheriges System mit den Divs.

5.2.5 Geschrieben von Sabeleis

GitHub Nutzername: Sabeleis
Matrikelnummer: 7050076

5.3 Sternenhimmel-Hintergrund

Das **Stylesheet** `stars.css` erzeugt einen animierten Sternenhimmel als Hintergrund für die Anwendung in allen Modi und Bildschirmen. Es nutzt mehrere Ebenen von unterschiedlich großen und schnell bewegten Sternen, um einen realistischen Parallaxe-Effekt zu erzielen.

5.3.1 Aufbau

1. Hintergrund

```
// frontend/src/css/stars.css
html {
  height: 200%;
  width: 100%;
  background: radial-gradient(ellipse at bottom, #1b2735
    ↪ 0%, #090a0f 100%);
  overflow: hidden;
}
```

   - Setzt einen radialen Farbverlauf als Nachthimmel.
   - Verhindert Scrollen durch overflow: hidden.

2. Sternebenen: Es gibt vier Ebenen: `#stars`, `#stars2`, `#stars3`, `#stars4`. Jede Ebene verwendet ein kleines, transparentes Element mit vielen weißen Schatten (`box-shadow`), die als Sterne erscheinen.

```
// frontend/src/css/stars.css
#stars {
  width: 4px;
  height: 4px;
  background: transparent;
  box-shadow: 743px 879px #FFF , 1145px 1260px #FFF , 1412
    ↪ px 672px #FFF , 507px 1211px #FFF; // (gekuerzt)
  animation: animStar 50s linear infinite;
}
#stars:after {
  content: "␣";
  position: absolute;
  top: 2000px;
  width: 4px;
  height: 4px;
  background: transparent;
  box-shadow: 743px 879px #FFF , 1145px 1260px #FFF , 1412
    ↪ px 672px #FFF , 507px 1211px #FFF; // (gekuerzt)
}
```

   - Die `:after`-Pseudo-Elemente sorgen dafür, dass die Animation nahtlos wiederholt wird.

- Die vier Ebenen sind unterschiedlich konfiguriert:
  `#stars`: Größte Sterne, schnellste Bewegung (50s)
  `#stars2`: Mittlere Sterne, schnelle Bewegung (100s), leicht transparent
  `#stars3`: Kleine Sterne, langsame Bewegung (150s), sehr transparent
  `#stars4`: Kleinste Sterne, langsamste Bewegung (250s)

3. Animation

```
// frontend/src/css/stars.css
keyframes animStar {
  from {
    transform: translateY(100px);
  }
  to {
    transform: translateY(-2000px);
  }
}
```

- Lässt die Sterne von unten nach oben wandern.
- Die unterschiedlichen Animationsdauern erzeugen einen Parallax-Effekt.

5.3.2 Verwendung

Um den Sternenhimmel zu nutzen, müssen im HTML folgende Elemente eingefügt werden:

```
frontend/src/pages/HomePage.tsx:
    <div id="stars"></div>
    <div id="stars2"></div>
    <div id="stars3"></div>
    <div id="stars4"></div>
```

Das Stylesheet muss eingebunden sein, z.B. in der `index.html`:

```
<link rel="stylesheet" href="src/css/stars.css">
```

5.3.3 Anpassung

- Die Anzahl und Position der Sterne sind anpassbare Werte in `boxshadow`.
- Die Geschwindigkeit kann durch die `animation`-Eigenschaft verändert werden.
- Die Farben können für andere Effekte geändert werden.

5.3.4 Geschrieben von GermanBrandt

GitHub Nutzername: GermanBrandt
Matrikelnummer: 7190678

5.4 Lokale und Globale Bestenlisten

Dieser Abschnitt dokumentiert die Implementierung der Bestenlisten-Systeme, die Spieler:innen ermöglicht, ihre Ergebnisse im Einzelspieler-Modus lokal und global zu vergleichen.

