2018

DartRogue Dokumentation

WEB-TECHNOLOGIE-PROJEKT – SOSE 2018

JANNIK SCHWARDT & RUBEN MAURER

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Code-Snippet-Verzeichnis

1. Einleitung

Im Rahmen des Web-Technologie-Projektes im Sommersemester 2018 der Fachhochschule Lübeck ist in einem Zweierteam ein DOM-Tree basiertes Single-Player Spiel zu entwickeln.

Die Ideenfindung hat schnell dazu geführt kein richtig typisches „Handyspiel für zwischendurch“ zu entwickeln, sondern ein Spiel mit Tiefgang, das mehr als nur wenige Minuten Spielinhalt bietet.

Daher ist die Wahl auf ein Spielprinzip gefallen, das auf den Spielen Rogue, NetHack, Diablo und Dark Souls basiert, aber, wie aus zahlreichen 2D-Rollenspielen bekannt, ein rundenbasiertes Kampfsystem verwendet.

Da diese Spiele aber alle einen relativ komplexen Einstieg haben, besteht hierbei die Herausforderung eine Mischung aus einfachem Spieleinstieg und geringer, aber doch motivierender Komplexität zu finden.

Ziel des Spiels ist es, den Spieler durch die bekannte „Sammelwut“ von Ausrüstungsgegenständen, bekannt aus den o.g. Spielen, stetig zu motivieren. Um den Wiederspielwert zu erhöhen wird ein Highscore-System genutzt, das den Spieler dazu ermutigen soll nach dem Durchspielen des Spiels erneut zu starten.

2. Anforderungen und abgeleitetes Spielkonzept

2.1. Anforderungen

AF-1: Single-Player-Game als Single-Page-App

- Das Spiel ist als Ein-Spieler-Game zu konzipieren.
- Das Spiel ist als HTML-Single Page App zu konzipieren.
- Das Spiel muss als **statische Webseite** von beliebigen Webservern (HTTP) oder Content Delivery Networks (CDN) bereitgestellt werden können (bspw. mittels GitHub Pages).
- Alle Spielressourcen (z.B. Level-Dateien, Bilder, CSS-Dateien, etc.) sind daher relativ und nicht absolut zueinander zu adressieren.

AF-2: Balance zwischen technischer Komplexität und Spielkonzept

- Sie sollen ein interessantes Spielkonzept entwickeln oder ein bestehendes Spielkonzept abwandeln. Spielkonzepte sind nicht immer technisch komplex (z.B. Memory).
- Sie sollen jedoch ein Spiel konzipieren, dass eine vergleichbare Komplexität mit den Spielen der Hall-of-Fame hat (Memory wäre bspw. zu einfach; Schach zu kompliziert, da sie für ein Ein-Spieler-Game eine Gegner-KI entwickeln müssten).
- Unabhängig von der inneren Komplexität soll das Spiel schnell und intuitiv erfassbar sein und angenehm auf einem SmartPhone zu spielen sein.

AF-3: DOM-Tree-basiert

- Das Spiel soll dem MVC-Prinzip folgen (Model, View, Controller).
- Das Spiel soll den DOM-Tree als View nutzen.
- Es sind keine Canvas-basierten Spiele erlaubt. *Hintergrund: Sie sollen Webtechnologien lernen und nicht wie man eine Grafikbibliothek programmiert.*

AF-4: Target Device: SmartPhone

- Das Spiel soll für eine SmartPhone Bedienung konzipiert werden.
- Entsprechende Limitierungen sind zu berücksichtigen.
- Als Target Devices sind die Plattformen Android und iOS zu berücksichtigen.
- Das Spiel soll mit HTML5 mobile Browsern auf den genannten Plattformen spielbar sein.

AF-5: Mobile First Prinzip

- Das Spiel soll bewusst für SmartPhones konzipiert werden.
- Das Spiel soll auch auf Tablets und Desktop PCs spielbar sein. Einschränkungen sind zu minimieren, werden aber billigend in Kauf genommen (z.B. fehlende 3D-Lage bei Desktop Browsern).
- Sie sollen typische mobile Interaktionen sinnvoll nutzen (z.B. Swipe, Wischen, 3D Lage im Raum, etc.).
- Übertragen sie nicht eine typische Desktop-Bedienung auf Mobile.
- Sie müssen allerdings keine mobilen Interaktionen sklavisch nutzen - wenn dies nicht sinnvoll für das Spielkonzept ist.

AF-6: Das Spiel muss schnell und intuitiv erfassbar sein und Spielfreude erzeugen.

