

The Hunt

Dokumentation des Spiels für das Modul „Webtechnologie Projekt“

Daniel Dornhof
Joe Phoomkhonsan

SoSe 2018

1. Einleitung

Grund für das Spiel

Das von uns entwickelte Spiel wurde im Rahmen des Moduls „Webtechnologie Projekt“ programmiert. Hierbei sollten wir am Beispiel der Spielentwicklung relevante Webtechnologien kennen lernen. Hierzu sollte die Darstellung des Spiels ausschließlich auf DOM-Tree-Manipulationen beruhen und auf den Einsatz von Canvas zu verzichten.

Diese Dokumentation dient dazu, das entwickelte Spiel genauer zu erläutern und um zu erklären wie die Anforderungen, an das Projekt, erfüllt wurden.

Ideenfindung

Bei der Ideensuche für unser Spiel, sind wir auf das Spiel „Blitzcrank's Poro Roundup“ (developed by Pure Bang Games and published by Riot games) gestoßen. Wir fanden das Grundprinzip dieses Spiels recht interessant und haben die Grundlagen dieses Spiel, genauer gesagt das fangen von Objekten in unsere Spielidee einfließen lassen und um weitere Objekte wie Gegner und Hindernisse erweitert. Am Ende ist unser Spiel zu einem anderem geworden und erinnert nur noch sehr gering an das Spiel, von dem unsere Idee ausging.

Rechtliches

Wir haben leider kein Recht an bzw. keine Lizenz für die verwendeten Assets / Bilder. Aufgrund dessen können wir einer Veröffentlichung des Spiels leider nicht zustimmen.

Vorweg

Um das Programm auf Github-Pages zum laufen zu bekommen, musste die „index.html“ und dementsprechend auch die „LevelConfig.json“ in das Hauptverzeichnis verschoben/kopiert werden. So entstehen 3 „index.html“ und 3 „LevelConfig.json“ Dateien.

2. Anforderungen und abgeleitetes Spielkonzept

2.1 Anforderungen

Das fertige Spiel sollte folgende, in der Tabelle (nächsten Seiten) aufgezeigten, Anforderungen erfüllen:

ID	Kurztitel	Anforderung
AF-1	Single-Player-Game als Single-Page-App	<ul style="list-style-type: none"> Das Spiel ist als Ein-Spieler-Game zu konzipieren. Das Spiel ist als HTML-Single Page App zu konzipieren. Das Spiel muss als statische Webseite von beliebigen Webservern (HTTP) oder Content Delivery Networks (CDN) bereitgestellt werden können (bspw. mittels GitHub Pages). Alle Spielressourcen (z.B. Level-Dateien, Bilder, CSS-Dateien, etc.) sind daher relativ und nicht absolut zueinander zu adressieren.
AF-2	Balance zwischen technischer Komplexität und Spielkonzept	<ul style="list-style-type: none"> Sie sollen ein interessantes Spielkonzept entwickeln oder ein bestehendes Spielkonzept abwandeln. Spielkonzepte sind nicht immer technisch komplex (z.B. Memory). Sie sollen jedoch ein Spiel konzipieren, dass eine vergleichbare Komplexität mit den Spielen der Hall-of-Fame hat (Memory wäre bspw. zu einfach; Schach zu kompliziert, da sie für ein Ein-Spieler-Game eine Gegner-KI entwickeln müssten). Unabhängig von der inneren Komplexität soll das Spiel schnell und intuitiv erfassbar sein und angenehm auf einem SmartPhone zu spielen sein.
AF-3	DOM-Tree-basiert	<ul style="list-style-type: none"> Das Spiel soll dem MVC-Prinzip folgen (Model, View, Controller). Das Spiel soll den DOM-Tree als View nutzen. Es sind keine Canvas-basierten Spiele erlaubt. <i>Hintergrund: Sie sollen Webtechnologien lernen und nicht wie man eine Grafikbibliothek programmiert.</i>
AF-4	Target Device: SmartPhone	<ul style="list-style-type: none"> Das Spiel soll für eine SmartPhone Bedienung konzipiert werden. Entsprechende Limitierungen sind zu berücksichtigen. Als Target Devices sind die Plattformen Android und iOS zu berücksichtigen. Das Spiel soll mit HTML5 mobile Browsern auf den genannten Plattformen spielbar sein.
AF-5	Mobile First Prinzip	<ul style="list-style-type: none"> Das Spiel soll bewusst für SmartPhones konzipiert werden. Das Spiel soll auch auf Tablets und Desktop PCs spielbar sein. Einschränkungen sind zu minimieren, werden aber billigend in Kauf genommen (z.B. fehlende 3D-Lage bei Desktop Browsern). Sie sollen typische mobile Interaktionen sinnvoll nutzen (z.B. Swipe, Wischen, 3D Lage im Raum, etc.). Übertragen sie nicht eine typische Desktop-Bedienung auf Mobile. Sie müssen allerdings keine mobile Interaktionen sklavisch nutzen - wenn dies nicht sinnvoll für das Spielkonzept ist.
AF-6	Das Spiel muss schnell und intuitiv erfassbar sein und Spielfreude erzeugen.	<ul style="list-style-type: none"> Das Spiel muss schnell und intuitiv erfassbar sein. Das Spiel muss Spielfreude erzeugen. Hintergrund: <i>Ihre Spiele sollen ggf. als Anschauungsbeispiele für Studieninteressierte Schüler auf Messen oder ähnlichen Veranstaltungen</i>