5.4.1 Erfassen eines Highscores

Um die Leistung eines Spielers in seinem Einzelspieler-Spiel festzustellen, wurde entschieden, die Länge der Spieler-Schlange zu nutzen. Dabei wird die maximale Länge der Schlange, die unmittelbar vor Spielende erreicht wurde, direkt als Punktzahl gewertet. Da das Spiel mit einer Schlange mit einem einzigen Segment beginnt, ist die Punktzahl von Beginn des Spiels an bei „1“. Die Höchstpunktzahl in einem Spiel wird auf Englisch auch „Highscore“ genannt. Der Highscore wird dann erfasst, wenn ein Game-Over ausgelöst wurde, etwa durch Kollisionen der Schlange mit sich selbst, den stationären oder bewegenden Hindernissen oder dem Spielfeldrand.[2]

Das Spielende, und damit auch Erfassung eines Highscores, wird in der Spielschleife (`snakeLoop()`) der Einzelspielerlogik (Pfad: `frontend/src/game/logic/SinglePlayerLogic.ts`) durch `isGameOver()` ausgelöst, die die oben benannten Kollisionen prüft. Um den Highscore zu generieren, werden drei Komponenten benötigt:

1. `playerName`: Der Spielername wird aus dem lokalen Speicher gelesen[3] und, sofern durch ein Fehler kein Name gespeichert ist, mit `'Unknown'` gekennzeichnet.
2. `score`: Die erreichte Punktzahl ist von der Anzahl der Schlangensegmente abhängig, wobei im Code zur Fehlerkorrektur diese Metrik um -1 und größer gleich 0 geprüft wird.
3. `this.stopWatch.getTime()`: Die verstrichene Zeit im Spiel wird beim Hochladen auf die globale Bestenliste zur Plausibilitätsprüfung genutzt, um unmögliche Punktzahlen abzufangen.

```
// frontend/src/game/logic/SinglePlayerLogic.ts
private snakeLoop = (): void => {
    // ... Weiterer Code ...
    if (this.isGameOver()) {
        const playerName = localStorage.getItem('playerName')
        ↪ || 'Unknown';
        const score = Math.max(this.snakeSegments.length - 1,
        ↪ 0);
        this.highscores.saveScore(playerName, score);
        this.highscores.uploadScore(
            playerName,
            score,
            this.stopWatch.getTime()
```

[2] Der Spielfeldrand wurde in der ausgelieferten Version ausgeschaltet: Die Schlange kann, ohne einen Game-Over auszulösen, durch den Spielfeldrand schleichen und erscheint auf der entgegengesetzten Seite des Spielfeldrands erneut.

[3] Der Spielername wird vor Spielbeginn auf der Hauptseite des Spiels im Namensfeld erfragt und von dort in den lokalen Speicher gelegt.

```
            );
            this.killSnake();
        }
        // ... Weiterer Code ...
}
```

Die Erfassung von Highscores wurde bewusst aus dem Mehrspieler-Modus entfernt, da es sich um einen anderen Spielmodus mit unterschiedlichen Grundvoraussetzungen und leicht veränderten Regeln handelt und daher eine andere Implementation nötig ist.

5.4.2 Lokale Bestenlisten

Wie im vorherigen Kapitel zu sehen war, wird nach dem Erfassen der Punktzahl zwei Methoden `saveScore()` und `uploadScore()` ausgeführt. Diese stehen jeweils für das lokale und globale Speichern des Highscores. In diesem Abschnitt geht es konkret um die lokale Bestenliste. Diese wird im Browser, mit der das Spiel gespielt wurde, gespeichert und ist damit selbst ohne Server immer erreichbar, solange das Frontend an sich geladen werden kann. Die lokale Bestenlisten-Verwaltung erfolgt durch die `LocalHighscoresManager`-Klasse (Pfad: `src/game/LocalHighscoresManager.ts`). Der Mechanismus verwendet den `localStorage` zur persistenten Speicherung der Highscores.

