Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim Geisenheim

Dokumentation zum Projekt

Jump & Run E-Learning Game
Bürgermeister Run

Team:
OneNil

Projektbeteiligte:
Abasin Halim, Eren Yildiz, Mirco Cesarano, Tobias Pawlik, Zhivko Antonov

Projektbetreuer:
Prof. Dr. Michael Ricken

Wahlprojekt der Hochschule RheinMain
Fachbereich DCSM WS17/18

Inhalt

1 Einführung in das Projekt

1.1 Projektentstehung

Im Rahmen des Wahlprojektes der Hochschule RheinMain für den Studiengang Wirtschaftsinformatik (B. Sc.) entstanden mehrere Projekte zum Thema „Spielerisches E-Learning für Wiesbadener Grundschulen“. Bei den Projekten handelt es sich um konkrete Umsetzungen zu den gegebenen Anforderungen der Stadt Wiesbaden welche im Rahmen von „Projekt Heimatschule“ erhoben wurden. Die Zielgruppe für das E-Learning Projekt sind Grundschüler der 3. und 4. Klassen. Das Team bestehend aus 5 Teilnehmern, beschäftigte sich mit der Konzeption und Umsetzung des Projekts, auf welches wir in dieser Dokumentation eingehen.

1.2 Projektbeteiligte

1.2.1 Gruppenmitglieder

Das Projektteam setzt sich aus den folgenden Studierenden zusammen

1. Abasin Halim (487690)
2. Mirco Cesarano (287583)
3. Tobias Pawlik (887733)
4. Zhivko Antonov (282371)
5. Eren Yildiz (282680)

1.2.2 Stakeholder

Folgende Stakeholder wurden von uns identifiziert:

Herr Dr. Weichel → Stellvertreter für die Stadt Wiesbaden und dem Projekt Heimatschule
Michael Ricken → Auftrag Beschaffer und universeller Ansprechpartner
Projektteam → Anforderungserheber und Umsetzer
Grundschüler der dritten/vierten Klasse (in Form von Personas) → Spielen das Spiel

1.3 Projektziel

Das Ziel des Projektes ist das konzipieren und umsetzten eines Spiels für die E-Learning Plattform der Stadt Wiesbaden. Das Spiel soll es ermöglichen Schülern der dritten und vierten Klasse, sich spielerisch Wissen zur Stadt Wiesbaden anzueignen. Das fertige Spiel soll auf der Seite der Stadt Wiesbaden spielbar sein.

Anforderungen

Um das Projektziel zu erreichen wurden uns mehrere Anforderungen genannt.

Hier eine Auflistung dieser:

- Es soll ein Browsergame entworfen und programmiert werden
- Als Basistechnologien sollen JavaScript, HTML 5 sowie CSS genutzt werden
- Der Inhalt des Spiels sollte dynamisch zu befüllen sein
- Das Spiel soll am Desktop und am Tablet spielbar sein
- Das Spiel muss auf einen Tomcat-Server abspielbar sein
- Die Zielgruppe sind Grundschulkinder in den Klassen 3 und 4
- Die Zielgruppe sollte Spaß am Spielen haben
- Das Spiel soll einen positiven Lerneffekt auf die Zielgruppe haben
- Es soll Wissen über die Stadt Wiesbaden vermittelt werden
- Die Ergebnisse sollen am Ende des Projektes präsentiert werden
- Die Abgabe des Spiels mit allen Komponenten ist am 30.01.18

1.4 Vorgehensweise

1.4.1 Organisation

Zur Umsetzung des Projekts wird eine projektinterne Organisation benötigt, diese bezieht sich auf die Kommunikation, Verteilung der Aufgaben sowie die Bereitstellung aller Dokumente für die Gruppenmitglieder.

1. Zur Kommunikation innerhalb der Gruppe wurden die Praktikumstermine der Hochschule wahrgenommen und weitere Gruppentreffen jeden Montag entgegengenommen. Zusätzlich wurde eine WhatsApp-Gruppe gegründet und Skype genutzt.

2. Für die Dokumentation und genaue Zuordnung der Aufgaben wurde scrumwise.com genutzt. Der kostenlose Zugang wurde durch Herrn Ricken ermöglicht.

3. Für die Bereitstellung zum Lesen, Erstellen und Verändern von allen Dokumenten des Projekts wurde Gitlab eingesetzt.

1.4.2 Projektstart und –Abschluss

Das Projekt startete im ersten Praktikum des Wahlprojekts am 23.10.17. Nach dem Zusammenschließen der Arbeitsgruppen wurde das entsprechende Projekt auf Scrumwise erstellt. Zu Beginn wurden uns die Grundlegenden Anforderungen genannt. Weiterer Input für die Vorgehensweise wurde uns durch die Vorlesung von Herrn Ricken gegeben.
Das Projekt musste zum Ende Semesters, am 29.01.18 abgegeben werden. Es wurden die Dokumentation und der Code abgegeben. Am 30.01.2017 stellen wir das gesamte Projekt in Form einer PowerPoint-Präsentation vor.

1.5 Schwerpunkte der Teammitglieder

Während des Projektes wurden den Gruppenmitgliedern Aufgaben zugeteilt. Damit hatte jedes Mitglied sein eignen Schwerpunkt innerhalb des Projektes. Die Schwerpunkte lassen sich nochmal in Kategorien eingliedern. Folgende Kategorien ergaben sich den bearbeiteten Aufgaben.

1. Konzeption und Gestaltung
2. Frontend-Entwicklung
3. Backend-Entwicklung

Für die Kategorie – **Konzeption und Gestaltung** – war **Mirco Cesarano** verantwortlich.

Folgende Aufgaben hat er während des Projekts bearbeitet:
- Erstellung eines Spielekonzepts inkl. Dokumentation
- Erstellung von Spielelementen wie Gegnern oder Objekte
- Erstellung der Map zum warmen Damm

Für die Kategorie – **Frontend-Entwicklung** – waren **Tobias Pawlik, Abasin Halim** und **Eren Yildiz** verantwortlich.

Folgende Aufgaben hat **Tobias Pawlik** während des Projekts bearbeitet:
- Codestrukturierung
- Erstellung der States zum Kartenladen (+ Tiledmap-Einbindung)
- Implementierung Objekte (Life, Score, Checkpoint, Coin)
- Menü- Funktionalitäten (+ Game-Over-Screen)
- Erstellung der Map zur russischen Kirche

Folgende Aufgaben hat **Abasin Halim** während des Projekts bearbeitet:
- Implementierung Objekte (Enemys, DestroyObjects)
- Tiledmap-Einbindung
- Menü- Funktionalitäten
- Menü – Design (+ Endscreen)
- Sounds
- Erstellung der Map zum Neroberg

Folgende Aufgaben hat **Eren Yildiz** während des Projekts bearbeitet:
- Implementierung der Hauptfunktionen in der Player Klasse (Fähigkeiten, Steuerelemente, Steuerlogik)
- Erstellung der Kurhaus-Map

Für die Kategorie – **Backend-Entwicklung** – war **Zhivko Antonov** verantwortlich.

Folgende Aufgaben hat er während des Projekts bearbeitet:
- Codestrukturierung
- Datenbankkonzept
- Implementierung Quizscreen Feature mit Frontend und Backend
- Deployments

2 Konzept und Spielidee

Die Spielidee

Um das Ziel des Projektes erreichen zu können, sind wir der Meinung gewesen, dass das Hauptmerkmal des Spiels für den Schüler der Spielspaß sein muss und nicht die sich dahinter verbergenden Informationen zur Stadt. So haben wir ein Spiel konzipiert, das das Wissen innerhalb des Spieles einbaut und am Ende auch genutzt werden muss. Dabei sind die Informationen aber so eingebaut, dass sie zum eigentlichen Bestandteil des Spieles gehören und so als Teil des Spiels akzeptiert werden. Im Folgenden werden wir die genaue Spielidee hinter unserem Spiel mit dem Namen „Bürgermeister Run“ weiter ausführen.

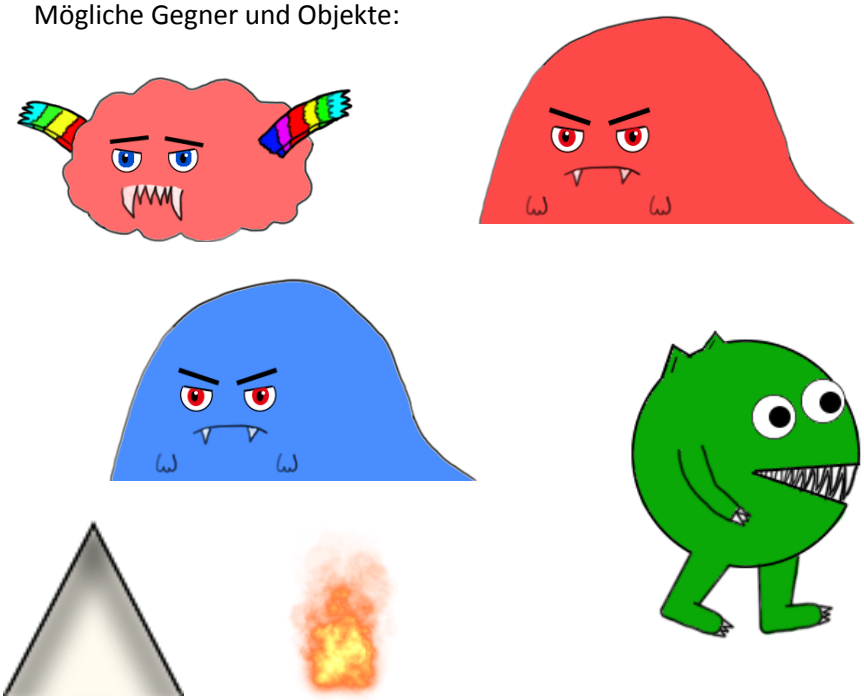
Bei dem Spiel Bürgermeister Run handelt es sich um ein Spiel aus Genre „Jump & Run“, bei dem der Spieler beim Spielen der Level thematisch gebundene Inhalte zur Stadt Wiesbaden erlernen soll. Die Hauptfigur wird wie der Name verrät durch den Bürgermeister der Stadt Wiesbaden repräsentiert.

Ziel des Spielers ist es die Level abzuschließen. Dabei gibt allerdings, im Gegensatz zu einem normalen Jump & Run, zwei Hürden:

1. Man muss das Level beenden ohne zu sterben
2. Man muss am Ende des Levels Fragen beantworten um das nächste Level freizuschalten

Zur 1: Der Spieler stirbt, wenn er aus der Map fällt, oder von einem Gegner getroffen wird. Damit nicht jede Berührung zum Tod führt, bekommt der Spieler vor jedem Level drei Leben. Sollte der Spieler von einem Gegner getroffen werden verliert er ein Leben, wenn er 0 Leben erreicht hat, gilt das Level als verloren und der Spieler muss das Level neustarten. Gegner können durch jegliche Art von Objekten repräsentiert werden. Damit sind auch Objekte wie Zacken oder Feuer als Gegner denkbar, da diese der Hauptfigur Schaden zufügen.

Mögliche Gegner und Objekte:



Der Bürgermeister hat auch sein eignes Hilfsmittel um die Gegner zu besiegen. Er ist in der Lage eine Papierkugel zu werfen welche den Gegner umbringt. Dies ist aber nicht einzige Möglichkeit, alternativ kann er auf den Kopf des Gegners springen.

Papierkugel des Bürgermeisters:

Die Leben retten den Spieler nicht vor dem Fallen aus der Map. Fällt der Spieler aus der Map muss er das Level neustarten oder von einem Checkpoint aus weiterspielen. Sollte der Spieler von einem Checkpoint aus weiterspielen, bekommt er für das Fallen ein Leben abgezogen.

Als weiteres Element des Spiels, sind auf der gesamten Map Coins verteilt. Die Coins sehen aus wie ein Stück Papier, welches einen Test mit der Note 1+ repräsentiert. Beim aufsammeln erhöht sich die Score des Spielers. Dies soll einen weiteren Anreiz für den Spieler darstellen, da der Schüler sich mit andern vergleichen kann. Zusätzlich generiert jeder aufgehobene Coin eine neue Papierkugel das als Hilfsmittel verwendet werden kann. Jede $1500 \cdot n$ Punkte durch die die Score erhält der Spieler ein weiteres Leben. Somit bietet die Coins viele Vorteile die vom Spieler genutzt werden sollten.

Coin im Spielfeld:

Ein Punktesystem dient dem Schüler als Anreiz sich mit andern Spielern zu messen. Dadurch wird mehr über das Spiel gesprochen, beispielsweise auf dem Pausenhof. Das wiederrum könnte die Anzahl der Spieler hochschrauben.
Das Punktesystem setzte sich wie folgt zusammen:

- Aufsammeln von Münzen → 50 Punkte
- Besiegen von Gegnern → Zwischen 100 und 200 Punkten (Abhängig vom Gegner)
- Alle 1500 Punkte → Ein Leben

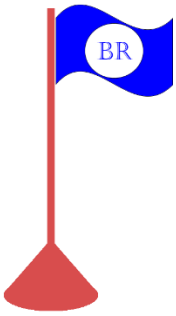
Score oben links:

Damit der Schüler nicht bei jedem Game-Over das Level neu starten muss, hat der Bürgermeister zu Beginn eines jeden Levels 3 Leben. Durch sammeln der Punkte kann sich das Leben erhöhen.