- Das Spiel muss schnell und intuitiv erfassbar sein.
- Das Spiel muss Spielfreude erzeugen.
- Hintergrund: *Ihre Spiele sollen ggf. als Anschauungsbeispiele für Studieninteressierte Schüler auf Messen oder ähnlichen Veranstaltungen gezeigt werden. Sie arbeiten also nicht für "die Tonne" - andere sollen ihre Spiele tatsächlich spielen.*
- *Tipp: Vermeintlich einfache Spielprinzipien haben ihre Stärken und sind dennoch komplex genug um die technischen Anforderungen dieses Projekts abzudecken.*

AF-7: Das Spiel muss ein Levelkonzept vorsehen

- Es ist ein steigender Schwierigkeitsgrad über mindestens 7 Level vorzusehen.
- Level sollen deklarativ in Form von Textdateien beschrieben werden können (z.B. JSON, XML, CSV, etc.).
- Diese Level sollen durch das Spiel nachgeladen werden können, so dass nachträglich Level ergänzt und abgeändert werden können, ohne die Programmierung des Spiels anpassen zu müssen.

AF-8: Ggf. erforderliche Speicherkonzepte sind Client-seitig zu realisieren

- Aufgrund der Zielgruppe sollen durch das Spiel gesammelte Daten auf den Geräten bleiben.
- Aufgrund des Demonstrationscharakters auf Messen dürfen keine zentralen Server für Highscores, etc. erforderlich sein oder angebunden werden.
- Ggf. erforderliche stateful Solutions sind mittels client-seitiger Storage-Konzepte zu lösen. D.h. z.B. mittels local storage.

AF-9: Dokumentation

- Das Spiel muss nachvollziehbar dokumentiert sein.
- Das Spiel muss analog dem SnakeGame dokumentiert sein. *Hinweis: Nutzen sie die SnakeGame-Dokumentation als Template.*

Hinzu kommen weitere Anforderungen:

- Als Programmiersprache ist Dart zu verwenden (Versionsnummer: 1.X)
- Nutzung von Canvas-basierten Darstellungstechniken ist explizit untersagt

2.2 Spielkonzept

Das Spielkonzept von DartRogue beruht darauf, dass man als Spieler verschiedene Monster besiegen muss, um durch gesammelte Erfahrung und Ausrüstungsgegenstände stärker zu werden.

Der Spieler kann seine Spielfigur per Klick bzw. Fingertipp auf das gewünschte Ziel durch die verschiedenen Level bewegen.

Ziel des Spiels ist es in jedem Level den Zwischenendgegner zu besiegen, um abschließend im Letzten Level gegen den Endgegner anzutreten.

Das Spiel ist vorbei, wenn der Spieler stirbt oder der Endgegner durch den Spieler getötet wird.

Nach Spielende wird der erreichte Punktestand in die Highscore-Liste eingetragen.

2.2.1 Spielercharakter

Der Spielercharakter ist die zentrale Figur im Spiel mit folgenden Eigenschaften:

- Wird durch Levelaufstiege und bessere Ausrüstung stärker
- Bekommt durch Siege Erfahrung
- Kann sich durch Benutzung von Tränken heilen
- Stirbt, wenn Lebenspunkte auf 0 fallen
- Hat ein Inventar mit zwölf Plätzen
- Kann von jeder Sorte von Gegenständen genau einen Gegenstand anlegen
- Hat vier verschiedene Angriffsfertigkeiten im Kampf

Im Charaktermenü lassen sich die Attribute, das Inventar sowie die angelegte Ausrüstung jederzeit einsehen.

Im Tränkeauswahlmenü lässt sich festlegen welcher Trank im Kampf eingesetzt werden soll. Zusätzlich wird angezeigt wie viele Tränke noch von jeder Sorte vorhanden sind.

2.2.2 Level

In den Leveln von DartRogue findet das Hauptspiel statt. Jedes Level ist wie folgt aufgebaut:

- Größe der Level beliebig festlegbar (n x m)
- Verschiedene Anzahl an Monstern
- Verschiedene Anzahl an Truhen
- Begehbare und nicht begehbare Bereiche vorhanden
- Durch Sieg über Endgegner freischaltbares Portal zum nächsten Level

Durch Kämpfe erlangte Ausrüstungsgegenstände werden direkt in das Inventar des Spielers gelegt. Sollte das Inventar nach einem Kampf voll sein und der Gegner würde Gegenstände fallen lassen, so erscheint eine Truhe im Level, sodass der Spieler nach Leerung des Inventars jederzeit Zugriff auf die Gegenstände hat.

2.2.3 Gegner

Die Gegner des Spielecharakters ähneln dem Spielercharakter, sind jedoch einfacher gehalten:

- Haben festes, nicht einsehbares Charakterlevel
- Können nur einen Standardangriff nutzen
- Können sich nicht heilen

Gegner bekommen feste Punkte in einem Level zugewiesen. Sobald sich der Spieler einem Gegner nähert, läuft dieser ihm entgegen bzw. hinterher.

Wenn ein Gegner stirbt, verschwindet er aus dem Level und hinterlässt gegebenenfalls für den Spieler nutzbare Ausrüstungsgegenstände und Tränke.

