
# Frogger
# Dokumentation

## Webtechnologie Projekt
## SoSe 2017

Prof. Dr. Nane Kratzke

von
Joshua Krieger
Sven Moeller

12. Juli 2017

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Listings

# 1. Einleitung

Im Rahmen des Webtechnologie Projektes haben wir uns für den Klassiker Frogger entschieden. Frogger wurde von Konami entwickelt und später von Sega im Jahre 1981 in Japan veröffentlicht. Es ist ein übliches Arcade Game welches man früher vorgefunden hat. Bis heute gibt es auch immer wieder Portierungen des Klassikers in unterschiedlichsten Variationen.

# 2. Anforderung und Spielkonzept

## 2.1 Anforderung

**Spielanforderung**

| ID | Kurztitel | Anforderung |
|---|---|---|
| AF-1 | Einplayer Game | Das Spiel soll ein Einzelspieler Spiel sein. |
| AF-2 | 2D Game | Das Spiel soll ein Einzelspieler Spiel sein. |
| AF-3 | Levelkonzept | Das Spiel sollte ein Levelkonzept vorsehen. |
| AF-4 | Parametrisierungskonzept | Das Spiel sollte ein Parameterisierungskonzept für relevante Spielparameter vorsehen. |
| AF-5 | N/A | Nicht verlangt dieses Jahr. |
| AF-6 | Desktop Browser | Das Spiel muss in Desktop Browsern spielbar sein. |
| AF-7 | Mobile Browser | Das Spiel muss auf SmartPhone Browsern spielbar sein. |

Tabelle 2.1: Tabelle mit den Spielanforderungen

**Dokumentationanforderung**

| ID | Kurztitel | Anforderung |
|---|---|---|
| D-1 | Dokumentationsvorlage | Die Dokumentation soll sich an vorliegender Vorlage orientieren. |
| D-2 | Projektdokumentation | Das Spiel muss geeignet dokumentiert sein, so dass es von projektfremden Personen fortgeführt werden könnte. |
| D-3 | Quelltextdokumentation | Der Quelltext des Spiels muss geeignet dokumentiert sein und mittels schriftlicher Dokumentation erschließbar und verständlich sein. |
| D-4 | Libraries | Alle verwendeten Libraries sind aufzuführen und deren Notwendigkeit zu begründen. |

Tabelle 2.2: Tabelle mit den Dokumentationanforderung

**Technische Randbedingungen**

| ID | Kurztitel | Anforderung |
|---|---|---|
| TF-01 | No Canvas | Die Darstellung des Spielfeldes sollte ausschließlich mittels DOM-Tree Techniken erfolgen. Die Nutzung von Canvas-basierten Darstellungstechniken ist explizit untersagt. |
| TF-02 | Levelformat | 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-03 | Parameterformat | Spielparameter sollten sich mittels deskriptiver Textdateien definieren lassen (z.B. mittels CSV, JSON, XML, etc.), so dass Parameter-Änderungen ohne Sourcecode-Änderungen des Spiels realisierbar sind. |
| TF-04 | HTML + CSS | Der View des Spiels darf ausschließlich mittels HTML und CSS realisiert werden. |
| TF-05 | Gamelogic in Dart | Die Logik des Spiels muss mittels der Programmiersprache Dart realisiert werden. |
| TF-09 | Browser Support | Das Spiel muss im Browser Chromium/Dartium (native Dart Engine) funktionieren. Das Spiel muss ferner in allen anderen Browserrn (JavaScript Engines) ebenfalls in der JavaScript kompilierten Form funktionieren (geprüft wird ggf. mit Safari, Chrome und Firefox). |
| TF-10 | MVC Architektur | Das Spiel sollte einer MVC-Architektur folgen. |
| TF-11 | Erlaubte Pakete | Erlaubt sind alle dart:* packages, sowie das Webframework start. |
| TF-12 | Verbotene Pakete | 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-13 | No Sound | Das Spiel muss keine Sounds unterstützen. |

Tabelle 2.3: Tabelle mit den Technische Randbedingungen

## 2.2 Spielkonzept

Grundlegend basiert das Spiel auf einem n x m Spielfeld, welches rasterartig aufgebaut ist. Der Spieler übernimmt die Rolle der Spielfigur Frogger und muss versuchen auf die andere Seite des Spielfeldes zu kommen. Dieses wird ihm erschwert durch unterschiedlichste Dinge wie eine befahrene Straße oder einem Fluss, bevor er den Geburtsteich erreicht, welches sein Ziel ist. Dieses muss er so oft wiederholen bis alle Geburtsteiche belegt sind. Dabei muss er alle vor ihm liegenden Gefahren ausweichen oder überwinden. Wie zum Beispiel unterschiedlichster Fahrzeuge, Objekte und Tieren. Er gewinnt Punkte für verschiedene Dinge auf dem Weg zum Geburtsteich. Die Objekte bewegen sich nur Horizontal auf ihrer eigenen Reihe und dies mit gleich bleibender Geschwindigkeit. Die Geschwindigkeit zwischen den Reihen kann allerdings unterschiedlich sein.

Die Schwierigkeit lässt sich variieren durch:

- Frequenz der Objekte die erstellt werden pro Reihe
- Abstand zwischen den beweglichen Objekten
- Länge der beweglichen Objekte
- Anzahl der Geburtsteiche im Level
- Anzahl der verschiedenen Objekte die in einer Reihe vorkommen können

Frogger könnte z.B. wie folgt variiert oder erweitert werden:

- Statische Hindernisse wie Büsche, Bäume oder parkende Autos
- Wegfallen von Grünstreifen durch Überflutung des Flusses
- Hinzufügen oder erweitern von neuen Objekten mit besonderen Eigenschaften
- Froscheigenschaften

# 3. Architektur und Implementierung

Da wir nach dem MVC Konzept unsere Architektur entwickeln sollten, haben wir unsere Spiellogik dementsprechend gegliedert.

Die Klasse FroggerView kapselt die Schnittstelle zum DOM-Tree und dient zur Visualisierung des Models und stellt dem Controller entsprechende Methoden bereit, um die Spieloberfläche zu aktualisieren. Näheres wird im Abschnitt 3.2 erläutert.

Die Klasse FroggerController ist die Schnittstelle zwischen dem Model und der View. Der Controller kontrolliert das Model und fordert die View auf sich zu aktualisieren. Näheres wird im Abschnitt 3.3 erläutert.

Das Model selbst ist komplex und kann nicht in einer einzelnen Klasse erfasst werden, viel mehr besteht das Model aus einem Zusammenspiel vieler einzelner Entitäten, die Komplexität überschaubar machen. Näheres wird im Abschnitt 3.2 erläutert.

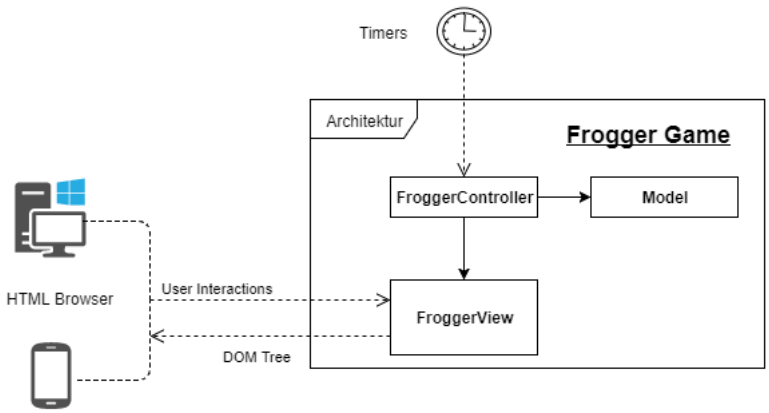


Abbildung 3.1: Architektur

Auf Grund der Größe, des gesamten UML, verweise ich hier auf einzelne Abschnitte des UML Diagramms. Trotzdem wird eine Gesamtübersicht mit beigelegt unter dem Dateinamen ÜMLComplete.png".

- UML Ausschnitt für die Entitäten ist in der Abbildung 3.2 zu sehen.
- UML Ausschnitt für das Spielfeld ist in der Abbildung 3.8 zu sehen.
- UML Ausschnitt für die Hilfsklassen, die verwendet werden, ist in der Abbildung 3.10 zu sehen.

## 3.1 Model

Aus dem grundlegenden Spielkonzept, wie in Abschnitt 2.2 dargelegt haben wir folgende Entitäten abgeleitet.

- Frosch (Frogger)
- Weiblicher Frosch (LadyFrog)
- Schildkröte (Turtle)
- Schlange (Snake)
- Baumstamm (Log)
- Fahrzeug (Vehicle)
- Krokodil (Crocodile)
- Spielfeld (Field)
- Spielkachel (Tile)

Im weiteren Verlauf der Entwicklung sind weitere Entitäten und Klassen dazu gekommen, die etwa der Unterstützung oder der Strukturierung des Model zu Gute kommen.

### 3.1.1 Schnittstelle zum Controller

#### 3.1.1.1 GameInstance

Die GameInstance ist die Schnittstelle zum Controller. Der Controller ruft periodisch Methoden der GameInstance auf und die GameInstance sendet Events (Abschnitt 3.1.1.2) zum Controller. Im Controller (Abschnitt 3.3) werden die Timer näher erläutert.

Der Zustand der GameInstance bildet sich aus

- einem StreamController, der die Aufgabe des EventController's übernimmt. (eventController)
- einer Warteschlange, in der die Spieler Bewegungen gespeichert werden bis sie vom Model übernommen wurden. (commandQueue)
- einem Spielfeld (Abschnitt 3.1.7.1) (field)
- einem Frogger als Spielfigur. (frogger)
- einem Integer für die Punktzahl. (score)
- einem Integer für die Leben des Spielers. (lives)
- einem Integer für die erreichten Ziele. (goalreached)
- einem Integer für die höchste Reihe, die zum Ziel erreicht wurde. (highestRowToGoal)
- einem Integer für die eingesammelten LadyFrog's. (collectedLadyFrogs)

Die Timer des Controllers rufen folgende Methoden auf:

- void update(). Sie bringt das Model in einen neuen Zustand.
- void dispatchCommandFromQueue(). Sie führt die nächste Aktion des Spielers aus, welche in der Warteschlange liegt.