Datenstruktur Highscores werden als Array von `LeaderboardEntry`-Objekten nach folgendem Schema gespeichert:

```
// frontend/src/game/LocalHighscoresManager.ts
type LeaderboardEntry = {
    name: string,
    score: number
};
```

Score-Speicherung Die Methode `saveScore` aktualisiert die Bestenliste mit garantierter Sortierung und aktualisiert bei bestehendem Nutzernamen die Punktzahl:

```
// frontend/src/game/LocalHighscoresManager.ts
public saveScore(name: string, score: number) {
    const leaderboard = JSON.parse(localStorage.getItem('
       ↪ leaderboard') || '[]');
    const existingEntry = leaderboard.find(entry => entry.name
       ↪  === name);

    if (existingEntry) {
        existingEntry.score = Math.max(
            existingEntry.score,
            score
        );
    } else {
        leaderboard.push({ name, score });
    }
```

```
    leaderboard.sort((a, b) => b.score - a.score);
    localStorage.setItem(
        'leaderboard',
        JSON.stringify(leaderboard)
    );
}
```

Anzeigelogik Die Darstellung erfolgt dynamisch auf der Homepage der innerhalb der linken Bestenliste:

```
// frontend/src/pages/HomePage.tsx
public displayLeaderboard() {
    const leaderboard: LeaderboardEntry[] = JSON.parse(
       ↪ localStorage.getItem('leaderboard') || '[]');
    const leaderboardContainer = document.querySelector('.
       ↪ leaderboard.left');

    if (leaderboardContainer) {
        leaderboardContainer.innerHTML = '<h3>Local Highscores
           ↪ </h3>';
        // Sort by score (descending) and take top 20
        leaderboard
            .sort((a, b) => b.score - a.score) // Highest
               ↪ score first
            .slice(0, 20) // Only keep top 20
            .forEach((entry: LeaderboardEntry) => {
                const entryElement = document.createElement('
                   ↪ div');
                entryElement.textContent = `${entry.name}: ${
                   ↪ entry.score}`;
                leaderboardContainer.appendChild(entryElement)
                   ↪ ;
            });
    } else {
        console.error('Leaderboard container not found');
    }
}
```

5.4.3 Globale Bestenlisten

Highscore-Upload Beim Spielende im Frontend wird der Highscore über eine HTTP-POST-Anfrage an den Endpunkt `/highscores` die Serveraddresse[4] gesendet. Der Request in der Methode `uploadScore()` enthält den Spielernamen, die erreichte Punktzahl und die Spielzeit in Sekunden (`playerName, score, gameDuration`). Das Backend empfängt diese Daten und validiert sie zunächst gegen ein festes Schema und dann gegen vorgegebene Parameter, wie erlaubte Länge und Zeichen im Namensfeld und einem plausiblen maximalen Punktwert[5]. Nach erfolgreicher Validierung wird der Datensatz über Drizzle ORM in die PostgreSQL-Datenbank eingefügt, wobei eine ID und das Erstellungsdatum automatisch generiert werden.

```
// frontend/src/game/LocalHighscoresManager.ts
public uploadScore(name: string, score: number, time: number)
    ↪ {
    const gameDuration = Math.floor(time / 1000); // calculate
        ↪  s from ms
    // ... Weiterer Code (Link-Generierung) ...
    fetch(`http${USE_SECURE === 'true' ? 's' : ''}:/ /${
        ↪ BACKEND_URL}/highscores`, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            playerName: name,
            score,
            gameDuration
        })
    })
    // ... Weiterer Code ... //
}

// --- Backend-Code
// backend/src/index.ts
const app = new Elysia()
    .use(corsConfig) // Cross Origin Resource Sharing
    .use(highscoresRoutes) // Routing for highscore API
    // ... Weiterer Code ...

// backend/src/routes/highscores.route.ts
export const highscoresRoutes = new Elysia()
    .post('/highscores', createHighscore) // POST Method (
        ↪ Incoming Highscores)
    .get('/highscores', getHighscores); // GET Method (
```

[4]Die Backend-Server-Adresse, an die das Frontend die HTTP-Anfragen sendet, wird in der Datei `/frontend/.env` festgelegt. Diese Implementation war wichtig, da sonst zum jedem Wechsel des Backends (lokal/extern) die URL an mehreren Stellen geändert werden muss.