2.2.4 Ausrüstungsgegenstände

Die im Spiel enthaltenen Ausrüstungsgegenstände unterscheiden sich in der Qualität und damit auch in den möglichen Boni. Die Boni beeinflussen die Attribute des Spielercharakters, so dass dieser durch das Anlegen von stärkerer Ausrüstung für höhere Level gewappnet ist.

Demnach wird das Spiel schwieriger, wenn weniger Gegner getötet werden, da einem wichtige Boni und Tränke nicht zur Verfügung stehen.

2.2.5 Gestrichene Features

Im ursprünglichen Konzept des Spiels sind noch weitere Features geplant gewesen, die es aber nicht ins fertige Spiel geschafft haben. Die Implementierung dieser hätte sowohl den Umfang, als auch die Komplexität dieses Projektes gemäß den Anforderungen überschritten.

Zu den nicht implementierten Features gehören:

- Attribut Glück (beeinflusst die Wertigkeit von fallengelassenen Gegenständen)
- Geschwindigkeitsunterschieden von Spieler und Gegnern (Fluchtmöglichkeit des Spielers)
- Zufällige Werte von Ausrüstungsgegenständen (benötigt für die Implementierung des Attributes Glück)
- Zwischenspeicherpunkte in Levels, an denen nach dem Spielertod das Spiel fortgesetzt wird

Teilweise sind diese Features noch im Code bzw. auch Spiel selbst ersichtlich und enthalten (z.B. die Attribute Glück und Geschwindigkeit im Heldenmenü).