#### 3.1.1.2 EventType

Dies ist ein Enum welches, Event Typen definiert, welche benutzt werden um zwischen GameInstance und Controller zu kommunizieren. Die Events, die die Bewegungen des Frogger's betreffen werden nur benötigt, damit die Camera (Abschnitt 3.2.3) sich parallel zum Frogger bewegt.

| **Event** | **Beschreibung** |
|---|---|
| Win | Wenn ein Level erfolgreich beendet wurde. |
| GameOver | Wenn der Spieler alle Leben verloren hat. |
| Loaded | Wenn das Spielfeld erfolgreich geladen wurde. |
| FroggerMovedLeft | Wenn Frogger sich nach Links bewegt hat. |
| FroggerMovedRight | Wenn Frogger sich nach Rechts bewegt hat. |
| FroggerMovedUp | Wenn Frogger sich nach Oben bewegt hat. |
| FroggerMovedDown | Wenn Frogger sich nach Unten bewegt hat. |
| ViewReset | Wenn Frogger ein Leben verliert und die Camera zurückgesetzt werden muss. |

Tabelle 3.1: Tabelle der Event Typen

### 3.1.2 Spiel- und Levelparameter

Alle wichtigen Spielparameter werden in der Klasse Config gehalten; Diese ist überall verfügbar. Sie hält neben wichtigen Parametern wie der Spielfeldgröße auch Festlegungen bezüglich eindeutiger Kennzeichnungen der einzelnen Entitäten und Pfade zu Ressourcen. Genaueres wird in der exakten Beschreibung der Config Klasse erläutert (Abschnitt 3.1.2.1).

Die Parameter, zu den einzelnen Leveln, werden in der Klasse Level gehalten. Näheres dazu wird in der exakten Beschreibung der Level Klasse erläutert (Abschnitt 3.1.2.2).

#### 3.1.2.1 Config

Die Config dient als zentraler Punkt für Globale Variablen und ist eine statische Klasse. Sie wird aus einer Json Datei heraus erstellt. Dies wird im Thema Parameterisierungskonzept (im Abschnitt 4.2) veranschaulicht.

Über ihr können folgende Einstellungen angegeben werden.

- Spielfeld Höhe (FieldRow)
- Spielfeld Breite (FieldCol)
- Kamera Höhe (ViewRow)
- Kamera Breite (ViewCol)
- Punkte für den Sprung in die richtige Richtung (Score_JumpInDirection)
- Punkte für erreichtes Ziel (Score_ Goal)
- Punkte für abgegebenen LadyFrog (Score_LadyFrog)
- Punkte für geschafftes Level (Score_CompletedLevel)
- Punkte für verbleibende Level Zeit (wird momentan nicht genutzt) (Score_TimeLeftPerUnit)
- Dateinamen Format in welchem die Level Jsons vorliegen (JsonLevelNamePattern)
- Das maximale Level, sprich wie viele Level Jsons es gibt (maxLevel)
- Timer Interval für das Model (updateInterval)
- Timer Interval für das aktualisieren der View (fieldPrintInterval)
- Timer Interval für das ausführen der Spieler Bewegungen (froggerDispatchMoveInterval)

Desweiteren sind alle IDs der Entitäten und Tiles hartcodiert.

**Spielfelder**

| Name | Bezeichnung | ID |
|---|---|---|
| Straße | Street | 1 |
| Gras | Grass | 2 |
| Wasser | Water | 3 |
| Ziel | Goal | 4 |
| Hohes Gras | HighGrass | 5 |

Tabelle 3.2: Tabelle mit den Übersicht der Spielfelder

**Entitäten**

| Name | Bezeichnung | ID |
|---|---|---|
| Nichts | noEntity | 0 |
| Baumstamm | Log | 1 |
| Krokodile | Crocodile | 3 |
| Schildkröte | Turtle | 4 |
| Schlange | Snake | 5 |
| Laster | Truck | 6 |
| Frosch | Frogger | 7 |
| Auto | Car | 8 |
| Weib. Frosch | LadyFrog | 9 |

Tabelle 3.3: Tabelle mit den Übersicht der Entitäten

#### 3.1.2.2 Level

Die Level Klasse hält alle Level Parameter und wird aus einer Json Datei geladen. Jedes mal wenn ein neues Level angefordert wird, wird die zugehörige Json Datei geladen. Mit Hilfe der Level Klasse werden die Parameter des Levels durch das Model gereicht. Das Einlesen aus der Json übernimmt die Klasse selbst. Die Level Parameter sind im Thema Levelkonzept (Abschnitt 4.1) aufgelistet und näher erläutert.

### 3.1.3 Entitäten

#### 3.1.3.1 Hierarchie

Die Hierarchie der Entitäten ist wie in Abbildung 3.2 aufgebaut. Das abstrakteste Objekt ist das GameObject (Abschnitt 3.1.3.3), von ihr leiten sich 2 verschiedene Arten von Klassen ab. Zum einem die Tiles, (Abschnitt 3.1.3.4) die die Spielfelder auf dem Spielfeld repräsentieren und zum anderen die Entities (Abschnitt 3.1.3.5). Sie wiederum ist die Basisklasse von alle Spielfiguren auf dem Spielfeld. Alle Spielfiguren im Spiel müssen diese Basisklasse erweitern.

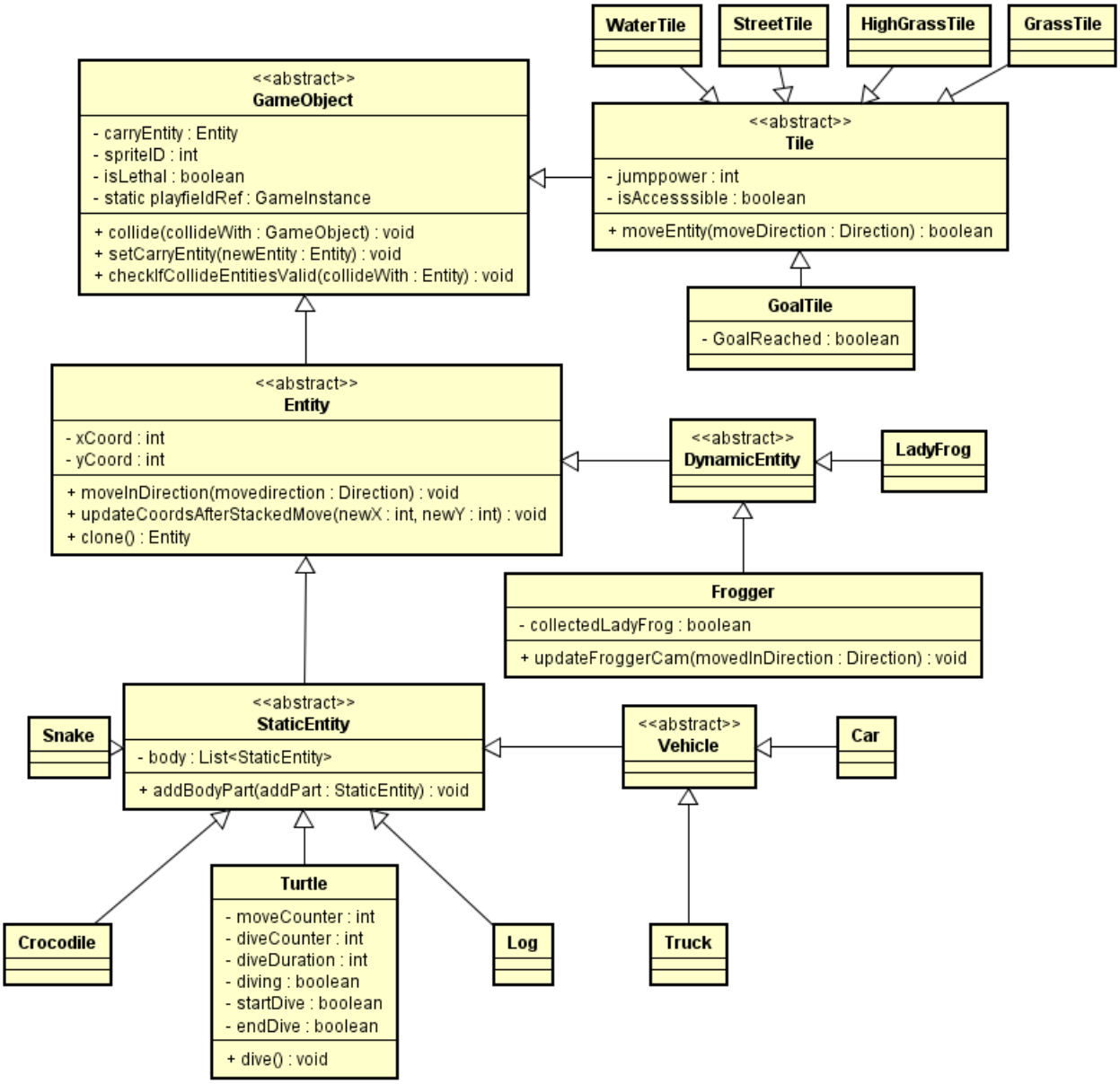


Abbildung 3.2: Entitäten Hierarchie

#### 3.1.3.2 Entitäten System

Das Entity-System ist aufgebaut wie eine Linked List. Das Vererbungssystem gibt vor, dass jede Entity von GameObject abgeleitet ist. Dadurch wird es möglich dem sog. EntityStack zu bauen. Da das GameObject eine CarryEntity definiert, hat jedes davon abgeleitete Objekt ebenfalls dieses Feld und kann somit eine Entität auf seinem „Rücken“ tragen.

Dabei kann zur Laufzeit ein Stack zum Beispiel wie in Abbildung 3.3 aussehen.

In Abbildung 3.3 kann man sehr gut die Linked List erkennen, die beim Benutzen diese Ansatzes entsteht. Jedes der in der Abbildung dargestellte Objekt leitet von GameObject ab.

