Dokumentation für das Spiel RabbitRinth im Modul "Web-Technologie-Projekt"

Marc-Niclas Harm und Bengt Claas Rhodgeß

Sommersemester 2018

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Diese Dokumentation entstand begleitend zur Veranstaltung "Web-Technologie-Projekt", deren Ziel es war an dem Beispiel einer Spieleentwicklung die Technologien HTML, DOM-Tree und Dart zu erlernen und praktisch anzuwenden.

Dabei war der Entwicklung eine Analyse der Anforderungen vorausgegangen, die einen groben Rahmen für die eigentliche Implementation vorgab.
Folgende Punkte waren daher maßgeblich bei den Überlegungen zum Spiel:

- Erstellung des Spiels als Single-Player-Game in einer Single-Page-App
- Balance zwischen technischer Komplexität und Spielkonzept
- Das Spiel soll DOM-Tree-basiert sein
- Das anvisierte Target Device: SmartPhone
- Wir folgen dem Mobile First Prinzip
- Das Spiel muss schnell und intuitiv erfassbar sein und Spielfreude erzeugen
- Das Spiel muss ein Levelkonzept vorsehen
- Ggf. erforderliche Speicherkonzepte sind Client-seitig zu realisieren
- Dokumentation

Gemessen an diesen Anforderungen und den im Projekt vorgegebenen Meilensteinen beginnen wir in dieser Dokumentation zunächst mit der detailierteren Aufstellung der Anforderungen und der Konzeptionierung des Spiels und widmen uns in späteren Kapiteln den Implementationsdetails.

2. Anforderungen und abgeleitetes Spielkonzept

2.1. Anforderungen

Ihr Spiel soll folgende Anforderungen erfüllen. Sie sollen dabei nachvollziehbar dokumentieren, wie diese Anforderungen durch ihre Realisierung umgesetzt werden. Das Spiel soll folgende in Tabelle 1 aufgeführten funktionalen Anforderungen, Dokumentationsanforderungen und technischen Randbedingungen erfüllen.

Id	Kurztitel	Anforderung
AF-1	Single-Player-Game als Single-Page-App	• Konzeption als Ein-Spieler-Game • Konzeption als Single-Page App • Bereitstellung als statische Webseite • relative Adressierung der Spielressourcen zueinander
AF-2	Balance zwischen technischer Komplexität und Spielkonzept	• Entwicklung eines interessanten Spielkonzepts • das Spielkonzept soll eine ähnliche Komplexität wie die Spiele der Hall of Fame aufweisen • das Spiel soll schnell und intuitiv erfassbar sein.
AF-3	DOM-Tree-basiert	• Spiel soll dem MVC-Prinzip folgen • Spiel soll den Dom-Tree als View nutzen • Canvas-basierte Spiele sind nicht erlaubt.
AF-4	Target Device: SmartPhone	• Spiel ist für die Smartphone-Bedienung konzipiert • Entsprechende Limitierungen sind zu berücksichtigen • Das Spiel soll mit HTML5 im mobilen Browser auf den genannten Plattformen spielbar sein.

AF-5	Mobile First Prinzip	• Spiel ist bewusst für Smartphones konzipiert • Spiel soll auch auf Tables und Desktop-Pcs spielbar sein. Einschränkungen sind zu minimieren. • Sinvolle Nutzung typisch mobiler Interaktionen (Swipen, 3D-Lage usw.) • Keine Übertragung typischer Desktop-Bedienung auf Mobile • Keine sinnlose Nutzung mobiler Interaktionen
AF-6	Das Spiel muss schnell und intuitiv erfassbar sein und Spielfreude erzeugen	• Spiel muss schnell und intuitiv erfassbar sein • Spiel muss Spielfreude erzeugen
AF-7	Das Spiel muss ein Levelkonzept vorsehen	• Es ist ein steigender Schwierigkeitsgrad über mindestens 7 Level vorzusehen. • Level sollen deklarativ in Textdateien beschrieben werden können • Spiel soll Level nachladen können, sodass Level nachträglich ergänzt und abgeändert werden können ohne die Spielprogrammierung anzupassen.

AF-8	Ggg. Erforderliche Speicherkonzepte sind Client-seitig zu realisieren	• Durch das Spiel gesammelte Daten sollen auf den Geräten bleiben • Aufgrund des Demonstrationscharakters auf Messen dürfen keine zentralen Server für Highscores etc. erforderlich sein • ggf. Erforderliche stateful solutions sind mittels client-seitiger Storage-Konzepte zu lösen (z.B. local storage)
AF-9	Dokumentation	• Das Spiel muss nachvollziehbar dokumentiert sein. • Dokumentation analog dem SnakeGame.

Tabelle 1: Anforderungen

2.2. Spielkonzept

Das Spiel RabbitRinth basiert auf der Idee eines Hasens/Kaninchens, das sich kurz vor Einbruch der Dunkelheit noch im Freien – und schlimmer – weit von seinem rettenden Bau entfernt befindet. Zwischen dem Startpunkt (S) des Kaninchens (K) und seinem Bau (O) befinden sich als Hindernis dichte Hecken (X), die ein Labyrinth darstellen, dass es zu durchqueren gilt.

Der Spieler bewegt das Kaninchen durch die 3D-Lageerkennung seines Smartphones und hat die Aufgabe das Tier durch das Heckenlabyrinth bis hin zu dem rettenden Loch zu steuern. Die Bewegungen erfolgen waagerecht und senkrecht, nicht aber diagonal. Dabei steht ihm bis zum Einbruch der Dunkelheit nur eine gewisse Zeit zur Verfügung.

Die Schwierigkeit des Spiels kann je nach Level in zwei Punkten variieren.

- Einerseits nimmt mit jedem Level die Komplexität des zu durchquerenden Labyrinths zu. Der Spieler stößt auf mehr Sackgassen und Kurven, die er umsteuern muss.
- Andererseits gibt es die Möglichkeit einen "Hart"-Modus auszuwählen, in dem die Zeit, die dem Spieler in jedem Level zur Verfügung steht mehr oder weniger drastisch zu reduzieren.

Ein Level ist beendet, wenn

- das Kaninchen vor Ablauf der Zeit seinen Bau erreicht
- oder die Zeit abläuft, ohne dass das Kaninchen am Bau angelangt ist.

Das Spielkonzept von RabbitRinth lässt sich mit folgenden Überlegungen leicht variieren:

- Anpassung der "Dunkelheit" über die Kontrasteinstellungen des Spielfeldes

- Das Kaninchen hat eine begrenzte Anzahl an Sprüngen mit denen es eine Hecke überwinden kann. Dies funktioniert, indem man auf dem Bildschirm auf die zu überwindende Hecke tippt.
- Einbau einer KI, die im Labyrinth Gegner auf das Kaninchen losgehen lässt, sobald es in eine bestimmte Sichtweite gerät. In dem Fall wäre auch das "Erwischt werden" das Ende eines Levels.