		<p>gezeigt werden. Sie arbeiten also nicht für "die Tonne" - andere sollen ihre Spiele tatsächlich spielen.</p> <ul style="list-style-type: none"> • <i>Tipp: Vermeintlich einfache Spielprinzipien haben ihre Stärken und sind dennoch komplex genug um die technischen Anforderungen dieses Projekts abzudecken.</i>
AF-7	Das Spiel muss ein Levelkonzept vorsehen	<ul style="list-style-type: none"> • Es ist ein steigender Schwierigkeitsgrad über mindestens 7 Level vorzusehen. • Level sollen deklarativ in Form von Textdateien beschrieben werden können (z.B. JSON, XML, CSV, etc.). • Diese Level sollen durch das Spiel nachgeladen werden können, so dass nachträglich Level ergänzt und abgeändert werden können, ohne die Programmierung des Spiels anpassen zu müssen.
AF-8	Ggf. erforderliche Speicherkonzepte sind Client-seitig zu realisieren	<ul style="list-style-type: none"> • Aufgrund der Zielgruppe sollen durch das Spiel gesammelte Daten auf den Geräten bleiben. • Aufgrund des Demonstrationscharakters auf Messen dürfen keine zentrale Server für Highscores, etc. erforderlich sein oder angebunden werden. • Ggf. erforderliche Stateful solutions sind mittels client-seitiger Storage-Konzepte zu lösen. D.h. z.B. mittels <u>local storage</u>.
AF-9	Dokumentation	<ul style="list-style-type: none"> • Das Spiel muss nachvollziehbar dokumentiert sein. • Das Spiel muss analog dem SnakeGame dokumentiert sein. <i>Hinweis: Nutzen sie die SnakeGame-Dokumentation als Template.</i>
Dokumentationsanforderung		
D-1	Dokumentationsvorlage	<ul style="list-style-type: none"> • Die Dokumentation soll sich an vorliegender Vorlage orientieren
D-2	Projektdokumentation	<ul style="list-style-type: none"> • Das Spiel muss geeignet dokumentiert sein, so dass es von projektfremden Personen fortgeführt werden könnte
D-3	Quelltextdokumentation	<ul style="list-style-type: none"> • Der Quelltext des Spiels muss geeignet dokumentiert sein und mittels schriftlicher Dokumentation erschließbar und verständlich sein.
D-4	Libraries	<ul style="list-style-type: none"> • Alle verwendeten Libraries sind aufzuführen und deren Notwendigkeit zu begründen
Technische Randbedingungen		
TF-1	No Canvas	<ul style="list-style-type: none"> • Die Darstellung des Spielfeldes sollte ausschließlich mittels DOM-Tree Techniken erfolgen. Die Nutzung von Canvas-basierten Darstellungstechniken ist explizit untersagt.
TF-2	Levelformat	<ul style="list-style-type: none"> • Level sollten sich mittels deskriptiver Textdateien definieren lassen (z.B. mittels CSV, JSON, XML, etc.), so dass Level-Änderungen ohne Sourcecode-Änderungen des Spiels realisierbar sind.
TF-3	HTML+CSS	<ul style="list-style-type: none"> • Der View des Spiels darf ausschließlich mittels HTML und CSS realisiert werden.
TF-4	Gamelogic in Dart	<ul style="list-style-type: none"> • Die Logik des Spiels muss mittels der Programmiersprache Dart realisiert werden.
TF-5	Browser Support	<ul style="list-style-type: none"> • Das Spiel muss im Browser Chromium/Dartium (native Dart Engine) funktionieren. Das Spiel muss ferner in allen anderen Browsern (JavaScript Engines) ebenfalls in der JavaScript kompilierten Form funktionieren (geprüft wird ggf. mit Safari, Chrome und Firefox).

TF-6	MVC Architektur	<ul style="list-style-type: none"> Das Spiel sollte einer MVC-Architektur folgen.
TF-7	Erlaubte Pakete	<ul style="list-style-type: none"> Erlaubt sind alle dart:* packages.
TF-8	Verbotene Pakete	<ul style="list-style-type: none"> Verboten sind Libraries wie Polymer oder Angular. (Sollten Sie Pakete verwenden wollen, die außerhalb der erlaubten Pakete liegen, holen Sie sich das Go ab, begründen sie bitte, wieso sie das Paket benötigen).
TF-9	No Sound	<ul style="list-style-type: none"> Das Spiel muss keine Sounds unterstützen.