3. Architektur und Implementierung

3.1 Model

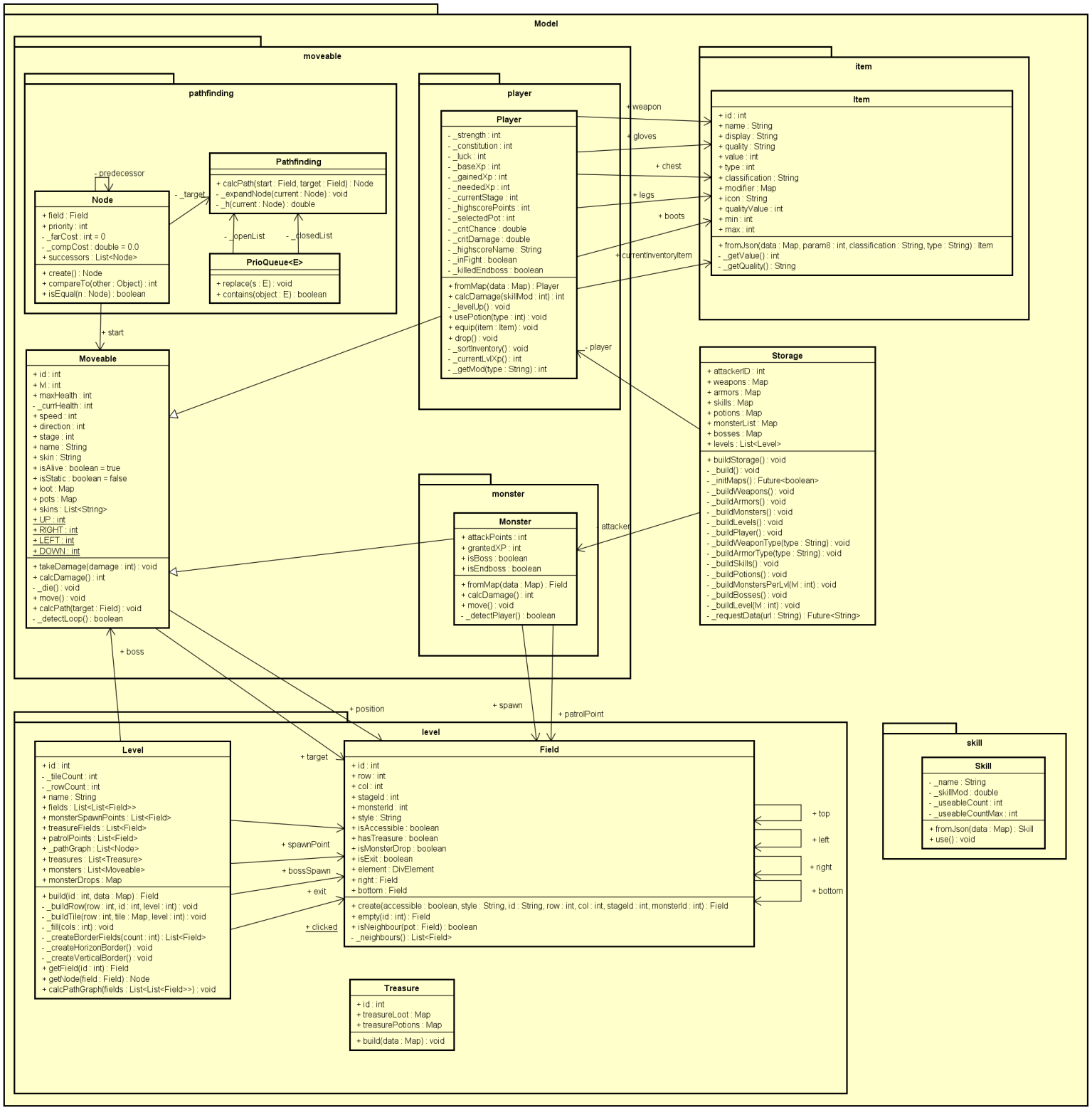

Abbildung 1 - Model Klassendiagramm

3.1.1 Item

Items sind alle Gegenstände, die vom Spieler im Spielverlauf durch das besiegen von Gegnern oder das Plündern von Schatztruhen erlangt werden können. Items haben verschiedene Attribute, durch welche sie beschrieben werden.

- *Value* bezeichnet den Wert eines Items. Wie genau dieser Wert im Spiel zum Tragen kommt hängt vom *type* des Items ab. Bei Waffen gibt *Value* den Schaden an und bei Rüstungen den Rüstungswert.
- *Modifier* bezeichnet eine Map welche verschiedene Modifikatoren enthalten kann, die sich auf den Spieler auswirken. So kann ein Item beispielsweise den Modifikator besitzen welcher dem Spieler mehr Stärke verleiht aber gleichzeitig auch Lebensenergie abzieht.
- *Icon* bezeichnet den Pfad zum Sprite eines Items, welches im Inventar angezeigt wird.

3.1.2 Level

Level sind aus einzelnen Feldern aufgebaut auf welchen dann später sowohl der Spieler als auch alle anderen Moveables und Kisten positioniert werden.

- *Boss* bezeichnet den Endgegner eines Levels, welcher besiegt werden muss um das nächste Level freizuschalten.
- *bossSpawn* bezeichnet den Ort, an dem der Boss Gegner eines Levels auftaucht.
- *monsterSpawnPoints* bezeichnet eine Liste von Feldern auf denen Gegner spawnen können.
- *pathGraph* bezeichnet eine Liste von Nodes. Diese Liste von Nodes wird benötigt um dem Spieler und den Monstern das bewegen durch ein Level zu ermöglichen.

Ein Level Objekt

- kann mittels eines Konstruktors erzeugt werden. Dabei wird die ID des neuen Levels und der Levelaufbau in Form einer Map übergeben. Beim erzeugen des Levels kommen mehrere andere Methoden zum Einsatz die alle entweder Felder auf der Vertikalen oder auf der Horizontalen anlegen.
- Kann mittels der Methode *getField()* ein bestimmtes Feld mit der übergebenen ID zurückgeben. Sollte sich kein Feld mit der übergebenen ID im Level befinden wir *null* zurückgegeben.
- Die Methode *getNode()* sucht für das übergebene Feld die passende Node und gibt diese zurück.
- Um es den verschiedenen Moveables zu ermöglichen sich durch ein Level zu bewegen muss über die Methode *calcPathGraph()* zunächst ein Graph des Levels erstellt werden. Auf die genauen Details der Wegfindung wird zu einem späteren Zeitpunkt eingegangen.

3.1.3 Moveable

Moveables bezeichnet alle Objekte, die sich in einem Level bewegen können.

- *Stage* gibt an in welchem Level sich ein Moveable befindet.
- Anders als die Items besitzen Moveables zusätzlich noch eine Liste mit Sprites für die vier verschiedenen Blickrichtungen.
- Position gibt die aktuelle Position an und *target* steht für das derzeit angesteuerte Ziel. In Verbindung zu diesen beiden steht auch noch *start*, welches die nächste Node steht die auf dem Weg zu *target* zu besuchen ist.

Ein Moveable Objekt

- Kann über die Methode *takeDamage()* einen übergebenen Schadenswert von seinem aktuellen Lebenspunkten abgezogen bekommen. Fallen die Lebenspunkte auf kleiner gleich Null, so stirbt der Moveable.
- Mit der Methode *calcDamage()* kann der Schadenswert berechnet werden der anderen Moveables zugefügt wird.
- Mittels der *move()* Methode wird die Position des Moveables im Model aktualisiert und passend zur Bewegung der aktuelle *skin* gewechselt.
- Um den Weg zu einem Zielpunkt zu berechnen, wird die Methode *calcPath()* verwendet welcher als Parameter das Zielfeld übergeben wird.

3.1.5 Player

Beim Player handelt es sich um ein Moveable, welches vom Spieler gesteuert werden kann. Im Vergleich zu Monstern ist der Player noch um einiges komplexer und wird deshalb eigenständig behandelt.

3.1.6 Node

Die Graphen, die für die Wegfindung in den einzelnen Leveln genutzt werden, sind aus Nodes aufgebaut. Ein Node steht dabei immer für ein bestimmtes Feld innerhalb des Levels.