Restzeit: 15 Sekunden					
S					
X	X		X	X	X
					OK

Abbildung 1: Kaninchen hat den Bau erreicht

Restzeit: 0 Sekunden					
					S
X		X	X	X	X
X					
X	X	X	X	X	
O	X	K			
			X	X	X

Abbildung 2: Kaninchen hat den Bau nicht vor Ablauf der Zeit erreicht

3. Architektur und Implementierung

Abbildung 3 zeigt die Architektur des Spiels RabbitRinth im Überblick. Diese Architektur orientiert sich – wie in der Aufgabenstellung gefordert – an dem MVC (Model, View, Controller) – Prinzip. Aus softwaretechnischer Sicht gliedert die Spiellogik sich in mehrere Kompetenten (Klassen) mit spezifischer funktionaler Verantwortlichkeit. Der Kern der Spielsteuerung befindet sich dabei in dem Controller (Klasse: *control*). Konkret kann der Controller

- Nutzerinteraktionen (dazu gehören Touch, 3D-Lageerkennung sowie Maus- und Tastatureingaben) sowie
- Zeitsteuerung (im vorliegenden Spiel insbesondere für die Bewegung des Hasen, sowie des Fuchses)

erkennen und in entsprechende Modelinteraktion umsetzen. Er wird im Detail in Abschnitt 3.3. erläutert.

Die Klasse *view* kapselt den DOM-Tree und bietet dem Controller Möglichkeiten den DOM-Tree zu manipulieren, damit sich ändernde Spielzustände im Browser angezeigt werden können. Die View wird näher im Abschnitt 3.2. erläutert.

Konzeptionell wird RabbitRinth in einem Model abgebildet. Dieses zeigt sich komplexer als die vorgenannten Klassen und untergliedert sich in mehrere logische Entities, die sich aus dem Spielkonzept unter 2.2. ableiten und im Abschnitt 3.1. erläutert werden.

Grundsätzlich ist keine Anbindunge an zentrale Server erlaubt, um das Spiel auf etwaigen Messen den Interessenten jederzeit präsentieren zu können. Trotzdem war es notwendig eine Methode der Speicherung zu finden, um bereits erreichte Spielstände nicht einfach verfallen zu lassen.
Dies wird über einen Local Store im Rahmen der Klasse *model* realisiert.

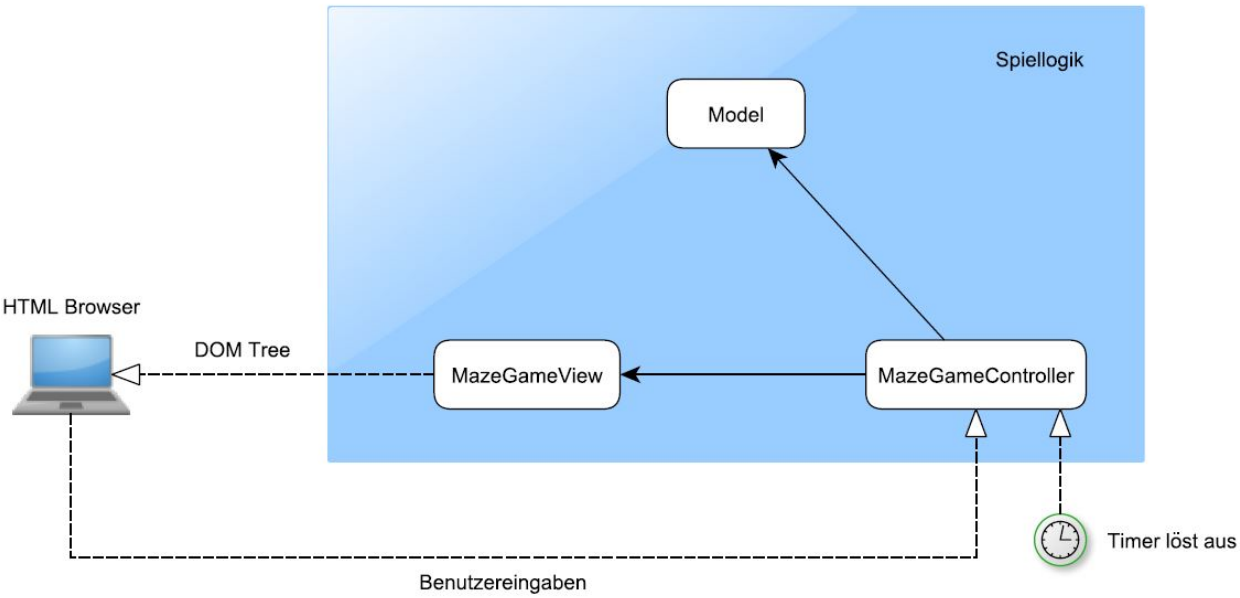

Abbildung 3: Architektur

3.1. Model

Aus dem Spielkonzept in Abschnitt 2.2. wurden zunächst ein Spielobjekt (G*ameObject*) und darauf basierend eine Kreatur (*Creature*) abgeleitet. Von diesem widerrum erbt das Kaninchen (R*abbit*) und die Feind-Klasse (*Enemy*), aus der sich der Fuchs (*Fox*) ableitet. Neben diesen Klassen findet sich hier die Klasse *NoSleep,* in der verhindert wird, dass der Bildschirm des genutzten Smartphones sich in den Energiesparmodus begibt. Die Klassen *Tile* und *TileType* dagegen beschreiben das Terrain auf dem Spielfeld. In der Klasse *MazeGameModel* finden sich die grundlegenden Mechaniken, wie beispielsweise die Initialisierung des Spiels nach dem Druck auf einen Knopf im Hauptmenü.

3.1.1. MazeGameModel

Der Controller interagiert in diesem Unterabschnitt nur mit dem MazeGameModel und nicht mit den dahinter liegendenen Entities. So ist es möglich im späteren Verlauf der Entwicklung noch weitere Entities oder Varianten derselben hinzuzufügen, ohne Gefahr zu laufen weitreichende Änderungen am Controller vorgehen zu müssen. Folgende Attribute sind Teil der *MazeGameModel* Klasse:

- *_level* stellt den Level als solchen bereit und speichert ihn für den Spielablauf zwischen.
- *levelNumber* speichert die Nummer des aktuellen Levels zur späteren Nutzung
- *_rabbit* hält das rabbit-Objekt
- *_enemies* enthält eine Liste der im aktuellen Level befindlichen Gegner.
- *_speedPowerups* hält eine Liste der im aktuellen Level befindlichen Speed-Powerups.
- *_gamestate* enthält symbolisch den aktuellen Aktivitätsstatus des Spiel. So kann in Erfahrung gebracht werden, ob das Spiel läuft oder beispielsweise pausiert ist.

Ein MazeGameModel-Objekt

- kann mit Hilfe eines Konstruktors erzeugt werden. Dabei wird zunächst die Nummer des ersten spielbaren Levels übergeben.
- lädt mit der Methode *loadCurrntLevel()* den entsprechenden Level mitsamt allen relevanten Levelinformationen (Gegner/Startpunkt, etc) ein.
- kann mittels der Funktion *get level* den aktuellen Level zurückgeben.
- *get stopped* und *get running* erlauben den aktuellen Aktivitätsstatus in einem binären Schema festzustellen.
- *get speedPowerups* gibt eine Liste der aktuell vorhandenen Speed-Powerups zurück.
- *get enemies* gibt eine Liste der aktuellen Feinde zurück
- *get rabbit* gibt das aktuelle *Rabbit*-Objekt zurück
- die Funktionen *start()* bzw. *stop()* starten/stoppen das Spiel indem sie seinen game-state entsprechend verändern.

3.1.2. GameObject

Das *GameObject* bildet die Grundlage für die weiteren Objekte des Spiels. Es verfügt über die Attribute

- *position*, die die Position des GameObjects abspeichert und
- *_type* die den Typen des GameObjects sichert

Über einen Konstruktor können weitere *GameObjects* erzeugt werden. Daneben besitzt das *GameObject* zwei Methoden bzw. Funktionen.