2.2 Spielkonzept

Der Spieler kontrolliert einen Charakter, welcher sich am linken Rand des Spielfeldes befindet. Der Charakter kann auf und ab bewegt werden, ein Netz zum fangen und Pfeile schießen.

- Mobile Geräte:
 - Bewegung: Auf und ab swipen.
 - Netz und Pfeile: Buttons am rechten unterem Rand des Bildschirms
- Computer
 - Bewegung: Pfeiltasten – hoch und runter
 - Netz: S
 - Pfeile: A

Das Spielfeld besteht aus mehreren Reihen, zwischen denen der Charakter / Spieler wechseln kann. Auf diesen Reihen bewegen sich Entitäten auf den Charakter / linken Rand des Bildschirms zu (Bisher: ein Gegner [Wolf], ein Hindernis [Fels] und ein Ziel [Hase]). Diese Entitäten werden bei jedem „Spawn-Prozess“ zufällig generiert.

Zu Beginn des Spiels hat der Spieler 3 Leben und 20 Schüsse Pfeile. Diese können allerdings durch Pick-Ups, welche von Gegnern fallen gelassen werden, erhöht werden. Der Spieler kann maximal 3 Leben haben, allerdings ist die Anzahl an Pfeilen unbegrenzt. Sollten dem Spieler die Pfeile ausgehen, kann dieser keine neuen Pfeile erhalten (Dies ist eine Art Survival-Element, welches beabsichtigt ist und den Spieler dazu anregen soll, über seine Ressourcen nachzudenken).

Weitere Pick-Ups in der derzeitigen Version sind ein „Double-Points-Power-Up“ und ein Power-Up, welches die Geschwindigkeit der Gegner und den „Spawn-Prozess“ verlangsamt.

Das Ziel des Spiels ist es, so viele Punkte wie möglich zu sammeln und dabei nicht zu sterben.

Möglichkeiten um Punkte zu erhalten sind folgende:

- Töte einen Gegner (Wolf) mit Pfeilen = 1 Punkt

- Fange ein Ziel (Hase) mit dem Netz = 5 Punkte
- Fange ein Ziel (Hase) in dem du es berührst = 1 Punkt

Möglichkeiten um Leben zu verlieren sind:

- Berühre ein Hindernis (Stein) oder einen Feind (Wolf)

Das Spiel erhöht das Level nach einer in der Level-Datei festgelegten Zeit. Bei jedem Level des Spiels kann man festlegen wie Schnell die Entitäten sich bewegen können sollen und wie schnell der „Spawn-Prozess“ wieder eintreten soll. Abgesehen davon, kann man für jedes Level einzeln festlegen, wie lange das Level andauern soll. Sollten keine weiteren Level deklariert sein, gibt es keinen Level Up-Prozess und das Level bleibt bis zum Game-Over bestehen.

Das Spiel ist so konzipiert, dass weitere Feinde und Geschosse einfach hinzugefügt werden können und nur minimale Änderungen am Code vorgenommen werden müssen.

3. Architektur und Implementierung

Die Architektur des Spiels ist das vorgegebene Model-View-Controller Prinzip. Durch das Prinzip teilt sich die Spiellogik auf die verschiedenen Klassen auf. Eine zentrale Rolle für die Spielsteuerung hat der Controller.

Der Controller verarbeitet die Nutzerinteraktionen (Das Betätigen der verschiedenen Buttons, Tastenanschläge auf dem Computer und Swipes auf dem Handy) und die Zeitsteuerung (die Trigger für die Bewegungen aller einzelnen Elemente, das Prüfen der aktiven Power-Ups und das Triggern des Level-Ups). Die Funktionsweise des Controllers wird Kapitel 3.3 beschrieben.

Die View wird verwendet um den DOM-Tree zu manipulieren und bietet dementsprechend Manipulationsmethoden für den Controller an, um den aktuellen Spielstand auf dem Browser zu repräsentieren. Die Funktionsweise des View wird in Kapitel 3.2 beschrieben.

Das Model, stellt das Spiel und die Spielstände dar. Da das Model die Spiellogik und die einzelnen Elemente des Spiels beinhaltet ist der Code in diesem Teil etwas größer und komplexer. Die Logik des Models wird in Kapitel 3.1 etwas genauer erklärt.

3.1 Model

Aus dem Spielkonzept gehen verschiedene Geschosse, Entitäten und Pick-Ups hervor. Abgesehen davon gibt es noch einen Charakter, welcher durch den Spieler gesteuert wird. Ein Spiel beinhaltet einen Charakter, kann mehrere Entitäten, Pick-Ups und Geschosse enthalten. Das Model können Sie der „Diagram.jpeg“ im Ordner dieses Dokuments entnehmen.

3.1.1 Game

Der Controller interagiert nur mit dieser Klasse aus dem Model, somit ist diese Klasse dafür zuständig alle Befehle an die weiteren Klassen weiterzuleiten und ist somit ein „Sammelpunkt“ für alle Klassen und alle Interaktionen zwischen diesen Klassen im Model.