- *Field* steht für das der Node zugeordnete Spielfeld.
- Bei *predecessor* handelt es sich um den Vorgängerknoten. Also der Knoten der vor dem jetzigen besucht wurde. *Successors* hingegen sind alle an den Knoten angrenzenden Knoten.

Node Objekte

- Können über einen Konstruktor erstellt werden, wobei ihnen eine Priorität und das Feld, das sie im Levelgraphen repräsentieren, zugewiesen werden.
- Mittels der *compareTo()* Methode werden zwei Nodes anhand ihres *f* Wertes verglichen.
- Die *isEqual()* Methode vergleicht ebenfalls zwei Nodes anhand ihrer zugeordneten Felder, bzw. deren ID.

3.2.7 Pathfinding

Die Pathfinding Klasse ist dafür zuständig im Level-Graphen den schnellsten Weg zu einem bestimmten Punkt zu finden. Um diese Aufgabe schnell und effizient zu lösen kommt der A*-Algorithmus zum Einsatz. Der Level-Graph wird bereits bei der Levelerstellung generiert. Sobald der Spieler ein Feld antippt, wird das Feld als Start und die aktuelle Spielerposition als Ziel übergeben.

So wird der Weg vom Ziel zum Spieler errechnet, der dann direkt abgegangen werden kann.

3.2 View

Der View dient der Darstellung des Spiels. Im Kern besteht dieser aus einem HTML-Dokument und einer clientseitigen Logik, geschrieben in Dart, die den DOM-Tree des Dokumentes manipuliert.

3.2.1 HTML-Dokument

Der View wird im Browser initial durch die *index.html* erzeugt. Im Verlaufe des Spiels wird der DOM-Tree des Dokumentes durch die Klasse RogueController manipuliert um Veränderungen im Spiel darzustellen.

```
[…]
<body>
    <div id="wrapper">
        <div id="home">
            <div id="start-menu" class="visible"></div>

            <div id="name-input-menu" class="invisible"></div>
            <div id="highscore" class="invisible"></div>
            <div id="how-to-play" class="invisible"></div>
            <div id="about" class="invisible"></div>

            <div id="game-over" class="invisible"></div>
            <div id="game-win" class="invisible"></div>
        </div>

        <div id="game" class="invisible">
            <div id="upper-bar">
                <div id="health-container"></div>
                <div id="stage-name-container"></div>
                <div id="command-buttons"></div>
            </div>

            <div id="dungeon">
                <div id="overlay">
                    <div id="hero-screen" class="invisible overlay"></div>
                    <div id="potions-menu" class="invisible"></div>
                    <div id="event-window" class="invisible centered"></div>
                    <div id="fighting-screen" class="invisible overlay"></div>
                </div>

                <div id="tiles"></div>
            </div>
        </div>

        <div id="portrait-message"></div>
    </div>
</body>
[…]
```

Code-Snippet 1 - index.html

3.2.2 RogueView als Schnittspiele zum HTML-Dokument

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

- Das Element mit der ID #home ist das Hauptmenü, in der der Spieler den Highscore einsehen kann und ein neues Spiel beginnen kann.
- Das Element mit der ID #game enthält das eigentliche Spiel.

- Das Element mit der ID #tiles ist das eigentliche Spielfeld, auf dem sich der Spieler bewegen kann.
- Das Element mit der ID #overlay bezeichnet eine Ebene, auf der alle Overlays angezeigt werden, beispielsweise das Inventar des Spielers oder aber der Kampfscreen.

Die CSS-Regeln sind entsprechend ihrer zuzuordnenden Kategorie in den Dateien enemies.css, player.css, style.css und tiles.css gespeichert. Die Logik wird über rogue.dart geladen.

Der RogueView selbst enthält keinerlei Logik, stellt lediglich 115 DOM-Tree-Elemente bereit und wird immer nur vom RogueController benutzt um bestimmte Manipulationen am DOM-Tree vorzunehmen.

3.3 Controller

Der Controller ist für die Ablaufsteuerung des Spiels und die Aktualisierung des Views zuständig. Er verarbeitet sowohl zeitgesteuerte Events, als auch durch den Nutzer ausgelöste Events.

Abbildung 2 - RogueController Klassendiagramm

3.3.1 Laufendes Spiel

Periodisch werden die Methoden update() und updateMoveablePositions() aufgerufen. Für jeder der Methoden gibt es einen eigenen Timer, welche unterschiedlich schnell ticken.

Beim Aufruf von update() werden zunächst die Erfahrungspunkte, die Attribute und die Lebensenergie des Spielers aktualisiert. Befindet sich der Spieler im Kampf wird der Kampfscreen abschließend aktualisiert.

Die Methode updateMoveablePositions() prüft zunächst ob sich der Spieler im Kampf befindet. Ist dies der Fall passiert weiter nichts und alle Moveables verbleiben auf ihrer aktuellen Position. Auf diese Art kann sichergestellt werden das der Spieler nicht von einem Kampf direkt in den Anderen gerät.