- *set type(String newType)*, mit deren Hilfe man dem *GameObject* einen eigenen Typen geben kann einerseits

- *get type*, mit der man sich den Typen des aktuellen *GameObjects* ausgeben kann andererseits.

3.1.3. Creature

Jede Kreatur im Spiel verfügt über eine Anbindung an das *MazeGameModel* und über das Attribut collisionType, das später genutzt wird, um die Art der Kollision auf dem Spielfeld zu untersuchen.

Die Klasse Creature verfügt über zwei Attribute:

- *belowGameObject* speichert das Feld zwischen auf den eine Kreatur gelaufen ist. Auf diese Weise kann es später wieder in der View hinzugefügt werden.
- *previousPosition* speichert die Position vor Ausführung der Bewegung ab.
- *direction* hält die Richtung, in die die Kreatur sieht, um sie einen linearen Weg verfolgen zu lassen und zu kontrollieren in welche Richtung sie laufen soll, wenn sie in der aktuellen Richtung auf ein Hindernis stößt.

Die Klasse Creature erweitert das *GameObject* und verfügt über einen eigenen Konstruktor und die Methoden

- *_updatePositions(int newRow, int newCol)* um die jeweilige Kreatur direkt an einen bestimmten Ort auf dem Spielfeld zu bewegen
- *move(int dRow, int dCol)* um die Kreatur Schritt für Schritt über das Feld zu bewegen. Dabei werden Kollisionen über ihren Switch-Case abgeprüft und entsprechend ausgewertet.
- *onCollideWithTerrain(GameObject collisionObject, int newRow, int newCol)* bildet die Bewegung ab, wenn die Kreatur lediglich auf ein begehbares Bodentile trifft.
- *onCollideWithGoal(GameObject collisionObject, int newRow, int newCol)* tut nichts und muss entsprechend überschrieben werden.
- *onCollideWithFox(GameObject collisionObject, int newRow, int newCol)* tut nichts und muss entsprechend überschrieben werden.
- *onCollideWithRabbit(GameObject collisionObject, int newRow, int newCol)* tut nichts und muss entsprechend überschrieben werden.
- *onCollideWithWall(GameObject collisionObject, int newRow, int newCol)* tut nichts und muss entsprechend überschrieben werden.
- *onCollideWithSpeedPowerup(GameObject collisionObject, int newRow, int newCol)* sorgt dafür, dass der Hase beim Überlaufen des Powerups mit dessen Effect verknüpft wird.
- *LEFT, RIGHT, UP* und *DOWN* in der inneren Klasse *Direction* lesen sich als ENUM für die Verwendung durch die anderen Methoden. Ihre Methode *getRowAndColIndication(final String direction)* gibt die Koordinatenveränderung beim Anwählen einer bestimmten Richtung zurück, während die Funktion *get types* alle möglichen Richtungen in einer Liste zurückgibt.

Zuletzt findet sich hier die Definition der UnknownDirectionException.

3.1.4. Enemy

Die Klasse *Enemy* erweitert die Klasse *Creature*. Sie verfügt über die Attribute

- *_movementType* beschreibt, ob der Feind sich vertikal oder horizontal bewegt oder ob er Jagd auf das Kaninchen macht,
- *_speed* zweiteres beschreibt, wie schnell die Feinde im Spiel sich zu bewegen versuchen,
- *_moveCountdown* beschreibt die Dauer des movement triggers.
- *_moveTimer* ist der Timer, der für die Bewegung der Feinde zuständig ist

und kann über ihren Konstruktor weitere *Enemy*-Objekte erschaffen. Außerdem hält die Klasse eine Instanz des *MazeGameModels*.

Die Klasse verfügt darüber hinaus über die (eventuell manipulierten) Methoden

- *onCollideWithRabbit(GameObject collisionObject, int newRow, int newCol)* beendet den Level und löst eine Niederlage aus.
- *startMoving(final MazeGameModel game)* startet die Bewegung des *Enemy*, indem sie den Timer startet
- *stopMoving()* beendet die Bewegung des *Enemy*, indem es den Timer beendet
- *makeMover()* um festzulegen, wie der Gegner sich bewegt. Über ein Switch-Case Konstrukt wird ausgewertet, ob der Gegner sich horizontal oder vertikal bewegt und in welche der verfügbaren Richtungen er dabei zuerst aufbricht. Außerdem kann hier die Verfolgung auf Sicht festgestellt werden.
- *moveHorizontalFirstRight(), moveHorizontalFirstLeft, moveVerticalFirstUp, moveVerticalFirstDown* und *moveIfRabbitInSight()* zur Ausführung dieser Bewegungsoptionen.

Die innere Klassen *EnemyMovementType* definiert als Hilfen entsprechende Konstanten, die das Navigieren für den Enemy wie ein Enum vereinfachen. Ihre *get types*-Funktion gibt eine Liste der möglichen Bewegungstypen zurück.

Zuletzt wird die UnknownMovementTypeException definiert.

3.1.5. Fox

Die Klasse *Fox* erweitert lediglich die Klasse *Enemy*, fügt aber – von einem Konstruktor abgesehen - keine weiteren Methoden oder Attribute hinzu. Der Fuchs bildet damit einen grundlegenden Standardgegner ab, der jederzeit durch weitere Gegnerarten unterstützt werden kann.

3.1.6. Rabbit

Die Klasse *Rabbit* erweitert die Klasse *Creature* und fügt ihr folgende Attribute hinzu:

- *isAbleToMove* speichert ab, ob der Hase sich schon wieder bewegen kann.
- *speed* speichert ab, wann der Hase sich zu bewegen versucht.
- *_moveCountdown* beschreibt die Dauer des movement triggers.

- *_moveTimer* ist der Timer, der für die Bewegung der Feinde zuständig ist

Sie überschreibt folgende Methoden, bzw. Fügt sie erst hinzu:

- *onCollideWithGoal(GameObject collisionObject, int newRow, int newCol)* löst den Gewinn/Abschluss des Levels aus.
- *onCollideWithFox(GameObject collisionObject, int newRow, int newCol)* beendet den Level und löst eine Niederlage aus.
- *startTimer(final MazeGameModel game)* startet die Bewegung des Hasen, indem sie den Timer startet.
- *StopTimer()* beendet die Bewegungen des Hasen indem es den zuständigen Timer stoppt.
- *_resetMovementState()* setzt die Bewegungsmöglichkeit des Hasen zurück, sodass er wieder laufen kann

Über den Konstruktor ist es möglich weitere *Rabbit*-Objekte zu erschaffen.

3.1.7. NoSleep

Die Klasse *NoSleep* verfügt über die Attribute *VIDEO_ENCODED* und *_video.* Beide werden genutzt, um ein Video bereitzustellen, das im Hintergrund des Spiels abgespielt wird, um den Sleep-Mode des SmartPhones zu verhindern. Dazu kommt *_rnd* als Seed für den Timer des Videos.

Die Klasse verfügt dabei über die Methoden

- *NoSleep()* in der dem Video die notwendigen Attribute mitgegeben werden, um unbemerkt vom Nutzer des Spiels zu arbeiten (Source des Videos, mute usw.).
- *enable()* spielt das Video ab.
- *disable()* stoppt das Video.

3.1.8. Tile

Die abstrakte Klasse *Tile* erweitert die Klasse *GameObject*. Dabei werden die Koordinaten des jeweiligen Tiles in der *position* abgespeichert.
Die inneren Klassen *Hedge*, *Terrain*, *Goal* und *Wall* erweitern die Grundklasse jeweils um den *TileType*, sodass damit in der Kollisionsabfrage gearbeitet werden kann.