- **rows** liefert die Anzahl der Reihen für das Spielfeld
- **level** bezeichnet den aktuellen Level, in dem sich das Spiel derzeit befindet.
- **score** gibt den aktuellen Punktestand des Spielers wieder
- **started** und **paused** werden dazu verwendet um festzustellen, ob das Spiel bereits gestartet wurde und wenn es gestartet wurde ob es derzeit pausiert ist.
- **activeSlowedGame** und **activeDoublePoints** werden verwendet um festzustellen ob einer der beiden Power-Ups aktiv ist.
- **slowedGamePickedUp** und **doublePointsPickedUp** werden dazu verwendet um zu prüfen ob eines der Power-Ups aufgehoben wurde.

Ein *Game* Objekt

- kann mittels des Konstruktors erzeugt werden, dabei wird die Anzahl der Reihen übergeben.
- Das Objekt kann mittels **moveEntities()**, **moveBullest()** und **movePickUps()** alle Objekte auf dem Feld bewegen. Abgesehen davon, werden diese Methoden dazu verwendet um die Kollisionen der einzelnen Elemente zu ermitteln und zu verarbeiten. Diese Funktionen entfernen die Objekte gegebenenfalls.
- Mittels **shootBullet()**, **shootNet()**, **spawnEntities()** und **spawnPickUps()** werden die verschiedenen Objekte auf dem Spielfeld erzeugt.
- Durch die Methoden **addHealth** und **addAmmo** werden dem Charakter, jeweils Leben und Munition / Pfeile hinzugefügt.
- Durch die Methode **addPoints**, werden dem Spielstand Punkte hinzugefügt
- Die restlichen Methoden dieser Klasse fungieren als getter und setter für den GameState (**started**, **paused**) und für die Power-Ups.

3.1.2 Character

- **currentRow** gibt die derzeitige Reihe des Characters wieder.
- **ammo** gibt die derzeitige Anzahl an Pfeilen wieder.
- **health** gibt die derzeitige Anzahl an Leben wieder.

- **alive** wird verwendet um zu Prüfen ob der Charakter noch lebt
- **netOut** wird dazu verwendet um zu schauen ob das Netz grad in Benutzung ist, um nicht mehrere Netze zur selben Zeit zu verwenden.
- **noAmmo** wird verwendet um zu schauen ob der Stand der Pfeile auf 0 ist um das schießen der Pfeile einzustellen, wenn dem so ist.

Der Character wird mittels Kontruktor erstellt und wird immer, in (einer) der mittleren Reihe platziert.

- Die **shootBullet()** und **gainAmmo**-Methoden modifizieren die Munition des Characters.
- Mithilfe von **shootNet()** und **regainNet()** wird **netOut** modifiziert
- **moveUp()** und **moveDown()** werden zum bewegen des Characters verwendet.
- **receiveDamage** und **gainHealth** werden dazu verwendet um die derzeitige Anzahl an Leben zu modifizieren und im gegebenen Fall den Character als tot zu deklarieren.

3.1.3 PickUp

Dies ist eine abstrakte Klasse. Diese Klasse wird dazu verwendet um eine beliebige Anzahl an verschiedenen Pick-Ups zu erstellen. Power-Ups benötigen weitere Implementation im Controller (beispielweise Timer).

Die jeweiligen Pick-Ups werden über jeweils eigene Konstruktoren erstellt. Pick-Ups werden von Enemy1 (Wolf) fallen gelassen und bekommen die Reihe und Position des Objekts, von dem sie fallen gelassen werden.

- **row** dient zur Bestimmung der Reihe des Objekts
- **currentPos** dient zur Bestimmung der Position des Objekts
- **type** dient zur Ermittlung des darzustellenden Objekts in der **View**
- **remove** dient zum feststellen ob dieses Objekt bei dem nächsten **movePickUps()**-Aufruf des Game-Objekts entfernt werden soll.
- **onPickUp()** löst den pickUp-Effekt des Objekts aus.
- **move()** bewegt das Objekt bzw. verändert die **currentPos** des Objekts
- **setRemoveTrue()** wird verwendet um das Objekt entfernen zu lassen.

3.1.4 Bullet

Dies ist eine abstrakte Klasse. Diese Klasse wird dazu verwendet um eine beliebige Anzahl an verschiedenen Bullets (Geschossen) zu erstellen. Diese zeichnen sich dadurch aus, dass sie sich als einziges Objekt im Spiel von links nach rechts bewegen.

- **row** dient zur Bestimmung der Reihe des Objekts
- **currentPos** dient zur Bestimmung der Position des Objekts
- **type** dient zur Ermittlung des darzustellenden Objekts in der **View**
- **remove** dient zum feststellen ob dieses Objekt bei dem nächsten
- **move()** bewegt das Objekt bzw. verändert die **currentPos** des Objekts

3.1.5 Entity

Dies ist eine abstrakte Klasse. Diese Klasse wird dazu verwendet um eine beliebige Anzahl an verschiedenen Entities (Gegner / Hindernisse / Ziele) zu erstellen.