Leben oben links:

Zusätzlich besitzt jedes Level einen Checkpoint. Dieser dient als Zwischenspeicher und ermöglicht dem Spieler beim Sterben nicht das gesamte Level neu zu starten, sondern vom einem bestimmten Punkt aus weiter zu spielen. Der Checkpoint wird beim Erreichen durch eine Berührung aktiviert. Dies soll den Frust des Spielers verhindern

Checkpoint im Spielfeld:



In der Hinsicht unterscheidet sich Bürgermeister Run nicht von einem klassischen Jump & Run, der wesentliche Unterschied wird in Punkt 2 erläutert.

Zur 2: Am Ende eines jeden Levels muss der Spieler Fragen zur Stadt beantworten. Die Antworten zu den Fragen sind in Form von Fakten in den Levels verteilt. Dabei bilden Kisten Fakten ab. Wenn der Spieler gegen eine Kiste springt wird ihm ein Fakt angezeigt. Diese Fakten beinhalten die Informationen um am Ende des Levels die Fragen zu beantworten. Die Kisten werden somit zum elementaren Teil des Spieles, welche der Spieler gerne sucht und findet.

Beispiel mit Kiste und Fakt:

Am Ende des Levels werden je 3 Fragen mit je 4 Antwortmöglichkeiten gestellt. Der Spieler hat bei der Beantwortung jeder Frage 2 Möglichkeiten. Beantwortet der Spieler eine Frage zweimal falsch, so muss er das Level erneut spielen. Dabei werden neue zufällige Fakten und Fragen zum Level ausgesucht.

Themen

Jedes Level gehört einem bestimmten Thema der Stadt Wiesbaden an.
Die Themen wurden aus dem uns zur Verfügung gestellten Dokumente der Stadt Wiesbaden abgeleitet und sind:

1. Neroberg
2. Russische Orthodoxe Kirche
3. Kurhaus
4. Warmer Damm

Dabei sind die Level so konzipiert, dass sie den Themen angepasst sind, d. h. beispielsweise das Level Nerberg ähnelt einer Berglandschaft. Die Fakten und die dazu passenden Fragen mit den entsprechenden Antwortmöglichkeiten werden vor jedem Spiel zufällig vom System gewählt, passen aber zur Thematik des Levels. So werden beispielweise im Neroberg Level ausschließlich Fakten und Fragen und Antworten zum Neroberg gewählt.

Die Fakten, Fragen und Antworten können vom Admin dem Spiel hinzugefügt werden. Dadurch wird der Inhalt des Spiels dynamisch erzeugt und das Spiel kann nach und nach erweitert werden.

Für die Pflege neuer Inhalte werden die einzelnen Datensätze in einem Formular eingetragen und beim Abschicken in eine .json-Datei hinzugefügt. Mehr Details dazu gibt es im Kapitel 4.6 Backend nachlesen.

3 Spielanleitung und Steuerung

3.1 Spieleanleitung

Die Spielanleitung dient dem Spieler um die Funktionsweise und Mechaniken des Spieles kennenzulernen. Da es sich um ein bekanntes Spielegenre handelt, wird die Anleitung nicht implizit angezeigt, sondern muss vom Spieler explizit im Main-Menü unter Optionen aufgerufen werden. Alternativ lässt sich die Anleitung auch im Pause-Menü unter Spielanleitung aufrufen.

Anleitung:

1. Versuche das Level mit mindestens 1. Leben abzuschließen, solltest du mit einem Leben sterben, musst du das Level neustarten.
   Du verlierst Leben, wenn:
   - Dich ein Gegner berührt
   - Dich eine Zacke oder Feuer berührt
   - Du in Wasser fällst
   - Du in Lava fällst
   - Du aus dem Spielfeld fällst

   Du kannst Gegner besiegen indem du:
   - Auf ihre Köpfe springst
   - Sie mit einer Papierkugel abwirfst

2. Um das Level abzuschließen musst du am Ende des Levels die Kiste berühren.
   Um das Ende zu erreichen musst du den Bürgermeister mit mindestens 1. Leben zum Ende navigieren, dabei warten mehrere Hindernisse auf dich die in Punkt 1 der Anleitung aufgeführt wurden.

3. Sammle die Coins ein. Mit jedem Coin den du sammelst erhältst du Punkte und einen weiteren Papierwurf der dir ermöglich deine Gegner zu besiegen. Des weiteren erhöhst du deinen Score um 50 Punkte. Alle 1500 Punkte erhältst du ein weiteres Leben.

4. Nutze den Doppelsprung um weite Sprünge auszuführen. Diese ermöglichen dir Hindernissen auszuweichen und größere Löcher im Spielfeld zu überqueren.

5. Besiehe Gegner, jeder besiegte Gegner bringt dir zwischen 100-200 Punkte. Alle 1500 Punkte erhältst du ein weiteres Leben.

6. Wenn du das Ende des Levels erreich hast, musst du 3 Fragen beantworten. Du hast für jede Frage zwei Versuche. Beantwortetest du eine Frage zweimal falsch, so hast du verloren und musst das gesamte Level erneut spielen.
   Um die Fragen beantworten zu können ist es wichtig, alle drei Fragezeichenkisten im Level zu finden und mit dem Bürgermeister zu berühren. Die Kisten zeigen dir dann einen Fakt an, welcher der für die Beantwortung der Frage helfen soll. Aber Achtung, es werden neue Fakten und Fragen erzeugt, wenn ein Level neu gestartet wird. Also aktiviere alle Kisten erneut und versuche dir die Fakten zu merken.

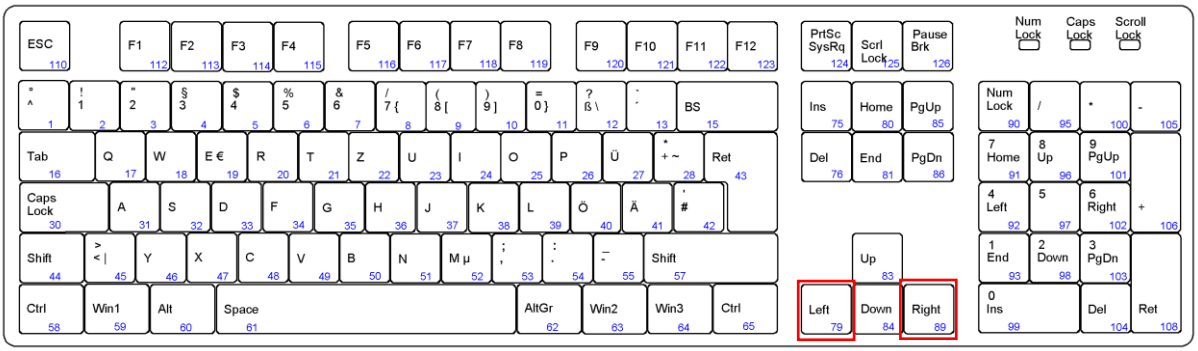
7. Eine Beschreibung zur Steuerung findest du im Menü „Steuerung“

3.2 Steuerung

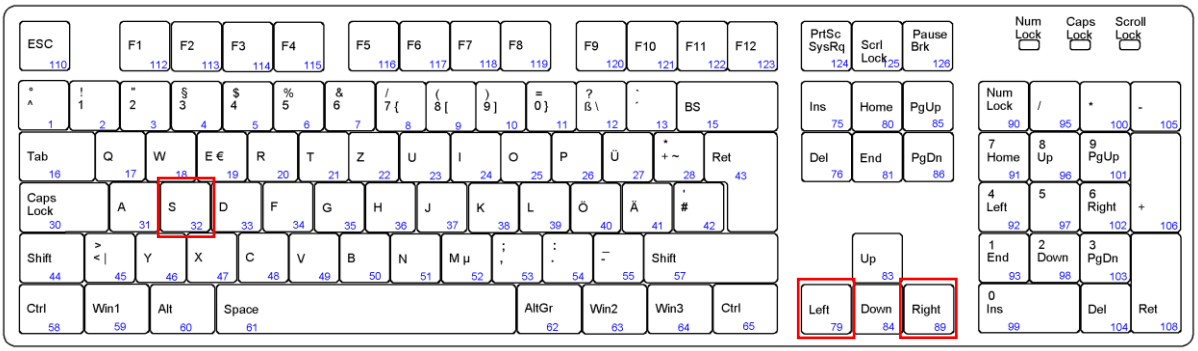
3.2.1 Desktop-PC

Am Desktop-PC erfolgt die Steuerung über die Tastatur.

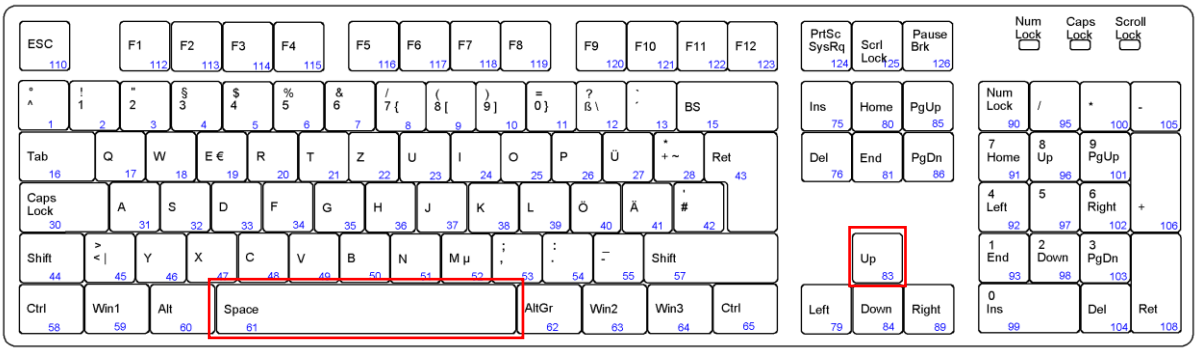
Die linke und rechte Pfeiltaste geben an in welche Richtung sich der Bürgermeister bewegt.



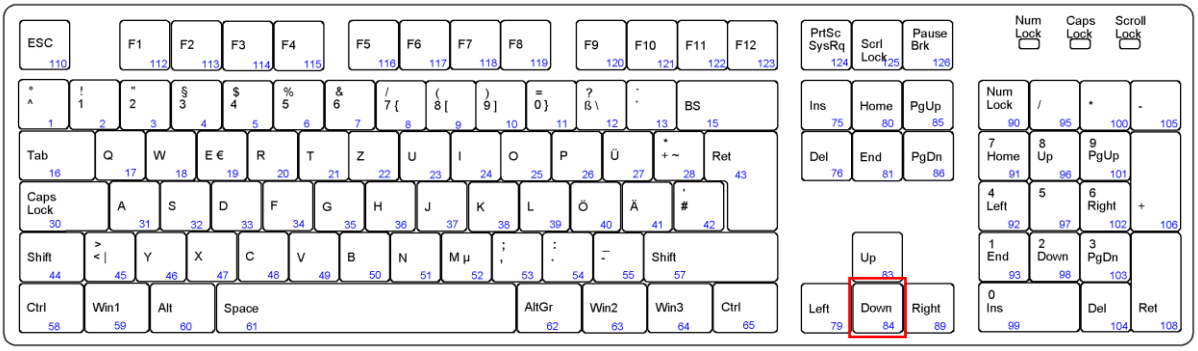
Drückt man zusätzlich die Taste „S“ bewegt sich die Figur mit doppelter Geschwindigkeit.



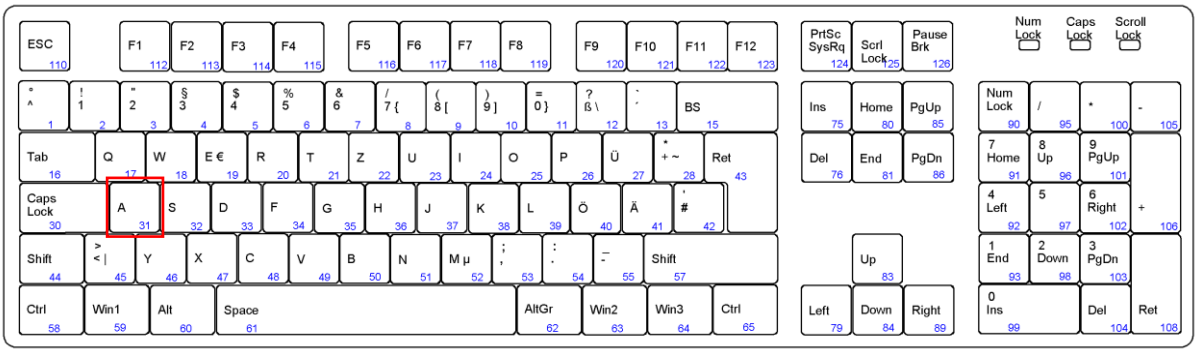
Mit der oberen Pfeiltaste kann man springen, ist man in der Luft und drückt erneut die obere Pfeiltaste kommt es zum Doppelsprung. Die selbe Funktionalität bietet die Leerteste.