3.1.9. TileType

Die Klasse *TileType* definiert die Konstanten der einzelnen Entitäten (*HEDGE, TERRAIN, GOAL, RABBIT, FOX, WALL*) und gibt sie in einer Liste per *get types* zurück. Darüber hinaus existiert hier eine innere Klasse die eine *UnknowTileTypeExeption* definiert.

3.1.10. Position

Die Klasse Position beinhaltet alle Informationen, die für die eigentliche Position der verschiedenen GameObjects notwendig sind. Sie beinhaltet die Attribute

- *row* für die Korrdinate der Reihe und
- *col* für die Koordinate der Spalte.

Darüber hinaus beinhaltet die Klasse die Methode *toString()*, um die Position ausgeben zu können und einen Konstruktor, um ein neues Positionsobjekt anlegen zu können.

3.1.11. Level

Die Klasse *Level* repräsentiert die einzelnen Level des Spiels. Sie hält die Attribute

- *name*, in dem der Name des Levels gespeichert wird
- *description*, das die Beschreibung des Levels speichert
- *timeTotal*, in dem festgelegt wird, wie viel Zeit dem Spieler zur Verfügung steht, um den Level abzuschließen
- *timeLeft*, wo zwischengespeichert wird, wie viel Zeit dem Spieler aktuell noch verbleibt um den Level abzuschließen
- *rows* und *cols* die festlegen, wie viele Reihen und Spalten der Level hat
- *gameOver*, das speichert, ob der Spieler das Level bereits verloren hat
- *done*, in dem festgehalten wird, ob der Spieler das Level bereits erfolgreich durchlaufen hat
- *objects*, in dem das Spielfeld in einer Liste von Listen repräsentiert wird.
- *_levelCountdown* speichert die Geschwindigkeit mit der der Countdown des Levels dekrementiert wird.
- *_levelTimer* ist der dazugehörige Timer

Die Klasse verfügt außerdem über die Methoden

- *start()* startet den Countdown des Levels.
- *stop()* hält den Leveltimer an
- *_updateTimer(Timer timer)* aktualisiert die Anzeige der noch verbleibenden Zeit für den Level und stoppt sie, wenn der Level beendet wurde.
- *isInBounds(final int row, final int col)* die abprüft, ob die angegebene Position sich innerhalb des Spielfeldes befindet, oder ob desssen Grenzen bereits überschritten wurden.
- *findGameObjectExact(finalString tileType)* gibt das erste Objekt zurück, dass dem übergebenen tileType entspricht und auf dem Spielfeld ist. Es wird null zurückgegeben, sollte es kein Objekt dieses Typs auf dem Feld geben.
- *findAllGameObjectsExact(final String tileType)* tut dasselbe, gibt allerdings alle passenden Objekte zurück.
- *getGameObjectAtPosition(final Position position)* gibt das GameObject zurück, dass sich an der angegebenen Position befindet.
- *getGameObjectAtRowAndCol(final int row, final int col)* tut dasselbe, kann aber mit den Koordinaten statt mit einer Position parametrisiert werden.
- *updateGameObjectAtRowAndCol(final int row, final int col, final GameObject gameObject)* und *updateGameObjectAtPosition(final Position position, final GameObject gameObject)* updaten ein *GameObject* an der gegebenen Position, das bisherige GameObject wird

zurückgegeben. Dabei wird auch das bisherige *GameObject* gespeichert, um es nach einer weiteren Bewegung der Kreatur wieder anzuzeigen.

Außerdem beinhaltet die Klasse *Level* die innere Klasse *LevelObjectAccessOutOfBoundsExeption*, eine eigene Fehlermeldung für den Fall, dass eines der Objekte außerhalb der Grenzen des Spielfeldes gefunden wird.

3.3.12. Levelloader

Die Klasse Levelloader ist dafür verantwortlich die entsprechenden Level einerseits zur Laufzeit in die View zu laden, andererseits aber auch dafür die Level schon beim erstmaligen Aufrufen des Hauptmenüs zur späteren Verwendung vollständig in den Cache zu laden. Sie verfügt über das Attribut *CACHED_LEVELS,* welches eine Map der bereits vorgeladenen Level beinhaltet.

Die Methoden des Levelloaders sind:

- *preloadAllLevels()*, die alle vorhandenen Level schon einmal vorlädt und in der *CACHED_LEVELS* Map ablegt.
- *resetInCache(final int levelNumber)* wird genutzt, um den beim Spielen veränderten Level zurückzusetzen. Auf diese Weise ist es möglich den Level bei Bedarf noch einmal zu spielen. Diese Funktion wird genutzt, während der Spieler bereits den nächsten Level spielt.
- *load(final int levelNumber, [final bool ignoreCache = false])* lädt den Level mit der entsprechenden levelNumber vor und gibt ihn als Objekt zurück.
- *_levelFromMap(final Map data)* extrahiert den Level aus der angegebenen Map.
- *_tilesFromMap(final List<Map> data, final int rows, final int cols)* überführt die Tiles des Levels in eine Liste von Listen, die später bei der Initialisierung des Spielfeldes genutzt werden kann.
- *_positionFromMap(final Map data)* gibt eine bestimmte Position aus der Map zurück.

3.3.13. Powerup

Die Klasse *Powerup* erweitert die Klasse *GameObject* und besitzt die Attribute

- *appearChance*, das festlegt, wie hoch die Chance ist, dass ein Powerup in einem Level vorkommt.
- *timeOnField* legt dagegen fest, wie lange das Powerup auf dem Feld bleibt, ehe es wieder verschwindet.
- *mayAppear* speichert ab, ob das Powerup überhaupt auftauchen darf oder nicht. Mögliche Werte sind *true* oder *false*.
- *hasAppeared* wird verwendet um festzustellen, ob das Powerup bereits aufgetaucht ist oder nicht.
- *used* gibt an, ob der Spieler das Powerup bereits eingesammelt hat.
- *timeOnFieldCountdown* ist die Dauer, die das Powerup auf dem Feld verbleibt und wird im Timer

- *timeOnFieldTimer* genutzt, um die Zeit herunterzuzählen.

Neben einem Konstruktor zur Erzeugung weiterer Powerups verfügt die Klasse über die Methoden

- *applyOn(final GameObject target)* ist abstrakt und bedarf einer konkreten Implementation.
- *appear()* setzt das Powerup auf das Feld, wenn es ihm erlaubt ist.
- *disappear()* entfernt das Powerup vom feld, wenn es Zeit dafür ist.

In der inneren Klasse *CarrotPowerup* findet sich neben einem Konstruktor das Attribut *_speedIncrease*, das angibt, um wie viel die Movement-Rate gesenkt werden soll, wenn der Spieler das Powerup einsammelt.

Hier wird auch die Methode *applyOn(finale GameObject target)* überschrieben und das Powerup auf den Hasen angewendet.

3.2. View

Die View dient der Darstellung des Spiels für den Spieler. Sie besteht aus einem HTML-Dokument (Siehe Abschnitt 3.2.1.) und einer clientseitigen Logik, die den DOM-Tree des HTML-Dokuments manipuliert. (Siehe Abschnitt 3.2.2.)

3.2.1. HTML-Dokument

Die View wird im Browser zu Beginn durch folgendes HTML-Dokument erzeugt. Im Verlaufe des Spiels wird dieses HTML-Dokuments durch die Klasse *MazeGameView* manipuliert, um den Spielzustand darstellen und Nutzerinteraktion ermöglichen zu können. Die Klasse *MazeGameView* wird dabei durch das Script *mazegameclient.dart* als clientseitige Logik geladen.