- **row** dient zur Bestimmung der Reihe des Objekts
- **currentPos** dient zur Bestimmung der Position des Objekts
- **health** dient zum feststellen der derzeitigen Leben des Objekts
- **speed** dient dazu um das Objekt sich schneller bewegen zu lassen. (Implementiert aber nicht genutzt).
- **points** wird verwendet um festzustellen, wieviel Punkte das Objekt bringt
- **type** dient zur Ermittlung des darzustellenden Objekts in der **View**
- **damage** wird verwendet um festzustellen, wieviel Schaden dieses Objekt dem Character zufügt.
- **alive** dient zum feststellen, ob dieses Objekt bei dem nächsten
- **onHit()** wird ausgeführt, wenn dieses Object von einem Pfeil getroffen wird
- **onDeath()** wird ausgeführt, wenn dieses Object alle Leben verliert.
- **onCatch()** wird ausgeführt, wenn dieses Object von einem Netz getroffen wird.
- **onTouch()** wird ausgeführt, wenn dieses Object den Character berührt.
- **move()** bewegt das Objekt bzw. verändert die **currentPos** des Objekts.

3.2 View

Die View dient zur Darstellung des Spiels auf dem Browser / für den Spieler. Die View besteht aus mehreren Teilen. Zu einem das HTML-Dokument (siehe Abschnitt 3.2.1) und einer clientseitigen Logik, die den DOM-Tree des HTML-Dokuments manipuliert (siehe 3.2.2).

3.2.1 HTML-Dokument

Die View wird im Browser initial durch folgendes HTML-Dokument (Ausschnitt) erzeugt. Im Verlauf des Spiels wird der DOM-Tree dieses HTML-Dokuments durch die View-Klasse manipuliert, um den Spielzustand darzustellen und die Nutzerinteraktion zu ermöglichen.

```
8      <title>TheHunt</title>
9      <link rel="stylesheet" href="build/web/styles.css">
10     <script defer="" src="build/web/main.dart.js"></script>
11  </head>
12  <body>
13    <div class="container">
14      <div id="instructions">
24     <div id="swipeArea"></div>
25     <div id="landscape"></div>
26     <div id="menu">
27       <div id="gameOver">Game Over</div>
28       <span id="start">Start</span>
29       <div id="endScore"></div>
30     </div>
31     <table id="gameField"></table>
32     <div id="hud">
33       <div id="score">Score</div>
34       <div id="ammo">Ammo</div>
35       <div id="slowPower"></div>
36       <div id="doublePower"></div>
37       <div id="health"></div>
38       <div id="level"></div>
39     </div>
40     <div id="buttons">
41       <div id="netButton">Net</div>
42       <div id="shoot">Arrow</div>
43     </div>
44  </div>
```

Abbildung 1: HTML-Ausschnitt

3.2.2 View als Schnittstelle zum HTML-Dokument

Folgende Elemente haben eine besondere Bedeutung und können dabei über entsprechende Attribute der Klasse View angesprochen werden.

- Das Element mit dem Identifier #gameField dient dazu das Spielfeld (Tabelle) einzublenden.
- Des Weiteren sind hier viele Elemente, die zum updaten des Overlays bzw. des HUDS verwendet werden, vorzufinden.
- Das Element mit dem Identifier #swipeArea wird für die mobilen Anwender als Fläche verwendet um die Swipes auszuführen. Damit andere Elemente die auf dem Bildschirm sind nicht in den Weg kommen und keine Komplikationen mit simultanen Button-Clicks entstehen

Alle CSS-Gestaltungen werden in der style.css vorgenommen. Nur Ein- und Ausblendungen, werden in der View durchgeführt.

Objekte der View-Klasse agieren nie eigenständig, sondern werden ausschließlich von Objekten der Controller-Klasse genutzt um die Ansicht zu aktualisieren. Dazu können folgende Methoden verwendet werden:

- Die **characterUpdate()** - Methode aktualisiert die Position des Characters im Table-Element #gameField. Sie wird nur durch Nutzer-Interaktionen aufgerufen.
- Die **update()** - Methode aktualisiert die Positionen aller Elemente im Table-Element #gameField. Die Daten für die Aktualisierung, kommen aus dem übergebenem Game-Objekt. Diese Methode wird bei jedem Bewegungs-Trigger ausgeführt.
- **updateHUD()** wird verwendet um die HUD upzudaten. Darunter zählen: Punkte, Level, Munition, Power-Ups und Leben. Diese Methode wird immer ausgeführt, wenn **update()** ausgeführt wird
- **createField()** wird beim Start des Spiels ausgeführt und baut die Tabelle, basierend auf den Daten, auf ,die in dem Game-Objekt hinterlegt sind. Die Methode blendet das Menü aus und blendet das HUD und die Buttons für die Mobile-Bedienung ein.
- **updateField()** wird bei einem Level-Up ausgeführt und baut die Tabelle mit all ihren Inhalten neu auf, um gegeben falls eine neue Reihe hinzuzufügen.
- **gameOver()** blendet die Buttons und das Spielfeld aus und blendet das Menu wieder ein. Abgesehen davon wird der letzte erreicht Score unter dem Start-/ Restart-Button eingeblendet.