Mit der unteren Pfeiltaste hingegen kann man sich bücken.



Die „a“ Taste bietet dem Spieler die Option eine Papierkugel zu werfen. Die Entscheidung die „a“ Taste zu verwenden, resultiert daraus, dass sie neben der „S“ Taste liegt und so auch während des Rennes Problemlos getätigt werden kann

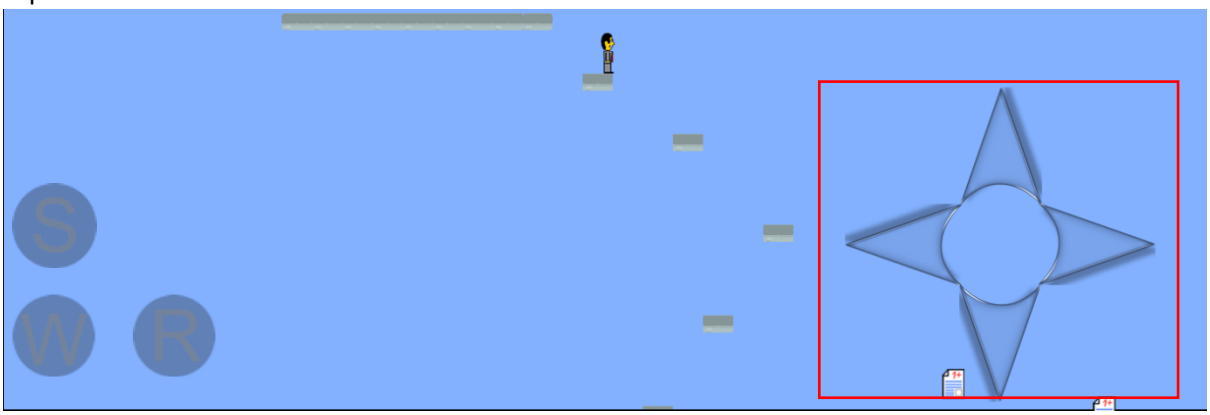


3.2.2 Tablet

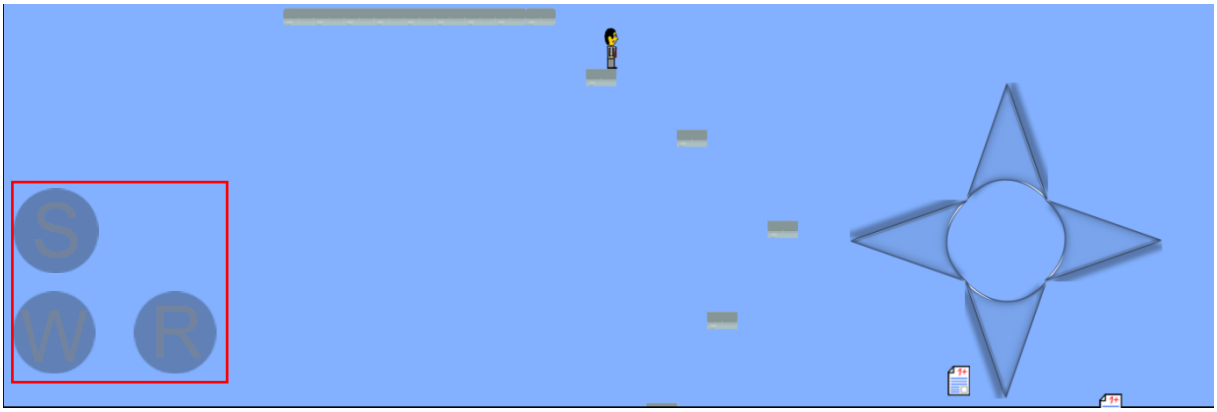
Um das Spiel auch Tablet spielen zu können, wird eine Optionale Steuerung angeboten. Diese wird automatisch eingeblendet, wenn der Spieler ein Tablet benutzt. Sollte dies nicht funktionieren, lässt sich die Optionale Steuerung auch Optionsmenü aktivieren.

Die optimale Steuerung wurde mittels virtuellen Buttons realisiert. Die Buttons repräsentieren dabei die verwendeten Tasten der Tastatur und haben auch die selbe Funktionalitäten.

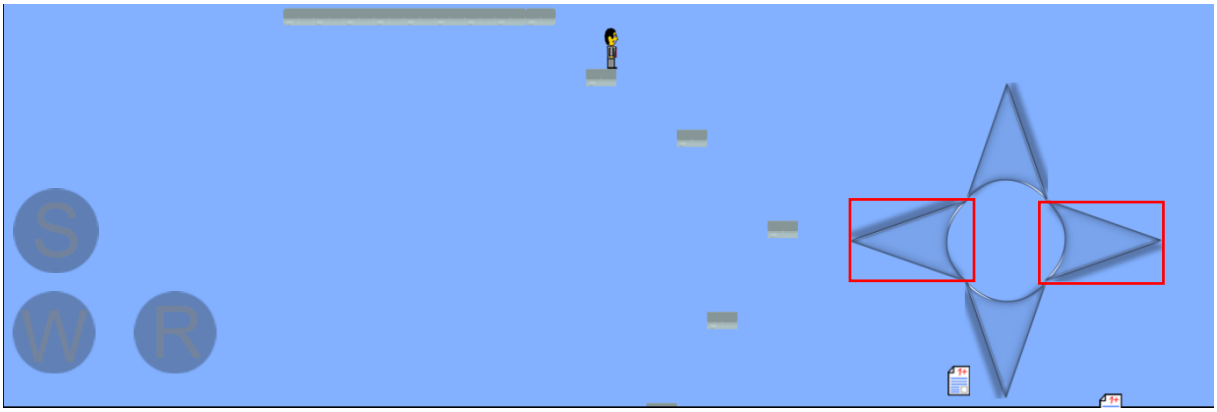
Im rechten Bereich des Spiels wird ein Steuerkreuz eingeblendet, dass alle vier Pfeiltasten repräsentiert.



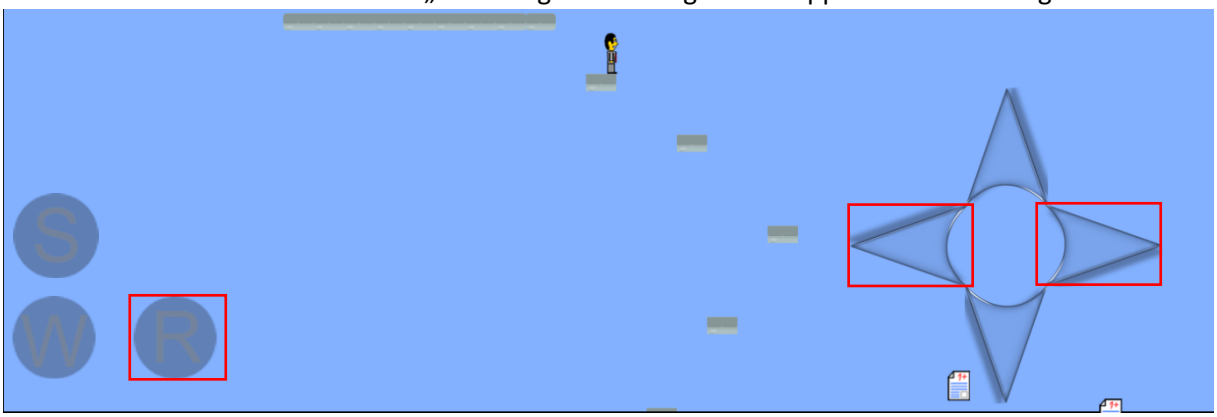
Im linken Bereich hingegen werden die Button „W“ für werfen, „R“ für rennen und „S“ für springen eingeblendet.



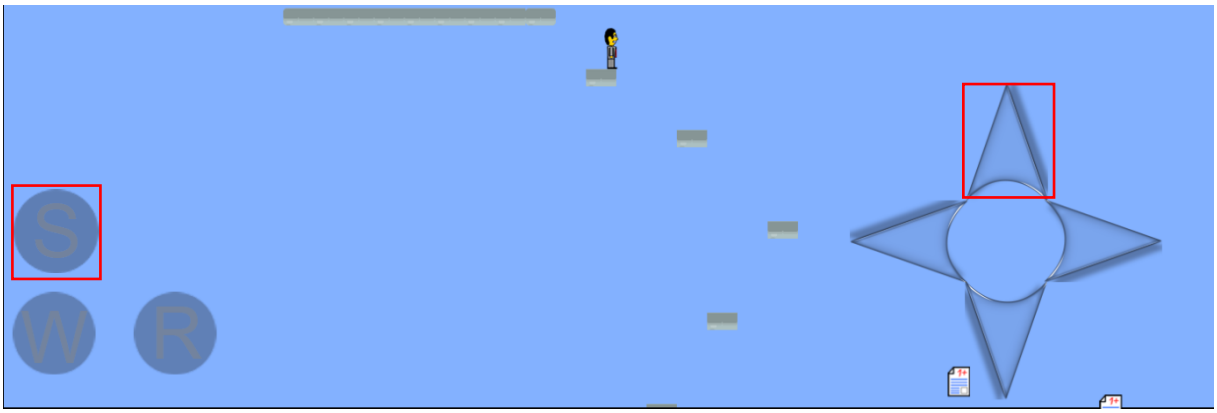
Die linke und rechte Pfeiltaste des virtuellen Steuerkreuzes geben an in welche Richtung sich der Bürgermeister bewegt.



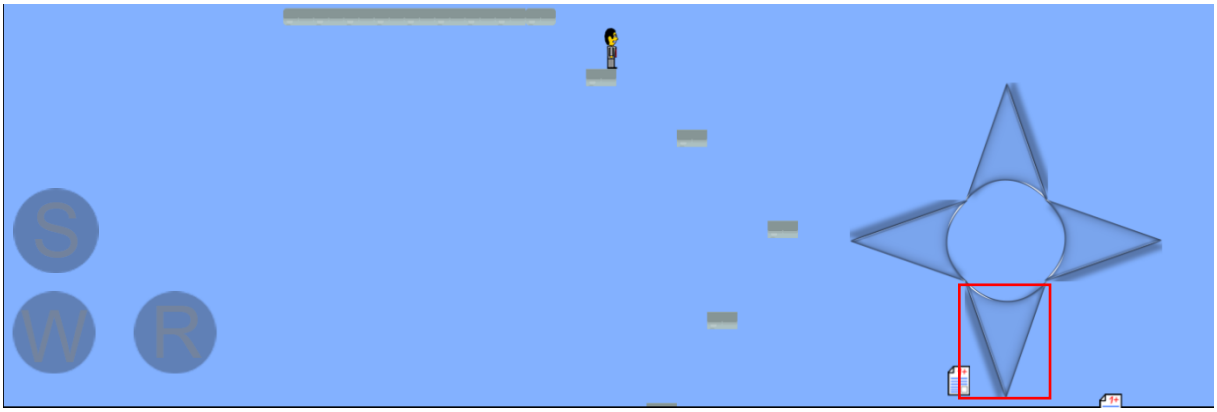
Berührt man zusätzlich den Button „R“ bewegt sich die Figur mit doppelter Geschwindigkeit.



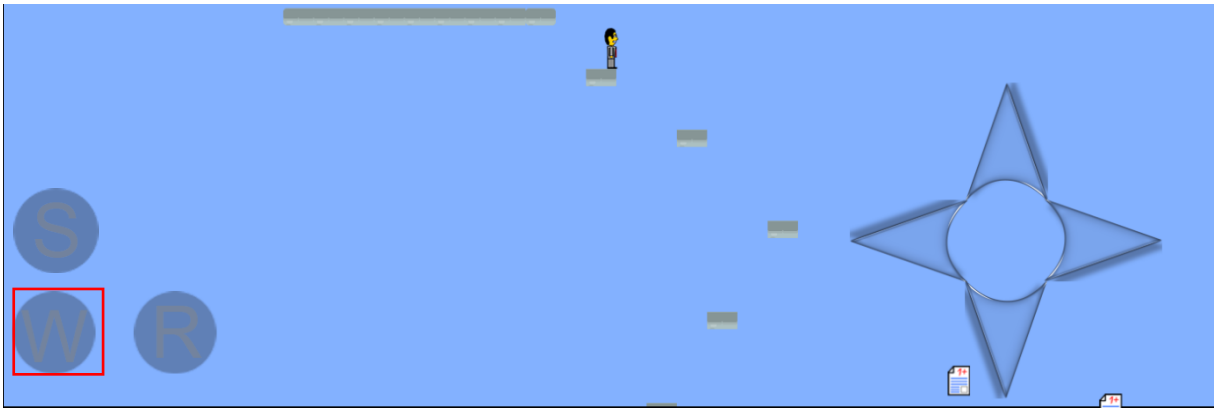
Mit dem oberen Pfeilbutton des virtuellen Steuerkreuzes kann man springen, ist man in der Luft und drückt erneut den oberen Pfeilbutton kommt es zum Doppelsprung. Die selbe Funktionalität bietet der Button „S“.



Mit dem unteren Pfeilbutton hingegen kann man sich bücken.



Der „W“ Button bietet dem Spieler die Option eine Papierkugel zu werfen.