Dieses HTML-Dokument wird genutzt, um darin das Spiel einzublenden (Siehe Abbildungen 4 und 5).

Abbildung 4: Hauptmenü

Abbildung 5: Laufendes Spiel

3.2.2. MazeGameView als Schnittstelle zum HTML-Dokument

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

- Das Element mit dem Identifier *#overlay* wird genutzt, um zwischen den einzelnen Leveln ein Menü einzublenden, dass es erlaubt zum nächsten Level voranzuschreiten oder in das Hauptmenü zurückzukehren. Über den Identifier *#overlay h2* wird der Titel des Menüs beschrieben und seine Beschreibung über den Identifier *#overlay p*.
- Das Element mit dem Identifier *#title* wird im Hauptmenü der Schriftzug "RabbitRinth" angesprochen. Der Name des Spiels ist daher darüber zu verwalten.

- Der Untertitel des Spiels ist über den Identifier *#subtitle* zu erreichen.
- Über den Identifier *#progress .label* wird das Element angesprochen, das im laufenden Spiel die verbleibende Zeit über der Progressbar anzeigt.
- Das Element, das die Progressbar selbst darstellt wird über den Identifier *#progressbar > div* angesteuert. Der Container, der sie beinhaltet über *#progress*.
- Das Element hinter dem Identifier *#game* ist die Tabelle, die das eigentliche Spielfeld bildet.
- Das Element mit dem Identifier *#landscape_warning* beschreibt das Div, in dem sich die Warnung befindet für den Fall, dass der Spieler auf seinem Smartphone in den Landscape-Modus wechselt. Der Text dafür befindet sich hier ebenfalls.
- Alle Elemente, deren Identifier mit *#btn* beginnt beschreiben die einzelnen Buttons, die im Spiel zu sehen sind. Die Benennungen im Identifier sind präzise genug, um eine einzelne Auflistung unnötig zu machen.

Alle CSS-Gestaltungen werden in der style.css vorgenommen. Die Applikationslogik wird wird über das mazegameclient.dart Script geladen.

Objekte der Klasse *MazeGameView* agieren nicht eigenständig. Es ist notwendig, dass sie von Objekten der Klasse *MazeGameController* genutzt werden, um die Aktualisierung der View vorzunehmen Der Controller kann sich dazu der folgenden Methoden und Attribute bedienen, um den DOM-Tree nicht selber manipulieren zu müssen (Separation of Concerns):

- *updateRabbit(Rabbit rabbit)* sorgt dafür, dass der Hase korrekt auf dem Spielfeld angezeigt wird.
- *updateEnemies(final MazeGameModel game)* updated die Position aller Feinde im Spiel.
- *updatePowerups(final MazeGameModel game)* sorgt für die korrekte Positionierung der Powerups im laufenden Spiel.
- *updateNotLivingTiles()* sorgt für die Anzeige aller nicht belebten Tiles (Hecken, Terrain).
- *updateTitleAndSubtitle()* aktualisiert die Anzeige des Titels und Untertitels.
- *UpdateTimerAndBrightness()* aktualisiert die Anzeige des Timers und die Gesamthelligkeit des Levels.
- *showGameOverlay(final int levelReached)* zeigt das Gameoverlay an und setzt dabei den erreichten Level ein.
- *showLevelFinishedOverlay(final int timeLeft, final int levelCompleted)* öffnet das Overlay nachdem ein Level beendet wurde. Das umfasst auch die Nutzung des *showGameOverlay(final int levelReached)*.
- *showTutorialOverlay()* öffnet die aktuelle Tutorialseite im Overlay des Spiels.
- *ShowAboutOverlay()* öffnet das About-Overlay und zeigt die grundlegenden Informationen zum Spiel an (Autoren, Rechte usw.).
- *_updateElementInGameFieldWithPosition(final Position position)* updated ein bestimmtes GameObject, das nach seiner Position bestimmt wird. Gleiches gilt für *_updateElementInGameField(final int row, final int col)*, nur das hier nicht die Position, sondern die Koordinaten verwendet werden.
- *resetToMainMenu()* setzt den Titel, den Unterntitel und den Container des Spiels in den initialen Zustand zurück.

- *animateRabbit(Rabbit rabbit, Function afterAnimation)* animiert den Hasen – dabei wird berücksichtigt, in welche Richtung er sich bewegt.
- *incrementTutorialPage()* erhöht die Zahl der aktuellen Tutorial-Seite, während *decrementTutorialPage()* das Gegenteil tut.
- *generateField(MazeGameModel game)* erzeugt eine Tabelle zur Darstellung des genutzten Spielfeldes. Dieses wird in den DOM-Tree in das TABLE-Element mit der Id #game eingeblendet.
- *_visibility(Element element, bool shouldAdd)* ändert die Sichtbarkeit des übergebenen Elementes je nach gefordertem Wert.
- Die Methoden *visible(Element element)* und *invisible(Element element)* greifen lediglich auf die *_visibility(Element element, bool shouldAdd)* Methode zu, um einen direkten Zugang zu ermöglichen.
- *closeOverlay()* und *openOverlay()* sind verkürzt dafür gedacht, das Overlay anzuzeigen oder unsichtbar zu machen.
- Mittels der Liste von Listen *fields* ist es der View möglich das Spielfeld zu befüllen.
- *currentTutorialPage* speichert die momentan geladene Seite des Tutorials
- *viewAnimationEnded* gibt Aussage darüber, ob die Animation des Hasen beendet ist oder nicht.