Befindet sich der Spieler in keinem Kampf wird der Spieler-Sprite zunächst von seiner aktuellen Position entfernt, bewegt und anschließend auf der neuen Position wieder angezeigt. Auf die selbe Art und Weise werden die Positionen der Monster aktualisiert.

Abschließend wird noch die Kamera wieder auf den Spieler zentriert und geprüft ob es auf den neuen Positionen zu einem Kampf kommt.

3.3.2 Spielende

Das Spiel kann auf zwei Arten enden. Entweder der Spieler besiegt den Endgegner und betritt das letzte Portal, oder aber der Spieler stirbt selbst.

In beiden Fällen bekommt der Spieler einen Endscreen zu sehen und der Highscore wird aktualisiert. Sobald der Spieler nun den Endscreen verlässt und ins Hauptmenü zurückkehrt, werden alle Timer gestoppt, alle Gegner, Items, Level und der Spieler selbst neu erstellt und auch das Inventar des Spielers geleert.

4. Level- und Parametrisierungskonzept

4.1 Levelkonzept

Durch die Datei ‚Storage.dart' werden bei Spielstart alle Level, Gegner, Ausrüstungsgegenstände, Fertigkeiten, sowie der Spieler selbst, ‚erstellt' und abgespeichert. Im Verlauf des Spiels werden von dort die benötigten Daten gelesen und manipuliert.

Auf die Erläuterung der Erstellung des Players sowie der Skills und Tränke per JSON Datei wird bewusst verzichtet, da nicht geplant ist dies zu erweitern.

4.1.1 Level

Jedes Level in DartRogue wird in einer JSON Datei deklarativ beschrieben.

```
[
  {
    "id": 0,
    "name": "The Breach",
    "boss": 0,
    "treasures": [
      {
        "id": 0,
        "axe": 0,
        "potions": [5, 0, 0]
      },
      {
        "id": 1,
        "hammer": 0,
        "gloves": 1,
        "potions": [0, 0, 0]
      }
    ],
    "rows": [
      {
        "id": 0,
        "row": [
          {
            "id": 0,
            "accessible": false,
            "style": "wall-top-1"
          },
          {
            "id": 1,
            "accessible": false,
            "style": "wall-top-2"
          }
[...]
```

Code-Snippet 2 - Auszug aus level0.json

Das Level selbst bekommt folgende Eigenschaften:

- id – Gibt an um welches Level es sich handelt
- name – Gibt den Anzeigenamen innerhalb des Spiels an

- boss – Gibt die Nummer des zu erscheinenden Endgegners an
- treasures – Gibt an wie viele Truhen inklusive deren Inhalt es im Level gibt
- rows – Beinhaltet alle Reihen des Levels

Level bestehen aus n x m Feldern, die durch einzelne Reihen ‚rows' beschrieben werden. Jede einzelne Reihe ‚row' beinhaltet wiederum einzelne Felder:

- id – Gibt an um welches Feld der Reihe es sich handelt
- accessible – boolscher Wert, ob das Feld vom Spieler oder einem Monster betreten werden kann
- style – Stellt den Namen der CSS-Klasse bereit zur Auswahl des richtigen Sprites
- monster – boolscher Wert, ob ein Monster auf diesem Feld stehen soll
- treasure – boolscher Wert, ob eine Truhe auf diesem Feld stehen soll
- boss – boolscher Wert, ob ein Endgegner auf diesem Feld stehen soll
- spawn – boolscher Wert, ob der Spieler auf diesem Feld im Level startet
- exit – boolscher Wert, ob das Portal zum nächsten Level sich auf diesem Feld befindet

4.1.2 Gegner

Gegner werden, ebenfalls wie Level, im JSON Format deklariert. Dabei gibt es zwei unterschiedliche Dateien, um zwischen normalen Gegnern und Endgegnern zu unterscheiden.

```
[…]
  {
    "stage": 0,
    "id": 5,
    "lvl": 2,
    "name": "Old_Magician",
    "hp": 20,
    "attack": 3,
    "speed": 10,
    "grantedXP": 20,
    "skin": "skull",
    "loot": {
      "legs": 1,
      "potions": [0, 0, 0]
    }
  },
  {
    "stage": 1,
    "id": 0,
    "lvl": 3,
[…]
```

Code-Snippet 3 - Auszug aus monster.json

Einen Gegner kann man wie folgt erstellen:

- stage – Gibt an welchem Level der Gegner zugeordnet ist
- id – Gibt die Nummer des Gegners in dem unter ‚stage' bestimmten Level an
- lvl – Gibt das Gegnerlevel an, um durch das unter 4.2 beschriebene Scaling die Stärke festzulegen