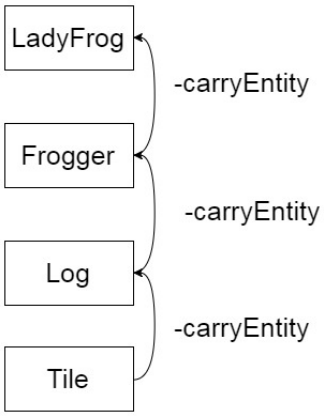


Abbildung 3.3: Entity Stack

Die Row (Abschnitt 3.1.7.2) Klasse hält ein Array der Tile Objekte. Die sind dann, im Level entsprechend angegeben, vom Typ Gras/Wasser/Straße oder Ziel. Jedes dieser Felder kann dann Entitäten tragen, die dann wieder Entitäten tragen können und so weiter. Man kann auf diese Weise beliebig große Entität-Stacks bauen, die dann ihre Kollisionen mit anderen Entitäten darüber handhaben, indem man eine Entität mit dem auf dem Feld liegenden Tile kollidieren lässt. Das Tile entscheidet dann entsprechend seiner Implementierung ob es die Kollision selber handhabt oder an die, auf ihm liegenden Entität, weitergibt.
Es wird folgende Ausgangssituation nach Abbildung 3.4 betrachtet.

Wenn man nun über die Eingabeoberfläche den Befehl gibt, dass sich Frogger auf das andere Feld begeben soll, dann läuft folgender Ablauf ab.
Frogger wird als erstes die Collide Methode des Tile's aufrufen von dem Feld, auf das er sich bewegen möchte. In Abbildung 3.5 gekennzeichnet durch die rote Markierung.

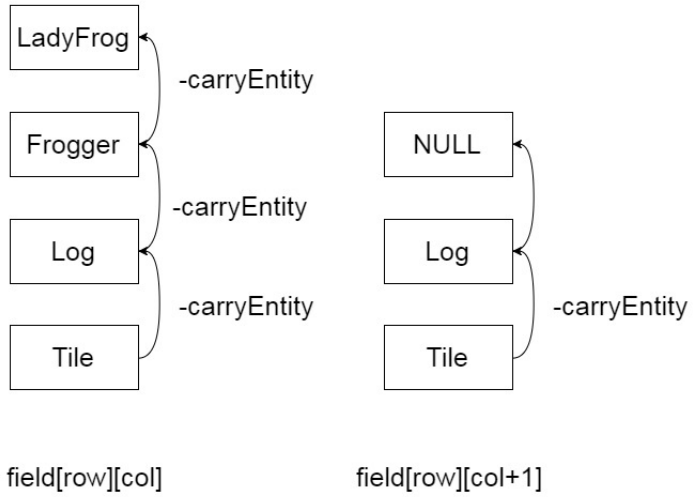


Abbildung 3.4: Ausgangssituation einer Kollision

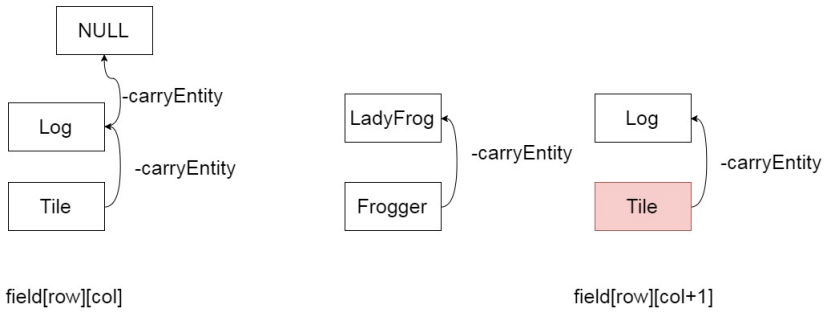


Abbildung 3.5: Kollision mit Tile

Da das Tile (mit seiner derzeitigen Implementierung) die Kollision nicht handeln wird, da er eine Carry-Entität besitzt, nimmt er das Argument des Collide-Aufrufs (in diesem Fall Frogger) und ruft die Collide Methode mit dem Argument auf, die es selbst erhalten hat.
Der in diesem Beispiel verwendete Log würde dann die Kollision handeln und Frogger als neue Carry-Entität setzen. Hätte der Log ebenfalls eine Carry-Entity, dann würde der Log an diese Entität die Kollision weitergeben. Da Frogger einen Verweis auf den auf ihm sitzenden Ladyfrog besitzt, werden dann durch Frogger die Koordinaten des Ladyfrogs ebenfalls upgedatet, nachdem Frogger seine eigenen Koordinaten geändert hat.

Nachdem die Kollision und damit auch der Move Befehl abgeschlossen ist. Hat man den Stackaufbau nach Abbildung 3.7 erreicht.

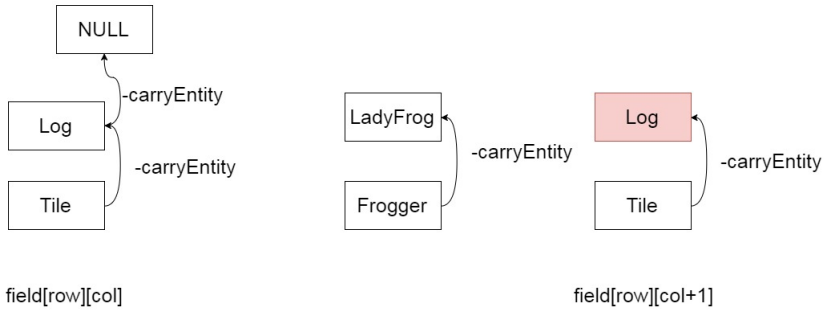


Abbildung 3.6: Kollision mit Log

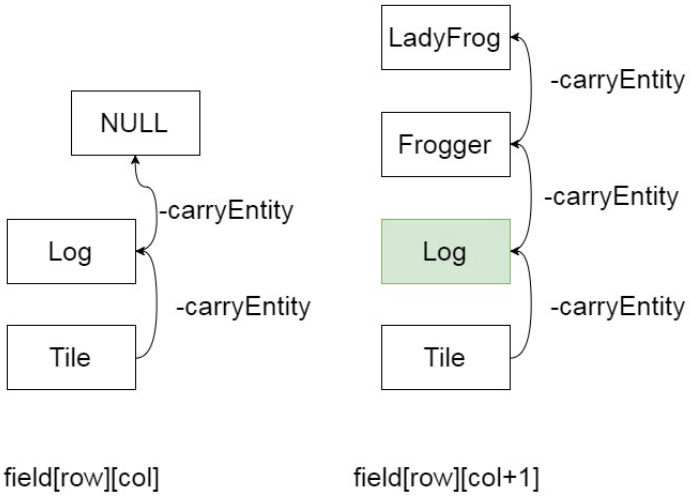


Abbildung 3.7: Ausgeführter Move Befehl

#### 3.1.3.3 GameObject

Das GameObject ist das grundlegende Element, von dem sich alle Entitäten ableiten. Die beiden direkten Ableitungen von GameObject sind die Entity (Abschnitt 3.1.3.5) und das Tile (Abschnitt 3.1.3.4) und auf Ihnen baut sich alles weitere auf. Dies ist ersichtlich aus der Architektur sowie genauer in der Hierarchie im Abschnitt 3.1.3.1.

Der Zustand des GameObject bildet sich aus:

- einer Entity die getragen werden kann. (carryEntity)
- einem Sprite welches genutzt wird vom SpriteManager (Abschhnitt 3.1.6) um die Grafiken zu verwalten, mit denen dieses GameObject auf dem Spielfeld angezeigt wird. (sprite)
- einem Boolean das Aussage darüber macht, ob eine Entität als tödlich zu betrachten ist. Das bedeutet, dass bei Kontakt mit Frogger diese Entität Frogger „töten“ wird und das alle Schritte abgearbeitet werden müssen um auf dieses Ereignis zu reagieren. (isLethal)

Zuzüglich zu den Instanz abhängigen Variablen hat das GameObject noch eine statische Referenz auf die GameInstance. Wir brauchen diesen Verweis, da alle GameObjects, also auch Entitäten, ihre Positionen selbständig auf dem Spielfeld verwalten können sollen. Mithilfe dieses Verweises, können die GameObjects auch untereinander kommunizieren.

#### 3.1.3.4 Tile

Die Tile Klasse ist die Superklasse für alle konkreten Tiles. Sie hat grundlegendes Verhalten zum setzen von Entitäten und gibt Kollisionen an anderen Entitäten weiter, die auf ihr sitzen. Sie legt außerdem fest, wie weit Frogger springt, wenn er auf dieser Tile sitzt. Auf diese Weise kann jede konkrete Tile eine „Powered up“ Version von sich selbst werden ohne für jede konkrete Tile eine neue Version erstellen zu müssen.

Der Zustand des Tile bildet sich aus:

- einem Boolean, der angibt ob Frogger sich auf diese Tile bewegen kann oder die Bewegung abgeblockt wird. (isAccessible)
- einem Integer, der bestimmt wie weit Frogger springen wird, wenn er sich von dieser Tile wegbewegen möchte. (jumpPower)

**Konkrete Tiles**
Die konkreten Implementierungen der Tiles. Haben grundlegend das selbe Verhalten. Sie unterscheiden sich nur in dem Typ des CollideResult (Abschnitt 3.1.4), welches sie zurückgeben. Die Water-Tile zum Beispiel würde auf Kontakt mit Frogger wasLethal zurückgeben.

#### 3.1.3.5 Entity

Die Entity Klasse ist die Superklasse für alle Typen von Entitäten. Direkt unter ihr liegen die statischen (Abschnitt 3.1.3.6) sowie die dynamischen Entitäten (Abschnitt 3.1.3.7).

Der Zustand der Entity bildet sich aus:

- einem Integer, der die logische x Koordinate dieser Entität auf dem Spielfeld darstellt. (coordX)
- einem Integer, der die logische y Koordinate dieser Entität auf dem Spielfeld darstellt. (coordY)

Die Entity besitzt eine wichtige Methode:

- updateCoordsAfterStackedMove(int newX, int newY)

Diese Methode erlaubt es von einer Entität aus alle auf ihr sitzenden Entitäten neue Koordinaten zu geben. Zusätzlich werden auch damit die Koordinaten der Entität umgesetzt, auf welcher diese Methode aufgerufen wurde.