3.3 Controller

Der Controller ist zuständig für die Steuerung des Ablaufes des Spiels. Er verarbeitet zeitgesteuerte Events und Nutzerinteraktionen. Während des Spiels hat der Spieler am Computer die Möglichkeit den Charakter mit Hilfe von den Pfeiltasten auf und ab zu bewegen (von reihe zu reihe). Der Spieler am Smartphone nutzt dafür das auf und ab swipen am Touch-Screen. Des Weiteren kann der Spieler einen Pfeil schießen, dies wird erreicht in dem man auf den Arrow-Button am unteren rechten Rand des Bildschirms klickt. Alternativ kann der Spieler am Computer auch die A-Taste drücken. Für das schießen des Netzes gilt dasselbe wie für die Pfeile, allerdings kann der Computer-Nutzer hierfür die S-Taste nutzen.

Während das Spiel läuft, werden zeitgesteuerte Event mithilfe der Timer-Klasse (dart.async) realisiert.

3.3.1 aktives Spiel

Wie bereits, erwähnt findet die User-Interaktion während des Spiels nur über das swipen, betätigen von Buttons und auf dem Computer durch Tastendrucke statt.

Abgesehen davon laufen während des aktiven Spiels mehrere periodische Timer, welche nach einer vorgegebenen Zeit Methoden triggern. Hierzu zählen der **spawnTrigger**, welcher die **Entities** spawnen lässt, der **entityTrigger**, welcher die **Entities** bewegt und der **bulletTrigger**, welcher die **Bullets** bewegt. Abgesehen davon läuft noch ein **levelUpTrigger**, welcher die levelUp-Methode triggert. Hinzukommen noch die Trigger, welche das Ende von Power-Ups triggern. Diese sind allerdings nur aktiv, wenn diese auch benötigt werden (das heißt, wenn Power-Ups aktiv sind). Das Spiel ist vorbei, sobald festgestellt wird, dass der Charakter keine Leben mehr hat. Daraufhin wird im Controller die **gameOver()**-Methode ausgeführt, die Timer gestoppt und die **gameOver()**-Methoden der View und des Models ausgeführt.

Sollte das Spiel aus dem Focus gelassen werden oder sollte der Spieler das Smartphone im Portrait-Modus halten, wird das Spiel pausiert. Und im 2. Fall wird eine Aufforderung angezeigt, die den Spieler darum bittet das Smartphone in den Landscape-Modus zu bringen. Sollte der Nutzer das Smartphone in den Landscape-Modus bringen bzw. wieder das Spiel fokussieren, wird das Spiel fortgeführt.

Wird das Spiel pausiert, werden alle Timer angehalten und die Nutzerinteraktionen deaktiviert. Für längere Timer, läuft nebenbei noch eine Stopwatch, die die verstrichene Zeit mitzählt, um diese dann beim neustarten des Timers (fortführen des Spiels) von der Timer-Zeit abzuziehen.

3.3.2 Menu/Game-Over-Bildschirm

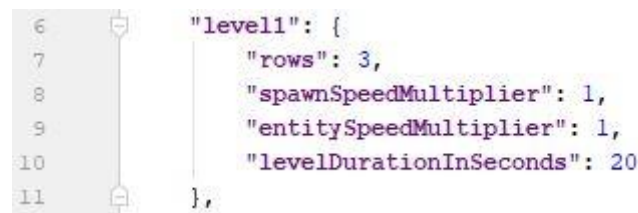
Hier ist jede Art von Interaktion mit dem Spiel ausgeschaltet und nicht möglich. Die einzige Interaktionsmöglichkeit des Spielers hier ist es den Start bzw. den Restart-Button zu betätigen. Dies führt dazu, dass das Spielfeld initialisiert wird, die Timer aktiviert werden und die View aktualisiert wird.

4 Level- und Parametrisierungs-Konzept

4.1 Levelkonzept

Das Levelkonzept wird realisiert indem, der Controller die Daten für die jeweiligen Level aus der „LevelConfig.json“ ausliest. Die „LevelConfig.json“ muss im selben Verzeichnis wie die „index.html“ liegen.