- name – Gibt den Namen des Gegners an und dient zur Zuordnung des richtigen Gegnersprites im Kampf
- hp – Gibt die Basislebenspunkte des Gegners an
- attack – Gibt den Basisangriff des Gegners an
- speed – Gibt die Geschwindigkeit des Gegners an
- grantedXP – Gibt die Basiserfahrungspunkte an
- skin – Dient der Zuordnung des richtigen Gegnersprites im Level
- loot – Gibt an welche Ausrüstungsgegenstände und Tränke ein Gegner für den Spieler fallen lassen kann, äquivalent zu dem Inhalt von Truhen nutzbar

4.1.3 Ausrüstungsgegenstände

Analog zu Leveln und Gegnern werden Ausrüstungsgegenstände ebenfalls in JSON deklariert

```
[…]
  {
    "id": 1,
    "name": "Heavy Boots",
    "display": "Boots",
    "type": 5,
    "value": 2,
    "quality": 1,
    "mods": {
      "health": 1,
      "strength": 1
    },
    "icon": "boots/boots_leather_big.png"
  },
  {
    "id": 2,
    "name": "Sabatons",
[…]
```

Code-Snippet 4 - Auszug aus boots.json

Ausrüstungsgegenstände werden wie folgt erstellt:

- id – Gibt die Nummer des Gegenstandes an
- name – Gibt den Namen des Gegenstandes an
- display – Gibt die Anzeigekategorie an
- type - Gibt die Art von Objekt an, jede Unterkategorie von Gegenständen hat eigene Nummer (Waffen: 0, Rüstung: 1-5)
- value – Gibt bei Waffen den Angriffswert, bei Rüstung den Verteidigungswert an
- quality – Gibt die Qualitätsstufe des Gegenstandes an (Werte: 0-4), ausschlaggebend für die Farbe des Hintergrundes und der Schrift des Gegenstandes
- mods – Gibt an welche Boni auf Attribute der Gegenstand dem Spieler bringt
- icon - Dient der Zuordnung des richtigen Sprites

4.1.4 Steigender Schwierigkeitsgrad

Der steigende Schwierigkeitsgrad wird während des Spielfortschritts dadurch gewährleistet, dass (wie unter 4.2 genauer beschrieben) einerseits die Gegner schneller stärker, als auch die Level größer werden und man gegen mehr Gegner als in vorherigen Levels kämpfen muss. Dabei sinkt stetig die

Zahl der von den Gegnern fallengelassenen Tränken, so dass der Spieler sich immer weniger selbst heilen kann, und der Spieler wird langsamer stärker als die Gegner. Daher ist er auf neue Ausrüstungsgegenstände angewiesen, um weiterhin den Gegnern standhalten zu können.

4.2 Parametrisierungskonzept

Die Parametrisierung des Spiels wird über statische Werte in der Datei ‚config/Settings.dart' realisiert. In dieser befinden sich sowohl Pfade bestimmten Ordnern, Konstanten für die Refreshrate, als auch Parameter zum Balancing.

- _dataPath – Pfad zu in JSON deklarierten Gegenständen, Leveln, Monster, Spieler, Fertigkeiten
- Image-Pfade – Pfade zu benötigten Sprites des Spiels

Über die folgenden Integer Werte lassen sich die Attribute des Spielers beeinflussen

- _strengthMod – Wert für einen Punkt des Stärkeattributs, beeinflusst den Schaden des Spielers
- _constMod – Wert für einen Punkt des Ausdauerattributs, beeinflusst das maximale Leben des Spielers
- _luckMod – Wert für einen Punkt des Glücksattributs, beeinflusst das Glück des Spielers

Über die folgenden Double Werte lässt sich maßgeblich das Balancing des Spiels beeinflussen:

- _monsterScaling – Gibt den Multiplikator der Gegnerattribute pro Level des Gegners an
- _playerStatScaling – Gibt den Multiplikator der Spielerattribute pro Level des Spielers an
- _playerXpScaling – Gibt den Multiplikator der benötigten Erfahrung des Spielers pro Level an

Die Implementation zielt darauf ab, dass die Werte exponentiell wachsen, so dass es für den Spieler gegen Gegner höheren Levels schwerer wird.

Über die folgenden Konstanten lassen sich die Aktualisierungsraten des Spiels festlegen

- gameRefreshRate – Eine Konstante, die festlegt in welchen Abständen das Spiel allgemein aktualisiert wird
- movementRefresRate – Eine Konstante, die festlegt in welchen Abständen die Bewegungen des Spielercharakters und der Gegner aktualisiert werden (primärer Wert für die Spielgeschwindigkeit)