#### 3.1.3.6 StaticEntity

Eine StaticEntity hält nur eine Liste von Body Elementen die wiederum auch StaticEntity Elemente sind. Diese werden bei einem Update der Entität, alle upgedatet. Auf diese Weise werden zusammenhängende StaticEntities wie Trucks oder Logs realisiert.

**3.1.3.6.1 Vehicle** Die Vehicle Klasse war wegen Punkten der Erweiterbarkeit als Überklasse für die beiden Fahrzeug basierten Klassen gedacht.

**3.1.3.6.2 Car** Die Car Klasse repräsentiert ein einfaches singuläres Objekt, welches sich linear über das Spielfeld bewegt und bei Kollision mit Frogger diesen tötet.

**3.1.3.6.3 Truck** Die Truck Klasse verhält sich genau so wie eine Kette aus Auto Objekten. Die Body Elemente werde dann nur in der Body Liste gehalten, wie in der Beschreibung der StaticEntity angegeben.

**3.1.3.6.4 Log** Der Log ist eine Kette von begehbaren Holzelementen, die sich in der Regel über Wasser bewegen. Diese Kette ist an allen Stellen frei begehbar. Es besteht eine gewisse Chance, das ein LadyFrog auf so einem Log Element mit spawnt.

**3.1.3.6.5 Crocodile** Die Crocodile Klasse ist ähnlich wie der Log, mit dem einzigen Unterschied, dass nur das vorderste Kopfelement für Frogger tödlich ist. Die anderen Elemente sind frei begehbar.

**3.1.3.6.6 Snake** Die Snake Klasse verhält sich ganz genau so, wie ein Auto Objekt, nur das es dafür gedacht war sich über den Grasstreifen zu bewegen.

**3.1.3.6.7 Turtle** Die Turtle ist die komplexeste der Entitäten. Da sie die Funktionalität implementiert unterzutauchen, mit vorherigem anzeigen das abgetaucht werden wird, und Auftauchen nach einer gewissen Tauchdauer. Das Untertauchen in mehreren Schritten ist mittels einer kleinen Statemachine gelöst.

Der Zustand der Turtle bildet sich aus:

- einem Integer, die als Zählvariable dient und angibt wie lange eine Turtle schon unter Wasser ist. (diveCounter)
- einem Integer, die das Limit angibt wie lange die Turtle insgesamt braucht um wieder den normalen „Über Wasser“ Zustand zu erreichen, nachdem sie angefangen hat unterzutauchen. (diveDuration)
- einem Boolean, der gibt an ob die Turtle grade unter Wasser ist. Ist als ein Zustand der oben erwähnten Statemachine zu verstehen. (diving)
- einem Boolean, der gibt an ob die Turtle grade in dem „halb unter Wasser“ Zustand ist. Ist als ein Zustand der oben erwähnten Statemachine zu verstehen. In diesem Zustand ist die Turtle noch nicht tödlich für Frogger. (startDive)
- einem Boolean, der gesetzt wird nachdem die Turtle einen kompletten Tauchvorgang abgeschlossen hat. Wird benutzt um in der Statemachine Werte wieder zurückzusetzen. (endDive)

Zuzüglich zu den Instanz abhängigen Variablen hat die Turtle noch folgende statische Werte:

- Konstanter Integer, der angibt wie lange die Turtles in dem Zustand des „halb unter Wasser sein“ verbleiben, bevor sie komplett abtauchen und dadurch auch tödlich für Frogger werden. (diveInit)
- Ein Zufallsgenerator, der genutzt wird um zu entscheiden, ob eine Turtle abtauchen wird. (RanDive)

#### 3.1.3.7 DynamicEntity

Die DynamicEntity war aus Gründen der Erweiterbarkeit als Überklasse für alle Entities gedacht, die sich, zu nicht festgelegten Zeitpunkten, frei in 4 Richtungen bewegen können.

**3.1.3.7.1 Frogger** Die Frogger Klasse ist die Spielfigur die der Spieler steuert.

Der Zustand des Froggers bildet sich aus:

- einem Boolean, der an gibt ob Frogger bereits ein LadyFrog eingesammelt hat. Dieses Flag nimmt ebenfalls Einfluss auf die visuelle Repräsentation von Frogger. Die Logik dazu ist im SpriteManager (Abschnitt 3.1.6) geregelt mit Hilfe des Flags. (collectedLadyFrog)

Es sind 2 wichtige Methoden am Frogger implementiert:

- updateFroggerCam(Direction movedInDirection) Die Kamera Klasse (Abschnitt 3.2.3), die genutzt wird um Frogger auf dem Feld darzustellen, braucht diese Events um sinnvolle Bewegungen auf dem Feld durchzuführen und so das „fensterartige“ Verhalten zu realisieren.
- collide(Entity collideWith) Frogger hat die besondere Eigenschaft, die meisten Kollisionen mit anderen Entitäten an die Entitäten zu geben, die mit ihm kollidieren. Auf diese Weise können wir das Kollisionsverhalten aus der Frogger Klasse herausziehen und sie an den Klassen implementieren, die dann die Entscheidung sowieso treffen sollen. Die einzige Kollision, die gehandelt wird ist die Kollision mit dem LadyFrog.

**3.1.3.7.2 LadyFrog** Der LadyFrog hat eine gewisse Chance auf einem Log zu spawnen und kann von Frogger aufgesammelt werden um beim Erreichen des Ziels extra Punkte zu generieren. Wenn Frogger bereits einen LadyFrog aufgesammelt hat, dann kann er kein Feld mehr betreten, welches von einem weiteren LadyFrog belegt wird.

### 3.1.4 CollideResult

Dies ist ein Enum welches Kollisionstypen definiert, die als Rückgabewert der Kollisionsfunktionen der Entitäten zurückgegeben werden können.

| **Collide** | **Beschreibung** |
|---|---|
| noValidMoveCoordinates | Falls man sich versucht auf eine Koordinate zu bewegen, welche aus Sicht des Felds als nicht gültig angesehen wird. |
| wasLethal | Wenn Frogger eine Kollision mit einer Entität hatte, die „tödlich verlief, wird dieses CollideResult zurückgegeben. |
| couldNotMoveTo | Dieses Result wird zurückgegeben, wenn Frogger sich auf ein Feld bewegen will, welches nicht begehbar ist, weil zum Beispiel hohes Gras den Weg blockiert. Auf diese Weise könnte man auch feste Hindernisse wie Wände realisieren. |
| wasNotHandled | Dieses Result ist mehr als eine Art Exception zu verstehen. Es wird nur dann als Ergebnis zurückgegeben, wenn eine Kollision von keiner Entität gehandelt wird. Dies sollte mit der derzeitigen Implementierung niemals auftreten. |
| goalReached | Wird beim Erreichen eines GoalTile durch Frogger zurückgegeben. |
| wasSet | Das Result wird zurückgegeben, wenn Frogger sich auf eine Tile bewegen möchte und die Kollision darin endet, dass Frogger von einem anderen GameObject als neue CarryEntität akzeptiert wurde. |
| ladyFrogPickedUp | Wenn Frogger sich auf ein Feld bewegt, auf dem ein LadyFrog sitzt, dann wird dies das Ergebnis der Kollision sein. |

Tabelle 3.4: Tabelle der Kollisionstypen

Diese Werte werden zurückgegeben als Ergebnis einer Kollision zweier Entitäten. In der Regel umfasst dies Frogger und eine andere Entität. Der Rückgabewert ermöglicht der aufrufenden Umgebung einfach auf das Ergebnis einer Kollision zu reagieren. Zum Beispiel das Result wasLethal wird zurückgegeben, wenn Frogger bei einer Kollision „stirbt“. Als Reaktion daraufhin kann dann in diesem Fall die GameInstance zum Beispiel ein Leben abziehen und Frogger auf eine bestimmte Position zurücksetzen.

### 3.1.5 Direction

Dies ist ein Enum welches einfach die Bewegungsrichtung für Objekte definiert.

- UP
- DOWN
- RIGHT
- LEFT

### 3.1.6 Sprite und SpriteManager

Um sämtliche Grafiken zu den einzelnen Entitäten und Tiles zu verwalten, wurde eine Sprite Klasse sowie auch ein SpriteManager erstellt. Die Sprite Klasse ist ein einfach nur ein Container mit einer ID und einen Pfad zur Bild Datei. Der Pfad wird später in die CSS Datei eingefügt für die jeweilige Entität oder Tile auf dem Spielfeld.

Der SpriteManager hingegen legt eine interne Struktur an um Sprites zu verwalten und diese zu setzen in Abhängigkeit ihres Verhaltens. Das Verhalten der Entitäten ist etwa über ihre Richtung oder Flags innerhalb der einzelnen Entitäten gegeben.

Der SpriteManager gibt auch die Möglichkeit für jedes Verhalten unterschiedliche Sprites zu haben. Zum Beispiel existieren unterschiedliche Sprites für das Spielfeld Gras. Wenn nun ein Grasfeld erstellt wird, wird ein zufälliges Sprite aus der Grasfeld Kategorie ausgewählt.

Um ein Sprite für ein GameObject zu setzen, wird einfach die setSprite Methode vom SpriteManager aufgerufen und das GameObject übergeben.

### 3.1.7 Spielfeld und Spielreihen

Der Zugriff auf das Spielfeld selbst wird durch die Klasse Field gekapselt. Genaueres zum Spielfeld wird im Abschnitt 3.1.7.1 selbst erläutert. Die Aufgaben des Spielfeld wurden nochmals in eigene Klassen unterteilt.

Da nach dem Spielkonzept jede Reihe von sich getrennt agiert und bewegt, so wie es im Abschnitt 2.2 erklärt wurde, haben wir eine Klasse Row (Abschnitt 3.1.7.2) eingeführt, die die Verantwortung einer einzelnen Reihe auf dem Spielfeld übernimmt.

Jede Row enthält einen RowSpawner, (Abschnitt 3.1.7.3) der wiederum die Verantwortung übernimmt Entitäten in einer Reihe zu spawnen.

Der RowSpawner entscheidet nicht selbst was und welche Entität gespawnt wird, dies übernimmt die Klasse BrainSpawn worauf im Abschnitt 3.1.7.4 eingegangen wird.