4 Technische Architektur

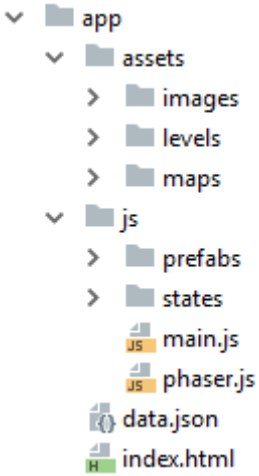
4.1 Frontend

4.1.1 Warum Phaser

Da es unser Ziel war innerhalb von drei Monaten ein funktionsfähiges Spiel zu entwickeln haben wir gezielt nach Frameworks gesucht, welche es uns ermöglichen dieses Ziel zu erreichen. Dabei haben wir uns mehrere Frameworks und auch Engines angeschaut und evaluiert. Wir sind zu dem Schluss gekommen Phaser zu benutzen. Phaser ist ein 2D-Spiele Framework, dass speziell für HTML5 Spiele entwickelt wurde und auf Desktop sowie Mobilen Endgeräten verwendet werden kann.

4.1.2 Aufbau des Frontends

Wir haben uns für folgende Projektstruktur entschieden:



Wie man sehen kann werden in einem Ordner assets alle Bilder, Levels und Maps separat abgelegt, was für eine gute Übersichtlichkeit sorgt.

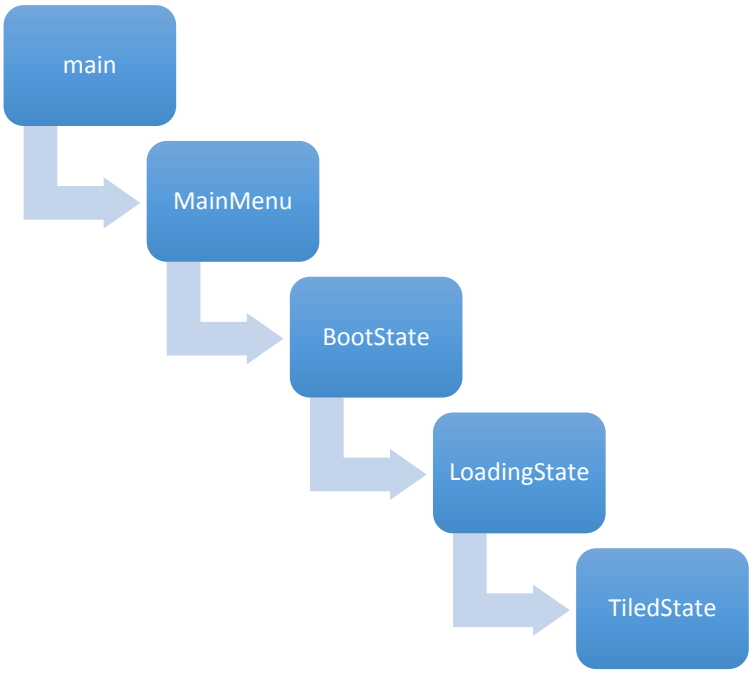
In dem Ordner images werden alle Bilder und Spritesheets abgelegt. Im levels Ordner befindet sich zu jedem Level eine JSON Datei, welche allen für das Level benötigten assets Schlüssel zuweist und Gruppen enthält. In dem maps Ordner sind die Karten für jedes Level, welche wir mithilfe des Programmes Tiled erstellt haben.
Unsere JavaScript Dateien werden im js Ordner in prefabs und States gegliedert, lediglich die main, sowie de phaser Datei bleiben im übergeordnetem Ordner. In der data.json werden die Fragen gespeichert.

Zunächst werden in der HTML Datei alle JavaScript Dateien wie folgend hinzugefügt.

```
<script src="js/states/xxx.js"></script>
<script src="js/prefabs/xxx.js"></script>
```

Des Weiteren wird die Klasse main gestartet.

```
<script type="text/javascript" src="js/main.js"></script>
```



In der Main wird zunächst das Game erstellt und die Fenstergröße auf 100% gesetzt.

```
var game = new Phaser.Game("100%", "100%", Phaser.CANVAS);
```

Wie folgt werden alle States hinzugefügt:

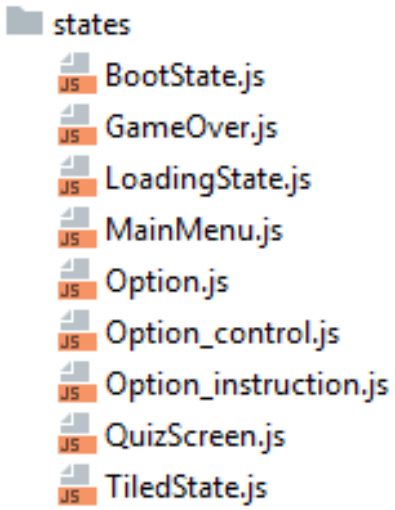
```
game.state.add("MainMenu", new BürgermeisterRun.MainMenu());
```

Das nächste Level wird auf 1 gesetzt.

```
var next_level=1;
```

Dann das Main Menu gestartet und das nächste Level wird übergeben.

```
game.state.start("MainMenu", true, false, next_level);
```



Im Main Menu wird zunächst in der init Funktion das weitergegebene nächste Level in eine Variable abgespeichert und dann werden in der preload Funktion die Images für den Hintergrund und die Buttons vorgeladen. Woraufhin in der create Funktion der Hintergrund gesetzt wird sowie die Buttons für alle spielbaren Levels erstellt werden. Die Buttons bekommen alle einen Listener der die Funktion up aufruft, welche den BootState mit dem passenden Level zum Button wie folgt aufruft.

```
game.state.start("BootState", true, false, "assets/levels/level1.json");
```

Im Boot State wird wieder zuerst die init Funktion aufgerufen, welche die weitergegebenen Level Daten in eine Variable abgespeichert. Die preload Funktion lädt dann den Text, welcher dann in der create Funktion genutzt wird um die Level Daten aus der JSON Datei zu laden. Mit diesen Daten wird dann der Loading State aufgerufen.

```
this.game.state.start("LoadingState", true, false, level_data);
```

Die Aufgabe des Loading States ist das Lader der Assets. Nachdem die weitergegebenen Level Daten in der int Funktion in eine separate Variable gespeichert wurden, werden in der preload Funktion alle assets geladen.

```
BürgermeisterRun.LoadingState.prototype.preload = function () {
    "use strict";
    var assets, asset_loader, asset_key, asset;
    assets = this.level_data.assets;
    for (asset_key in assets) { // load assets according to asset key
        if (assets.hasOwnProperty(asset_key)) {
            asset = assets[asset_key];
            switch (asset.type) {
            case "image":
                this.load.image(asset_key, asset.source);
                break;
            case "spritesheet":
                this.load.spritesheet(asset_key, asset.source, asset.frame_width, asset.frame_height,
asset.frames, asset.margin, asset.spacing);
                break;
            case "tilemap":
                this.load.tilemap(asset_key, asset.source, null, Phaser.Tilemap.TILED_JSON);
                break;
            }
        }
    }
};
```

Daraufhin werden die Level Daten, in der create Funktion, dem GameState (TiledState.js) weitergegeben.

```
this.game.state.start("GameState", true, false, this.level_data);
```

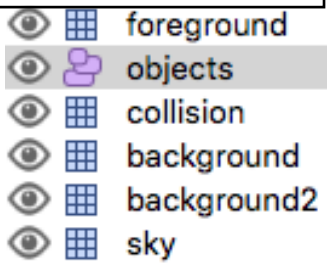
Im Konstruktor von der TiledState Klasse werden zunächst alle prefab Klassen in prefab_classes gespeichert, wobei es wichtig ist die Schlüssel genau wie die typen der zu ladenden Objekte zu benennen.

```
this.prefab_classes = {
    "player": BürgermeisterRun.Player.prototype.constructor,
    "ground_enemy": BürgermeisterRun.GroundEnemy.prototype.constructor,
    "flying_enemy": BürgermeisterRun.FlyingEnemy.prototype.constructor,
    "running_enemy": BürgermeisterRun.RunningEnemy.prototype.constructor,
    "goal": BürgermeisterRun.Goal.prototype.constructor,
    "checkpoint": BürgermeisterRun.Checkpoint.prototype.constructor,
    "coin": BürgermeisterRun.Coin.prototype.constructor,
    "score": BürgermeisterRun.Score.prototype.constructor,
    "life": BürgermeisterRun.Life.prototype.constructor,
    "infobox": BürgermeisterRun.Infobox.prototype.constructor,
    "jag": BürgermeisterRun.Jag.prototype.constructor,
    "fire": BürgermeisterRun.Fire.prototype.constructor,
    "green_enemy": BürgermeisterRun.WalkingEnemy.prototype.constructor
};
```

In der init Funktion werden hier nicht nur die Level Daten in eine neue Variable gespeichert, sondern auch die Karte skaliert, die Spiel Physik aktiviert, woraufhin man die Gravitation setzten kann. Die Karte wird dann mithilfe der zuvor gemerkten Level Daten geladen und die Teilsets werden der map hinzugefügt.

```
this.map = this.game.add.tilemap(level_data.map.key);
tileset_index = 0;
this.map.tilesets.forEach(function (tileset) {
    this.map.addTilesetImage(tileset.name, level_data.map.tilesets[tileset_index]);
    tileset_index += 1;
}, this);
```

Daraufhin werden in der create Funktion zunächst die Ebenen (Layer) geladen. Von dem Colision Layer wird zusätzlich jede Kachel (Tile) durchgegangen und am Ende die Kollision aktiviert. Zum Schluss wird noch die Kartengröße angepasst.




```
var group_name, object_layer, collision_tiles;

this.layers = {};
this.map.layers.forEach(function (layer) {
    this.layers[layer.name] = this.map.createLayer(layer.name);
    if (layer.properties.collision) { // collision layer
        collision_tiles = [];
        layer.data.forEach(function (data_row) { // find tiles used in the layer
            data_row.forEach(function (tile) {
                // check if it's a valid tile index and isn't already in the list
                if (tile.index > 0 && collision_tiles.indexOf(tile.index) === -1) {
                    collision_tiles.push(tile.index);
                }
            }, this);
        }, this);
        this.map.setCollision(collision_tiles, true, layer.name);
    }
}, this);
```

```
// resize the world to be the size of the current layer
this.layers[this.map.layer.name].resizeWorld();
```

Danach werden die Gruppen erstellt und für jedes Objekt auf dem objectlayer die Funktion create_object aufgerufen.

In der create_object Funktion wird zunächst die Position angepasst, danach wird geschaut ob der Objekt Typ oben in den prefab_classes aufgelistet ist und gegeben falls wird die Entsprechende prefab Klasse mit den entsprechenden Eigenschaften aufgerufen.

```
BürgermeisterRun.TiledState.prototype.create_object = function (object) {
    "use strict";
    var position, prefab;
    // tiled coordinates starts in the bottom left corner
    position = {"x": object.x + (this.map.tileHeight / 2), "y": object.y - (this.map.tileHeight / 2)};
    // create object according to its type
    if (this.prefab_classes.hasOwnProperty(object.type)) {
        prefab = new this.prefab_classes[object.type](this, position, object.properties);
    }
    this.prefabs[object.name] = prefab;
};
```

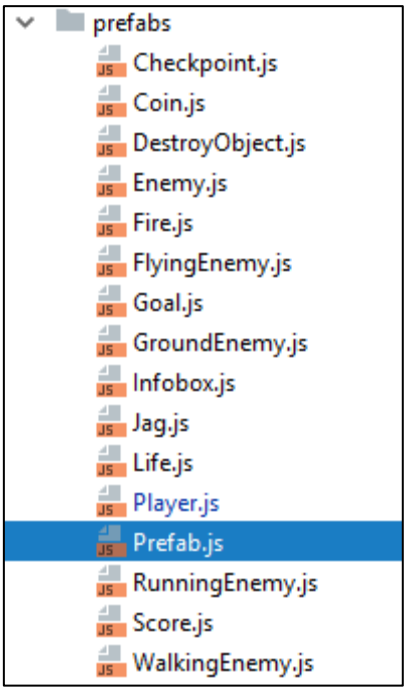
Als letztes wird in der create Funktion bestimmt, dass die Kamera dem Player folgt.

```
this.game.camera.follow(this.prefabs.player);
```

Des Weiteren stellt die TiledState Klasse eine Funktion bereit, welche im Falle des Todes des Spielers schaut, ob er noch leben hat und ihn dann gegebenen falls am Anfang oder am Checkpoint wieder spawnt. Falls der Spieler keine Leben mehr hat wird der GameOver State mit dem aktuellen Level gestartet.

Der GameOver State arbeitet genau wie das Main Menu. Es wird zuerst in der init Funktion das Level in eine Variable gespeichert und dann die Bilder in der preload Funktion vorgeladen. In der create Funktion wird dann wieder der Hintergrund gesetzt und der Button erstellt, welcher das Menu mit dem aktuellen Level aufruft.

