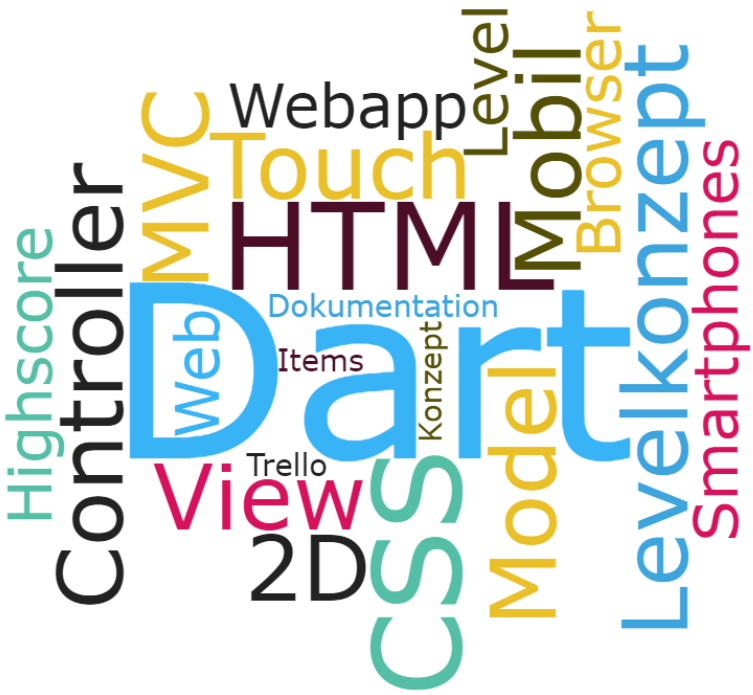
Webtechnologien Projekt

StarSpeeder Dokumentation



Niclas zum Felde
Tim-Pascal Lau
Léon Klick

Inhalt

1. Einleitung

1.1 Allgemein

Die folgende Dokumentation erläutert die Entwicklung des Spiels „StarSpeeder“ und gewährt tiefe Einblicke in verschiedene Aspekte, ausgehend von der Projektidee bis hin zur Umsetzung und Fertigstellung.

Es werden grundlegende Leitpunkte hinsichtlich Spielkonzept, Anforderungen, Architektur und Implementierung, sowie detailreiche konzeptionelle Entwicklungsaspekte erläutert.

Mit dem Ziel, diese möglichst nachvollziehbar und transparent darzustellen, wird von uns auf selbsterstellte Grafiken und selbst konzipierte Modelle zurückgegriffen, welche während der verschiedenen Entwicklungsphasen entstanden sind.

1.2 Ideenfindung

Es war uns von Anfang an sehr wichtig ein Spiel zu entwickeln, welches den Nutzer fordert, zugleich aber nicht überfordert.

Wir nahmen uns während der Findungsphase mehrere große Arcade-Klassiker wie zum Beispiel Tetris, Space Invaders, Frogger und Pacman vor, um diese zu analysieren und herauszufinden was sie so besonders macht, dass sie selbst heute noch von so vielen Menschen unterschiedlicher Personen- und Altersgruppen gespielt werden.

Allgemein lassen sich all diese Klassiker in unseren Augen auf wenige, jedoch bedeutsame und vor allem effektive Eigenschaften reduzieren:

Einfaches, oft stark abstraktes Setting

- Pacman: Spieler steuert eine Gelbe Kugel mit Gesicht und versucht Geistern zu entkommen, während er Punkte im Level sammelt
- Tetris: Kein konkretes Objekt mit dem sich der Spieler identifizieren kann, lediglich verschiedene Figuren/Formen werden möglichst vorteilhaft gestapelt
- Space Invaders: Spieler steuert Raumschiff und wehrt wiederum feindliche Raumschiffe mittels simplem Waffensystem ab
- Frogger: Spieler steuert einen Frosch über eine stark befahrende Straße und anschließend über einen Fluss mit verschiedenen Hindernissen um das Spielziel zu erreichen

Einfache Gameplay-Mechaniken

- Meist wenige und einfache Steuermöglichkeiten des Spielers (Horizontale und vertikale Bewegung/Schießen/Drehen/Ausweichen)
- Interaktion mit Umgebung (Sammeln von Gegenständen/Kollision mit Level Elementen)

Einfache aber konkrete Spielmotivation

- Einfaches Punktesystem (Ständig präsenter Punktestand/Einfache Punkte/Extrapunkte durch besonderes Geschick des Spielers)
- Vergleichbarkeit mit sich selbst oder anderen Personen (Verlangen der beste Spieler zu sein)

Weniger ist mehr

- Wenige, jedoch eindeutige Spiel- und Grafikelemente (Z.B. Tetris: Nur fünf verschiedene Spiel-Blöcke bei einem der berühmtesten Spiele der Welt seit über 30 Jahren)

Adhoc-Spielprinzip

- Keine komplizierten Hintergrundgeschichten (Jeder versteht den Kontext ohne Vorwissen)
- Kurze Spiel-Sessions möglich (Spiel dauert je nach Anspruch und Erfahrung wenige Sekunden bis Minuten)

1.3 Projektkoordination

Im Rahmen des Webtechnologien-Projekts „StarSpeeder“ hat das Team beschlossen, eine Projektkoordinierungssoftware zu verwenden. Einstimmig hat man sich hierbei auf die als Web- und Mobilapplikation verfügbare Software „Trello“ geeinigt, da die Teammitglieder bereits erste Erfahrungen mit „Trello“ sammeln konnten. Für die zentrale Verwaltung von Dokumenten und Dateien hat sich das Team für die Verwendung einer Google Drive-Freigabe entschieden, sodass ein zentrales Archiv für alle Teammitglieder vorhanden ist.

Des Weiteren wurde zu Anfang eine Soll-Timeline für Projektabgaben definiert, sodass diese Daten in Trello eingepflegt werden konnten.

In Trello selbst agieren wir nach einem Kanban-Board-Prinzip. Dabei haben wir verschiedene Karten (eine Karte = eine Aufgabe), welche von Mitgliedern in Kategorien (= Bearbeitungsstadien) „gepullt“ werden können. Karten können von jeder Person jederzeit erstellt werden. In folgende Kategorien/Bearbeitungsstadien können Karten „gepullt“ werden:

- **Ice-Box:** Hier befinden sich die Karten, welche noch bearbeitet werden müssen, es ist also der Anfangspunkt einer jeden Karte
- **In Progess:** Karten die hier einsortiert sind, werden zurzeit bearbeitet
- **Complete:** Hier werden Karten abgelegt, welche fertig bearbeitet sind (bedeutet, dass Aufgabe fertiggestellt ist)

Dabei schreibt die Person, die die Karte „pullt“, ihren Namen und den Bearbeitungsstatus, in welchen die Karte verschoben wird, als Kommentar in die Karte.

Dieses Prinzip fordert von Natur aus, dass jedes Teammitglied eine gewisse Arbeitsbereitschaft mit sich bringt. Da die Gruppenkonstellation bereits andere Projekte mit diesem Board erfolgreich bewältigt hat, kann guten Gewissens die Verantwortung an das komplette Projektteam übertragen werden statt diese durch eine zentrale Person zu verwalten.

Durch dieses Prinzip gelingt eine Selbstdokumentation und Verlaufsverfolgung, sodass bei möglichen Problemen direkt zu erkennen ist, welches Teammitglied für die Problemlösung anzusprechen ist.

Besonders zur Finalisierung des Spiels haben sich zwei weitere Kategorien als hilfreich herausgestellt:

- Bugs
- BugsComplete

Durch die Trennung von Bugs und regulären Aufgaben war es zunehmend einfacher, die Übersicht zu behalten.

1.4 Projektkommunikation

Damit eine reibungslose Kommunikation gelingt, verwenden wir ein Nachrichtenforum in Trello, welches alle Projektmitglieder bei neuen Nachrichten in Trello selbst und zusätzlich via Email benachrichtigt. Dieses Prinzip funktionierte schon in vorangegangenen Projekten, weshalb es auch hier beibehalten wird.

2. Anforderungen und abgeleitetes Spielkonzept

2.1 Anforderungen

Zusätzlich zu den bereits gegebenen Anforderungen haben wir uns zusätzlich eigene definiert, um so definierte und messbare Ziele für das finalisierte Spiel zu haben.

Im Folgenden sind die Anforderungen auf Funktionale Anforderungen, Dokumentationsanforderungen, Technische Randbedingungen und Designanforderungen aufgeteilt und die einzelnen Anforderungen erklärt.

Funktionale Anforderungen:

ID	Titel	Anforderung
AF-1	Singleplayer	Bei dem Spiel muss es sich um ein Einspielerkonzept handeln, Mehrspieleroptionen sind nicht vorgesehen.
AF-2	2D-Raster	Das Spielkonzept muss auf einem 2D-Raster basieren.
AF-3	Levelkonzept	Das Spiel sollte ein Levelkonzept vorsehen, wobei eine abschnittsweise Steigerung der Schwierigkeit vorhanden sein sollte
AF-4	Parametrisierungs-konzept	Das Spiel sollte ein Parametrisierungskonzept für relevante Spielparameter vorsehen
AF-3	Mobile Geräte	Das Spiel ist für einen mobilen Browser vorgesehen und muss auf einem Smartphone/Tablet spielbar sein.
AF-4	Desktop Geräte	Das Spiel ist in einem Desktop Browser spielbar
AF-5	Highscore speichern	Der Highscore eines Spielers kann auf dem Endgerät gespeichert und eingesehen werden.
AF-6	Spielmotivation	Es muss eine eindeutige Spielmotivation vorhanden sein
AF-7	Item „Invulnerable“ hinzufügen	Ein besonderes Item soll integriert werden. Diese neue Anforderung wurde im letzten Sprint bekannt

Dokumentationsanforderungen:

ID	Titel	Anforderung
DF-1	Projektdokumentation	Die Dokumentation des Projektes muss so gestaltet sein das eine projektfremde Person sich einlesen kann und das Projekt fortführen kann.
DF-2	Quellcodedokumentation	Der Quellcode muss mittels der schriftlichen Dokumentation erschließbar sein. Klassen und Methoden sind kommentiert, ggf. werden komplexere Algorithmen ebenfalls kommentiert.
DF-3	Libraries	Verwendete Libraries werden in der Dokumentation kenntlich gemacht und die Notwendigkeit ihres Einsatzes begründet.

Technische Randbedingungen:

ID	Titel	Anforderung
TF-1	Canvas	Die Verwendung von Darstellungstechniken die Canvas verwenden ist ausdrücklich untersagt. Die Darstellung des Spielfeldes darf nur über DOM-Tree Techniken erfolgen.
TF-2	Levelfomat	Die einzelnen Levels des Spiels sollen sich mittels einer JSON-

		Datei beschreiben lassen, so dass Änderungen an Levels ohne Änderung des Quellcodes möglich sind.
TF-3	Parameterformat	Spielparameter sollen mittels einer JSON-Datei beschrieben werden. Änderungen an Spielparametern sollen möglich sein ohne dass der Quellcode verändert werden muss.
TF-4	View	Es dürfen nur HTML und CSS zur Realisierung der View des Spiels verwendet werden.
TF-5	Spiellogik	Zur Realisierung der Logik des Spiels darf nur die Programmiersprache Dart verwendet werden.
TF-6	Browser	Das Spiel unterstützt in der JavaScript kompilierten sämtliche Browser mit einer JavaScript Engine. Es werden auch die Browser mit einer native Dart Engine unterstützt. Ferner werden die besonderen Rahmenbedingungen von Safari unter iOS 10 beachtet.
TF-7	MVC	Das Spiel folgt der MVC-Architektur.
TF-8	Highscore im Localstorage	Der Highscore soll lediglich (wenn verfügbar) im LocalStorage des Browsers gespeichert werden, es ist eine Online-Speicherung vorgesehen

Designanforderung:

ID	Titel	Anforderung
DS-1	Grafiken	Sämtliche im Spiel verwendete Grafiken und Bilder sind entweder selbst erstellt oder von ihrem Ersteller zur freien Verwendung lizenziert worden. Auf lizenzierte Grafiken und Bilder wird in der Dokumentation hingewiesen.