Der BrainSpawner erstellt mit Hilfe der Levelparameter (Abschnitt 3.1.2.2)

SpawnObjekte die mit der Klasse SpawnObject (Abschnitt 3.1.7.5) modelliert werden.

Die Klassen BrainSpawner sowie SpawnObject sollten eigentlich als 'Nested Classes' implementiert werden, allerdings unterstützt Dart dieses Konzept nicht. Auf Grund dessen wurden sie als eigenständige Klassen implementiert, aber logisch gesehen sind sie ein Teil des RowSpawner's um Funktionalität strukturiert zu kapseln.

Veranschaulichen kann man sich dies an der Abbildung 3.8.

#### 3.1.7.1 Field

Die Klasse Field besteht aus

- einer Liste von Reihen (Row)
- dem aktuellen Level (Level)

Das Spielfeld stellt neben den üblichen CRUD Methoden auf dem Spielfeld selbst zwei wichtige Methoden bereit.

Die Aktionen auf dem Spielfeld werden über die Update Methode in Bewegung gesetzt, indem das Spielfeld jeder Reihe mitteilt das sie sich updaten soll, wie in Listing 3.1 zu sehen.

```
1  void update()
2  {
3    // update every row
4    field.forEach((r)
5          {
6              r.update();
7          });
8  }
```

Listing 3.1: Field - Update Method

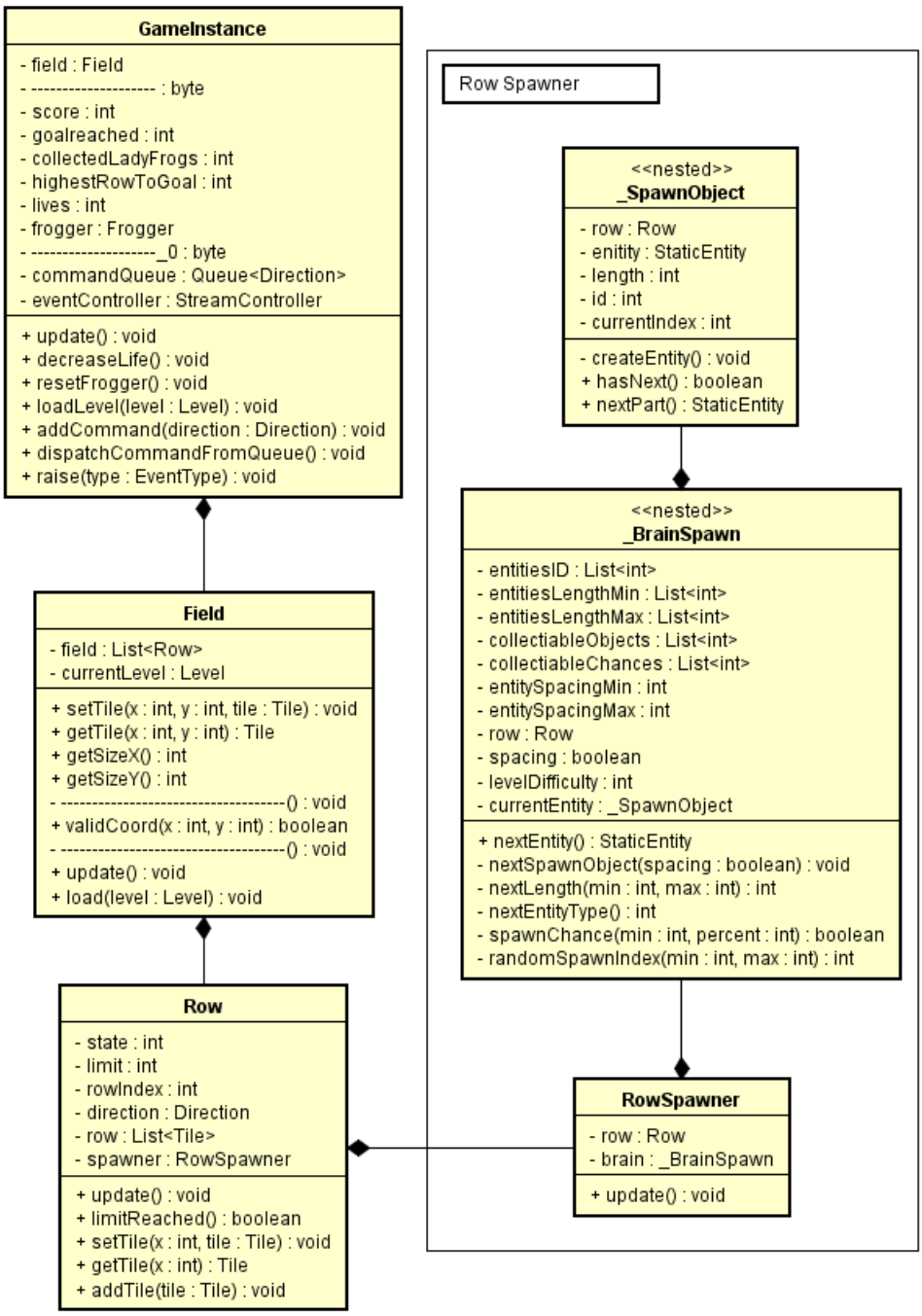


Abbildung 3.8: UML - Spielfeld

Das Level wird über eine Methode in das Spielfeld geladen, mit dessen Hilfe das Spielfeld erstellt wird. Im Listing 3.2 ist ein Ausschnitt dessen zu sehen.

```
bool load(Level level)
{
  /*        ...                */

    // create tiles in field
    for(int r = 0; r < level.getLevelHeigth(); r++)
    {
      // for each row
      for(int tile = 0; tile < Config.FieldCol; tile++)
      {
        switch(level.getTileTypeForRow(r))
        {
          case Config.Grass:
            field[r].addTile(new GrassTile());
            break;
          case Config.Street:
            field[r].addTile(new StreetTile());
            break;
          case Config.Water:
            field[r].addTile(new WaterTile());
            break;
          case Config.Goal:
            field[r].addTile(new HighGrassTile());
            break;
        }
      }
    }

  /*        ...                */

    // tell the controller the level is loaded
    GameObject.playFieldRef.raise(EventType.Loaded);
    return true;
}
```

Listing 3.2: Field - Load Method

#### 3.1.7.2 Row

Der Zustand der Klasse Row wird wie folgt dargestellt:

- einem Index, der die Reihe identifiziert auf dem Spielfeld (rowIndex)
- einem Limit, bis wohin gezählt wird bis ein Update geschehen soll (limit)
- ein Zustand, der hoch zählt bis zum Limit (state)
- eine Richtung. Diese sagt aus in welche Richtung sich die Entities auf der Reihe bewegen (direction)
- eine Liste von Spielfeldkacheln (Tiles) (tiles)
- einem Spawner, der neue Entitäten in die Reihe spawnt (spawner)

Bei jedem Update, welches auf der Reihe ausgeführt wird, wird die 'state' Variable inkrementiert, bis sie das Limit der Reihe erreicht hat. Wenn sie das Limit erreicht, wird 'state' wieder auf 0 gesetzt und alle Entitäten in der Reihe bewegen sich in Richtung "direction". Die "limit"Variable ist einer der Levelparameter die in der Json festgehalten ist. Je kleiner das 'limit', desto schneller werden die Entitäten in der Reihe sich bewegen.

Neben dem eigentlichen Bewegen der Entitäten wird hier auch die Auswertungen von Kollisionen behandelt. Diese treten ein wenn eine Entität mit Frogger kollidiert. Hier einmal an dem Beispiel für die Behandlung wenn sich die Reihe nach Links bewegt.

```
case Direction.LEFT:
  // move row left
  for(int tX = 0; tX < _tiles.length; tX++)
  {
    if(_tiles[tX].getCarryEntity() != null)
    {
      // don't update frogger
      var entity = _tiles[tX].getCarryEntity();
      if(entity is Frogger)
        continue;

      // move entity and check collide result
      // (can collide with frogger)
      CollideResult result = entity.moveInDirection(_direction);
        if(result == CollideResult.noValidMoveCoordinates)
          _tiles[tX].setCarryEntity(null);
        else if(result == CollideResult.wasLethal)
        {

          /*         ...                  */

          GameObject.playFieldRef.raise(EventType.ViewReset);
          GameObject.playFieldRef.decreaseLife();
        }

        /*         ...                  */
    }
  }
break;
```