4.1.3 Klasse Prefab




```
BürgermeisterRun.Prefab = function (game_state,
position, properties) {
    "use strict";
    Phaser.Sprite.call(this, game_state.game,
position.x, position.y, properties.texture);

    this.game_state = game_state;
this.game_state.groups[properties.group].add(this);
};
```

Um nicht überall einzeln die Texturen etc. einzufügen, wird die Prefab Klasse genutzt, um diese in jeder Klasse zu integrieren. Dadurch werden die Positionen (X und Y-Koordinaten) und die Texturen in allen synchronisiert. Dies wird automatisch durch die Phaser-interne Methode bewerkstelligt.
Des Weiteren werden die einzelnen Objekte einer Gruppe hinzugefügt.

4.1.3.1 Checkpoint

Eigenschaft	Wert
Objekt	
ID	89
Name	checkpoint
Typ	checkpoint
Sichtbar	☑
X	4.692,62
Y	943,02
Breite	35,00
Höhe	37,00
Drehung	-359,81
Spiegelung	
Horizontal	☐ Falsch
Vertikal	☐ Falsch
Benutzerdefinierte Eigenschaften	
group	checkpoints
texture	checkpoint_image

Jedes Objekt hat einen Typ, Koordinaten, sowie eine Gruppe und Textur, wie man auf dem Bild des bereits eben genannten Checkpoints sehen kann. Der Typ ist dafür da, dass Objekt einer prefab Klasse zuweisen zu können, daher ist es so wichtig im Konstruktor der TiledState Klasse beim Zuweisen der prefab_classes die richtigen Schlüssel zu nehmen. Die Koordinaten sind wichtig für die spätere Position im Spiel. Die von uns in den Benutzerdefinierten Eigenschaften erstellte Gruppe dient dazu die Objekte den richtigen Gruppen zuzuweisen. Der String texture muss mit dem in der levelX.json übereinstimmen um das zugewiesene Image oder Sprite laden zu können.

Dem Checkpoint wird der game_state die angepasste Position und die Objekteigenschaften übergeben, mit welchen er die Prefab Funktion aufruft, welche sowohl die Position, als auch Textur setzt. Da der Checkpoint am Anfang noch nicht erreicht ist wird checkpoint_reached = false gesetzt. Dann wird die Spiel Physik aktiviert, um den Checkpoint kollidieren lassen zu können. Danach wird der Bildanker, welcher normalerer weise immer oben links vom Bild sitzt angepasst.

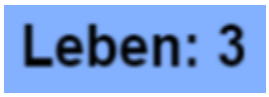
```
var BürgermeisterRun = BürgermeisterRun || {};

BürgermeisterRun.Checkpoint = function (game_state, position, properties) {
  "use strict";
  BürgermeisterRun.Prefab.call(this, game_state, position, properties);

  this.checkpoint_reached = false;

  this.game_state.game.physics.arcade.enable(this);
```

```
    this.anchor.setTo(0.5);
};

BürgermeisterRun.Checkpoint.prototype = Object.create(BürgermeisterRun.Prefab.prototype);
BürgermeisterRun.Checkpoint.prototype.constructor = BürgermeisterRun.Checkpoint;
```

Die update Funktion Kollidiert den Checkpoint mit dem Kolisions Layer und schaut ob er sich mit dem Player überlappt. Wenn dies der Fall ist wird die reach_checkpoint Funktion aufgerufen, welche den boolean checkpoint_reached auf true setzt.

```
BürgermeisterRun.Checkpoint.prototype.update = function () {
    "use strict";
    this.game_state.game.physics.arcade.collide(this, this.game_state.layers.collision);
    this.game_state.game.physics.arcade.overlap(this, this.game_state.groups.players,
this.reach_checkpoint, null, this);
};

BürgermeisterRun.Checkpoint.prototype.reach_checkpoint = function () {
    "use strict";
    // checkpoint was reached
    this.checkpoint_reached = true;
};
```

4.1.3.2 Life



Die Klasse Life zeigt die Anzahl der Leben oben links an.

Mithilfe der Update-Funktion wird die Lebensanzeige aktuell gehalten.

```
BürgermeisterRun.Life.prototype.update = function () {
    "use strict";
    // update text to player current score
    this.text = "Leben: " + this.game_state.prefabs.player.life;
}
```

4.1.3.3 Score

Die Klasse Score zeigt die Anzahl der Punkte die man im Spiel erreicht hat an und die Papierbälle, die man noch zur Verfügung hat.

```
BürgermeisterRun.Score.prototype.update = function () {
    "use strict";
    // update text to player current score
    this.text = "Score: " + this.game_state.prefabs.player.score + "
Papierbälle: "+ _iPaperball;
};
```

4.1.3.4 Coin

Der „Coin“ ist ein Blatt Papier mit dem man seinen Punktestand um eine gewisse Anzahl an Punkten erhöhen kann. Ab einer bestimmten Anzahl an Punkten, erhöht sich die Anzahl der Leben.
Des Weiteren bekommt man für jedes Blatt auch einen Papierball bzw. ein Geschoss zum Werfen.

Dem Coin wird der jeweilige Punktewert zugewiesen, den man in der Tiledmap eingestellt hat.

```
this.score = +properties.score;
```

Der Coin soll weder bewegt werden können, noch soll er aufgrund der Schwerkraft auf den Boden fallen.

```
this.body.immovable = true;
this.body.allowGravity = false;
```

Die Animation des Coins wird hinzugefuegt.

```
this.animations.add("turning", [0, 1, 2], 3, true);
this.animations.play("turning");
```

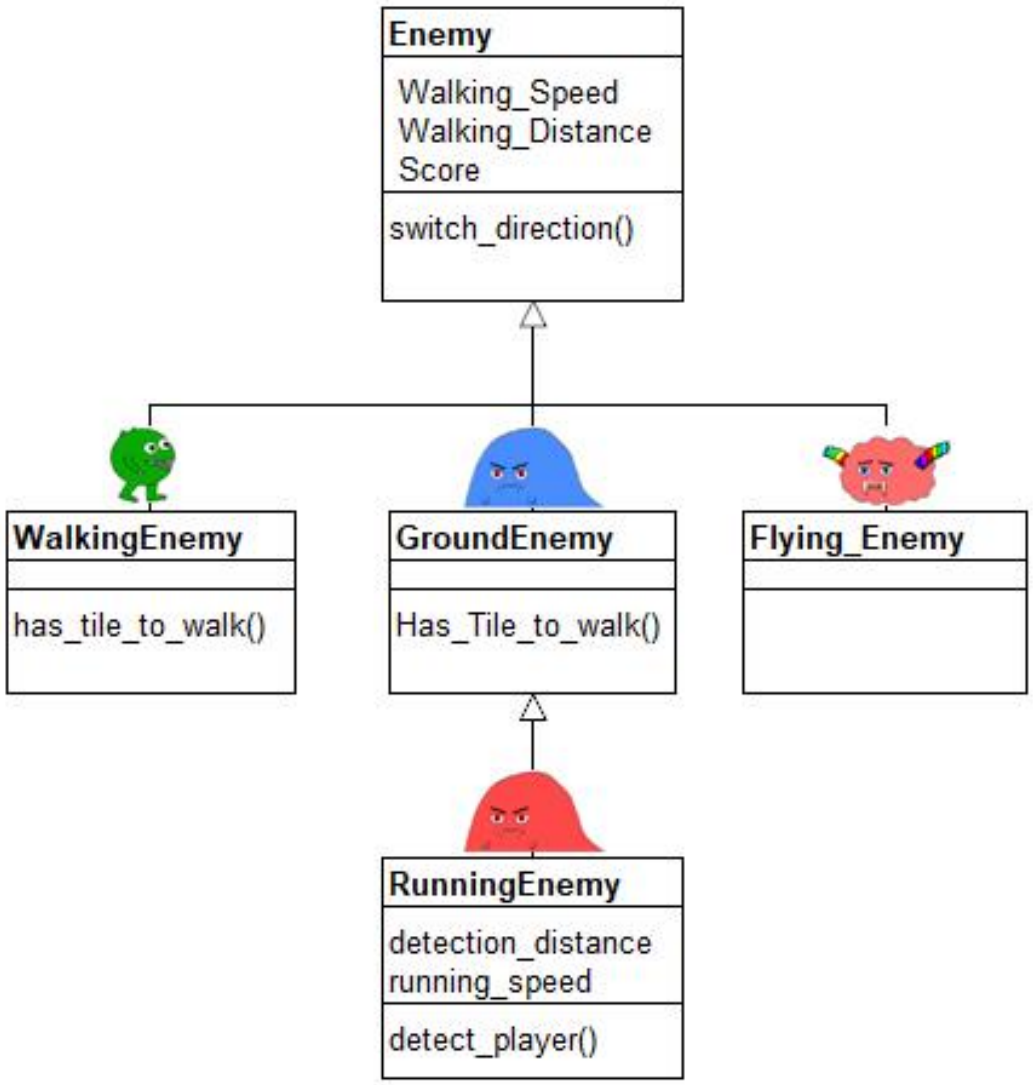
Beim Einsammeln wird ein kurzer Sound abgespielt.

```
coinCollect = this.game.add.sound('coinCollect',0.4,false);
```

Wenn man der Player den Coin berührt, wird der Coin zerstört, der Punktestand (Score) und die Anzahl an Papierbällen erhöht und der Sound für das Einsammeln abgespielt.

```
BürgermeisterRun.Coin.prototype.collect_coin = function (coin, player) {
    "use strict";
    // kill coin and increase score
    this.kill();
    player.score += this.score;
    _iPaperball++;
    coinCollect.play();

};+
```

4.2 Enemy (Gegner)



Klasse Enemy

Jeder Enemy besitzt die folgenden Eigenschaften ():

```
this.walking_speed = +properties.walking_speed;
this.walking_distance = +properties.walking_distance;
this.score = +properties.score;
```

Walking_Speed ist die Geschwindigkeit, mit der sich der Enemy bewegen kann.
Walking_Distance ist die Distanz, welche der Enemy kontrolliert bzw. in der er sich fortbewegt.
Score ist der Punktewert, den man für das besiegen des Enemy erhält.
Um die Distanz die der Enemy gegangen ist zu speichern, nutzen wir die Variable x

```
// saving previous x to keep track of walked distance
this.previous_x = this.x;
```

Die Arcade-Physik wird dem Enemy zugeteilt.

```
this.game_state.game.physics.arcade.enable(this);
```

Die X-Koordinate des Gegners wird bestimmt, indem der Enemy, je nach direction (Richtung in die er schaut), mit seiner Geschwindigkeit fortbewegt.

```
this.body.velocity.x = properties.direction * this.walking_speed;
```

Das Object wird erstellt

```
BürgermeisterRun.Enemy.prototype =
Object.create(BürgermeisterRun.Prefab.prototype);
BürgermeisterRun.Enemy.prototype.constructor =
BürgermeisterRun.Enemy;
```

Die Update-Funktion überprüft ständig, ob der Enemy mit der Collisions-Layer (In TilesMap Collision genannt) kollidiert (Die collide Funktion lässt den Enemy damit kollidieren, damit der Gegner drauf laufen kann)

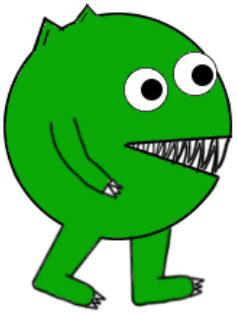
Die Richtung in die sich der Enemy bewegt wird ständig berechnet mithilfe der Variable X bzw. es wird der Wert den der Enemy gelaufen ist überprüft und falls man die genannte walking_distance gelaufen ist, wird die Richtung gewechselt.

```
BürgermeisterRun.Enemy.prototype.update = function () {
    "use strict";
    this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);

    // change the direction if walked the maximum distance
    if (Math.abs(this.x - this.previous_x) >= this.walking_distance) {
        this.switch_direction();
    }
};
```

Wenn diese Funktion aufgerufen wird, wechselt der Enemy die Richtung

```
BürgermeisterRun.Enemy.prototype.switch_direction = function () {
    "use strict";
    this.body.velocity.x *= -1;
    this.previous_x = this.x;
    this.scale.setTo(-this.scale.x, 1);
};
```

Klasse WalkingEnemy



Eigenschaften etc. vom Enemy aufrufen mit call

```
BürgermeisterRun.WalkingEnemy = function (game_state,
position, properties) {
    "use strict";
    BürgermeisterRun.Enemy.call(this, game_state, position,
properties);
```

Die Animationen und die Frames werden dem Gegner entsprechend angepasst.

```
this.animations.add("moving", [0, 1, 2], 5, true);
this.animations.play("moving");
```

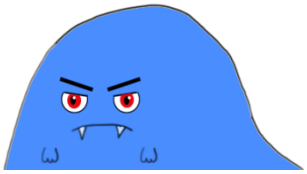
Update-Funktion: Wie bei GroundEnemy

```
BürgermeisterRun.WalkingEnemy.prototype.update = function () {
    "use strict";
    BürgermeisterRun.Enemy.prototype.update.call(this);

    if (this.body.blocked.down && !this.has_tile_to_walk()) {
        this.switch_direction();
    }
};
```

has_tiel_to_walk-Funktion: Wie bei GroundEnemy

```
BürgermeisterRun.WalkingEnemy.prototype.has_tile_to_walk = function () {
    "use strict";
    var direction, position_to_check, map, next_tile;
    direction = (this.body.velocity.x < 0) ? -1 : 1;
    // check if the next position has a tile
    position_to_check = new Phaser.Point(this.x + (direction *
this.game_state.map.tileWidth), this.bottom + 1);
    map = this.game_state.map;
    // getTileWorldXY returns the tile in a given position
    next_tile = map.getTileWorldXY(position_to_check.x,
position_to_check.y, map.tileWidth, map.tileHeight, "collision");
    return next_tile !== null;

};
```