2.2 Spielkonzept

Bei der Entwicklung unseres Spiels priorisierten wir das Anstreben, des im letzten Kapitel erwähnten Arcade-Spielprinzips, um besonders auch für Gelegenheitsspieler einen möglichst einfachen Zugang zu Spiel-Elementen und Gameplay-Mechaniken zu bieten und so den Spaß- und Suchtfaktor in die Höhe zu treiben:

- Der Spieler steuert ein Raumschiff, den sogenannten „StarSpeeder“, durch das Weltall und sammelt vergoldete Ringe oder selten vorkommende vergoldete Sterne ein, während er entgegenkommenden Hindernissen ausweicht um möglichst lange zu überleben
    - Vergoldeter Ring: Der Spieler erhält einen Punkt
    - Vergoldeter Stern: Der Spieler erhält fünf Punkte
    - Asteroidenfeld: Der Spieler verliert einen Teil seiner Punkte
    - Großer Asteroid: Der Spieler geht GameOver
    - Multiplikator: Multipliziert alle von dem Spieler eingesammelten Punkte für kurze Zeit mit dem Faktor Zwei
    - SpeedDown: Erhöht die Spielgeschwindigkeit für kurze Zeit
    - SpeedUp: Senkt die Spielgeschwindigkeit für kurze Zeit
- Der Spieler bewegt sich auf einer fest definierten horizontalen Bahn
- Die Spielgeschwindigkeit nimmt von Level zu Level zu
- Die Anzahl der Bahnen, auf denen sich der Spieler bewegt nimmt von Level zu Level zu
- Ein ständig sichtbares Punktesystem zeigt dem Spieler seine aktuelle Leistung
- Der Spieler hat die Möglichkeiten Extrapunkte zu sammeln, sollte er selten vorkommende Items einsammeln
- Sollte der Spieler GameOver gehen, so kann er seinen Highscore speichern
- Ein neues Spiel beginnt immer in dem ersten Level

3 Architektur und Implementierung

3.1 Allgemein

Die zugrundeliegende Architektur verfolgt den Aufbau des MVC Architekturmusters, sodass eine Trennung von Datenmodell, Präsentation und Programmsteuerung stattfindet. Die Aufgaben der einzelnen Komponenten bezogen auf das von uns implementierte Spiel sind dabei wie folgt aufgeteilt:

- Model: Enthält Spiellogik (z.B. Item- und Playerbewegung, Item-Player-Kollisionen, …)
- Controller: Ist für Nutzerinteraktionen und Zeitsteuerung (=Spielschritte) verantwortlich
- View: Repräsentiert den Spielzustand des Models, hierbei durch Manipulation des DOM-Tree

Im Folgenden werden die einzelnen Komponenten unserer MVC-Implementierung beschrieben, sodass der Aufbau unseres Spiels verdeutlicht wird.

3.2 Model

Da das Model die höchste Komplexität aufweist, ist es hier logisch auf mehrere Komponenten aufgeteilt (Siehe Abbildung 1). Damit unsere Diagramme dem realen Aufbau des Projekts so nah wie möglich kommen, sind die jeweiligen Klassen der Abbildung innerhalb von Packages, welche die einzelnen Dart-Dateien im Projekt darstellen. Somit ist die Klasse *Game* in der *game.dart*-Datei in der Projektstruktur. Dadurch sind unsere Überlegungen im Zusammenhang mit dem Quellcode schneller nachzuvollziehen.

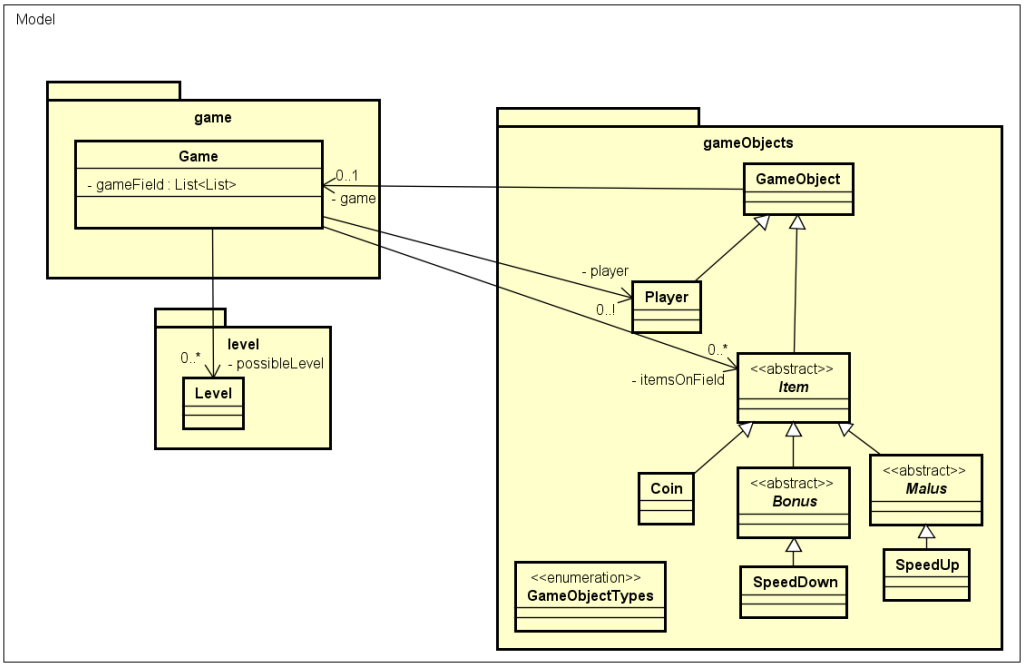


Abbildung 1 Model Klassendiagramm

3.2.1 Objekte auf dem Spielfeld

Auf einem Spielfeld existieren typischerweise verschiedene Objekte, welche meist ähnliche Attribute und Abhängigkeiten aufweisen. Somit lag der Gedanke nahe, eine Superklasse für Spiel-Objekte des Spielfelds zu erstellen, welche hier die Bezeichnung *GameObject* trägt. Ein *GameObject* hat dabei immer folgende Attribute:

- `int _row` — (Private) Aktuelle Position des GameObject als Zeilenangabe
- `int _col` — (Private) Aktuelle Position des GameObject als Spaltenangabe
- `Map<String, int> get position` — Aktuelle Position eines GameObject als Map, öffentlich verfügbar für andere Komponenten. Als Key wurde dabei ein String verwendet, da es ohne sprechende Bezeichner schnell zu Verwechslungen kommen kann (hier verwendet: „row“ und „col“ als Key)
- `GameModel _game` — (Private) Jedes GameObject kennt das dazugehörige Game. Grund hierfür liegt in der Logik der Items. Das Einsammeln eines Items vereinfacht oder erschwert das Spiel. Somit ist klar, dass ein GameObject mit dem GameModel kommunizieren sollte

Von der abstrakten Super-Klasse *GameObject* werden verschiedene konkretere Klassen abgeleitet. Diese Vererbungshierarchie ist in Abbildung 1 innerhalb des Packages gameObjects erkennbar. Auf der ersten Vererbungshierarchie abgehend von *GameObject* befinden sich *Player* und *Item*.

Die konkrete Klasse *Player* entspricht der Repräsentation des realen Spielers. Sie ist notwendig, um innerhalb der Spiellogik Kollisionen und Bewegungen zu realisieren und ist für das Spiel logischerweise essentiell.

Dem gegenüber steht die abstrakte Klasse *Item*. Items sind Objekte des Spielfelds, welche mit dem Player kollidieren und das daraufhin Spiel vereinfachen, erschweren oder dem Spieler Punkte verschaffen können. Eine Kollision der Items untereinander ist ausgeschlossen. Daher besitzt *Item* eine abstrakte Methode `void execute()`, welche die definierte Ausführung bei einer Kollision mit dem Player darstellt. Auf der zweiten Vererbungsebene abgehend von *GameObject* sind die Klassen *Coin*, *Malus* und *Bonus* zu finden.

Coin ist eine konkrete Klasse, welche die Methode `void execute()` implementiert und um ein Attribut `int _amountCoins` erweitert wurde, sodass der Highscore beim Aufruf von der Kollisionsmethode `void execute()` im *GameModel* dementsprechend angepasst werden kann. Beim Erstellen eines Coins muss demnach eine Anzahl der zu verwendenden Punkte angegeben werden.

Malus und *Bonus* sind im Gegensatz zur konkreten Klasse *Coin* abstrakte Klassen, welche eine Aufteilung der Weiteren Items vollziehen und so die Übersichtlichkeit und Erweiterbarkeit wahren. Wie die Bezeichner schon vermuten lassen, teilen diese beiden Klassen die spielvereinfachenden- und erschwerenden Items auf. Konkrete Implementierungen dieser abstrakten Klassen sind in Abbildung 1 beispielhaft zu erkennen.

Im Folgenden eine Übersicht, welche Items in das Spiel integriert sind und welche Aufgaben sie besitzen:

Item	Beschreibung	Besonderheit
Coin	Coins bringen beim Einsammeln Punkte, welche auf den aktuellen Punktestand addiert werden.	-
SpeedDown	Die Spielgeschwindigkeit wird für n Spielschritte verlangsamt	Solange aktiv bis n-ter Spielschritt oder neues Level erreicht wurde
Premium	Der Spieler erhält beim Einsammeln einige Extra-Punkte	-
Multiplicator	Eingesammelte Coins werden für n Spielschritte mit Konstante multipliziert	Gilt nur für Coins Solange aktiv bis n-ter Spielschritt oder neues Level erreicht wurde
Invulnerable	Malus-Items haben bei Kollision keinen Einfluss auf Spieler, gleichzeitig agiert ein Autopilot, sodass möglichst viele Premium und Coin Items eingesammelt werden	Solange aktiv bis n-ter Spielschritt oder neues Level erreicht wurde
SpeedUp	Die Spielgeschwindigkeit wird für eine Zeit t erhöht	Solange aktiv bis n-ter Spielschritt oder neues Level erreicht wurde
PointReduce	Der Spieler erhält bei Einsammeln einen gewissen Punkteverlust	-
Wall	Beendet das aktuelle Spiel, also GameOver	-

Um die View konzeptionell möglichst unabhängig von konkreten Itemobjekten auf dem Spielfeld zu halten, aber auch verfügbare Items in Leveln einfacher abbilden zu können, verwenden wir eine Enumeration, welche die möglichen Items als Typ vorhält:

```
enum GameObjectType {
	PLAYER,
	EMPTYCELL,
	WALL,
	COIN,
	POINTREDUCE,
	PREMIUM,
	SPEEDUP,
	SPEEDDOWN,
	MULTIPLICATOR,
	INVULNERABLE
}
```

Durch diesen Aufzählungstyp fällt es leichter, eine konzeptionelle Trennung zu konkreten Objekten des GameModels zu wahren.

3.2.2 Level-Klasse

Da ein Level viele Parameter besitzen kann, ist es hier sinnvoll diese zu kapseln. Hierfür dient die Klasse *Level*, wobei ein Levelobjekt ein einziges Level im Spiel darstellt. *Level* haben verschiedene Attribute, welche das Spielgeschehen und die Spieldarstellung beeinflussen können (Informationen über das Erstellen der Level folgen später im Abschnitt „Levelkonzept“).