[5]Das Spielprinzip von Snake gibt vor, dass die Schlange nicht größer wachsen kann als die Anzahl an Feldern im Spielfeld.

```
        ↪ Outgoing Highscores)

// backend/src/controllers/highscores.controller.ts
export const createHighscore = async ({ body, set }: Context<{
   ↪  body: HighscoreInput }>) => {
// Zu lange fuer LaTeX-Compiler. Siehe in Z.17-61 in .ts-Datei
}
```

Highscore-Abfrage Die Abfrage der globalen Bestenliste erfolgt über einen einfachen GET-Request an denselben Endpunkt in der Methode `getAllHighscores()` im zugehörigen `highscores.service.ts`. Das Backend liefert daraufhin die 20 höchsten Scores aus der Datenbank, absteigend sortiert nach Punktzahl. Die Antwort wird im Frontend direkt in die Bestenlistenansicht eingebunden.

```
// frontend/src/pages/HomePage.tsx
// ... Weiterer Code (HomePage) ...
useEffect(() => {
    displayLeaderboard();
    // Fetch global highscores from backend
    fetch(`http${USE_SECURE === 'true' ? 's' : ''}://${
       ↪ BACKEND_URL}/highscores`)
      .then(res => res.json())
      .then((data: Highscore[]) => {
        setGlobalHighscores(data);
      })
      .catch(() => setGlobalHighscores([]));
  }, []);
// ... Weiterer Code (HomePage) ...


// siehe Abschnitt "Highscore-Upload" fuer Routing im Backend
// backend/src/controllers/highscores.controller.ts
export const getHighscores = async () => {
    try {
        return await highscoresService.getAllHighscores();
    } catch (error) {
        console.error('Error fetching highscores:', error);
        throw new Error('Failed to retrieve highscores');
    }
};

// backend/src/services/highscores.service.ts
getAllHighscores = async () => {
    return await db.query.highscores.findMany({
        orderBy: [desc(highscores.score)],
        limit: 20
    });
}
```

Sicherheit und Abhängigkeiten Das System nutzt Zod für robuste Inputvalidierung, Drizzle ORM für typsichere Datenbankinteraktionen und Elysia.js als effizientes Backend-Framework. Sicherheitsmaßnahmen umfassen Rate-Limiting (5 Requests/10s pro IP), CORS-Restriktionen und automatische Prepared Statements zur SQL-Injection-Prävention. Die Datenbankanbindung erfolgt über den Neon Serverless Driver für optimierte Performance in der Cloud.

5.4.4 Einordnung in das Projekt

Der Bestenlisten-Service ist als modularer Dienst in die Projektarchitektur eingebunden:

1. **Systemressourcen:**
    - Wiederverwendung der Spielzeitmessung für Plausibilitätschecks
    - Wiederverwendung des Spielernamens aus der Mehrspieler-Namenseingabe
    - Minimale Eingriffe in die Kernspielmechaniken
    - Nahtlose Nutzererfahrung durch Hintergrund-Synchronisation
2. **Backend-Integration:**
    - Läuft als eigenständiger Service auf der gemeinsamen Elysia.js-Instanz
    - Nutzt dieselbe Infrastruktur wie der Mehrspieler-Modus
    - Verfügbar unter `/highscores` (REST-Endpoints)
    - Einfache integrierte Sicherheitsmechanismen (Plausibilitätscheck, Typenvalidierung, Injektions-Schutz durch Prepared Statements beim Upload auf die Datenbank, Rate-Limit von 5 Requests/10s)
3. **Datenbankanbindung:**
    - Dedizierte `highscores`-Tabelle in der gemeinsamen PostgreSQL-Instanz
    - Lastverteilung durch Serverless-Architektur der Neon-Datenbank
    - Automatische Backups der `highscores`-Tabelle durch Neon

5.4.5 Geschrieben von ultra-ms

GitHub Nutzername: ultra-ms
Matrikelnummer: 63222372