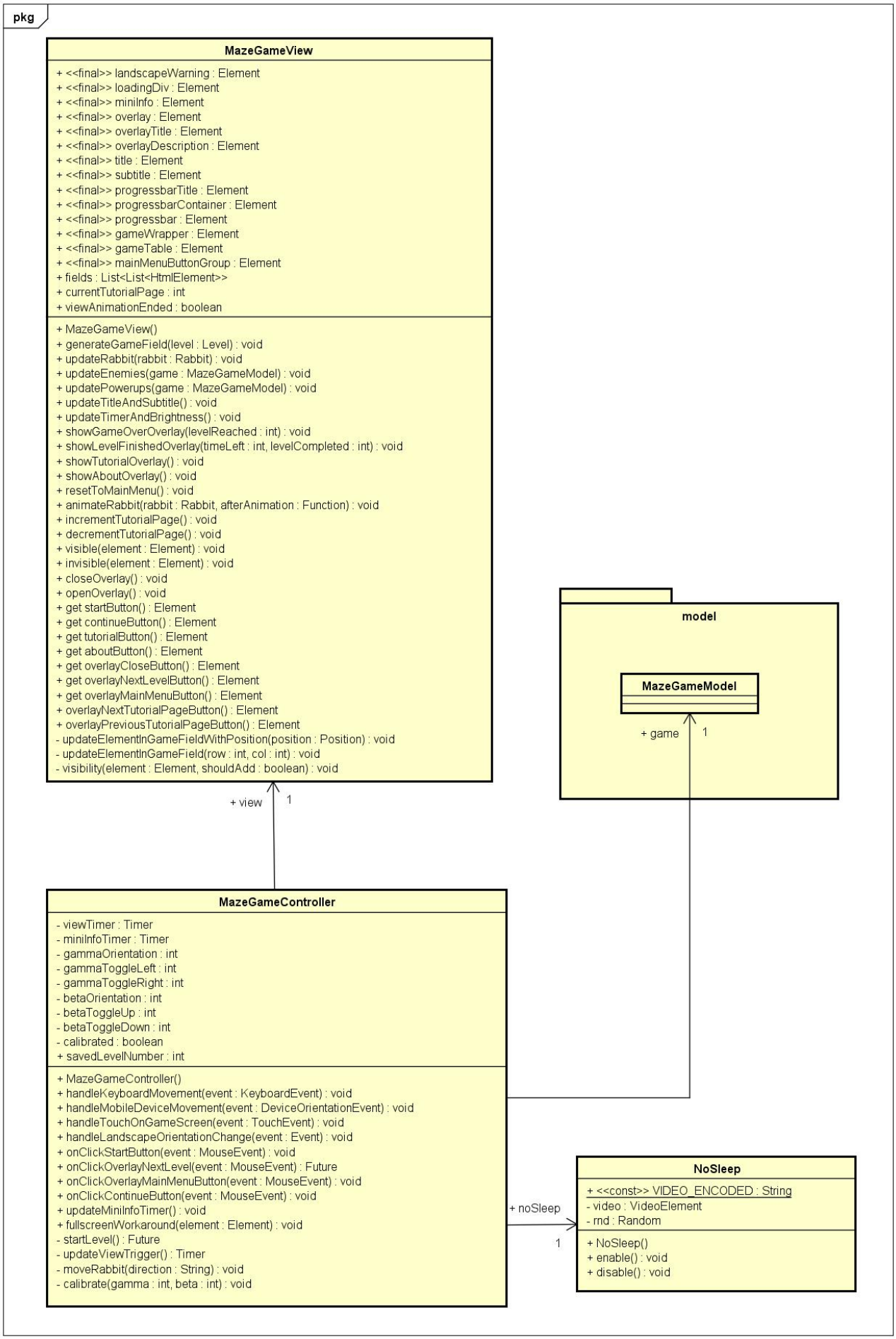

Abbildung 6: View-Controller

3.3. Controller

3.3.1. Laufendes Spiel

Zu Beginn des Spiels wird über die *onClickStartButton()*-Methode der Initialisierungsprozess gestartet. Hier wird über die *noSleep.enable()*-Methode das Video gestartet, um dem Sleep-Mode des Smartphones zuvor zu kommen. Das Spiel wird in den Fullscreenmodus gebracht und die Lageerkennung des Smartphones kalibriert. Außerdem wird das Spielfeld inklusive Progressbar und Timer initialisiert.

Während des Spielablaufs wird die View von der *handleMobileDeviceMovement(DeviceOrientationEvent event)*-Methode getriggert, wenn der Hase sich bewegt hat. Über den *moveTimer* wird ein *moveCountdown* (beide Teil der Klasse *Rabbit*) genutzt und schließlich in der *_moveRabbit(final String direction)* in Bewegung umgesetzt. Um den Zeitpunkt des Viewupdates zu bestimmen nutzt das Spiel im Controller die *VIEW_UPDATE_COUNTDOWN* Konstante um den *updateViewTrigger(Timer timer)* anzustoßen.

Auf einem Desktop-PC übernimmt diese Aufgabe die Methode *handleKeyboardMovement(KeyboardEvent event)*.

Der Fuchs wird über den *enemyMoveTrigger* periodisch bewegt. Über diesen Aufruf wird auch die View entsprechend geupdated.

Ursprünglich als Methode gedacht dem Sleep-Mode zuvor zu kommen hat sich herausgestellt, dass die Spieler während des Spiels oft vergessen in welcher Ausgangsposition ihre Lageerkennung kalibiert wurde. Die Methode *handleTouchOnGameScreen(TouchEvent event)* ermöglicht daher eine Neukalibrierung während des laufenden Spiels.

Dreht der Spieler sein Smartphone in den Landscape-Modus, triggert das die *handleLandscapeOrientationChange(final Event event)*-Methode, die dem Spieler dann die entsprechende Warnung zeigt. Dreht der Spieler das Gerät wieder zurück, so verschwindet die Warnung und das Spielfeld wird wieder angezeigt.

Über die Ableitung des Enemy von Creature und GameObject ist es möglich ohne größeren Aufwand neue, stärkere Gegner für den Hasen zu implementieren und sie mit den verschiedenen Bewegungsmöglichkeiten zu verknüpfen.
Da die Anzahl der Gegner bereits im Level selbst angegeben werden kann sind auch mehrere Gegner kein Problem. Der Level wäre in diesem Fall lediglich wie in Abbildung 6 gezeigt zu erweitern.

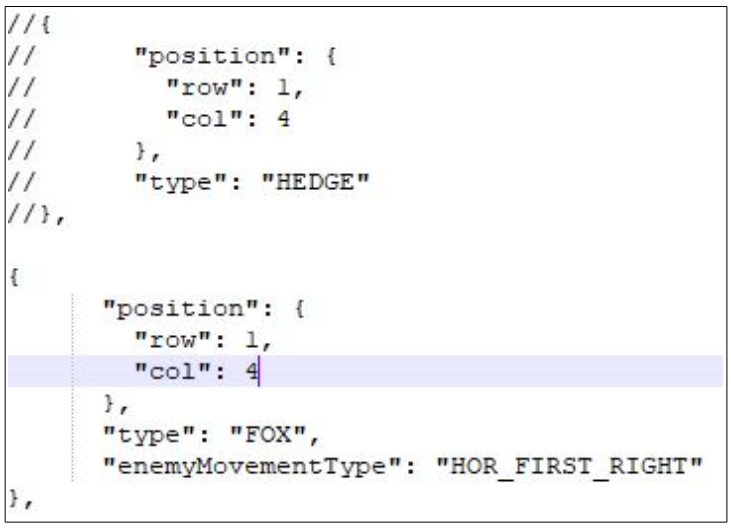

Abbildung 7: Einfügen eines weiteren Fuchses

3.3.2. Level Done

Ein Level gilt dann als erfolgreich beendet, wenn der Hase das Loch erreicht, ehe die *timeLeft* des Levels =< 0 erreicht und ohne, dass er mit einem Feind kollidiert wäre. Die Kollision mit dem Ziel triggert die Methode *onCollideWithGoal()* in der das *done*-Attribut des Levels auf *true* gesetzt wird. Beim nächsten Feuern des *levelCountDownTriggers* werden die Trigger des Levels, des Feindes und des Hasen gestopt und das Spiel angehalten. Es öffnet sich dann das Overlay und ermöglich dem Spieler in den nächsten Level voran zu schreiten oder ins Hauptmenü zurückzukehren. Zeitgleich wird die *levelNumber* des nächsten Levels in den Local Storage geladen. Auf diese Weise wird es dem Spieler ermöglicht seinen Spielfortschritt auch dann zu behalten, wenn er sich dafür entscheidet, das Spiel vorerst zu beenden.

3.3.3. Game Over

Ein Verlieren des Spiels ist auf zwei Weisen möglich. Einerseits wird das gameOver-Attribut des Levels auf true gesetzt, wenn der *levelCountdownTrigger* feuert und die *timeLeft* des Levels <= 0 ist. Der Level gilt dann als nicht geschafft und das Overlay wird angezeigt.

Die zweite Weise ist, dass der Hase mit einem Feind kollidiert. Abgeprüft wird dabei sowohl, ob der Hase in den Gegner gelaufen ist, als auch ob der Gegner in den Hasen gelaufen ist. Die entsprechenden Methoden finden sich – wie oben bereits beschrieben in den Klassen *Enemy* und *Rabbit* in den Methoden onCollideWithRabbit(GameObject colissionObjekt, int newRow, int newCol) und onCollideWithFox(GameObject colissionObjekt, int newRow, int newCol).