Jedes Level besitzt verschiedene Attribute, welche im Folgenden tabellarisch vorgestellt werden. Angemerkt sei, dass jedes der vorgestellten Attribute als privat deklariert ist und für jedes Attribut Abfragemethoden definiert sind, sodass externe Komponenten auf die Informationen der Level zugreifen können.

- `int _levelnumber`	Levelnummer zur Identifikation des Levels
- `int _columns`	Anzahl der Spalten für das aktuelle Level. Mit einer höheren Spaltenanzahl wird auch das Spiel automatisch schwieriger
- `double _gamespeedMultiplicator`	Multiplikator für die Spielgeschwindigkeit. Im Regelfall sollte dieser im Intervall $(0,1]$ liegen
- `Map<GameObjectType, int> _itemDistribution`	Verteilung der Items basierend auf enum GameObjectType. Der Key ist eines der verfügbaren Items, wobei dessen Wert das Vorkommen dieses Items für das Level definiert.
- `bool get levelDone`	Zeigt, ob das aktuelle Level bereits fertig gespielt ist. True, wenn Prüfung auf Summe der verfügbaren Values aus `Map _itemDistribution` gleich 0 ist, sprich wenn kein Item mehr in der Map vertreten ist

Ein Level stellt zudem Operationen `GameObjectType getRandomItemType()` bereit, um ein zufälliges Item aus der Itemverteilung (`Map _itemDistribution`) zu entnehmen und zu retournieren, sowie eine weitere Operation `void reduceItemAmountByOne(GameObjectType type)`, um die Anzahl der Items eines bestimmten Typs der `_itemDistribution` um eins zu verringern.

`getRandomItemType()` ist prinzipiell eine einfache Möglichkeit, einen zufälligen Itemtyp aus der gegebenen Map der Itemverteilung zu bestimmen. Es wird zunächst die Summe s über alle verbleibenden Itemvorkommen der Map gebildet und auf Basis dieser Summe s eine (Pseudo)-Zufallszahl innerhalb des Intervalls $[0, s]$ generiert. Daraufhin müssen schlichtweg die Vorkommen für jedes Item der Map addiert und dabei geprüft werden, ob die generierte Zufallszahl im aktuell aufsummierten Bereich liegt. Da die Map im Regelfall nur so groß ist, wie die Anzahl der im Spiel zur Verfügung stehenden Items, ist diese Operation laufzeittechnisch (bisher) marginal und somit unproblematisch für jeden Spielschritt ausführbar.

3.2.3 GameModel-Klasse

Die Klasse *GameModel* (*game.dart*) repräsentiert die Hauptklasse des Models, wobei diese ebenfalls als Kommunikationsschnittstelle für Game*View* und Game*Controller* agiert. Da das Model jegliche Spiellogik beinhaltet und der Aufbau dementsprechend komplex ist, werden im Folgenden die Funktionalitäten des Models durch typische Teilaktionen während eines Spieldurchlaufs erklärt. Dadurch ist der Kontext der Methoden und Attribute verständlicher.

Game-Initialisierung

Beim Erstellen eines *GameModel* muss die initiale Größe des Spielfelds angegeben werden. Die Größe setzt sich dabei aus der Anzahl der anfänglichen Spalten und Zeilen zusammen. Diese beiden wichtigen Parameter müssen dementsprechend in einem Attribut gespeichert werden, welche im Quellcode als Integer mit der Bezeichnung `width` und `height` versehen und als *private* deklariert sind. Soll ein Spiel gestartet werden, so muss die Methode `newGame(List<Level> levelConcept)` aufgerufen werden. Hierdurch findet die Initialisierung des gesamten Models statt, wobei alle verfügbaren, zuvor definierten und eingelesenen *Level* in einer Liste übergeben werden müssen. Die Initialisierung des Levelkonzepts im Model geschieht in einer privaten Methode `initLevel(List<Level> levelConcept)`, dazu aber später mehr. Da ein Spiel typischerweise einen Spieler auf dem Spielfeld benötigt, wird bei der Initialisierung ebenfalls ein neues Player-Objekt erstellt. Nach der Initialisierung des Games wird der Status des Spiels zunächst auf „STOPPED“ gesetzt, sodass ein definierter Spielstatus existiert.

Spielstatus ermitteln

Der Spielstatus ist eine gute Möglichkeit, allen Komponenten des MVC-Musters einen einfachen Zugang zum aktuellen Zustand des Spiels zu ermöglichen. Der Status ist innerhalb des Models durch ein `enum GameState` realisiert, welches die Zustände `RUNNING`, `STOPPED` und `GAMEOVER` bereitstellt. Der jeweilige Status wird in einem privaten Attribut `GameState gameState` gekapselt, wobei Abfragemethoden bereitgestellt sind, welche die Abfrage eines spezifischen Zustands realisieren und diesen als boolschen Wert retournieren. Eine solche Abfragemethoden ist wie folgt aufgebaut (Aufbau äquivalent für `STOPPED` und `GAMEOVER`):

```
bool get running => _gameState == GameState.RUNNING
```

Dadurch können externe Komponenten einfach erfragen, in welchen Status sich das Model befindet und dementsprechend ihre eigenen Aktionen ausführen.

Initialisierung der Level

Das Levelprinzip im Model wird durch eine private Methode `initLevel(SplayTreeMap<int, Level> levelConcept)` initialisiert. Hierbei werden alle levelbasierenden Parameter in *GameModel* auf jene Werte gesetzt, welche das erste Level der übergebenen SplayTreeMap beinhaltet (zu levelbasierenden Parametern später mehr). Die Level müssen dabei in einer `SplayTreeMap` vorliegen, da so im default eine aufsteigende Sortierung (nach Levelnummern) vorliegt. Vorteil dabei ist, dass Fehler im Levelkonzept ausgeglichen werden können. Beispielsweise könnte beim Erstellen der Level inmitten der Aufzählung ein Level vergessen worden sein ([1, 2, __, 4, 5]). Dadurch, dass die `SplayTreeMap` jedoch aufsteigend nach Levelnummern sortiert ist, wird schlichtweg nach Level 2 das nächsthöhere Level verwendet. Dadurch wird das Levelkonzept toleranter gegenüber Fehlern.

Alle verfügbaren Level werden in einem privaten Attribut `SplayTreeMap<int, Level> possibleLevel` gespeichert, sodass diese für das gesamte Model verfügbar sind.

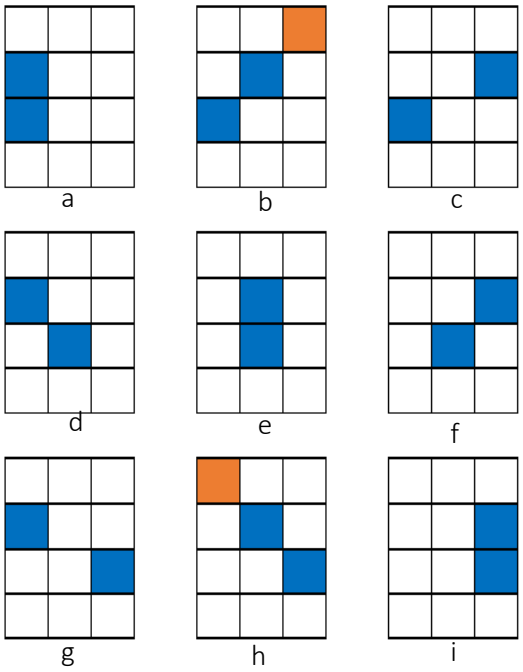


Abbildung 2 Item-Kombinationen bei drei Spalten

Erzeugen eines Items

Items werden mit jedem Spielschritt erzeugt - zumindest immer dann, wenn dem Spieler weiterhin eine faire Situation vorgelegt werden kann. Unfaire Situationen sind solche Situationen, in denen „Sackgassen“ von spielerschwerenden Items entstehen und der Spieler somit gezwungen ist, eines dieser Malus-Items einzusammeln.

Das Erzeugen eines Items ist auf vier Methoden aufgeteilt:

- `addItem()`: Entspricht der Hauptmethode der Item-Spawn-Logik
- `getNextItem()`: Ermittelt, welcher Itemtyp (aus GameObjectType-enum) als nächstes auf das Spielfeld gesetzt werden soll und retourniert ein konkretes Item
- `createNextItem()`: Erstellt das Item auf Basis des ermittelten Typs
- `generateItemColPosition()`: Ermittelt, in welcher Zeile ein Item gespawnt werden soll um unfaire Situationen zu verhindern

In Abbildung 2 sind auf Basis eines dreispaltigen Spielfelds alle möglichen Konstellationen für Itemplatzierungen angegeben (a-i). Dabei sind die Konstellationen b) und h) genau jene, die unfaire Situationen provozieren und müssen demnach vermieden werden.

Da unser Spielkonzept auch mehr als drei Spalten zulässt, muss dies in einem entsprechenden Algorithmus zur Vermeidung genannter Situationen berücksichtigt werden.

Um zu verhindern, dass es im Spiel zu solchen unfairen Situationen kommt, wird mittels eines Algorithmus das Spawnen von Items, welche einen spielerschwerenden Effekt haben, kontrolliert. Eine solche Situation tritt genau dann auf, wenn in jeder Zeile ein Malusobjekte genau linksversetzt oder rechtsversetzt unter einem Malusobjekt der darüber liegenden Zeile platziert ist.

Darum wird in der Methode *_generateItemColPosition(),* in welcher bestimmt wird in welcher Spalte des Spielfelds ein Item positioniert wird, eine besondere Logik für Malusobjekte ausgeführt. Items, welche spielerleichternd sind, müssen dabei nicht geprüft werden, schließlich ist mit solchen Items keine unfaire Situation provozierbar.

Um zu prüfen ob eine Sackgassen-Situation auftritt, wird angenommen, dass die Position des zu platzierenden Malusobjekts die des Items in der obersten Zeile ist. Dies ist möglich da beim Aufruf der Methode *_generateItemColPosition()* bereits alle Items ihre Bewegung abgeschlossen haben und die oberste Zeile frei ist. Diese Position wir in die Variable *lastItemPosition* gespeichert. Dort wird immer die Spalte des zuletzt gefundenen Items gesichert. Dann wird in einer Schleife das Spielfeld durchsucht. Ist eine Zelle nicht leer, wird geprüft, ob in der Zeile darüber direkt, links- oder rechtsversetzt ein Item liegt. Ist dies der Fall wird die Variable *seriesCount* um eins erhöht. Wenn *seriesCount* gleich der Anzahl an Spalten des Spielfeldes minus eins ist, gibt die Methode *_generateItemColPosition()* minus eins zurück, dies bedeutet das Item konnte nicht platziert werden. Sonst wird die Spaltennummer, an der das Item platziert werden soll, zurückgegeben. Im Listing 1 wird die prüfende Schleife gezeigt.

```
int lastItemPosition = possibleColPosition;
if(type == GameObjectType.WALL || type == GameObjectType.SPEEDUP || type ==
GameObjectType.POINTREDUCE) {
  for(int row = 1; row < _width; row++){
    for(int col = 0; col < _width; col++){
      if(_actualField[row][col] != GameObjectType.EMPTYCELL){
        if(col+1 == lastItemPosition || col-1 == lastItemPosition || col ==
lastItemPosition){
          if(++seriesCount >= _width-1){
            return -1;
          }
        }
        lastItemPosition = col;
      }
    }
  }
```

Listing 1: Item-Spawn Logik

Den Spieler bewegen

Um den Spieler im *GameModel* zu bewegen, wird die Methode `movePlayer(int directionX)` verwendet. Der übergebene Integer `directionX` stellt dabei die Bewegung in x-Richtung dar, sodass die Position des `Player`s um einen passenden Wert verändert werden kann. Ist der ausgerechnete Wert außerhalb der Spielfeldgrenzen, so wird die Position des `Player`s insofern angepasst, dass bei einem Wert < 0 eine Anpassung auf 0 und bei einem Wert größer der maximalen Spaltengröße n eine Anpassung auf $n - 1$ vollzogen wird. So ist sichergestellt, dass sich der Spieler nicht über das Spielfeld hinausbewegen kann.