Listing 3.3: Row - Snippet Update Method

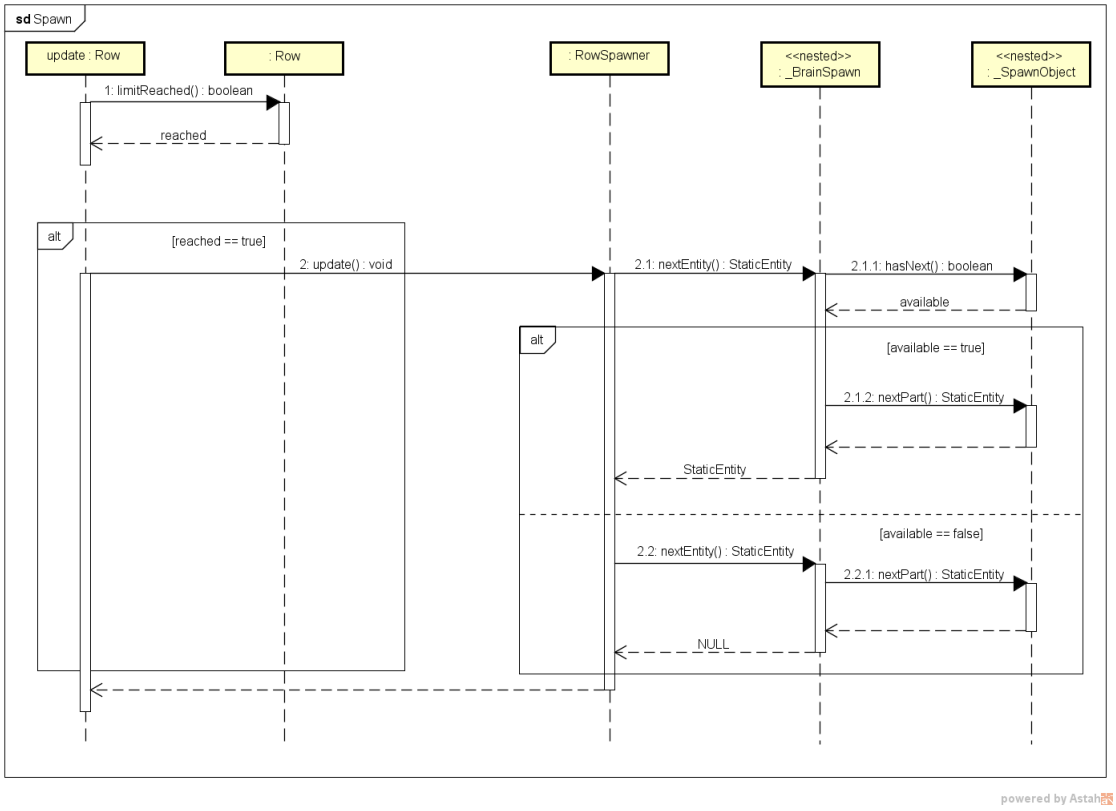


Abbildung 3.9: Sequenzdiagramm für das erstellen eines Elementes in der Reihe

#### 3.1.7.3 RowSpawner

Der RowSpawner selber beinhaltet nicht viel, da er hier nur dafür verantwortlich ist ein Teil einer Entität in der Reihe zu spawnen. Daher besteht der RowSpawner nur aus einer Referenz der Reihe (row) auf der er agiert, sowie ein BrainSpawn (brain) der ihm das nächste Teil einer Entität gibt, die er spawnen soll.

```
1            /*check if there are something to spawn*/
2            if((entity = _brain.nextEntity()) != null)
3
4            /*set entity part into row (example for direction left)*/
5            _row.getTile(_row.getRowLength() - 1)
6              .setCarryEntity(entity);
7
```

Listing 3.4: RowSpawner - Snippet Update Method

#### 3.1.7.4 BrainSpawn

Das BrainSpawn extrahiert alle Parameter aus der Level Klasse, (Abschnitt 3.1.2.2) die etwas mit Entitäten zu tun haben. Diese benötigt es um jeweils ein SpawnObject zu erstellen.

Daraus bildet sich folgender Zustand der Klasse.

- Liste aus Entität Arten die in der Reihe vorkommen können. (entityID)
- Liste aus minimal Längen der jeweiligen Entität Art. (entityLengthMin)
- Liste aus maximal Längen der jeweiligen Entität Art. (entityLengthMax)
- Liste aus minimal Abstand zwischen zwei Entitäten in der Reihe. (entitySpacingMin)
- Liste aus maximal Abstand zwischen zwei Entitäten in der Reihe. (entitySpacingMax)
- Liste aus aufsammelbaren Objekt Arten die in der Reihe vorkommen können. (collectibleObjects)
- Liste aus den jeweiligen Chancen das so ein aufsammelbares Objekt spawnen kann. (collectiableChances)
- Referenz der Reihe auf der er agiert. (row)
- Einem Boolean der Aussagt ob als nächstes eine Entität oder ein Freiraum gespawned wird. (spacing)
- Einem Integer der noch ein wenig die Schwierigkeit des Levels beeinflussen kann, dieser fließt mit in die Standard Spawnchance der Entitäten und Objekte ein. (levelDifficulty)
- Ein SpawnObject welches, jeweils immer eine komplette Entität dargestellt, die gespawned werden soll. Diese ist solange gültig bis sie komplett im Spielfeld ist, danach wird es neu erstellt für das nächste Objekt.

**Welche Entität wird als nächstes gespawned ?**

In der Initialen Erzeugung besteht eine 60% Chance das als erstes eine Entität gespawned wird.

Wenn es dazu kommt das eine Entität gespawned werden soll, dann wird als ersten geschaut ob in der Reihe mehr als eine Entität vorkommen kann. Sollte es der Fall sein, dass mindestens 2 Entitäten pro Reihe vorkommen können, dann gibt es eine weitere 20% Chance um **nicht** die Standard Entität der Reihe zu spawnen. Die Standard Entität der Reihe ist immer die, die im Level als erstes in der Liste für die jeweilige Reihe steht. Auf der festen 20% Spawnchance wird noch die Level Schwierigkeit (levelDifficulty) addiert, damit man noch weiter an der Schwierigkeit des Levels drehen kann.

Nachdem entschieden ist was für eine Entität gespawned werden soll, wird die Länge ermittelt. Dafür wird die minimal Länge der jeweiligen Entität herangezogen sowie eine 50% Chance für ein weiteres Körperelement bis hin zu dem maximal Wert der jeweiligen Entität. Sollte einmal die 50% Chance versagen, dann wird auch nicht mehr weiter versucht neue Körperelemente zu generieren.

Mit all den jetzt vorhandenen Informationen wird ein SpawnObject erstellt. Dieses übernimmt die Verantwortung die richtige Instanz der Entität zu erstellen und zu sagen welches Körperelement als nächstes gespawned werden muss.

#### 3.1.7.5 SpawnObject

Das SpawnObject ist eine Hilfsklasse, um die Entscheidung welches Körperelement von einer Entität gespawned werden soll zu kapseln. Sie besteht immer nur solange bis alle Körperelemente der Entität auf dem Spielfeld sind. Danach wird das Objekt verworfen und das jeweilige BrainSpawn erstellt eine neue Instanz für die nächste Entität oder Freiraum.

Das SpawnObject selber besteht nur aus

- einer Referenz der Reihe. Diese wird benötigt um die Startkoordinaten zu setzen. (row)
- der ID der Entität. (id)
- der Länge der Entität, die durch das BrainSpawn entschieden wurde. (length)
- einem Index, der angibt welches Körperelement als nächstes ist. (currentIndex)
- einer StaticEntität um die jeweilige Entität zu halten welche erstellt wird. Näheres zu der Hierarchie der Entitäten gibt es im Abschnitt 3.1.3. (entity)

Das SpawnObject stellt 2 Methoden dem BrainSpawn zur Verfügung.

Die ”hasNext()”Methode gibt eine Boolean zurück der aussagt ob es noch etwas zu spawnen gibt.

```
1           bool hasNext()
2           {
3             return (this._currentIndex < (this._length));
4           }
5
```

Listing 3.5: SpawnObject - hasNext Method

Die ”nextPart()”Methode gibt etwa ein StaticEntity Element der Entität zurück oder wenn es sich um einen Freiraum handelt null.

```
1           StaticEntity nextPart()
2           {
3             if(_entity != null)
4             {
5               StaticEntity result =
6                   _entity.getBodyPartList()[_currentIndex++];
7               return result;
8             }
9             else
10              {
11                  _currentIndex++;
12                  return null;
13              }
14          }
15
```

Listing 3.6: SpawnObject - nextPart Method

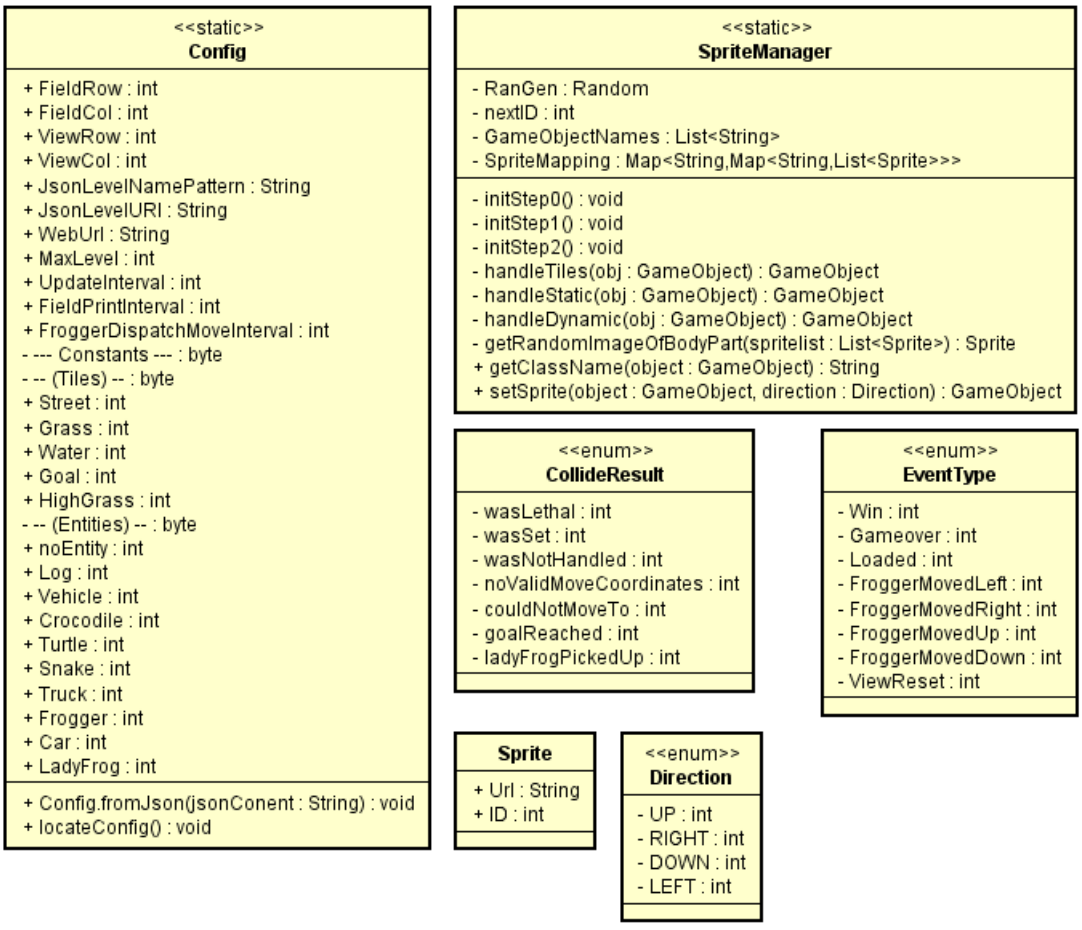


Abbildung 3.10: UML von den Hilfsklassen

## 3.2 View

Die Darstellung wird durch das HTML Dokument erreicht welches, den Kern der View darstellt und mit Hilfe der FroggerView manipuliert. Das Aussehen der einzelnen Elemente der Oberfläche wird durch CSS geregelt und ebenfalls durch die FroggerView manipuliert.