Tritt eine dieser beiden Möglichkeiten ein, wird die *stop()*-Methode des Models aufgerufen und das Spiel auf diese Weise angehalten. Außerdem werden die Trigger für den *levelCountdown*, die Bewegung des Hasen und die Bewegung des Feindes angehalten. Dann wird das Overlay aufgerufen mit dem der Spieler in das Hauptmenü zurückkehren kann. Dort besteht für ihn die Möglichkeit das Spiel ganz von vorne zu beginnen (Klick auf den Start-Button) oder den eben verlorenen Level zu wiederholen (Klick auf den Continue-Button). Im zweiten Fall wird über die Methode *onClickContinueButton()* die im Local Storage abgespeicherte *levelNumber* eingeladen, um das Spiel sofort mit dem zuletzt gestarteten Level zu beginnen.

3.4. Local Storage zur Speicherung des Spielfortschrittes

In anbetracht der sich aus der gestellten Aufgabe ergebenen Anforderungen war es notwendig ein Speicherkonzept zu finden, das unabhängig von einer Internetverbindung ist. Um dennoch den Spielfortschritt des Spielers festzuhalten und ihn im Bedarfsfall dort wieder mit dem Spiel beginnen zu lassen, wo er zuletzt aufgehört hat, haben wir auf den *Local Storage* zurückgegriffen.

Dieser entspricht einem (bei Bedarf auch mehreren) Key/Value-Paar, das im Cache-Speicher des Browsers abgelegt werden kann. Auf diese Weise wird der Spielfortschrit nicht nur während des laufenden Spiels persistiert, sondern auch über das Schließen des Browsers und sogar über einen Neustart des Rechners oder Smartphones hinweg. Lediglich das gezielte Löschen des Browser-Caches entfernt den Spielfortschritt.
Grundsätzlich erfolgt die Speicherung des Spielfortschrittes dabei wie folgt: Immer dann, wenn ein Spieler einen Level erfolgreich abschließt, wird die Nummer des folgenden Levels in den *Local Storage* geschrieben. Kehrt der Spieler dann in das Hauptmenü zurück, reicht ein Klick auf den Continue-Button, um die *levelNumber* des anstehenden Levels zurück in das Spiel zu laden und dort anzufangen, wo man aufgehört hat.
Dabei wird die Nummer jedes Mal überschrieben, sodass nach Abschluss des Levels 3 die *levelNumber* 3 durch die *levelNumber* 4 ersetzt wird. Daraus resultiert auch, dass bei einem Neustart des Spiels über den Start-Button und darauf folgendem Abschließen des ersten Levels die 2 als *levelNumber* gespeichert wird, sodass höhere *levelNumber*-Einträge nicht mehr verfügbar sind.

4. Level- und Parametrisierungskonzept

4.1. Levelkonzept

Die Level von RabbitRinth werden als JSON-Dateien geliefert. Diese beinhalten zunächst die Attribute, wie sie unter Abschnitt 3.1.11. genannt werden und dazu eine Liste von Listen (also ein zweidimensionales Array), indem jeweils die notwendigen Informationen für jedes *Tile* unseres Spielfeldes festgehalten werden. Diese werden im *Levelloader* auf Dart umgesetzt und später zum Spielfeld initialisiert.

Die Steigerung des Schwierigkeitsgrades zwischen den Leveln ergibt sich aus mehreren Parametern. Zum ersten werden die späteren Level ein wenig größer als die, mit denen der Spieler sich zu Beginn konfrontiert sieht. Dieser Stellschraube sind allerdings durch die Größe des Smartphone-Displays Grenzen gesetzt.
Zum zweiten steigert sich von Level zu Level die Komplexität des Levels. Während der erste Level zum Einstieg nur fordert geradeaus in eine Richtung zu gehen stellen sich dem Spieler im zweiten Level bereits mehr Hindernisse in den Weg und in späteren Leveln ist das Loch lediglich von einer Seite aus begehbar.
Drittens variieren die Zeiten, die dem Spieler zur Erreichung des Lochs verbleiben. Gemessen an der Größe und Komplexität des Levels steigert das den Schwierigkeitsgrad für den Spieler, auch wenn die eingeblendete Zeit zu Beginn des Levels absolut betrachtet höher sein kann.
Zuletzt gibt es die Möglichkeit den Spieler über das Auftauchen von Feinden – im Augenblick nur Füchse – herauszufordern. Ab dem vierten Level tauchen diese Gegenspieler auf und führen bei einer Kollision zu einem Game Over für den Spieler. Dabei wird innerhalb der Füchse noch einmal

der Schwierigkeitsgrad unterschieden. So gibt es einerseits Füchse, die sich auf einer gleichbleibenden, linearen Bahn bewegen und dem der Spieler nur ausweichen muss, andererseits aber auch solche, die den Spieler bei Sichtkontakt zu verfolgen beginnen. Daraus resultiert die Notwendigkeit für den Spieler dem Fuchs auszuweichen, ihn aber möglicherweise auch erst von einem blockierten Durchgang weglocken zu müssen.

Im Folgenden (Abbildung 7) finden Sie den Ausschnitt eines Levels, um einen Eindruck von seinem Aufbau zu erhalten.

```
{
  "name": "Level 4",
  "description": "Look out! There is a fox around.",
  "time": 25,
  "rows": 7,
  "cols": 7,
  "tiles": [
    {
      "position": {
        "row": 0,
        "col": 0
      },
      "type": "TERRAIN"
    },
    {
      "position": {
        "row": 0,
        "col": 1
      },
      "type": "TERRAIN"
    },
    {
      "position": {
        "row": 0,
        "col": 2
      },
      "type": "TERRAIN"
    },
```

Abbildung 8: Ausschnitt aus einer Level-Datei

4.2. Parametrisierungskonzept

Die Parametrisierung von RabbitRinth erfolgt über die Klasse *Constants*. Sie hält die Konstanten

- *MAX_LEVEL* in dem die levelNumber des letzten Levels im Spiel festgelegt wird. Damit weiß das Spiel, wann der Spieler alle Level erfolgreich abgeschlossen hat,
- *RABBIT_MOVEMENT_SPEED*, in dem festgelegt werden kann, wie schnell der Hase sich bewegt und
- *ENEMY_MOVEMENT_SPEED*, in dem die Geschwindigkeit der vorhandenen Gegner gesteuert werden kann.
- *MINI_INFO_COUNTDOWN* das die Dauer Angibt, während der die Sichtbarkeit der Mini Info im Footers gelten soll, nachdem man darauf geklickt hat.
- *DEVICE_MOTION_TOGGLE_HORIZONTAL* und *DEVICE_MOTION_TOGGLE_VERTICAL* die angeben, wie weit das Gerät bewegt werden muss, ehe die Lageerkennung feuert.
- *VIEW_UPDATE_COUNTDOWN* in der die Zeit gesteuert wird, die verstreichen muss, ehe die View periodisch geupdated wird.
- *MAX_TUTORIAL_PAGES* bestimmt die Höchstanzahl der Seiten des Tutorials.
- *TUTORIAL_MESSAGES* beinhaltet den Inhalt der verfügbaren Seiten des Tutorials.