Einen Spielschritt im GameModel ausführen

Um einen Spielschritt im Model auszuführen, muss die Methode `void nextGameStep()` ausgeführt werden.

Zunächst einige Informationen dazu, wie gekennzeichnet wird, ob ein neues Level gesetzt wurde. Hierfür wird ein Flag `bool _newLevel` verwendet. Hauptsächlich dient dieser dem Controller als Abfragepunkt um zu erkennen, ob ein neues Level geladen wurde. Dies ist aufgrund dessen notwendig, weil in bestimmten Leveln die Spielgeschwindigkeit angepasst wird. Da der Controller jedoch für die Zeitsteuerung verantwortlich ist, muss dieser davon erfahren, schließlich müssen seine Eigenschaften dementsprechend angepasst werden.

Der erste Schritt befasst sich mit der Prüfung, ob das aktuelle *Level* durchgespielt und nicht das letzte *Level* ist, um bei einer positiven Auswertung des Ausdrucks das neue Level zu setzen. Ist das letzte Level erreicht, soll dieses als endlose Schleife bis zum GameOver gespielt werden, daher die zusätzliche vorherige Prüfung, ob das letzte Level bereits erreicht ist.

Die erste Dynamik des Spieles tritt im Folgeschritt von `nextGameStep()` ein. Nun werden zum ersten Mal Positionsänderungen der Items auf dem Spielfeld vollzogen. Hierzu durchläuft die Methode `moveItems()` alle auf dem Feld befindlichen Items, welche in einer Liste `itemsOnField` gespeichert sind. Dabei wird bei allen Items die y-Koordinate um eins erhöht. Items, die aufgrund ihrer aktuellen Position nicht mehr auf dem Spielfeld sind, werden aus der Liste gelöscht.

Nachdem alle Items ihre neue Position eingenommen haben und die erste Zeile somit wieder Platz für ein neues Item besitzt, wird innerhalb von `nextGameStep()` nun die Methode `addItem()` ausgeführt. Da die Logik schon im vorigen Unterkapitel beschrieben wurde, folgt nun die Erklärung der Kollisionserkennung zwischen Item und Player, welche direkt nach dem Hinzufügen des neuen Items vollzogen wird. Hierbei wird die Liste der auf dem Feld vorhandenen Items `itemsOnField` durchlaufen und für jedes Item die Kollision mit dem Player geprüft. Natürlich wäre hier eine Optimierungsmöglichkeit, nur jene Elemente zu betrachten, die überhaupt für eine Kollision in Frage kommen – ein Item in der ersten Zeile wird im aktuellen Spielschritt wahrscheinlich keine Kollision mit dem Player erreichen. Jedoch haben wir uns gegen eine solche Betrachtung aus zwei Gründen entschieden:

1. Eine mögliche Anforderungsänderung im Zusammenhang mit der Position des Players auf dem Spielfeld (besonders bezogen auf seine y-Koordinate) hätte viele Überlegung zur Logik hinfällig gestalten können
2. Bei einer sinnvollen[1] Höhe h des Spielfeldes und der Bedingung, dass nur ein Item pro Zeile existiert, verschlechtert der Durchlauf der kompletten Liste (Größe der Liste also $\leq h$) die Laufzeit nur vernachlässigbar gering

Aufgrund dieser Aspekte haben wir uns gegen die (minimale) Optimierung und für die Flexibilität auf Anforderungsänderungen entschieden.

Diese genannten Schritte sind notwendig, um im *GameModel* einen Spielschritt durchzuführen. Grundsätzlich wäre man so bereits in der Lage, das grundlegende Spiel auf Basis des *Model*s auszuführen. Auf die Erklärung von kleineren Hilfsmethoden wurde hierbei verzichtet.

Aufgrund dessen betrachten wir im Folgenden den *Controller* genauer, um unter Andrem die Interaktionen zwischen *Model* und *Controller* zu analysieren.

[1] Mehr als 20 Zeilen sind auf derzeitigen Geräten für dieses Spiel weniger geeignet

3.3 Controller

3.3.1 Allgemein

Die Ablaufsteuerung des Spiels wird durch den Controller geregelt. Dieser kontrolliert dabei sowohl die Nutzerinteraktionen mit der View, das Laden der Level und reagiert auch auf zeitgesteuerte Ereignisse. Der Spieler verfügt dabei auf mobilen Geräten über die Möglichkeit das Raumschiff im Spiel durch das drücken des Bildschirms oder die Rotation seines Gerätes, entlang der z-Achse zu steuern. Auf Desktopbrowsern kann der Spieler das Raumschiff mittels der Cursortasten oder durch klicken in die linke oder rechte Bildschirmhälfte steuern.

3.3.2 Steuerung

Die Steuerung über die Cursortasten ist dabei die leichteste und wird lediglich mittels eines Event Listeners für KeyDown-Events realisiert. Dieser wird in Listing 1 dargestellt

```
window.onKeyDown.listen((KeyboardEvent ev) {
  switch (ev.keyCode) {

    case KeyCode.LEFT:
      if(_model.running)
        _handleMovePlayerTap(_MOVE_LEFT);
      break;

    case KeyCode.RIGHT:
      if(_model.running)
        _handleMovePlayerTap(_MOVE_RIGHT);
      break;

    case KeyCode.P:
      if(_model.stopped)
        _model.start();
      else if(_model.running)
        _model.stop();
      break;
  }});
```

Listing 2: Listener für Keyboardinput

Die Steuerung mittels klicken oder berühren der Bildschirmhälften der funktioniert ähnlich einfach. Es werden mittels HTML und CSS zwei transparente Steuerungsfelder wie in Abbildung 3 über die Seite gelegt.

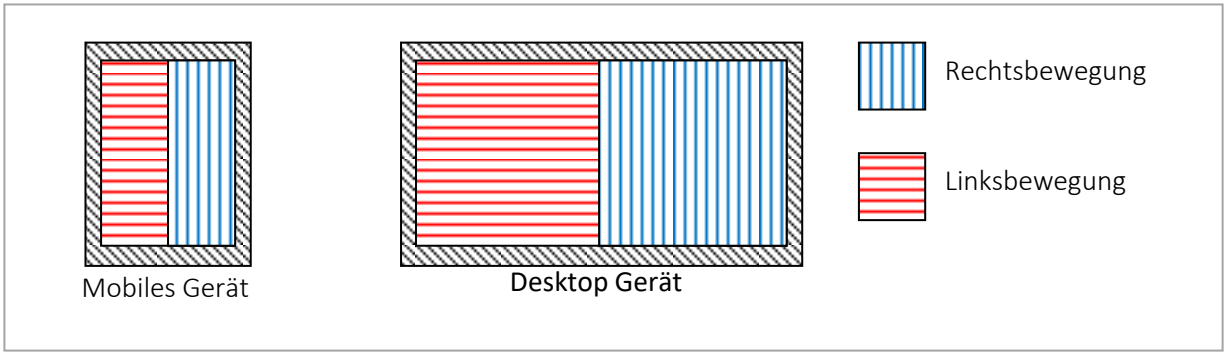


Abbildung 3: Tap-Steuerung

Die beiden Elemente werden dabei im HTML-Dokument mit den IDs LEFT und RIGHT gekennzeichnet. Im Controller wird dann mittels eines QuerySelectors auf das jeweilige Element ein onClick Event Listener angewendet. Diese werden im Listing 3 dargestellt.

```
querySelector('#left').onClick.listen((ev){
  if(_model.running && mobileControlType == MobileControlType.TOUCH)
    _handleMovePlayerTap(_MOVE_LEFT);
});

querySelector('#right').onClick.listen((ev){
  if(_model.running && mobileControlType == MobileControlType.TOUCH)
    _handleMovePlayerTap(_MOVE_RIGHT);
});
```

Listing 3: Listener für Maus- und Touchinput

Um Spielern auf mobilen Geräten noch eine weitere Möglichkeit der Steuerung zu bieten, verfügt das Spiel ebenfalls über eine Möglichkeit zur Steuerung über die Rotation entlang der z-Achse. Um zu verhindern, dass die Steuerung über Rotation oder Touchscreen gleichzeitig genutzt werden wird, in der globalen Variable `MobileControlType mobileControlType` des Controllers die vom Spieler gewünschte Steuerungsart gespeichert. Der Spieler kann dabei im Hauptmenü des Spiels die von ihm bevorzugte Steuerungsart selektieren. `MobileControlType` ist ein Enum, sollten also neue Steuerungskonzepte für mobile Geräte zum Projekt hinzugefügt werden, ist dies leicht erweiterbar.

Die Steuerung über Rotation entlang der z-Achse ist wesentlich komplexer als die anderen beiden Steuerungskonzepte. Es werden zusätzlich zum onDeviceRotation-Listener noch die Variablen `initRotationZ` und `_ROTATIONCONSTANT` benötigt. In `initRotationZ` wird die Position des Gerätes während des ersten Events gespeichert. Die `_ROTATIONCONSTANT` wird vom Controller genutzt um den unterschiedlichen Spalten, in denen sich das Raumschiff befinden kann, Grenzen zu zuweisen zwischen denen eine Rotation, abhängig von der initialen Position liegen muss, damit sich das Raumschiff in die Spalte bewegt. In Listing 3 sehen wird die Logik mit der geprüft wird, in welche Spalte das Raumschiff sich aufgrund einer Rotation bewegt.

Im Listener werden zusätzlich die Variablen `rotZ`, `upperLimit`, `lowerLimit`, `checkLimit` und `moveMade` deklariert. Wie weit und in welche Richtung das Gerät rotiert wurde, wird in `rotZ` gespeichert. Dazu wird die im Event festgehaltene aktuelle Position des Geräts minus der initialen Position, die in `initRotationZ` gespeichert ist, gerechnet. In `upperLimit` und `lowerLimit` wird die Ober- und die Untergrenze gespeichert die eine Rotation erreichen muss, um den Spieler z.B. in die Spalte 0 oder Breite des Spielfeldes-1 zu bewegen. Um diese Limits zu berechnen wird die Breite des Spielfeldes durch zwei geteilt und aufgerundet und dann mal die `_ROTATIONCONSTANT` genommen. Es wird beim Dividieren aufgerundet, damit bei einer ungeraden Breite des Spielfelds ein ungerades Ergebnis entsteht. Sonst kann für die mittlere Spalte keine Grenze mehr gefunden werden. Für die Bestimmung des `lowerLimit` wird das Ergebnis der Berechnung zusätzlich noch einmal mit -1 multipliziert. Durch die Variable `checkLimit` wird bestimmt, wie viele Spalten wir prüfen müssen. Zu der Spalte i kann auch immer die Spaltenbreite des Spielfeld-(1+i) überprüft werden, daher muss immer bei einer geraden Anzahl von Spalten nur die Hälfte aller Spalten geprüft werden. Bei einer ungeraden Anzahl von Spalten, muss zusätzlich die mittlere Spalte geprüft werden. In `moveMade` wird `true` gespeichert, sobald das Raumschiff bewegt wurde. Dies ist notwendig, da zuerst die äußeren Spalten geprüft werden.