Klasse GroundEnemy



Die Klasse ruft mithilfe von „Enemy.call“ die game_state, posititon und Eigenschaften des Enemy auf.
Da der Klasse beim Laden ein Spritesheet zugewiesen wird, haben wir mithilfe der „animations“ die jeweiligen Spriteposition angegeben und die Frames, in denen es abgespielt werden soll

```
BürgermeisterRun.GroundEnemy = function (game_state, position, properties)
{
    "use strict";
    BürgermeisterRun.Enemy.call(this, game_state, position, properties);

    this.animations.add("moving", [0, 1], 5, true);
    this.animations.play("moving");
```

Ein Objekt GroundEnemy wird erstellt mithilfe des Enemy.prototype um weitere Funktionen hinzuzufügen.

```
BürgermeisterRun.GroundEnemy.prototype =
Object.create(BürgermeisterRun.Enemy.prototype);
BürgermeisterRun.GroundEnemy.prototype.constructor =
BürgermeisterRun.GroundEnemy;
```

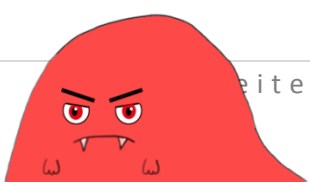
Die Updatefunktion überprüft ob der GroundEnemy auf etwas steht (body.blocked.down) und sich vor ihm kein Tile befindet. Falls er vor sich keinen Begehbaren boden findet, dreht er um, indem switch_direction aufgerufen wird.

```
BürgermeisterRun.GroundEnemy.prototype.update = function () {
    "use strict";
    BürgermeisterRun.Enemy.prototype.update.call(this);

    if (this.body.blocked.down && !this.has_tile_to_walk()) {
        this.switch_direction();
    }
};
```

Diese Funktion überprüft ob der Enemy vor sich einen begehbaren Bereich hat

```
BürgermeisterRun.GroundEnemy.prototype.has_tile_to_walk = function () {
    "use strict";
    var direction, position_to_check, map, next_tile;
    direction = (this.body.velocity.x < 0) ? -1 : 1;
    // check if the next position has a tile
    position_to_check = new Phaser.Point(this.x + (direction *
this.game_state.map.tileWidth), this.bottom + 1);
    map = this.game_state.map;
    // getTileWorldXY returns the tile in a given position
    next_tile = map.getTileWorldXY(position_to_check.x,
position_to_check.y, map.tileWidth, map.tileHeight, "collision");
    return next_tile !== null;
};
```

Klasse RunningEnemy



Da der RunningEnemy den Spieler beachten soll und ihm nachlaufen soll, werden ihm die Eigenschaften detection_distance (Der Bereich in dem der Spieler entdeckt werden soll) und running_speed (die Geschwindigkeit des Gegner, wenn der Spieler entdeckt wird, da er ihm hinterherlaufen soll)

```
BürgermeisterRun.RunningEnemy = function (game_state, position, properties)
{
    "use strict";
    BürgermeisterRun.GroundEnemy.call(this, game_state, position,
properties);

    this.detection_distance = +properties.detection_distance;
    this.running_speed = +properties.running_speed;
};
```

In der Update-Funktion ständig überprüft, ob der Gegner entdeckt worden ist. Falls das der Fall ist, läuft der Gegner in Richtung des Spielers und das mit erhöhter Geschwindigkeit (running_speed) Falls der Spieler nicht in der Nähe ist, verhält sich der Enemy wie ein ganz normaler und läuft seine Distanz.

```
    if (this.detect_player()) {
        // player is inside detection range, run towards it
        direction = (this.game_state.prefabs.player.x < this.x) ? -1 : 1;
        this.body.velocity.x = direction * this.running_speed;
        this.scale.setTo(-direction, 1);
        this.previous_x = this.x;
    } else {
        // player is outside detection range, act like a regular enemy
        direction = (this.body.velocity.x < 0) ? -1 : 1;
        this.body.velocity.x = direction * this.walking_speed;
        this.scale.setTo(-direction, 1);
        BürgermeisterRun.GroundEnemy.prototype.update.call(this);
    }
};
```

Die Funktion überprüft die Koordinaten des Spielers mit seinen eigenen Koordinaten. Falls der Spieler die selben Y-Koordinaten hat und sich in der „detection_distanz“ sprich dem Bereich, den wir vorher angegeben haben, befindet, wird True zurückgegeben.

```
BürgermeisterRun.RunningEnemy.prototype.detect_player = function () {
    "use strict";
    var distance_to_player;
    distance_to_player = Math.abs(this.x -
this.game_state.prefabs.player.x);
    // the player must be in the same ground y position, and inside the
detection range
    return (this.bottom === this.game_state.prefabs.player.bottom) &&
(distance_to_player <= this.detection_distance);
};
```

Klasse FlyingEnemy

Eigenschaften etc. des Enemy aufgerufen mit call

```
BürgermeisterRun.Enemy.call(this, game_state, position, properties);
```

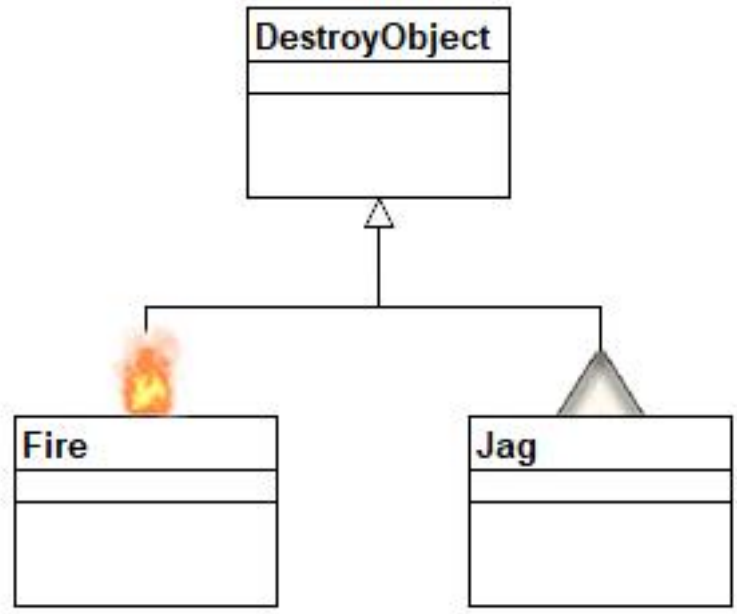
Da der fliegende Gegner nicht von der Schwerkraft beeinflusst wird, schalten wir die Gravitation für ihn aus

```
this.body.allowGravity = false;
```

Die Animationen werden an den Sprite angepasst

```
this.animations.add("flying", [0, 1], 5, true);
this.animations.play("flying");
```

4.3 DestroyObject



Klasse DestroyObject

Die arcade Physik wird den DestroyObject hinzugefügt.
Da sich die Objekte nicht bewegen sollen, wird der Wert immovable (nicht bewegbar) auf true gesetzt. Die Gravitation für die Objekte wird ebenfalls ausgeschaltet, damit man diese überall hinsetzen kann (z.B. Feuer im Himmel oder für zukünftige Erweiterungen)

```
this.game_state.game.physics.arcade.enable(this);
this.body.immovable = true;
this.body.allowGravity = false;
```

Die Kollision mit dem Kollisions-Layer der Map wird hergestellt

```
this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
```

Klasse Jag

Die Eigenschaften von DestroyObject werden aufgerufen

```
BürgermeisterRun.DestroyObject.call(this, game_state,
position, properties);
```

In der Update-Funktion wird die Kollision mit dem Kollisions-Layer der Map überprüft

```
this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
```

Klasse Fire

Die Eigenschaften von DestroyObject werden aufgerufen

```
BürgermeisterRun.DestroyObject.call(this, game_state, position,
properties);
```

Da das Feuer eine Animation bzw. einen Spritesheet besitzt, werden diese Werte angepasst.

```
this.animations.add("fire", [0,
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,
29,30,31], 5, true);
//
this.animations.play("fire");
```

Die Kollision wird ebenfalls in der Update-Funktion überprüft

```
this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
```

4.4 Die Klasse Player

Wie der Name schon vermuten lässt, sind in der Player Klasse alle Hauptfunktionen des Players implementiert. Dazu gehören das Verhalten des Playerobjektes auf die Spielwelt, sowie dessen Fähigkeiten und Steuerung.
Zunächst ruft die Klasse mithilfe von seiner Superklasse bereitgestellten Funktion, „Prefab.call“ die game_state, posititon und Eigenschaften des Players auf.

```
BürgermeisterRun.Player = function (game_state, position, properties) {
    "use strict";
    BürgermeisterRun.Prefab.call(this, game_state, position, properties);
```

Da die Umsetzungen der Attribute, Animationen und Skalierungen dem unter 4.1.2 vorgestellten Enemy Klassen sehr ähneln, wird hier nicht näher darauf eingegangen.

Neu in der Funktion ist, dass hier die Steuerelemente des Players implementiert werden.
Einmal für die Benutzung der Tastatur an einem Desktop PC:

```
this.cursors = this.game_state.game.input.keyboard.createCursorKeys();
this.speedRunSKey = this.game_state.game.input.keyboard.addKey(Phaser.Keyboard.S);
this.shootKey = this.game_state.game.input.keyboard.addKey(Phaser.Keyboard.A);
spacebarKey = game.input.keyboard.addKey(Phaser.Keyboard.SPACEBAR);
this.muteSound = game.input.keyboard.addKey(Phaser.Keyboard.M);
```

und einmal wird für die Benutzung mobiler Geräte virtuelle Buttons erzeugt.

```
   if(! Phaser.Device.desktop) {
        game.scale.setGameSize(1400, 850);

        shootButton = game.add.button(10, 705, 'shootButton', this.shoot, this);
        shootButton.fixedToCamera = true;

        jumpButton = game.add.button(10, 580, 'jumpButton', this.jump, this);
        jumpButton.fixedToCamera = true;

        jumpButtonDpad = game.add.button(1111, 465, 'jumpButtonDpad', this.jump,
this);
        jumpButtonDpad.fixedToCamera = true;

        leftButton = game.add.button(978, 600, 'leftButton', null, this);
        leftButton.fixedToCamera = true;
        leftButton.events.onInputDown.add(function () {
            left = true;});
        leftButton.events.onInputUp.add(function () {
            left = false;});

        rightButton = game.add.button(1200, 598, 'rightButton', null, this);
        rightButton.fixedToCamera = true;
        rightButton.events.onInputDown.add(function () {
            right = true;});
        rightButton.events.onInputUp.add(function () {
            right = false;});

        downButton = game.add.button(1111, 690, 'downButton', null, this);
        downButton.fixedToCamera = true;
        downButton.events.onInputDown.add(function () {
            duck = true;});
        downButton.events.onInputUp.add(function () {
            duck = false;});


        speedButton = game.add.button(140, 705, 'speedButton', null, this);
        speedButton.fixedToCamera = true;
        speedButton.events.onInputDown.add(function () {
            speedrun = true;});
        speedButton.events.onInputUp.add(function () {
            speedrun = false;});

   }
```

Als Parameter bekommen die virtuellen Buttons ihre x und y Koordinaten und rufen gegebenenfalls eine im Folgenden beschrieben Funktionen auf. Die Buttons die keinen Funktionsaufruf als Parameter haben, bekommen gesondert Listener, die nach Usecase eine boolean Variable true und danach false setzen. Die genauere Funktionsweise wird in der folgenden Update Funktion ersichtlich.

Abschließend wird noch ein „shootobjekt“ generiert und gestaltet. Dieser findet in der später folgenden shoot Funktion Verwendung.

```
//shootobject config
shootobject = game.add.group();
shootobject.enableBody = true;
shootobject.physicsBodyType= Phaser.Physics.ARCADE;
shootobject.createMultiple(100, 'paperball');
shootobject.setAll('anchor.x' , 0.5);
shootobject.setAll('anchor.y' , 0.5);
shootobject.setAll('scale.x' , 1);
shootobject.setAll('scale.y' , 1);
shootobject.checkWorldBounds = true;
shootobject.outOfBoundsKill = true; };
```

Nachdem das Playerobjekt erzeugt wird,

```
/**
 * Player object extend from Prefab class
 * @type {BürgermeisterRun.Prefab}
 */
BürgermeisterRun.Player.prototype =
Object.create(BürgermeisterRun.Prefab.prototype);
BürgermeisterRun.Player.prototype.constructor = BürgermeisterRun.Player;
```

überprüft die update Funktion ständig diverse Spielbedingungen des Players.

```
BürgermeisterRun.Player.prototype.update = function () {
    "use strict";
    this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
    this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.enemies, this.hit_enemy, null, this);
    this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.destroyObject, this.hit_destroyObject, null, this);
    this.game_state.game.physics.arcade.overlap(shootobject,
this.game_state.groups.enemies, this.shoot_enemy, null, this);
this.game_state.game.physics.arcade.collide(shootobject,
this.game_state.layers.collision);
```

Zunächst werden alle Objekte mit dem der Player direkt oder indirekt, durch Treffen mit seinem shootobject, kollidiert (Player → collision Layer der Tilesmap, enemies, destroyobject; shootobject → enemies). Bei Kollisionen mit den Enemies und Destroyobjects werden anschließend adäquate Funktionen aufgerufen.