In folgender Abbildung, ist das Format der Level-Konfiguration zu sehen:



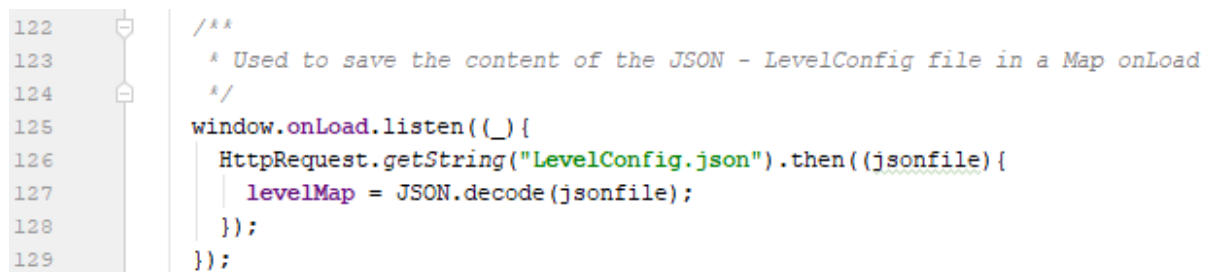
```
6 |  
7 |  
8 |  
9 |  
10 |  
11 |  
    "level1": {  
        "rows": 3,  
        "spawnSpeedMultiplier": 1,  
        "entitySpeedMultiplier": 1,  
        "levelDurationInSeconds": 20  
    },
```

Abbildung 2: Level-Konfiguration

Jedes Level hat als Schlüssel, den Begriff „level“ mit einer angehängten Nummer (startend von 1).

- **rows** wird dazu verwendet die Anzahl der Reihen in dem Level festzulegen.
- **spawnSpeedMultiplier** dient dazu um die Abstände zwischen den „Spawn-Prozessen“ zu modifizieren. Da diese Zahl mit den festen Abständen multipliziert wird, erhöht alles über 1 die Abstände und alles unter 1 verkleinert die Abstände. Hier können auch Zahlen mit Kommastellen verwendet werden.
- **entitySpeedMultiplier** dient dazu um die Abstände zwischen den „Bewegungs-Prozessen“ der Entities zu modifizieren (Das heißt die Geschwindigkeit der Entities zu modifizieren). Da diese Zahl mit den festen Abständen multipliziert wird, erhöht alles über 1 die Abstände und alles unter 1 verkleinert die Abstände. Hier können auch Zahlen mit Kommastellen verwendet werden.
- **levelDurationInSeconds** wird dazu verwendet um festzustellen, wie viele Sekunden das Level anhält, vorausgesetzt es gibt ein nächstes Level.

Die „LevelConfig.json“, wird beim Öffnen der Seite, in einer dafür angelegten Map im Controller gespeichert. Diese Map wird dazu verwendet um alle Inhalte der Level-Datei, wenn benötigt aufzurufen.



```
122 |  
123 |  
124 |  
125 |  
126 |  
127 |  
128 |  
129 |  
    /**  
    * Used to save the content of the JSON - LevelConfig file in a Map onLoad  
    */  
    window.onload.listen(() {  
        HttpRequest.getString("LevelConfig.json").then((jsonfile) {  
            levelMap = JSON.decode(jsonfile);  
        });  
    });
```

Abbildung 3: "Save Json in Map" – Snippet

4.2 Parametrisierungskonzept

Die Parameter werden ebenfalls aus der „LevelConfig.json“ erzeugten, Map gelesen und beinhalten nur zwei Parameter.

```

2  "durations": {
3      "slowDownDurationInSeconds": 30,
4      "doublePointsDurationInSeconds": 30

```

Abbildung 4: Parameter

Diese Parameter dienen dazu die Dauer der jeweiligen Power-Ups festzulegen.

5. Nachweis der Anforderungen

Nachfolgend wird erläutert wie die in Kapitel 2 aufgeführten Anforderungen erfüllt bzw. eingehalten werden. Anschließend wird angegeben, wer im Team welche Verantwortlichkeiten hatte.

5.1 Nachweis der funktionalen Anforderungen

ID	Titel	Erfüllt	Teilw. erfüllt	Nicht erfüllt	Erläuterung
AF-1	Single-Player-Game als Single-Page-App	X			Das Spiel kann nur von einem Spieler zurzeit gespielt werden, da es nur einen Charakter gibt. (siehe Kapitel 2) Das Teilen der Steuerung sehen wie nicht als Multiplayer. Das Spiel ist eine statische Webseite, dies geht aus den Vorführungen im Praktikum hervor.
AF-2	Balance zwischen technischer Komplexität und Spielkonzept	X			Das Spielkonzept, wie wir es entwickelt haben, haben wir so in der Hall-of-Fame noch nicht gesehen, desweiteren ist die Komplexität auf einem Niveau, der den Spielen der Hall-of-Fame ähnlich ist. Es wird in diesem Spiel keine KI verwendet, es gibt nur Zufallsgenerierte Abläufe. Dadurch, dass das Spiel für Smartphones entwickelt wurde ist es angenehm auf einem Smartphone spielbar.
AF-3	DOM-Tree-basiert	X			Die View des Spiels basiert rein auf der HTML- /der CSS- Datei und der Manipulation des DOM-Trees durch die View bzw, durch den Controller, der die View steuert. In Kapitel 3 wird die MVC Struktur des Projekts erläutert.
AF-4	Target Device: SmartPhone	X			Das Spiel besitzt Buttons und eine Swipe-Mechanik für Smartphones um die Steuerung des Charakters zu ermöglichen. Das Spiel ist auf allen gängigen Browsern spielbar (Mobile Geräte und Computer)
AF-5	Mobile First Prinzip	X			Es wird das Swipen auf den mobilen Geräten genutzt, um die Steuerung zu ermöglichen, desweiteren gibt es Buttons um die Schuss-Funktionen auszuführen.