Im Listing 4 wird dargestellt wie die prüfende Schleife arbeitet.

```
for(int col = 0; col <= checkLimit; col++){
  if(!moveMade) {
    if (upperLimit == _ROTATIONCONSTANT) {
      if (rotZ < upperLimit && rotZ > lowerLimit) {
        _handleMovePlayerRotation(rotZ, col);
      }
    } else {
      if (rotZ >= upperLimit || rotZ <= lowerLimit) {
        _handleMovePlayerRotation(rotZ, col);
        moveMade = true;
      }
    }
    upperLimit = upperLimit == _ROTATIONCONSTANT? _ROTATIONCONSTANT:
upperLimit - _ROTATIONCONSTANT;
    lowerLimit = -1 * upperLimit;
}
```

Listing 4: Prüfen in welche Spalte eine Rotation führt

Der in der If-Anweisung `if(upperLimit == _ROTATIONCONSTANT)` beschriebene Fall tritt dann auf, wenn das Spielfeld eine ungerade Anzahl von Spalten besitzt. In der Methode `_handleMovePlayerRotation()` wird dann entschieden, ob das Raumschiff in die linke oder rechte Spalte bewegt wird. Dazu wird geprüft, ob die Rotation positiv ist. Dies wird im Listing 5 gezeigt.

```
void _handleMovePlayerRotation(int rotation, int positionX) {
  if(rotation >= 0){
    _model.movePlayer(_model._width-(1+positionX));
  } else {
    _model.movePlayer(positionX);
  }
  _view.updatePlayer();
}
```

Listing 5: Spielerbewegung bei Rotation verarbeiten

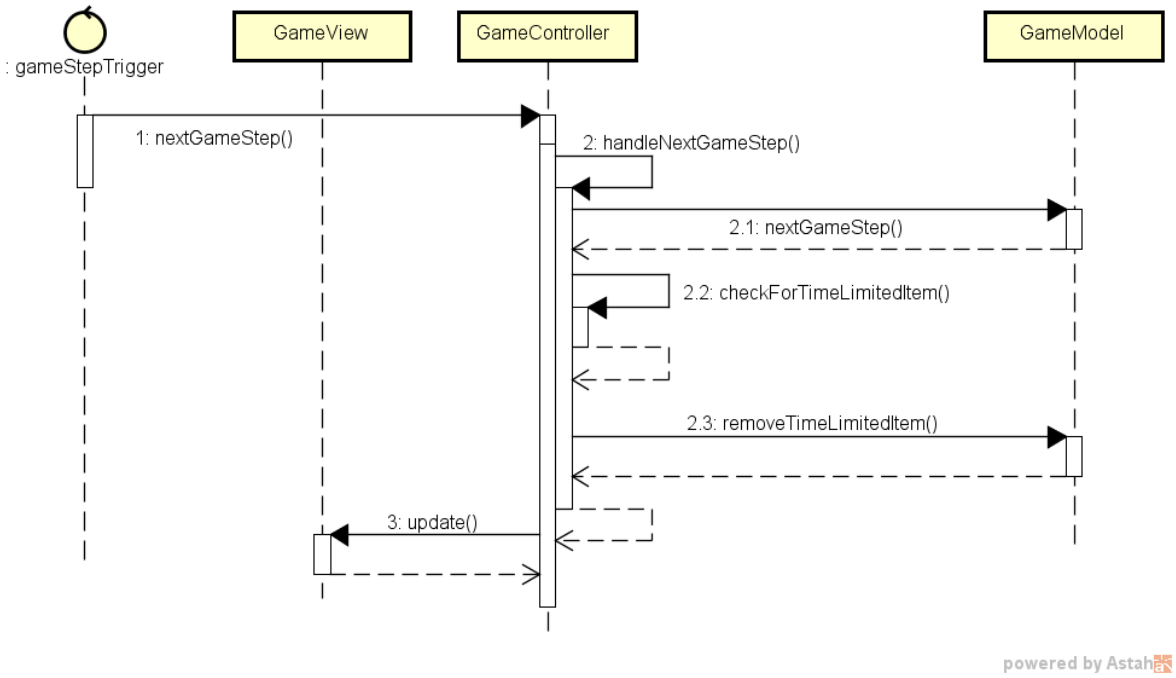


Abbildung 4: Sequenzdiagramm nextGameStep()

3.3.2 Laufendes Spiel

Über den `_gameStepTrigger` wird periodisch die Methode `_nextGameStep()` aufgerufen (siehe Abbildung 4. Bei der Initialisierung dieser Periode wird diese Zeit mit der Variable `_levelGameSpeedAcceleration` multipliziert. So wird das Spiel pro Level unterschiedlich schnell.

In der Methode `_nextGameStep()` wird überprüft, in welchem Zustand sich das Model befindet. Daraufhin wird die Methode `_handleNextGameStep()` aufgerufen. Innerhalb dieser wird zunächst das Model in den nächsten Schritt gesetzt. Dann wird geprüft, ob ein Item, welches sich auf die Geschwindigkeit oder den Multiplikator auswirkt, aktiv ist (genaueres im Abschnitt 3.4.2). Sollte ein Multiplikator aktiv sein, so prüft der Controller mittels `_multiplicatorStepsDone` und der Konstanten `_MAXSTEPSMULTIPLICATOR`, ob der Multiplikator die maximale Anzahl an für ihn defnierten Schritten erreich hat. Sollte dies der Fall sein, wird im Model die Variable `_multiplicateCoins` auf `false` gesetzt und `_multiplicatorStepsDone` auf 0 gesetzt. Sonst wird `_multiplicatorStepsDone` lediglich um eins erhöht. Zum Abschluss überprüft der Controller mit seiner Methode `_checkForCollectedItem()`, ob im Model ein Item mit limitierter Dauer gesammelt wurde und entfernt diese dann damit es im nächsten Spielschritt fälschlicherweise nicht wieder erkannt wird. Auf die Fälle *SpeedDown* und *SpeedUp* wird in Abschnitt 3.4.3 eingegangen. Für den Fall, dass ein Multiplikator eingesammelt wurde, wird `_multiplicatorStepsDone` auf eins und `_multiplicateCoins` im Model auf `true` gesetzt.

Wenn das Model über die Variable `newLevel` signalisiert, dass ein neuer Level geladen wurde, ruft der Controller zusätzlich die Methode `_handleNewLevel` auf. Dort werden dann die Zähler `_multiplicatorStepsDone` und `_speedStepsDone` auf 0 zurückgesetzt. Aus dem Model wird ein neuer Wert für die Variable `_levelGameSpeedAcceleration` geladen. In der View wird die Methode `_generateHtmlField()` aufgerufen, um die Tabelle auf dem HTML Dokument zu aktualisieren. Am Ende des Ablaufs wird über die Methode `_startNewGameTrigger()` ein neuer `_gameStepTrigger` gestartet.

Sollte sich das Model im Zustand `stopped` befinden, wird mit einer Vielzahl von onClick-Listenern die Steuerung durch die Menüs realisiert. Besonders interessant ist dabei das Öffnen der Highscoretabelle, da hierbei mittels der Methode `_getLocalStorage()` die gespeicherten Highscoreeinträge aus dem LocalStorage des Browsers lädt. Diese werden der View dann in der Methode `setTopPlayer()` als Map übergeben. Genauer wird auf diesen Speichermechanismus im Abschnitt 3.4.6 eingegangen.

Auf die Abläufe des Controllers, wenn sich das Model im Zustand `GAMEOVER` befindet, wird im Abschnitt 3.4.4 eingegangen.

Am Ende der Methode `_nextGameStep()` aktualisiert der Controller die View über deren Methode *update()*, so dass dem Spieler die aufgetretenen Veränderungen angezeigt werden.

3.3.3 Speedup und Speeddown

Da im Spiel Items enthalten sind die eine Manipulation der Spielgeschwindigkeit vornehmen, hält der Controller drei Konstanten `_NORMALGAMESPEED`, `_SPEEDUPGAMESPEED` und `_SPEEDDOWNGAMESPEED`, wobei diese Konstanten durch die Parametrisierung initialisiert werden (Siehe Abschnitt „Level- und Parametrisierungskonzept“). Für das Auslösen des nächsten Spielschritts ist der `_gameStepTrigger` des Controllers verantwortlich. Dieser wird mittels der Methode `_startNewGameTrigger()`, die im Listing 6 dargestellt wird, gestartet.

```
void _startNewGameTrigger(Duration newSpeed) {
  if(_gameStepTrigger != null && _gameStepTrigger.isActive)
  _gameStepTrigger.cancel();
  _gameStepTrigger = new Timer.periodic(newSpeed *
_levelGameSpeedAcceleration, (t) => _nextGameStep());
}
```

Listing 6: Starten des Triggers

Werden Items eingesammelt die nur für eine bestimmte Dauer aktiv sein sollen, dann wird dies in der Methode `_checkForCollectedItem()`, die schon im Abschnitt 3.4.2 erwähnt wurde, festgestellt.

Gleichgültig ob es sich um ein Item vom Typ *SpeedUp* oder *SpeedDown* handelt, wird dann die Variable `_speedStepsDone` des Controllers auf eins gesetzt. Zusätzlich wird mittels der Methode `_startNewGameTrigger()` der `_gameStepTimer` neugestartet. Dabei wird der Methode als Argument, abhängig davon welches Item eingesammelt wurde, entweder `_speeddownGameSpeed` *oder* `_speedupGameSpeed` übergeben.

In der ebenfalls schon in Abschnitt 3.4.2 erwähnten Methode `_handleNextGameStep()` überprüft der Controller, ob Items die eine limitierte Dauer besitzen, abgelaufen sind. Dazu wird geprüft, ob momentan ein *SpeedUp* oder SPEEDDOWN Item aktiv ist. Wenn dies der Fall ist, wird mit Hilfe der Konstante `_MAXSTEPSSPEED` und `_speedStepsDone` überprüft, ob das Item abgelaufen ist. Ist das nicht der Fall, wird `_speedStepsDone` um eins erhöht.

Wenn das Item abgelaufen ist, wird `_speedStepsDone` auf 0 gesetzt und die Methode `_startNewGameTrigger()` wird aufgerufen und dabei als Argument `_normalGameSpeed` übergeben. So kehrt das Spiel wieder zu seiner normalen Geschwindigkeit zurück.

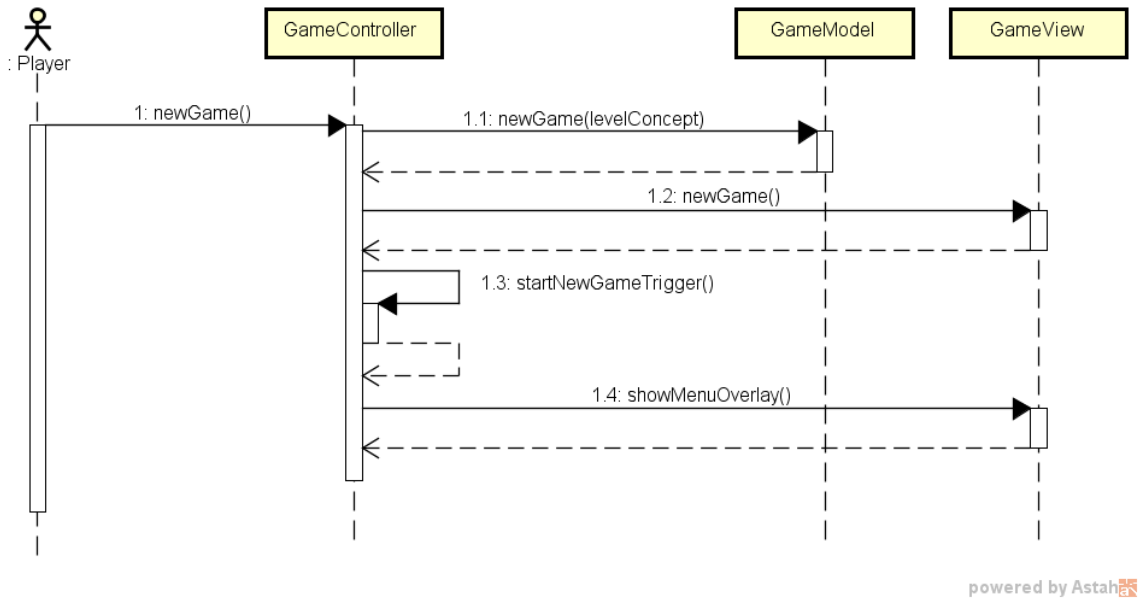


Abbildung 5: Sequenzdiagramm newGame()

3.3.5 Gameover

Sollte sich das Model am Abschluss der Methode `_nextGameStep()` des Controllers im Zustand `GAMEOVER` befinden, wird die Methode `_handleGameOver()` aufgerufen.