5. Nachweis der Anforderungen

5.1 Nachweis der funktionalen Anforderungen

id	Kurztitel	Erfüllt	Teilw. erfüllt	Nicht erfüllt	Erläuterung
AF-1	Single-Player-Game als Single-Page-App	x			DartRogue ist ein Single-Player-Game, das als Single-Page-App geschrieben wurde. Erreichbar ist das Spiel unter schrotty.github.io/DartRogue (GitHub Pages).
AF-2	Balance: technische Komplexität und Spielkonzept	x			Um die technische Komplexität nicht zu übersteigen, sind ein paar Features, unter 2.2.5 erwähnt, nicht in das Spiel implementiert worden. Dies führt auch dazu, dass das Spielkonzept von DartRogue nicht zu komplex geworden ist.
AF-3	DOM-Tree-basiert	x			Wie in ‚Architektur und Implementierung' beschrieben, wird das MVC-Prinzip genutzt und der DOM-Tree als View genutzt. Änderungen am View werden nur durch die Manipulation des DOM-Trees erreicht.
AF-4	Target Device: Smartphone	x			Das Design der Menüs und des eigentlichen Spiels sind für Smartphones konzipiert, jedoch aber auch in einem Browser auf dem PC tendenziell spielbar. Lediglich Firefox bietet nicht die volle Unterstützung.
AF-5	Mobile First Prinzip	x			Die gesamte Steuerung, sowie die Positionierung der einzelnen Elemente ist auf mobile Endgeräte zugeschnitten worden. Auf spezifisch mobile Eingabemethoden wurde bewusst verzichtet, da diese dem Spielkonzept nicht dienlich gewesen wären.
AF-6	Schnell & intuitiv erfassbar, erzeugt Spielfreude	x			Verschiedene Testpersonen (von absoluter Laie bis täglicher Spieler) haben bestätigt, dass man sich im Spiel schnell zurechtfindet. Im Verlauf des Spiels stellt sich ein gewisser ‚Sammeldrang' ein, so dass man nach immer besseren Ausrüstungsgegenständen sucht, um den Endgegner zu besiegen. Darüber hinaus wird eine gewisse Spieltiefe geboten.
AF-7	Levelkonzept	x			Level werden deklarativ in Form von JSON Dateien beschrieben. Im Verlauf des Spiels werden die Level größer und beinhalten schwierigere Gegner.
AF-8	Speicherkonzept Client-seitig	x			Für das Konzept der Client-seitigen Speicherung wird der local storage benutzt. Dieser dient lediglich zur Speicherung der erzielten Highscores.
AF-9	Dokumentation	x			Die Dokumentation nutzt die Referenzdokumentation des SnakeGames als Vorlage.

Tabelle 1 - Nachweis der Anforderungen

5.2 Verantwortlichkeiten im Projekt

Komponente	Detail	Asset	Ruben Maurer	Jannik Schwardt	Anmerkung
Model	Moveable	lib/src/content/moveable/*	V		
	Player	lib/src/content/moveable/player/*	U	V	
	Player-JSON	web/data/player/*	V	U	
	Monster	lib/src/content/moveable/monster/*	U	V	
	Monster-JSON	web/data/monster/*	U	V	
	Item	lib/src/content/item/*	U	V	
	Item-JSON	web/data/item/*	U	V	
	Pathfinding	lib/src/content/pathfinding/*	V		
	Level	lib/src/content/level/*	V	U	
	Level-JSON	web/data/level/*	U	V	
	Storage	lib/src/content/*	V	U	
	Skill	lib/src/content/skill/*		V	
	Skill-JSON	web/data/skill/*			
View	HTML-Dokument	web/index.html	V	U	
	Gestaltung	web/style/style.css	V	U	
		web/style/enemies.css	U	V	
		web/style/player.css	U	V	
		web/style/tiles.css	V	U	
	Sprites	web/img/*	V	U	
	RogueView	lib/src/RogueView.dart	V	U	
Controller	Eventhandling	lib/src/RogueController.dart	U	V	Eigentlich Arbeitsteilung
Dokumentation	DartRogueReport	docs/DartRogueReport.docx	U	V	Eigentlich Arbeitsteilung

Tabelle 2 - Verwantwortlichkeiten im Projekt

V = verantwortlich
U = unterstützend

5.3 Zusätzlich verwendete Libraries

Da es lediglich im Browser ‚Firefox' zu Problemen kam, ist die Library ‚browser_detect' in der Version 1.0.4 verwendet worden. Diese wird dazu genutzt um Browserabhängig andere Parameter beim Scrollen zu verwenden, so dass der Spielercharakter auch in Firefox dauerhaft zentriert ist.

5.4 Quellen für Sprites und Assets

Von folgenden Personen sind Sprites und Assets in DartRogue verwendet worden:

Ausrüstungsgegenstände: twitter.com/DerNaderer
Level-, Monster- und Spielersprites: 0x72.itch.io

Die verwendeten Sprites und Assets stehen unter CC-0 Lizenz.
Ebenfalls sind die o.g. Personen im Startmenü von DartRogue unter ‚About' erwähnt und einsehbar.

5.5 Freigabe zu Präsentationszwecken

Hiermit bestätigen wir, dass das Spiel ‚DartRogue' von der Fachhochschule Lübeck zu Präsentationszwecken verwendet werden darf.