					Die Buttons bleiben auf dem Bildschirm auch, wenn auf dem Computer gespielt wird. Dies indiziert, dass das Spiel für Smartphones konzipiert wurde.
AF-6	Das Spiel muss schnell und intuitiv erfassbar sein und Spielfreude erzeugen.	X			Durch die Anleitung im Startbildschirm ist des Spiels, erhält der Spieler, noch vor Beginn des Spiels Informationen über das Spiel. Damit ist es leichter in das Spiel reinzukommen. Durch die vielen Sachen, auf die man Achten sollte und die man falsch machen kann (Pfeile verschießen/ Hasen töten) generiert das Spiel in unseren Augen Spielfreude.
AF-7	Das Spiel muss ein Levelkonzept vorsehen	X			Wie in den vorherigen Kapiteln erklärt, hat das Spiel ein Level-Konzept, welches das Level nach einer bestimmten Zeit erhöht.
AF-8	Ggf. erforderliche Speicherkonzepte sind Client-seitig zu realisieren	X			Es wird kein Speicherkonzept benötigt und somit mussten wir auch keines realisieren
AF-9	Dokumentation	X			Sowohl der Code als auch das Spiel selber sind dokumentiert.

5.2 Nachweis der Dokumentationsanforderungen

ID	Titel	Erfüllt	Teilw. erfüllt	Nicht erfüllt	Erläuterung
D-1	Dokumentationsvorlage	X			Diese Dokumentation hält sich an die Struktur der Snake-Dokumentation.
D-2	Projektdokumentation	X			Diese Dokumentation, dient als Dokumentation für die einzelnen Elemente des Projekts.
D-3	Quelltextdokumentation	X			Alle Methoden und Datenfelder wurden, durch Inline-Kommentare erläutert.
D-4	Libraries	X			Es wurden nur dart: * packages verwendet.

5.3 Nachweis der Einhaltung technischer Randbedingungen

ID	Titel	Erfüllt	Teilw. erfüllt	Nicht erfüllt	Erläuterung
TF-1	No Canvas	X			Die Dartstellung des Spiels erfolgt ausschließlich durch den DOM-Tree und dessen Manipulation. Die Nutzung von Canvas ist ausschließlich untersagt.
TF-2	Levelformat	X			Wie in Kapitel 4 beschrieben, hat das Spiel ein Levelformat

TF-3	HTML + CSS	X			Die View des Spiels beruht ausschließlich auf HTML und CSS (siehe web/index.html)
TF-4	Gamelogic in Dart	X			Die Logik des Spiels ist mittels der Programmiersprache Dart realisiert worden.
TF-5	Browser-Support	X			Das Spiel ist in allen gängigen Browsern (JavaScript Engines) spielbar.
TF-6	MVC-Architektur	X			Das Spiel folgt durch die Ableitung mehrerer Modell-Klassen (siehe lib/src/model.dart), einer View-Klasse (siehe lib/src/view.dart) und dem zentralen Controller (siehe lib/src/controller.dart) einer MVC-Architektur. Die View greift auf das Model nur lesend zu und nicht manipulierend.
TF-7	Erlaubte Pakete	X			Es sind keine Pakete, außer den erlaubten Dart: * Paketen genutzt worden. Siehe pubspec.yaml.
TF-8	Verbotene Pakete	X			Es sind keine Pakete, außer den erlaubten Dart: * Paketen genutzt worden. Siehe pubspec.yaml.
TF-9	No Sound	X			Das Spiel hat keine Soundeffekte.

5.4 Verantwortlichkeiten im Projekt

Komponente	Detail	Asset	Daniel D.	Joe P.	Anmerkungen
Model	Game	lib/src/model.dart	V	U	Beinhaltet Inline-Dokumentation
	Character	lib/src/model.dart	V	U	Beinhaltet Inline-Dokumentation
	PickUp-Klassen	lib/src/model.dart	V	U	Beinhaltet Inline-Dokumentation
	Bullet-Klassen	lib/src/model.dart	V	U	Beinhaltet Inline-Dokumentation
	Entity-Klassen	lib/src/model.dart	V	U	Beinhaltet Inline-Dokumentation
View	HTML-Dokument	web/index.html	V	U	
	Gestaltung	web/style.css	V	U	
		web/images/*	V	U	
	Viewlogik	lib/src/view.dart	V	U	Beinhaltet Inline-Dokumentation
Controller	Eventhandling	lib/src/controller.dart	V	U	Beinhaltet Inline-Dokumentation
	Level und Duration-Parameter	web/LevelConfig.json	V	U	
Dokumentation	Game Report	doc/*.*	V	U	
Testen	Game-Testing		V	U	Keine Dokumentation zu den Tests

V = verantwortlich

U = unterstützend