In der Methode werden zuerst die Items mit limitierter Dauer zurückgesetzt, dies geschieht über die die Methode `_resetItemsWithDuration()`. Wenn der `_gameStepTimer` noch aktiv ist, wird dieser abgebrochen.

Dann wird mit zwei onClick-Listenern auf Events auf den beiden im GameOver Overlay gezeigten Buttons gelauscht. Drückt der Spieler den Button „Skip“, wird vom Controller die Methode `_newGame()` (siehe Abbildung 5) aufgerufen in der das Model, die View und Parameter des Controllers auf ein neues Spiel vorbereitet.

Wenn der Spieler sich dazu entscheidet den Button „Save“ zu drücken, wird mittels der Methode `showSaveOverlay()` View dazu gebracht, das Speichermenü zu zeigen. Der Controller ruft dann die Methode `_saveHighscore()` auf. Dort wird mit zwei onClick-Listenern auf Events reagiert, die auf den beiden angezeigten Buttons geschehen. Drückt der Spieler den Cancel-Button, wird die oben beschriebene Methode `_newGame()` ausgelöst.

Drückt der Spieler den Button „Save“ wird über die View der in das InputField geschriebene Name in eine Variable `name` gespeichert. Diese Variable wird dann der Methode `_addToLocalStorage()` als Argument übergeben. Der konkrete Ablauf des Speicherns wird im Abschnitt 3.4.6 erläutert. Zum Abschluss wird ebenfalls die Methode `_newGame()` ausgelöst, die den Spieler in das Hauptmenü zurückführt.

3.3.6 Local Storage

Um es dem Spieler zu ermöglichen seinen Highscore lokal zu speichern, werden im Controller die zwei Methoden `_addToLocalStorage()` und `_getLocalStorage()` realisiert. Zusätzlich zu den Methoden hält der Controller ebenfalls die Konstante `GAMEIDENT` deren Wert der String ".`STRSPDR`" ist. Damit werden Einträge, die zum Spiel StarSpeeder gehören, im LocalStorage gekennzeichnet. Außerdem verfügt der Controller über eine Map `_savedLocalStorage`, deren Keys die Namen der Spieler sind und deren Values die zum Spieler zugehörigen Highscores repräsentieren.

Die Methode `_getLocalStorage()` durchsucht in einer Schleife den LocalStorage, sobald einer der Strings den *GAMEIDENT* beinhaltet, wird dieser Eintrag in der Map `_savedLocalStorage` platziert. Aus dieser Map werden die Einträge dann noch einmal in einer Schleife in eine sortierte TreeMap `sortedMap` überführt. Diese wird der View in deren Methode `setTopPlayer()` als Argument übergeben. Die Methode wird im Listing 6 dargestellt.

```
void _getLocalStorage() {
  _savedLocalStorage = new Map<String, int>();
  window.localStorage.forEach((k, v) {
    if(k.contains(GAMEIDENT)) {
      _savedLocalStorage.putIfAbsent(k, () => int.parse(v));
    }
  });
  SplayTreeMap<int, String> sortedMap = new SplayTreeMap<int, String>();
  _savedLocalStorage.forEach((k, v) => sortedMap.putIfAbsent(v, () =>
k));
  _view.setTopPlayer(sortedMap);
}
```

Listing 6: Laden aus dem Local Storage

In der Methode `_addToLocalStorage()` wird durch den Aufruf von `_getLocalStorage()` zunächst sichergestellt, dass `_savedLocalStorage` auf dem aktuellsten Stand ist. Wenn die als Argument an `_addToLocalStorage()` übergebene Variable `name` einen leeren String hält, wird nicht mit dem speichern fortgefahren. Ist dies nicht der Fall, wird mithilfe von `GAMEIDENT` geprüft, ob es zu dem Namen bereits einen Eintrag im Local Storage gibt. Gibt es keinen Eintrag, wird ein neuer angelegt und auf der View ein Dialog geöffnet der anzeigt, dass gespeichert wurde. Sollte es bereits einen Eintrag für den Namen gibt, wird geprüft ob der gespeicherte Highscore kleiner ist als der aktuelle. Trifft dies zu, wird der alte Highscore überschrieben. Wenn nicht, bleibt der alte Eintrag bestehen. Die Methode wird im Listing 6 dargestellt.

```
void _addToLocalStorage(String name) {
  _getLocalStorage();
  if(name != "") {
    name = GAMEIDENT + name;
    if(_savedLocalStorage.containsKey(name)) {
      int actValue = int.parse(window.localStorage[name]);
      if(_model.highscore > actValue) {
        window.localStorage[name] = '${_model.highscore}';
        this._view.showAlert("highscore saved");
      }
    }
    else {
      window.localStorage.putIfAbsent(name, () => '${_model.highscore}');
      this._view.showAlert("highscore saved");
    }
  }
}
```

Listing 7: Speichern im Local Storage

3.4.7 Anpassungen für die Bonusaufgabe

Um das gewünschte Item Autopilot zu realisieren wurden am Controller einige Änderungen vorgenommen. In den Methoden `_handleMovePlayerTap()` und `_handleMovePlayerRotation()` wird geprüft, ob sicher das Model sich im Zustand *invulnerable* befindet. Ist dies der Fall, wird nicht auf den Steuerinput des Spielers reagiert. Zusätzlich verfügt der Controller nun über die Konstante `_MAXSTEPSINVULNERABLE` und den Zähler `_invulnerableStepsDone`. Mit diesen wird - wie im Abschnitt 3.3.2 für das Multiplikator-Item und im Abschnitt 3.3.3 für *SpeedDown*- und *SpeedUp*-Items in der Methode `_handleNextGameStep()`- geprüft, ob das Item noch aktiv ist. Solange `_invulnerableStepsDone` dabei kleiner ist als `_MAXSTEPSINVULNERABLE` wird auf der View signalisiert, dass der Autopilot aktiv ist. Sollte die verbleibende Dauer des Items genau so groß sein wie der Index der Reihe des Raumschiffes wird im Model der Itemspawn deaktiviert. So wird dem Spieler Zeit gegeben die Steuerung wieder zu übernehmen. Ist `_invulnerableStepsDone` genau so groß wie `_MAXSTEPSINVULNERABLE` wird die View angewiesen aufzuhören das Feedback Overlay für den Autopiloten anzuzeigen. Dem Model wird über die zwei Flags `_invulnerable` und `skipAddItem` mitgeteilt wieder zum normalen Verhalten zurückzukehren und `_invulnerableStepsDone` wird dabei auf 0 zurückgesetzt. Das Einsammeln des Autopilot-Items wird analog um Multiplikator-, SpeedUp- und SpeedDown-Item in der Methode `_checkForCollectedItem()` verarbeitet. Dazu wird `_invulnerableStepsDone` auf eins gesetzt und die Variable `_invulnerable` des Models auf `true`.

3.4 View

Die Darstellung des Spiels und damit inbegriffen die visuelle Synchronisation mit dem aktuellen Zustand des Models ist die Aufgabe der View. Hier werden durch Nutzerinteraktion, wie zum Beispiel das Bewegen des Spielers, verschiedene Operationen ausgelöst, welche sich durch das Verändern der Benutzeroberfläche bemerkbar machen.

3.4.1 Synchronisation und Darstellung

Das Spielfeld wird zur optischen Wahrnehmung für den Nutzer hierbei durch eine HTML Tabelle repräsentiert. Diese Tabelle muss in der View ständig geändert werden, da sich der Zustand des Spielfelds während eines Spiels ununterbrochen ändert.

Um dies zu ermöglichen haben wir diverse Methoden implementiert, welche im Folgenden erläutert werden:

```
void update() {
  if(_model.running) {
    updateCompleteGameField();
    updateHighscore();
    updateLevel();
  }
}
```

Listing 8: update im Model

Diese Update-Methode, welche vom Controller in regelmäßigen Abständen aufgerufen wird, sorgt für das kontinuierliche Aktualisieren der visuellen Spielrepräsentation, hierbei werden verschiedene weitere Methoden aufgerufen, welche auf den folgenden Seiten erklärt werden.

```
void _generateHtmlField() {
  final field = _model.field;
  String table = "";
  for(int col = 0; col < field.length; col++) {
    table += "<tr>";
    for(int row = 0; row < field[0].length; row++) {
      final tdPos = "field_${col}_${row}";
      final tdClass = _getTdClass(field[col][row]);
      table += "<td id='$tdPos' class='$tdClass'></td>";
    }
    table += "</tr>";
  }
  _gameField.setInnerHtml(table);
  _saveTdElements(field);
}
```

Listing 9: Spielfeld generieren

Sobald eine neue View erstellt wird, wird diese zunächst initialisiert. Die Initialisierung findet dadurch statt, dass eine Methode `_generateHtmlField()` aufgerufen und innerhalb dieser HTML Code zum Erstellen einer gewöhnlichen Tabelle generiert wird. Dabei wird eine Zeichenkette für jede Reihe

durch <tr> (HTML-Tag für eine Reihe) und für jede Zelle durch <td> (HTML-Tag für eine Zelle) mit bestimmten Eigenschaften erweitert. So entsteht eine HTML-Tabellen-Struktur, welche unser Spielfeld darstellt.

Da das Model der View bekannt ist, kann diese direkt auf den internen Spielzustand lesend zugreifen und die Informationen des Spielfelds in eine formatierte tabellarische Darstellung bringen.

Die verschiedenen Zellen bekommen das Attribut `id='$tdPos'`, welches der Position der aktuellen Zelle im Model entspricht. Dies ist ein wichtiger Schritt um die Zellen der HTML-Tabelle im weiteren Verlauf ansprechen zu können.

Darüber hinaus wird noch das Attribut `class='$tdClass'` gesetzt, welche im anfänglichen Zustand immer EMPTYCELL entsprechen wird, da das Feld in der initialisierungsfase noch vollkommen leer ist.

```
void _saveTdElements(final field) {
  _htmlFields = new List<List<HtmlElement>>(field.length);
  for (int row = 0; row < field.length; row++) {
    _htmlFields[row] = [];
    for (int col = 0; col < field[row].length; col++) {

_htmlFields[row].add(_gameField.querySelector("#field_${row}_${col}"));
    }
  }
}
```

Listing 10: TD-Elemente speichern

Da das ständige Erstellen und Nutzen vom „querySelector“-Operator sehr aufwändig hinsichtlich der Rechenzeit wäre, wird einmalig für jedes Feld jeweils ein „querySelector“ erstellt, über welche man später unter anderem in der Update-Methode `updateCompleteGameField()` Änderungen an der HTML Repräsentation vornehmen kann (Dazu im Folgenden mehr).

```
void updateCompleteGameField() {
  final field = _model.field;
  for (int row = 0; row < field.length; row++) {
    for (int col = 0; col < field[col].length; col++) {
      final td = _htmlFields[row][col];
      if (td != null) {
        td.classes.clear();
        td.classes.add(_getTdClass(field[row][col]));
      }
    }
  }
}
```

Listing 11: Gesamtes Spielfeld aktualiseren

Die bereits angesprochene Methode zur visuellen Aktualisierung des Spielfelds ändert lediglich die Klasse der jeweiligen Zelle der HTML-Tabelle über die angesprochenen und vorbereiteten „querySelector“-Operatoren. So ist ein gewisser Performancevorteil vorhanden.