Als nächstes überprüft die update Funktion ob ein Steuerelement (Tastatur oder virtueller Button) betätigt wurde und löst dann die Steuerlogik aus.

```
if(this.muteSound.isDown){
    if(!this.game.sound.mute){
        this.game.sound.mute = true;
    }
   else{
        this.game.sound.mute=false;
    }
}

if (this.cursors.down.isDown||duck && this.body.velocity.x === 0) {
    // cower
    if(lookRight===true){
        this.frame = 10;

    }
    else{
        this.frame = 0;

        }
    if(!fart.isPlaying){
        fart.play();
    }

} else if (this.cursors.right.isDown||right && this.body.velocity.x >= 0) {
    // move right
    if(this.speedRunSKey.isDown||speedrun)this.body.velocity.x = this.walking_speed
*1.5;
    else this.body.velocity.x = this.walking_speed;
    this.animations.play("right");
```

```
    this.scale.setTo(1, 1);
    lookRight = true;

} else if (this.cursors.left.isDown ||left && this.body.velocity.x <= 0) {
    // move left
    if(this.speedRunSKey.isDown||speedrun)this.body.velocity.x =-
this.walking_speed *1.5;
    else this.body.velocity.x = -this.walking_speed;
    this.animations.play("left");
    this.scale.setTo(1, 1);
    lookRight = false;

} else {
    // stop
    this.body.velocity.x = 0;
    this.animations.stop();
    this.scale.setTo(1, 1);
    if(lookRight===true){
        this.frame = 6;
    }
    else{
        this.frame = 1;
    }
}

this.cursors.up.onDown.add(this.jump,this);
spacebarKey.onDown.add(this.jump,this);
//shoot
if (this.shootKey.isDown) {
    this.shoot();

}

// dies if touches the end of the screen
if (this.bottom >= this.game_state.game.world.height) {
    this.game_state.restart_level();
}
};
```

Die Klasse Player endet mit verschiedenen Funktionen die überwiegend in der Update Funktionen aufgerufen werden.

Die Funktionen hit_enemy, shoot_enemy und hit_detstroyobject finden ihren usecase in den collide Funktionen. Hierbei geht es im Allgemeinen um das Reseten oder rausnehmen der kollidierten Elemente.

```
BürgermeisterRun.Player.prototype.hit_enemy = function (player, enemy) {
    "use strict";
    // if the player is above the enemy, the enemy is killed, otherwise the player dies
    if (enemy.body.touching.up) {
        this.score += enemy.score;
        enemy.kill();
        player.y -= this.bouncing;

    } else {
        this.game_state.restart_level();
    }
};

BürgermeisterRun.Player.prototype.hit_destroyObject = function (player, destroyObject) {
    "use strict";
    // kill coin and increase score
    this.game_state.restart_level();
};
```

```
BürgermeisterRun.Player.prototype.shoot_enemy = function (player, enemy) {
    "use strict";

        this.score += enemy.score;
        enemy.kill();
        paperball.kill();
        shootTime = game.time.now;
};
```

Die Funktionen check_touched_ground, limit_fall_speed und check_score werden in der update Funktion aufgerufen. Während die ersten beiden Funktionen dem Verhindern von Bugs dienen, überpruft die check_score Funktion ständig ob der Score sich um 1500 Punkte erhöht hat. Jedesmal wenn dies geschieht, bekommt der Player als Belohnung ein Leben dazu.

```
BürgermeisterRun.Player.prototype.check_touched_ground = function (player) {
    "use strict";
    if (this.body.blocked.down){
        touchedGround = true;
    }
};



//Check if player is falling to fast, if yes the falling velocity will be limited
BürgermeisterRun.Player.prototype.limit_fall_speed = function (player) {
    "use strict";
    if (this.body.velocity.y > MAX_FALL_SPEED){
        this.body.velocity.y = MAX_FALL_SPEED;
    }
};


BürgermeisterRun.Player.prototype.check_score = function (player) {
    if((this.score / (1500 * scoreMultiplier)) > 1){
        scoreMultiplier++;
        this.life++;
    }
};
```

Die jump Funktion stellt einen normalen Sprung und einen weiteren in der Luft zur Verfügung.

```
BürgermeisterRun.Player.prototype.jump = function (player) {
    "use strict";
    if (touchedGround) {
        this.body.velocity.y = -this.jumping_speed;
        doubleJump = true;
        touchedGround = false;
    }else {
        if (doubleJump){
            this.body.velocity.y = -this.jumping_speed;
            doubleJump = false;
        }
    }};
```

Die Klasse endet mit der shoot Funktion. Die Funktion ermöglicht es dem Player shootobjects zu werfen. Dabei wird der Schussbestand um eins reduziert.

```
BürgermeisterRun.Player.prototype.shoot = function () {
    "use strict"

    if (game.time.now > shootTime && this.alive && _iPaperball > 0) {
        paperball = shootobject.getFirstExists(false);

        if(paperball&& lookRight == true){
            paperball.reset(this.x,this.y);
            paperball.body.velocity.x = 1000;
            paperball.body.velocity.y = -115;
            shootTime = game.time.now + 1000;
            _iPaperball--;
        }
        if(paperball&& lookRight == false ){
            paperball.reset(this.x,this.y);
            paperball.body.velocity.x = -1000;
            paperball.body.velocity.y = -115;
            shootTime = game.time.now + 1000;
            _iPaperball--;
        }
    }
};
```

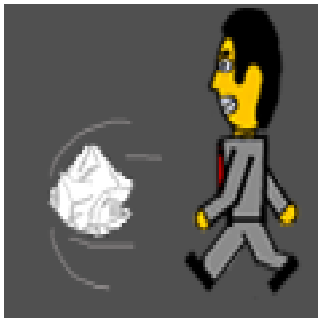
4.5 Sound

Player-Klasse

„Furz“-Geräusch

```
if (this.cursors.down.isDown||duck && this.body.velocity.x
=== 0) {
    // cower
    if(lookRight===true){
        this.frame = 10;
    }
    else{
        this.frame = 0;
        }
    if(!fart.isPlaying){
        fart.play();
    }
```

„Wurf“-Geräusch



```
//sound for Paperthrowing
if (!paperThrow.isPlaying) {
    paperThrow.play();
}
```

Hintergrundmusik

```
levelSound = game.add.sound('levelSound',0.2,true);
levelSound.play();
```

Coin Klasse

Coin-Einsammel-Sound


```
coinCollect.play();
```

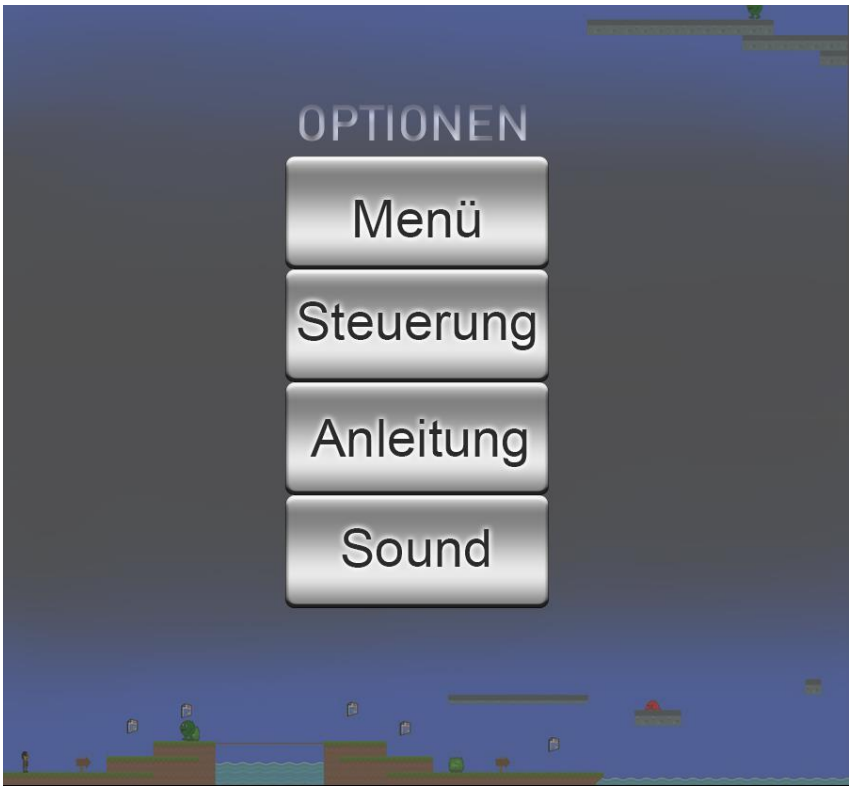
Infobox Klasse

Wenn man gegen die Box springt, ertönt ein Sound und die Info wird angezeigt.

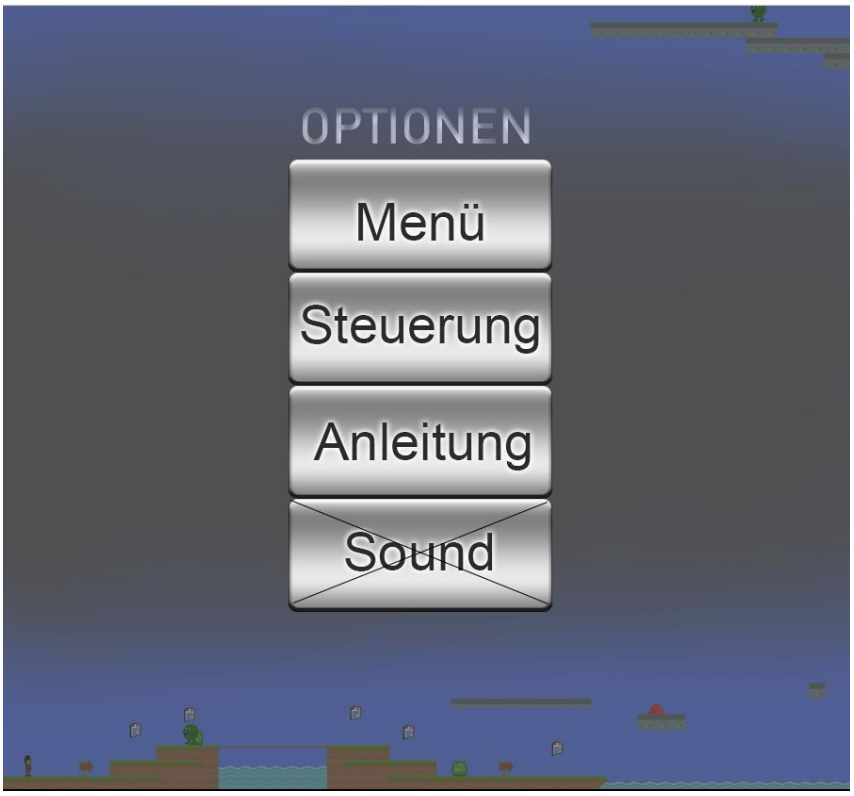
```
infoboxSound.play();
```

4.6 Menü

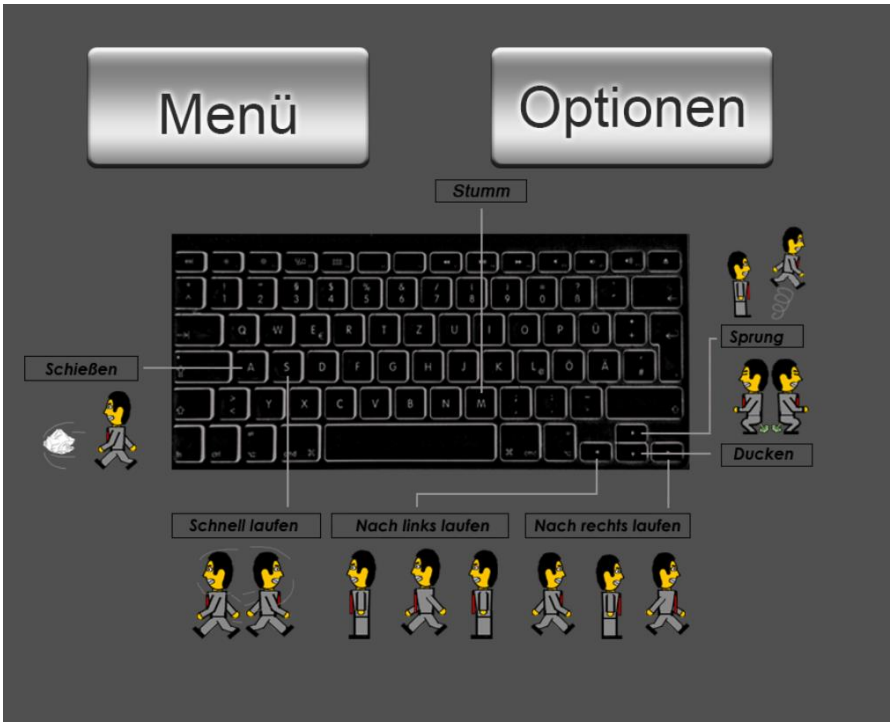
Von Optionen aus kann man in die Verschiedenen Untermenüs oder wieder zurück ins Menü springen (**State Option**)