### 3.2.1 HTML-Dokument

Im folgenden ist die Basis Darstellung des HTML Dokuments zu sehen, bevor es durch die FroggerView manipuliert wurde. Während der Spielzeit wird dauerhaft der DOM Tree manipuliert und die Tabelle zu erstellen und zu aktualisieren. Dafür ist die Tabelle mit der ID 'gameplayField' vorgesehen, hierin wird das Spielfeld erstellt. Ebenfalls wird die Tablle mit der ID 'infoTable' immer wieder aktualisiert, da diese die Spielübersicht darstellt. Hier sind Informationen zu Level, Punkte, Leben und wie viele LadyFrog's man eingesammelt hat ersichtlich.

Die DIVs mit den IDs 'menuArea' und 'flashMessageContainer' haben besondere Bedeutung in dem Sinne, dass hier während der Laufzeit Buttons und Nachrichten erzeugt werden und ggf. ein- oder ausgeblendet werden.

```
<html>
<head>
  [...]
    <title>Frogger</title>
    <link rel="stylesheet" href="styles.css">
    <script defer src="main.dart" type="application/dart"></script>
    <script defer src="packages/browser/dart.js"></script>
</head>
<body>
<div id="gameArea" class="gameArea CenterPos">
    <div id="menuArea" class="menuArea CenterPos"></div>

    <div id="flashMessageContainer" class="flashMessageContainer
    Hide CenterPos">
        <div id="flashMessage" class="flashMessage CenterPos">

        </div>
    </div>

    <div id="fieldDiv" class="fieldDiv CenterPos">
        <table id = "gameplayField" class="gameplayField CenterPos"
    ></table>
    </div>

    <div id="infoArea" class="infoArea Hide">
        <table class="infoTable">
            <tr>
                <th id="levelCell" class="tg-031e">Level</th>
                <th class="tg-031e">Lives</th>
                <th class="tg-031e">Score</th>
                <th class="tg-yw4l">Collected LadyFrogs</th>
            </tr>
            <tr>
                <td id="levelNameCell" class="tg-031e"></td>
                <td id="lifeCell" class="tg-031e"></td>
                <td id="scoreCell" class="tg-031e"></td>
                <td id="collectedCell" class="tg-yw4l"></td>
            </tr>
        </table>
    </div>
</div>
</body>
</html>
```

Listing 3.7: Index - Basisdokument

### 3.2.2 FroggerView

Die FroggerView dient als Schnittstelle zum HTML und CSS Dokument und ist nur vom Controller (siehe Abschnitt 3.3) benutzbar. Es werden neben Manipulationen auf dem DOM Tree weitere wichtige Methoden für den Controller bereit gestellt.

- Die render(GameInstance instance) Methode aktualisiert die Oberfläche anhand der übergebenen GameInstance.
- Die updateHud(GameInstance instance) Methode aktualisiert die Spielinformationen auf der Oberfläche.
- Die generateField(int fieldX, int fieldY) Methode erstellt die Tabelle für das Spielfeld selbst, wie im Abschnitt 3.2.1 erwähnt.

Weitere Besonderheit die dem Controller zu Verfügung gestellt wird ist, dass für die Methoden

- showMenu(Function func())
- setFlashMessage(String message, Function func())

eine Funktion übergeben werden kann, die die Funktionalität des Buttons, welcher durch die Methoden selbst erstellt wurden, zu belegen. Dies musste so geregelt werden, damit man der Erstellung der Buttons dem Controller noch die Möglichkeit geben kann auf das OnClick Event der Buttons zu reagieren.

Eine Besonderheit des FroggerView ist, dass sie eine Hilfsklasse Camera (Abschnitt 3.2.3) verwendet, mit der man nur einen Teilausschnitt des gesamten Spielfeldes, welches durch das Model geben ist anzeigen kann. Für die Camera Klasse werden 2 Funktionen bereit gestellt.

- Function printFieldCell([...])
- Function printEntitiesStackInTile([...])

Die beiden Methoden dienen dazu nur den sichtbaren Bereich der Camera zu aktualisieren, so werden z.B. Spielfelder die außerhalb der Camera sind nicht aktualisiert in der View sondern nur intern im Model.

### 3.2.3 Camera

Die Camera Klasse zeigt einen Teilausschnitt der View um einen übergebenen Fixpunkt auf dem Spielfeld an.

Dafür wird bei der Erstellung der Camera folgendes benötigt:

- Die Breite des Ausschnittes (viewSizeX)
- Die Höhe des Ausschnittes (viewSizeY)
- Ein Fixpunkt in der Reihe, an dem sich die Camera ausrichtet. (originX)
- Ein Fixpunkt in der Spalte, an dem sich die Camera ausrichtet. (originY)
- Referenz auf das Model, welches er von der FroggerView bekommt. (instance)
- Referenz auf den DOM Tree, welcher er von der FroggerView bekommt. (viewSection)

Die Camera bietet 3 Methoden der FroggerView sowie dem Controller an.

- void moveInDirection(Direction dir) verschiebt die Camera in die übergebene Richtung.
- void resetOrigin() setzt den Fixpunkt der Camera wieder auf dem Frogger zurück.
- void applyCamera(Function func(int row, int col, Tile currentTile, HtmlElement element)) übernimmt eine Funktion, die auf dem Camera Bereich ausgeführt wird.

## 3.3 Controller

Der Controller ist die Schnittstelle zwischen dem Model und der View. Der Controller beinhaltet 3 periodische Timer, mit zwei von Ihnen wird das Model in einen neuen Zustand überführt und der letzte aktualisiert die View periodisch.

Anfänglich wurden 2 separate Controller für Eingaben erstellt die wiederum ein eigenes Interface implementieren sollten. Allerdings unterstützt Dart das Konzept von Interfaces nicht, da man es genau so gut mit einer abstrakten Klasse umsetzen kann. Im Verlauf der Entwicklung wurde die Erkenntnis erlangt, dass keine Separierung notwendig ist. Als Resultat dessen ist die Klasse IControlManager überflüssig geworden und der damalig noch bestehende KeyBoardController ist in den TouchController mit integriert wurden. Da wir auf die Umarbeitung des FroggerController's verzichten wollten, besteht der IControlManager weiterhin.

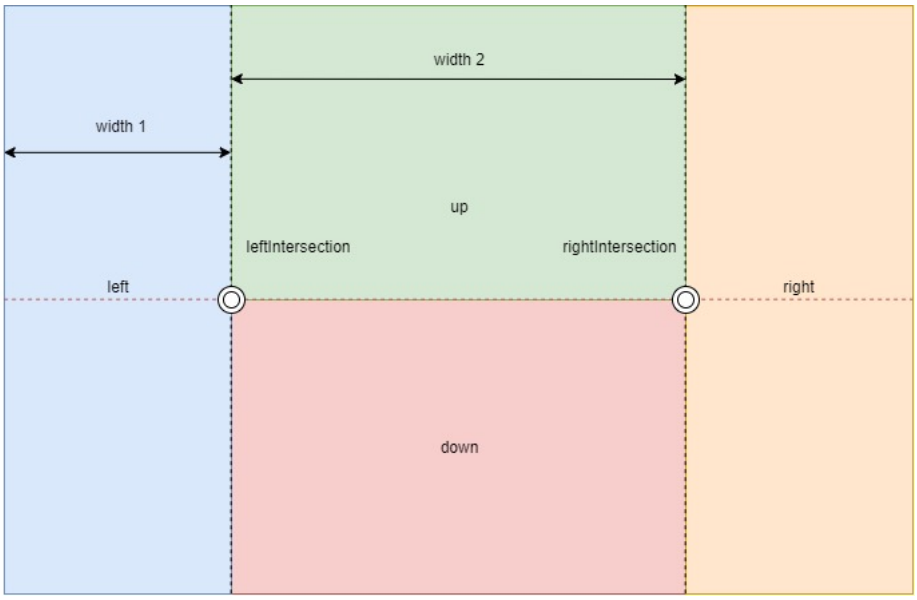


Abbildung 3.11: Touchcontrol on Mobile Devices

### 3.3.1 TouchController

Das Steuerungsschema ist nach folgenden Gesichtspunkten umgesetzt:

- es wird von einer statischen Pixel basierten Größe des Spielfelds ausgegangen.
- anhand der Pixel basierten Höhe und Breite des Spielfelds werden die Touch Bereiche ermittelt.

Das “Steuerkreuz” ist im Spiel transparent und erstreckt sich über das gesamte Spielfeld. Es wird Abbildung 3.11 im weiteren zur Anschauung verwendet.

Die Variable _upDownFieldWidthRatio gibt das Verhältnis der Gesamtbreite ”width2”genannten Strecke an. Die Breite der Variable “width1” wird den restlichen vorhandenen Platz einnehmen.
Um eine Berührung einem der 4 Felder und damit auch direkt einer der 4 möglichen Richtungen zuzuordnen gehen wir folgendermaßen vor:

1. Wir berechnen die beiden, left- und rightIntersection Punkte mithilfe der _upDownFieldWidthRatio neu, für den Fall dass die Oberfläche vom Nutzer geresized wurde.

2. Prüfen ob die berührten Koordinaten links von dem Punkt "leftIntersection"liegt. Ist dies der Fall dann wird die Eingabe als "nach links gehenümgesetzt.

3. Prüfen ob die berührten Koordinaten rechts von dem Punkt "rightIntersection"liegt. Ist dies der Fall dann wird die Eingabe als "nach rechts gehenümgesetzt.

4. Prüfen ob die berührten Koordinaten rechts der "leftIntersectionünd links der "rightIntersection", sowie über den beiden genannten Punkten liegt. Ist dies der Fall dann wird die Eingabe als "nach oben gehenümgesetzt.

5. Trifft keine der in Punkt 2-4 genannten Punkte zu, dann muss der Spieler den unteren-mittleren Teil des Steuerkreuzes berührt haben und wir interpretieren die Eingabe als "nach unten gehen"

Es gibt im Touchcontroller zwei Hilfsfunktionen isStartPointOverEndPoint und isStartPointLeftOfEndPoint, diese sind dafür da um boolesche Auswertungen zu abstrahieren, damit oben genannten Überprüfungen einfacher zu lesen sind. Da die Funktionen als private markiert sind, wurde auf weitere Erläuterung verzichtet. Aufgrund ihrer simplen Form wäre dies sowieso nicht nötig.