Dies geschieht, indem die Klasse der aktuellen Zelle der HTML-Tabelle so angepasst wird, dass sie mit der passenden Zelle des Spielfelds im Model übereinstimmt.
Daraufhin ändert sich für den Benutzer die Darstellung des aktuellen Spielfelds, wodurch im Spielfluss das Gefühl für den Spielenden entsteht, dass sich die Items auf ihn zu bewegen.

Die benötigte Klasse wird mit Hilfe der Methode `_getTdClass()` ermittelt, indem ihr die Zelle des Models an der Stelle `field[row][col]` übergeben und in einer Switch-Case auf deren Klasse geprüft wird.

```
String _getTdClass(var cell) {
  switch(cell) {
    case GameObjectType.EMPTYCELL:
      return 'empty';
    case GameObjectType.PLAYER:
      return 'player';
    case GameObjectType.WALL:
      return 'wall';
    case GameObjectType.COIN:
      return 'coin';
    case GameObjectType.POINTREDUCE:
      return 'pointreduce';
    case GameObjectType.PREMIUM:
      return 'premium';
    case GameObjectType.SPEEDUP:
      return 'speedup';
    case GameObjectType.SPEEDDOWN:
      return 'speedown';
    case GameObjectType.MULTIPLICATOR:
      return 'multiplikator';
  }
  return "";
}
```

Listing 12: Mapping für GameObjectTypes

Die jeweiligen retournierten Strings müssen dabei mit den definierten Klassen der CSS-Datei übereinstimmen und sind somit vordefiniert.

```
void updatePlayer() {
  int rowPlayer = _model._player.position['row'];
  final field = _model.field;
  for(int col = 0; col < field[rowPlayer].length; col++) {
    final td = _htmlFields[rowPlayer][col];
    if (td != null) {
      td.classes.clear();
      td.classes.add(_getTdClass(field[rowPlayer][col]));
    }
  }
}
```

Listing 13: Nur Player-Update

Um eventuell auftretenden Performance-Problemen bei weniger gut bezüglich der kompletten Aktualisierung bei jedem Spielschritt und jeder Player-Bewegung entgegenzuwirken, wurde eine weitere Methode integriert, welche nur die Spielerfigur betrachtet und diese separat aktualisiert. Dazu wird die in Listing 13 verwendete Methode `updatePlayer()` verwendet.

Soll bei einem Spielschritt das komplette Spielfeld aktualisiert werden, so wird hingegen die Methode `updateCompleteGameField()` aufgerufen.

```
void updateLevel() {
  int levelNumber;
  if(_model.actualLevel == null)
    levelNumber = 1;
  else
    levelNumber = _model.actualLevel.level;
  this._level.setInnerHtml("<h5>LEVEL <br>$levelNumber</h5>");
}
```

```
void updateHighscore() {
  int highscore = _model.highscore;
  _score.setInnerHtml("$highscore");
}
```

Die beiden Methoden `updateLevel()` und `updateHighscore()` aktualisieren lediglich die beiden Statusinformationen des laufenden Spiels. Explizit geht es hier zum einen um den derzeitig erreichten Highscore des Spielers, welcher sich bei jedem eingesammelten Ring, Stern oder Asteroidenfeld ändert und einer Levelbezeichnung, welcher einer ganzen Zahl entspricht und repräsentiert wie weit es der Spieler zum aktuellen Zeitpunkt im Spiel geschafft hat.

3.3.2 Die Navigationsmöglichkeiten

Die Verwendung des „querySelector“-Operators, welcher von der Dart Api zur Verfügung gestellt wird, ermöglicht uns das bilden der Schnittstelle zwischen Programm-Code und den Elementen des Dom-Tree.

Auf diese Weise ist es uns möglich, die visuelle Darstellung des Spiels für den Nutzer, dem Spielzustand entsprechend, dynamisch zu verändern und so eine lebendige Oberfläche mit Nutzerinteraktionsreaktionen zu kreieren.

Bei der Darstellung von verschiedenen Spiel- und Menübereichen, nutzen wir verschiedene konstruierte Overlays und Elemente, welche diese repräsentieren und im Nachfolgenden kurz beschrieben werden:

Der Ladebildschirm

```
final _loadingOverlay     = querySelector("#loadingOverlay");
```

Wird zu Beginn angezeigt, bis alle Inhalte des Spiels eingelesen und geladen wurden.
Sollte dies erfolgt sein, so verschwindet der Ladebildschirm und der Spieler befindet sich im Menü.

Das Menü

final _menuOverlay = querySelector("**#menuOverlay**");
Stellt das Navigationsmenü des Spiels dar und dient darüber hinaus auch als Pausebildschirm, bei unterbrechung des Spiels durch den Spieler.
Der Bildschirm enthält die Navigationsmöglichkeiten

Play	-	Startet das Spiel
Description	-	Öffnet das _manualOverlay
Gameplay	-	Öffnet das _gameplayOverlay
Highscore	-	Öffnet das _highscoreOverlay

Um das Verhalten der verschiedenen Overlay Sektionen zu managen und explorative Feedbackmöglichkeiten zu bieten, sind verschiedene Methoden geschrieben worden, welche lediglich als Werkzeug dienen, das erscheinen und verschwinden von den in CSS designten Overlays umzusetzen. Da diese Methoden immer nach dem gleichen Schema aufgebaut sind, wird an dieser Stelle lediglich eine Variante gezeigt:

```
void showMenuOverlay() {
  this._menuOverlay.style.setProperty("visibility", "visible");
`
```

```
void closeMenuOverlay() {
  this._menuOverlay.style.setProperty("visibility", "hidden");
`
```

Listing 14: Overlays öffnen und schließen

Die Itembeschreibungen

final _manualOverlay = querySelector("**#manualOverlay**");
Enthält kurze Informationen zu den verschiedenen Items des Spiels, um dem Spieler die Regeln bewusst zu machen.

Die Gameplayeinstellungen

final _gameplayOverlay = querySelector("**#gameplayOverlay**");
Stellt Optionen für den Spieler bereit, die Steuermöglichkeiten des Spiels mittels einer Checkbox zu wählen.

- **final _gameplayImg** = querySelector("**#gameplayExplanation**");
  Darüber hinaus, zeigt sich je nach Wahl der Option eine dementsprechende einfache animierte Grafik, welche als visuelles Feedback, auf die Auswahl des Spielers fungiert und die ausgewählte Optionen visuell darstellt.

Der Highscorebereich

final _highscoreverlay = querySelector("**#highscoreOverlay**");
Zeigt dem Spieler die bisherigen erreichten Highscores.

- **final topthree** = querySelector("**#topThreeScores**");

Enthält eine geordnete Aufzählung der besten 3 Ergebnisse, wobei der höchste Highscore an der Spitze steht.

Der Creditsbereich

```
final _creditsOverlay      = querySelector("#creditsOverlay");
```

Enthält Informationen zu den Mitwirkenden und Einflussgebern.

Das Spielfeld

```
final _game       = querySelector("#game");
```

Stellt das gesamte Spielfeld dar, inklusive der Statusleiste, welche Informationen über die Level- und Punktestand des Spielers enthält und darüber hinaus auch den Menü-Button um das Spiel zu pausieren.

```
final _gameField = querySelector("#gameField");
```

Stellt ausschließlich den Teil dar, in dem das eigentliche Spiel stattfindet.

```
final _score      = querySelector("#scoreField");
```

Enthält die Informationen über den aktuellen Punktestand des Spielers.

```
final _level = querySelector("#level");
```

Enthält Informationen über den aktuellen Levelstand des Spielers.

Der Gameoverbildschirm

```
final _gameoveroverlay    = querySelector("#gameoveroverlay");
```

Dieser Bildschirm erscheint, nach dem der Spieler das Spiel verloren hat. Der Spieler hat nun Navigationsmöglichkeiten, entweder seinen Highscore zu speichern oder diesen Schritt zu überspringen.

- ```
  final _saveOverlay = querySelector("#saveoverlay");
  ```
  Dieser Bildschirm erscheint als Reaktion auf die Auswahl des Spielers, den Highscore zu speichern und bietet dem Spieler die Möglichkeit seinen Namen anzugeben und zu bestätigen.
  Der Nutzer hat weiterhin die Möglichkeit über einen Button die Aktion abzubrechen um dann wieder im Menü zu landen.

  ```
  InputElement _saveInput     = querySelector("#textFieldSave");
  ```
  Eingabefeld, in welches der Spieler seinen Namen einträgt.
  ```

4 Level- und Parametrisierungskonzept

4.1 Allgemein

Um die Initiierung des Spiels durch eine externe Komponente vorzunehmen und Level-Prinzipien austauschbar zu gestalten, haben wir uns für eine Parametrisierung durch JSON-Dateien entschieden. Dies hat entscheidende Vorteile, wobei besonders die zentralisierte Änderungsmöglichkeit für das Projekt im Vordergrund stand. Dabei fand eine Trennung von Level- und Spielparametern statt, sodass beide Konzepte auf zwei unterschiedliche JSON-Dateien aufgeteilt wurden (*level.json*, *config.json*).

4.2 Levelkonzept

Im Kapitel „Architektur und Implementierung“ wurde die Klasse *Level* bereits genauer behandelt, jedoch fand noch keine Beschreibung statt, von welcher Komponente diese Level befüllt und erstellt werden, dies soll nun folgen.

Da ein *Level*-Objekt bestimmte Attribute besitzt, müssen diese von anderen Komponenten bereitgestellt und befüllt werden. Das Bereitstellen der Daten geschieht dabei durch die JSON-Datei *level.json*, das Erstellen und Befüllen der *Level* wird dabei vom *Controller* vollzogen.

Die JSON-Datei der Level ist dabei aus einer Liste von JSON-Objekten (=einzelne Level) aufgebaut, welche jene Informationen bereitstellen, die die Attribute der Level-Klasse benötigen.