5. Nachweis der Anforderungen

Nachfolgend wird erläutert, wie die in Kapitel 2 eingeführten funktionalen Anforderungen, die Dokumentationsanforderungen und die technischen Randbedingungen erfüllt bzw. Eingehalten wurden. Dies erfolgt argumentativ. Abschließend ist die jeweilige Zuständigkeit der Teammitglieder angegeben.

5.1. Nachweis der funktionalen Anforderungen

Id	Kurztitel	Erfüllt	Teilw. Erfüllt	Nicht erfüllt	Erläuterung
AF-1	Single-Player-Game als Single-Page-App	X			RabbitRinth ist ein Einpersonen Spiel, wie aus dem Spielkonzept in Abschnitt 2.2. hervorgeht.
AF-2	Balance zwischen technischer Komplexität und Spielkonzept	X			Das Spielkonzept von RabbitRinth (Abschnitt 2.2.) basiert auf analogen Labyrinthspielen, in denen man eine physisch vorhandene Kugel durch ein Labyrinth bewegen muss, indem man das Spiel kippt. Das Interesse an diesem Konzept zeigte sich bei der ersten Vorführung. Das Spielprinzip ist schnell verstanden und führte lediglich in einem Fall zu Rückfagen. Technisch war es anspruchsvoll genug, um einen guten Einblick in Web-Technologien zu erhalten. Das Spiel wird im Augenblick über die GitHub Pages bereit gestellt und alle Ressourcen sind relativ zueinander adressiert.
AF-3	DOM-Tree-basiert	X			Die Implementation und Umsetzung der View basiert wie in Abschnitt 3.2.2. ausgeführt auf einer Manipulation des DOM-Trees. Da ausdrücklich verboten war Canvas-basierte Spiele zu erstellen, wurde darauf verzichtet. Das MVC-Prinzip wurde geplant und eingehalten.
AF-4	Target Device: SmartPhone		X		Das Spiel ist zwar für die Bedienung durch ein SmartPhone konzipiert und nutzt auch die Unterstütung von HTML5 mobile Browsern, funktioniert allerdings nur auf Android. Der Grund dafür ist, dass in der Implementation Funktionalitäten genutzt wurden, die der Sierra-Browser nicht unterstützt (bspw. Vollbild-Modus im Browser)

AF-5	Mobile First Prinzip	X			RabbitRinth ist bewusst für SmartPhones konzipiert. Wie bereits im Spielkonzept unter Abschnitt 2.2. ausgeführt nutzen wir die Lagesteuerung des Smartphones, sowie die Touchfunktion, um das Spiel zu steuern. Auf dem Desktop PC und dem Tablet ist es zwar spielbar, verfügt aber ggf. über weniger Features (Lageerkennung fehlt beispielsweise). Die native Entwicklung erfolgte allerdings ausdrücklich für das Smartphone.
AF-6	Das Spiel muss schnell und intuitiv erfassbar sein und Spielfreude erzeugen	X			Wie bereits oben angedeutet, haben Testspieler Freude an dem Spiel geäußert – einige haben sogar musikalische Untermalung und weitere Level gefordert. Die Steuerung an sich erschien zunächst contraintuiitiv und wurde daher nochmal angepasst – spätere Rückfragen wie das Spiel zu steuern sei blieben auch bei neuen Testspielern aus.
AF-7	Das Spiel muss ein Levelkonzept vorsehen	X			Das Spiel beinhaltet sieben Level, die wie im Abschnitt 4.1. genauer ausgeführt, kontinuierlich schwieriger werden. Hierbei bestimmen die oben genannten vier Faktoren die Gesamtschwierigkeit eines Levels. Deklarativ sind unsere Level in JSON-Dateien beschrieben. Ein Nachladen neuer Level ist grundsätzlich kein Problem, solange in der Parametrisierung die Maximalzahl der Level entsprechend angepasst wird. Eine Änderung der Programmierung ist dafür nicht erforderlich.
AF-8	Ggf. erforderliche Speicherkonzepte sind clientseitig zu realisieren	X			Um die Anforderung einzuhalten war es nötig auf die Anbindung an zentrale Server zu verzichten. Aus diesem Grund wurde vorliegend die local storage Lösung verwendet, wie in Abschnitt 3.4. genauer beschrieben.

Tabelle 2: Nachweis der funktionalen Anforderungen

5.2. Nachweis der Dokumentationsanforderungen

Id	Kurztitel	Erfüllt	Teilw. Erfüllt	Nicht Erfüllt	Erläuterung
AF-9	Projektdokumentation	X			Vorliegende Dokumentation erläutert die übergeordneten Prinzipien und verweist an geeigneter Stelle auf den Quelltext und seine interne Dokumentation.
AF-9	Quelltextdokumentation	X			Es wurden alle Methoden und Datenfelder sowie Konstanten durch Kommentare erläutert.
AF-9	Libraries	X			Es wurden lediglich das normalize.min.css genutzt, um die Interoperabilität der Browser zu garantieren, sowie der Press Start 2P Google Font. Beide sind in der index.html bzw. der styles.css aufgeführt. Weitere Libraries wurden nicht eingebunden.

Tabelle 3: Nachweis der Dokumentationsanforderungen

5.3. Verantwortlichkeiten im Projekt

Komponente	Detail	Asset	Marc-Niclas Harm	Claas Rhodgeß	Anmerkungen
Model	MazeGame Model	lib/src/model.dart	V	U	Beinhaltet Dokumentation per Kommentar
	Creature	lib/src/creature.dart	U	V	Beinhaltet Dokumentation per Kommentar
	Enemy	lib/src/enemy.dart	U	V	Beinhaltet Dokumentation per Kommentar
	Fox	lib/src/fox.dart	U	V	Beinhaltet Dokumentation per Kommentar
	GameObject	lib/src/gameobject.dart	V	U	Beinhaltet Dokumentation per Kommentar
	Level	lib/src/level.dart	V	U	Beinhaltet Dokumentation per Kommentar
	LevelLoader	lib/src/levelloader.dart	V	U	Beinhaltet Dokumentation per Kommentar
	NoSleep	lib/src/nosleep.dart	V	U	Beinhaltet Dokumentation per Kommentar
	Position	lib/src/position.dart	V	U	Beinhaltet Dokumentation per Kommentar
	Rabbit	lib/src/rabbit.dart	U	V	Beinhaltet Dokumentation per Kommentar
	Tile	lib/src/tile.dart	V	U	Beinhaltet Dokumentation per Kommentar
	TileType	lib/src/tiletype.dart	V	U	Beinhaltet Dokumentation per Kommentar
Controller	MazeGameController	lib/src/controller.dart	V	U	Beinhaltet Dokumentation per Kommentar

	Parametrisierung	lib/src/constants.dart	U	V	Beinhaltet Dokumentation per Kommentar
	Level	web/assets/lvl	V	U	
Dokumentation	RabbitRinth Dokumentation	doc/*	U	V	
Local Storage	Einbindung des Local Storage	lib/src/model.dart	U	V	

Tabelle 4: Verantwortlichkeiten im Projekt

V = verantwortlich (hauptführend, kann nur einmal pro Zeile vergeben werden)
U = unterstützend (Übernahme von Teilaufgaben)

6. Nutzung von RabbitRinth zum Zwecke öffentlicher Vorführung und genutzte Grafiken

Wir sind ausdrücklich damit einverstanden, dass das Spiel RabbitRinth öffentlich genutzt und bereitgestellt wird wird. Ferner möchten wir darauf hinweisen, dass sämtliche genutzte Grafiken von uns selbst erstellt wurden und in keinster Weise gegen die Rechte Dritter verstoßen.