# 4. Level- und Parametrisierungskonzept

## 4.1 Levelkonzept

Das Levelkonzept ist über Json Files realisiert, wobei jedes einzelnes Level eine eigene Json Datei besitzt. So kann man einfach neue Level erstellen in dem man eine neue Json Datei anlegt und in der Config Json das maximal Level erhöht. Im Model finden sich die Parameter in der Level Klasse (Abschnitt 3.1.2.2) wieder.

Die Parameter sind größtenteils selbsterklärend, und eine nähere Erklärung lässt sich zur Klasse BrainSpawn im Abschnitt 3.1.7.4 herauslesen, da dort das meiste heraus extrahiert wird. Was noch zu erwähnen ist, ist dass alle Arrays die in der Json vorkommen mindestens so viele Einträge haben müssen, wie es Reihen auf dem Spielfeld gibt. Die Codierung der Abstände, (entitiesSpacing) sowie der Längen (entititesLength) sind als String in folgender Form codiert "X-Y". Hierbei ist X der minimal Wert und Y der maximal Wert für die jeweilige Entität (entities) an selbiger Stelle im Array.

```
{
  "level":
  {
    "levelName":"Hot Summer Street (LVL 1)",
    "updateRate":[0,5,5,3,0,20,45,30,0],
    "tileType":[2,1,1,1,2,3,3,3,4],
    "entities":[[0],
                [0],
                [0],
                [0],
                [0],
                [0],
                [0],
                [0],
                [0]],
    "entitiesSpacing":[ "0-0",
              "2-4",
              "2-3",
              "6-10",
              "0-0",
              "2-2",
              "3-3",
              "1-1",
              "0-0"],
    "entitiesLength": [["0-0"],
                       ["2-3"],
                       ["2-3"],
                       ["2-3"],
                       ["0-0"],
                       ["2-3"],
                       ["2-3"],
                       ["2-3"],
                       ["0-0"]],
    "directions":[0,1,0,1,0,1,0,1,0],
    "goalAmount":2,
    "spawnChanceLadyFrog": 0,
    "levelDifficulty": 0
  }
}
```

Listing 4.1: Level - Level Json

## 4.2 Parameterisierungskonzept

Hier soll noch einmal gezeigt werden, dass man das Spiel über eine Json neu einstellen kann. Näheres zu den einzelnen Einstellungsmöglichkeiten kann man im Abschnitt 3.1.2.1 nachlesen.

```
{
  "config":
  {
    "FieldRow" : 9 ,
    "FieldCol" : 19 ,
    "ViewRow" : 7 ,
    "ViewCol" : 17 ,

    "Score_JumpInDirection" : 10 ,
    "Score_Goal" : 50 ,
    "Score_LadyFrog" : 200 ,
    "Score_TimeLeftPerUnit" : 10 ,
    "Score_CompletedLevel" : 1000 ,

    "JsonLevelNamePattern" : "Level_" ,
    "maxLevel": 4,

    "updateInterval" : 50 ,
    "fieldPrintInterval" : 50 ,
    "froggerDispatchMoveInterval" : 20
  }
}
```

Listing 4.2: Config - Config Json

# 5. Nachweis der Anforderungen

## 5.1 Nachweis der funktionalen Anforderungen

| ID | Kurztitel | Erfüllt | Teilw. erfüllt | Nicht erfüllt | Erläuterung |
|---|---|---|---|---|---|
| AF-1 | Einplayer Game | X | | | Das Spiel ist nur mit einer Person spielbar zur Zeit. |
| AF-2 | 2D Game | X | | | Es spielt nur auf einen 2D Raster und besitzt keine 3te Dimension. |
| AF-3 | Levelkonzept | X | | | Level Jsons können beliebig bearbeitet werden ohne das das Spiel neu kompiliert werden muss. |
| AF-4 | Parametrisierungskonzept | X | | | Config Json kann beliebig bearbeitet werden ohne das das Spiel neu kompiliert werden muss. |
| AF-6 | Desktop Browser | X | | | Ist in Desktop Browser Spielbar. Wurde in Chrome, Firefox, Opera sowie Safari getestet. |
| AF-7 | Mobile Browser | | X | | Ist nur teilweise in Mobile Browser Spielbar, angesichts bei der Menge auch nicht 100% möglich. Wurde unter Android 4.4 und 7.0 mit den nativen Browsern getestet. Ebenfalls mit Chrome auf einem Android. |

Tabelle 5.1: Tabelle mit den Nachweis der funktionalen Anforderungen

## 5.2 Nachweis der Dokumentationsanforderungen

| ID | Kurztitel | Erfüllt | Teilw. erfüllt | Nicht erfüllt | Erläuterung |
|---|---|---|---|---|---|
| D-1 | Dokumentationsvorlage | X | | | Dokumentation richtet sich nach Referenzdokumentation des Snake Spieles. |
| D-2 | Projektdokumentation | X | | | Diese Dokumentation deckt alles vorhandene im Projekt ab. Diese werden mindestens in ihren Prinzipien erläutert und ggf. durch Grafiken und/oder Quelltexte unterstützt. |
| D-3 | Quelltextdokumentation | X | | | Fast ausschließlich alles wurde durch Inline - Kommentare oder JSDoc erläutert. |
| D-4 | Libraries | X | | | Wir haben keine externen Libraries benutzt. |

Tabelle 5.2: Tabelle mit den Nachweis Dokumentationsanforderungen

## 5.3 Nachweis der Einhaltung technischer Randbedingungen

| ID | Kurztitel | Erfüllt | Teilw. erfüllt | Nicht erfüllt | Erläuterung |
|---|---|---|---|---|---|
| TF-01 | No Canvas | X | | | Das Spiel funktioniert allein durch klassische HTML Tags und DOM-Tree Manipulation. |
| TF-02 | Levelformat | X | | | Es sind ladbare Level Jsons vorhanden. Die Level besitzen ein einheitliches Format. |
| TF-03 | Parameterformat | X | | | Es ist eine ladbare Config Json vorhanden, die durch das Format der Config Klasse definiert ist. |
| TF-04 | HTML + CSS | X | | | Die Visualisierung basiert allein auf HTML und CSS bzw. deren Manipulation. |
| TF-05 | Gamelogic in Dart | X | | | Die komplette Spiellogik wurde durch Dart realisiert. |
| TF-09 | Browser Support | X | | | Das Projekt ist sowohl in Dartium als auch in Javascript Form auf einen Webserver ausführbar und funktionsfähig. |
| TF-10 | MVC Architektur | X | | | Das Spiel ist durch die Vorlagen im MVC Konzept entworfen und implementiert worden. |
| TF-11 | Erlaubte Pakete | X | | | Wir haben ausschließlich die durch die Vorgabe erlaubten Dart Libraries verwendet. |
| TF-12 | Verbotene Pakete | X | | | Es sind keine Pakete, außer den erlaubten genutzt worden. Siehe pubspec.yaml der Implementierung |
| TF-13 | No Sound | X | | | Das Spiel besitzt keinen Sound. |

Tabelle 5.3: Tabelle mit den Nachweis Technische Randbedingungen

## 5.4 Verantwortlichkeiten im Projekt

| Komponente | Detail | Asset | Sven Möller | Joshua Krieger |
|---|---|---|---|---|
| Model | Config | Config.dart | V | U |
| | Level | model\Level.dart | | V |
| | Sprite | model\Sprite.dart | V | |
| | SpriteManager | model\SpriteManager.dart | V | |
| | GameInstance | model\GameInstance.dart | V | U |
| | EventType | model\EventType.dart | V | |
| | Row | model\Row.dart | V | |
| | RowSpawner | model\RowSpawnerFusion.dart | V | |
| | BrainSpawn | model\RowSpawnerFusion.dart | V | |
| | SpawnObject | model\RowSpawnerFusion.dart | V | |
| | Field | model\Field.dart | V | |
| | Direction | model\gameObjects\Direction.dart | | V |
| | CollideResult | model\gameObjects\GameObject.dart | | V |
| | GameObject | model\gameObjects\GameObject.dart | | V |
| | Entity | model\gameObjects\Entity.dart | | V |
| | Tile | model\gameObjects\Tile.dart | | V |
| | StaticEntity | model\gameObjects\StaticEntity.dart | | V |
| | DynamicEntity | model\gameObjects\DynamicEntity.dart | | V |
| | Frogger | ... \dynamicEntities\Frogger.dart | | V |
| | LadyFrog | ... \dynamicEntities\LadyFrog.dart | | V |
| | Car | ... \staticEntities\Car.dart | | V |
| | Crocodile | ... \staticEntities\Crocodile.dart | | V |
| | Log | ... \staticEntities\Log.dart | | V |
| | Snake | ... \staticEntities\Snake.dart | | V |
| | Truck | ... \staticEntities\Truck.dart | | V |
| | Turtle | ... \staticEntities\Turtle.dart | U | V |
| | Vehicle | ... \staticEntities\Vehilce.dart | | V |
| | GoalTile | ... \Tiles\GoalTile.dart | | V |
| | GrassTile | ... \Tiles\GrassTile.dart | | V |
| | HighGrassTile | ... \Tiles\HighGrassTile.dart | | V |
| | StreetTile | ... \Tiles\StreetTile.dart | | V |
| | WaterTile | ... \Tiles\WaterTile.dart | | V |
| View | FroggerView | view\FroggerView.dart | U | V |
| | Camera | view\Camera.dart | U | V |
| | Index | view\index.html | V | |
| | Style | view\style.css | V | U |
| Controller | FroggerController | controller\FroggerController.dart | V | U |
| | TouchControler | controller\touchControler.dart | | V |
| | IControlManager | controller\IControlOutput.dart | | V |

| | | | | |
|---|---|---|---|---|
| Dokumentation | Dokumentation | doc\FroggerDokumentation.pdf | V | U |
| | Dokumentation | doc\FroggerDokumentation.tex | V | |

Tabelle 5.4: Tabelle mit der Projektverantwortlichkeiten