```
[
   {
      "level": 1,
      "columns": 3,
      "gamespeedMultiplicator": 1.0,
      "itemDistribution": {
         "coin": 10,
         "wall": 10,
         "premium": 4,
         "pointreduce": 10,
         "speedup": 6,
         "speeddown": 0,
         "multiplicator": 3
         "invulnerable": 0
      }
   },

   {
      "level": 2,
      "columns": 4,
      "gamespeedMultiplicator": 0.7,
      "itemDistribution": {
         "coin": 15,
         "wall": 20,
         "premium": 5,
         "pointreduce": 6,
         "speedup": 5,
         "speeddown": 1,
         "multiplicator": 3
         "invulnerable": 1
      }
   }
]
```

Man erkennt hierbei, dass ein Level eine Levelnummer (**`level`**), eine Spaltenanzahl (**`columns`**), einen Multiplikator für die Spielgeschwindigkeit (**`gamespeedMultiplicator`**) und ein JSON-Objekt für die

Verteilung der Items für das angegebene Level (**`itemDistribution`**) besitzt. Die Itemverteilung besteht wiederrum aus einzelnen Parametern, welche für jedes Item das Vorkommen für das gegebene Level aufweisen. So wird in Level 1 zehn Mal das Coin-Item, zehn Mal das Wall-Item, usw. verwendet. Es kann somit durchaus simpel gesteuert werden, welche Items in welchen Leveln wie oft auftreten werden.

Eine steigende Schwierigkeit kann durch Anpassen der Parameter **`columns`**, **`gamespeedMultiplicator`** und der Änderung der Itemvorkommen in **`itemDistibution`** erreicht werden. Dabei sollten die angegebenen Werte logischerweise sinnvoll eingetragen werden. Eine steigende Schwierigkeit wird nur erreicht, wenn die Spaltenanzahl größer[2] und der Multiplikator kleiner[3] wird. Durch das Erhöhen der Vorkommen von Malus-Objekten (Speedup, Wall, Pointreduce), kann die Schwierigkeit ebenso erhöht oder verringert werden.

Unser Levelkonzept sieht dabei vor, dass das letzte Level solange ausgeführt wird, bis es zu einem GameOver kommt. Diese Idee besteht daher, weil unser Spiel auf einem Highscore-Prinzip aufbaut und es somit keinen Sinn macht, das Spiel vorzeitig zu beenden. Ebenso kann die Schwierigkeitssteigerung nur bis zu einem bestimmten Punkt weitergeführt werden, irgendwann wäre das Spiel sonst unspielbar.

4.3 Parametrisierungskonzept

Um auf hartcodierte Parameter zu verzichten, haben wir uns für eine zusätzliche JSON-Parameterdatei entschieden[4]. *config.json* enthält dabei verschiedene Parameter, welche besonders für die Steuerung (sprich für den *Controller*) relevant sind.

```
{
  "rows": 10,
  "rotationConstant": 15,
  "maxStepsSpeed": 10,
  "maxStepsMultiplicator": 10,
  "maxStepsInvulnerable": 10,
  "normalStepTrigger": 400,
  "speedupStepTrigger": 250,
  "speeddownStepTrigger": 650
}
```

Es kann dabei die initiale Zeilenlänge[5] (**`rows`**), eine Rotationskonstante für den Fall der Steuerung via Bewegungssensor von Endgeräten (siehe im Abschnitt „Controller“), die maximale Anzahl an Spielschritten für verschiede zeitlimitierte Items (**`maxSteps`**) und die Zeitangaben (in Millisekunden) für den Aufruf der `nextGameStep()`-Methode im *Controller* angegeben werden. Dabei werden drei Zeitangaben unterschieden: Zunächst die Zeitangabe für den normalen Spielbetrieb (kein *SpeedUp* oder *SpeedDown* eingesammelt) (**`normalStepTrigger`**), dann die Angabe für den Fall, dass ein *SpeedUp*-Item eingesammelt wurde (**`speedupStepTrigger`**) und äquivalent dazu eine Angabe für den Fall, dass ein *SpeedDown*-Item eingesammelt wurde (**`speeddownStepTrigger`**). Diese Parameter haben sich als sinnvoll zur Auslagerung herausgestellt, da diese oft für Testzwecke verändert werden mussten.

---

[2] Empfehlung nach Versuchen: Spaltenanzahl ≤ 8, um auch auf mobilen Endgeräten ein angenehmes Spielerlebnis zu haben

[3] Im Regelfall sollte der Spielgeschwindigkeits-Multiplikator im Intervall $(0,1]$ liegen

[4] Nicht alle wichtigen Parameter sind über die JSON steuerbar. Beispielsweise ist der Identifikationsstring für den LocalStorage hartcodiert, schließlich sollte dieser eindeutig sein

[5] Empfehlung nach Versuchen: Zeilenanzahl ≤ 15, um auch auf mobilen Endgeräten ein angenehmes Spielerlebnis zu haben

5 Nachweis der Anforderungen

5.1 Nachweis der funktionalen Anforderungen

ID	Titel	Erfüllt	Tw. Erfüllt	Nicht erfüllt	Erläuterung
AF-1	Singleplayer	X			Es ist ein Einspieler-Konzept implementiert. Im *GameModel* kann maximal ein Spieler zurzeit existieren, wobei dieser Bewegungen ausführen und Kollisionen mit Items verursachen kann
AF-2	2D-Raster	X			Es wird innerhalb der *View* für die Darstellung des Spielfeld ein HTML-Table mit einzelnen Zellen verwendet, sodass ein 2D-Raster vorhanden ist
AF-3	Levelkonzept	X			Es ist ein Levelkonzept vorhanden, eine steigernde Schwierigkeit haben wir durch verschiedene Parameter wie Spielgeschwindigkeit, Spaltenanzahl und Itemverteilung erreicht. Siehe: Level-und Parametrisierungskonzept
AF-4	Parametrisierungs-konzept	X			Relevante Spielparameter wie z.B. initiale Zeilenanzahl, Durations für Timer, … können unabhängig vom Quellcode angepasst werden. Siehe: Level-und Parametrisierungskonzept
AF-3	Mobile Geräte	X			Eigene CSS-Dateien für aufgeteilte Displaygrößen von Endgeräten, welche die Haupt-CSS erweitern
AF-4	Desktop Geräte	X			Gelöst durch eigene CSS-Datei, welche als Hauptdatei dient (screen.css)
AF-5	Highscore speichern	X			Durch Verwendung der LocalStorage-Eigenschaft von Browsern kann der Highscore im Zusammensetzung mit einem Namen des Spielers gespeichert werden. Sofern diese Funktion in einem Browser nicht verfügbar ist, wird eine Fehlermeldung beim Versuch des Speicherns ausgegeben
AF-6	Spielmotivation	X			Das Spiel besitzt verschiedene Items, welche verschiedene Aktionen ausführen, die Schwierigkeit steigert sich mit höheren Leveln, das Spiel ist einfach zu verstehen und durch das Speichern des Highscores wird die Spielmotivation abermals gesteigert
AF-7	Item „Invulnerable“ hinzufügen	X			Das Item wurde nach Absprache mit Prof. Kratzke integriert. Dabei musste die Vererbungshierarchie der GameObjects angepasst, die Level-JSON-Datei sowie das Model um dieses neue Item angepasst werden. Eine neue Grafik wurde zusätzlich erstellt

5.2 Nachweis der Dokumentationsanforderungen

ID	Titel	Erfüllt	Tw. Erfüllt	Nicht erfüllt	Erläuterung
DF-1	Projektdokumentation	X			Detaillierte Beschreibung der wichtigsten Komponenten ist erfolgt, Diagramme und Abbildungen für schnelles Verstehen der Sachlage wurden verwendet
DF-2	Quellcodedokumentation	X			Der Quellcode wurde so dokumentiert, dass externe Personen schnell einsteigen können. Um sprachliche Barrieren zu umgehen, sind Kommentare auf Englisch formuliert
DF-3	Libraries	X			In *main.dart* ist aufgezeigt, welche Library für welche Aktionen verwendet wird

5.3 Nachweis der Einhaltung technischer Randbedingungen

ID	Titel	Erfüllt	Tw. Erfüllt	Nicht erfüllt	Anforderung
TF-1	Canvas	X			Es wurde keine Canvas verwendet, der Aufbau ist lediglich durch eine Tabelle mit Bildern gegeben
TF-2	Levelkonzept	X			Level können innerhalb von *level.json* verändert werden. Datei wird beim Spielstart eingelesen und die Level einzelnen werden dementsprechend im Spielverlauf angepasst
TF-3	Parameterformat	X			Spielparameter können innerhalb von *config.json* verändert werden. Datei wird beim Spielstart eingelesen und Spielparameter werden dementsprechend beim Spielstart angepasst
TF-4	View	X			Es wurde innerhalb der View lediglich auf Manipulation des DOM-Tree und Veränderung der CSS gesetzt und keine anderen Komponenten
TF-5	Spiellogik	X			Die Spiellogik basiert ausschließlich auf der der Programmiersprache Google Dart
TF-6	Browser	X			Das Spiel unterstützt in der JavaScript kompilierten Version sämtliche Browser mit einer JavaScript Engine. Es werden auch die Browser mit einer native Dart Engine unterstützt. Um Randbedingungen von iOS 10 gerecht zu werden, wurde bei der Steuerung zusätzlich auf eine weitere Komponente durch Verwendung der Bewegungssensoren gesetzt. So kann der User seine bevorzugte Steuerung auswählen.
TF-7	MVC	X			Das Spiel folgt der MVC-Architektur

5.4 Nachweis der Designanforderungen

ID	Titel	Erfüllt	Tw. Erfüllt	Nicht erfüllt	Anforderung
DS-1	Grafiken	X			Das Hintergrundbild unseres Spiels entstammt der Website pixabay.com und ist als CC0 (freie Nutzung, kein Bildnachweis erforderlich) gekennzeichnet[6]. Alle Bilder der Items sind ausschließlich selbst erstellt. Ansonsten sind keine weiteren Grafiken verwendet worden.

5.5 Verantwortlichkeiten im Projekt

Durch die in „Kap. 1 Projektkoordination“ beschriebene Möglichkeit, Verantwortlichkeiten und unterstützende Tätigkeiten innerhalb der Trello-Karten zu verfolgen, war es ein leichtes diese auf eine passende Tabelle zu übertragen.

Komponente	Detail	Asset	Tim-Pascal Lau	Niclas zum Felde	Léon Klick	Anmerkungen
Model	Items	src/gameObject.dart			V	Beinhaltet Inline Dokumentation Beinhaltet zusätzlich (diese) Dokumentation
	Level	src/level.dart		V	U	Beinhaltet Inline Dokumentation Beinhaltet zusätzlich (diese) Dokumentation
	GameModel	src/model.dart	U	U	V	Beinhaltet Inline Dokumentation Beinhaltet zusätzlich (diese) Dokumentation
	Levelkonzept	etc/level.json	U	U	V	Beinhaltet Inline Dokumentation Beinhaltet zusätzlich (diese) Dokumentation
View	HTML-Dokument	/index.html	V	U		
	Gestaltung durch CSS	css/largeTablet.css css/rotateDevice.css css/screen.css css/smartphone.css css/tablet.css	V			
	Viewlogik	src/view.dart	V		U	Beinhaltet Inline Dokumentation Beinhaltet

[6] Siehe https://pixabay.com/de/science-fiction-nebula-3-space-1424446/

						zusätzlich (diese) Dokumentation
Controller	Eventhandling	src/controller.dart		V		Beinhaltet Inline Dokumentation Beinhaltet zusätzlich (diese) Dokumentation
	Parametrisierung	src/controller.dart			V	Beinhaltet Inline Dokumentation Beinhaltet zusätzlich (diese) Dokumentation
	Steuerung	src/controller.dart	U	V		Beinhaltet Inline Dokumentation Beinhaltet zusätzlich (diese) Dokumentation
	Highscore-speicherung	etc/config.json			V	Beinhaltet Inline Dokumentation Beinhaltet zusätzlich (diese) Dokumentation

U = Unterstützende Tätigkeit
V = Verantwortlich für dieses Modul

6 Anmerkungen

Das Spiel wurde in verschiedenen Phasen ausgiebig getestet. Hierfür wurde auf einem verfügbaren Webspace die aktuellste Version des Spiels veröffentlicht, sodass möglichst viele Personen StarSpeeder testen und dementsprechend Feedback geben konnten. Logischerweise kann trotzdem nicht garantiert werden, dass das Spiel auf jedem Webbrowser gespielt werden kann. Trotzdem konnte eine Vielzahl der verfügbaren Webbrowser getestet werden, darunter: Google Chrome auf Android und PC, Safari auf Mac OS und iOS, Firefox auf Android und PC und Samsung Internet-Webbrowser auf Android.