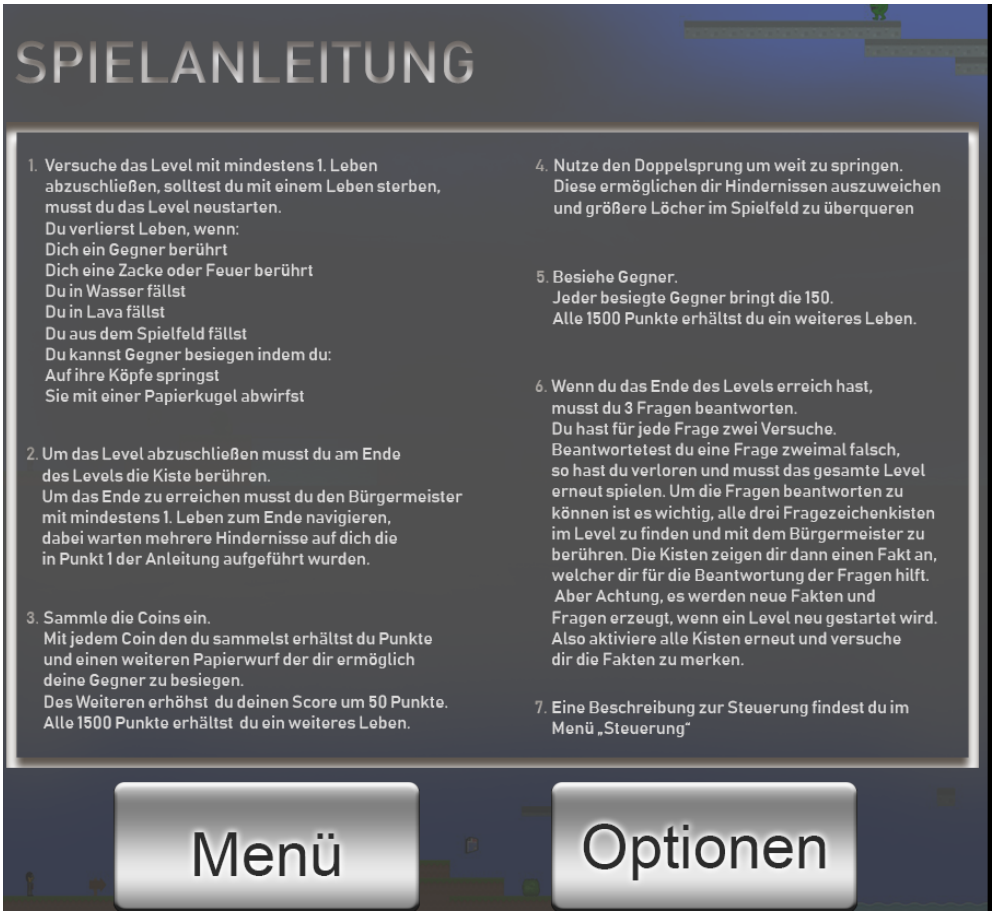
Man kann mithilfe des Sounds Button den Ton des Spiels ausschalten. Falls der Sound des Spieles stumm geschaltet ist, erhält man beim aufrufen des Options-Menü den „Sound-off“-Button



Im Steuerungsmenü kann man sich anschauen, wie man das Spiel spielt (**State Option_control)**.



Eine kurze Spielanleitung gibt es um zu verstehen welche Regeln es in dem Spiel gibt und wie man es erfolgreich abschließt (**State Option_instruction)**.



Gameover – Wenn man stirbt

Endscreen – Wenn man alle Level erfolgreich beendet hat

4.7 Backend

4.7.1 Anforderungen

Fakten

Die Fakten sollten Themenbasiert und passend zum aktuellen Level sein.
Beispielsweise wenn das Level Neroberg gespielt wird, sollten auch Fakten zum Neroberg im jeweiligen Level auftauchen.

Es sollten drei Fakten pro Level erscheinen, sobald man die Infobox anspringt.

Die Fakten die aus der Datenbank geladen werden, sollten zufällig im Level gesetzt werden und mehr als drei Fakten zum jeweiligen Level eingepflegt haben.

Es sollte durch Admins bzw. Sachbearbeiter möglich sein neue Fakten einzuspielen.

Quiz

Das Quiz sollte am Ende eines jeden Levels auftauchen.

Es sollten drei Fragen gestellt werden, passend zu den jeweiligen Fakten im Spiel.
Ein Fehlversuch pro Frage ist erlaubt.

Sollten alle drei Fragen korrekt beantwortet werden, dann soll das Main Menü erscheinen und das nächste Level sollte Freigeschaltet sein und Spielbereit zur Verfügung stehen.

Sollte zweimal eine Frage falsch beantwortet werden, dann soll ein Game Over Bildschirm erscheinen um zu kennzeichnen, dass man Verloren hat.

4.7.2 Idee und Konzept

Es stellte sich zunächst die Frage wie die Datenbank mit den Fakten aufgebaut sein soll.
Nach reifer Überlegung haben wir uns dazu entschlossen als Datenbank eine einfache JSON (JavaScript Object Notation) Datei zu verwenden, da dies vollkommen ausreichend ist für das Feature.

Diese JSON Datei soll vom Spiel selbst gelesen werden und an den richtigen Stellen die Daten einspielen.

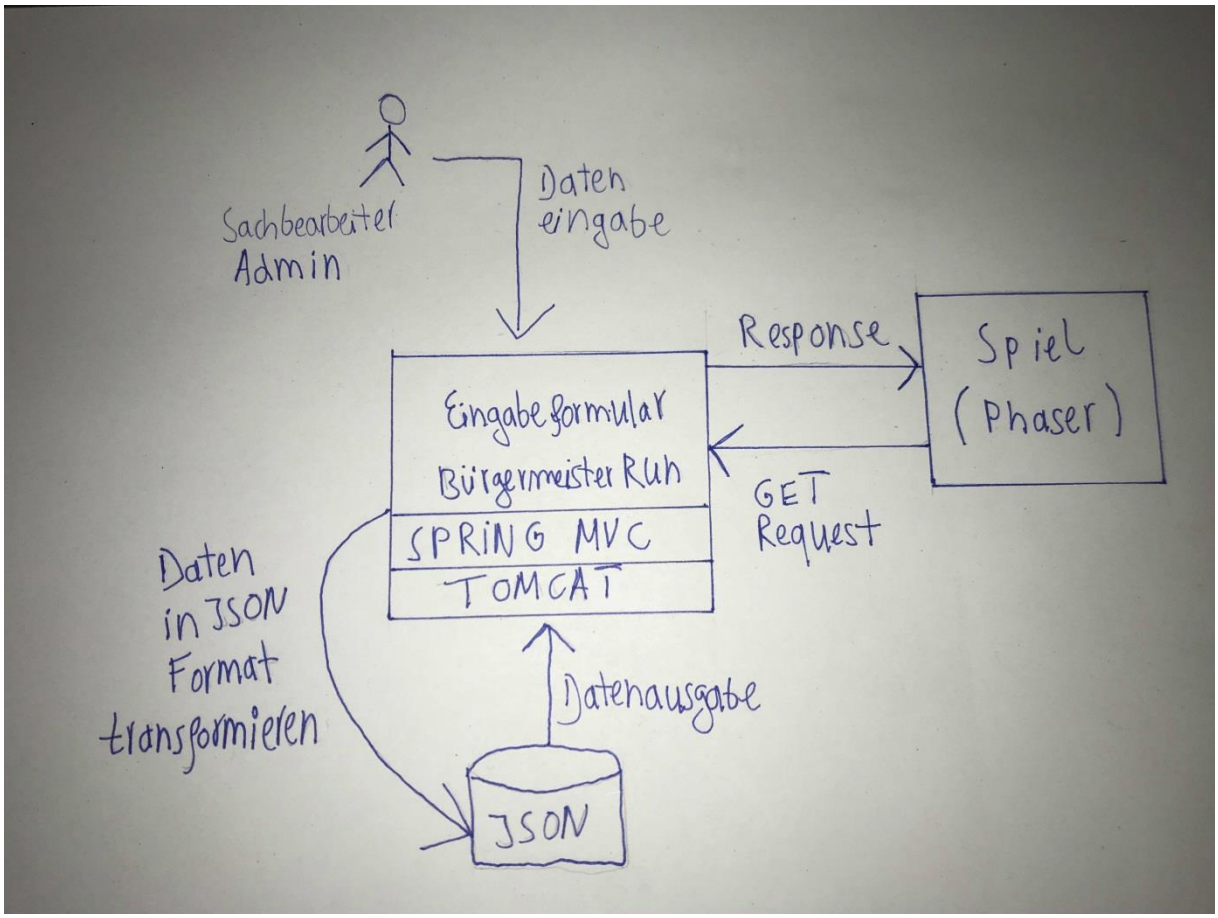
Das Eingabeformular basiert auf HTML und soll eine Möglichkeit bieten neue Fakten einzupflegen.

4.7.3 Implementierung

Eingesetzte Technologien

- JAVA als Serverseitige Programmiersprache
- Tomcat Webserver
- Spring Framework mit der MVC Komponente
- JSP die das HTML Eingabeformular liefert
- JSON als Datenbank
- JQuery für den GET Request

Architektur



A. Szenario

1. Spiel wird gestartet.
2. Es wird ein GET Request als AJAX call mit jQuery gemacht.
3. Dieser fordert eine JSON Datei an.
4. Die JSON Datei wird von Der Spring MVC App gelesen und geliefert.

B. Szenario

1. Admin/Sachbearbeiter ruft das Eingabeformular auf
2. Dieses wird von der Spring MVC App geliefert.
3. Eingabe erfolgt durch den Admin/Sachbearbeiter.
4. Die Eingabe wird zu einem JAVA Objekt serialisiert.
5. Serverseitige Validierungen der Daten werden durchgeführt.
6. Daten werden in die JSON Datei mithilfe von JAVA eingepflegt
7. Start des A. Szenario

Datenbankaufbau (JSON Datei)

```
[
    {
      "subject" : "russische-kirche",
      "fact" : "Die Tuerme der Kirche sind goldenfarbig und zwiebelfoermig",
      "question" : "Welche Farbe haben die Tuerme der Russischen Kirche?",
      "correctAnswer" : "Gold",
      "wrongAnswer1" : "Silber",
      "wrongAnswer2" : "Gelb",
      "wrongAnswer3" : "Braun"
    },
    {
      "subject" : "russische-kirche",
      "fact" : "Die Kirche steht auf dem Neroberg",
      "question" : "Wo befindet sich die Kirche?",
      "correctAnswer" : "Auf dem Neroberg",
      "wrongAnswer1" : "Im Nerotal",
      "wrongAnswer2" : "In der Innenstadt",
      "wrongAnswer3" : "Am Hauptbahnhof"
    },
    {
      "subject" : "russische-kirche",
      "fact" : "Zwiebelfoermige goldene Tuerme besitzt die Kirche",
      "question" : "Welche Form haben die Tuerme der Kirche?",
      "correctAnswer" : "Zwiebelfoermig",
      "wrongAnswer1" : "Quadratisch",
      "wrongAnswer2" : "Viereckig",
      "wrongAnswer3" : "Kartoffelfoermig"
    },
```

Eine Validierte JSON Datei mit bereits eingespielten Basisdaten mit jeweils zehn Mengen von Themen, Fakten, Fragen und vier Antwortmöglichkeiten wurden auf den Tomcat Webserver geladen.

Diese Datei wird von dem Spiel mithilfe eines AJAX Calls (GET Request) angefragt.
Geliefert wird die Datei in der gleichen Struktur und demselben Aufbau wie sie in JSON vorliegt.

```
/**
 * Loads the quiz data from JAVA app with via GET request
 * @returns string[]
 */
BürgermeisterRun.LoadingState.prototype.get_quiz_data = function (){
    var result = "";
    $.ajax({
        url:"http://localhost:63342/OneNil/app/data.json",
        async: false,
        success:function(data) {
            result = data;
        }
    });
    return result;
};
```

Der AJAX Call

Das Benutzen der Daten für weitere Verarbeitung bzw. Pflege in das Spiel ist nun dank der JSON Struktur sehr einfach wie das folgende Beispiel zeigt.

```
// Example
quiz_data = this.get_quiz_data();
console.log(quiz_data[0]['fact']);
```

Serverseitige Implementierung

Die Antwort auf GET Requests wird durch folgenden Code von dem Spring MVC Programm geliefert.

```
/**
 * Parse the submit.json file and return its content as string
 * @return String
 */
@CrossOrigin(origins = "*", allowCredentials = "true", allowedHeaders = "*")
@RequestMapping
(value = "/json", method = RequestMethod.GET,produces = MediaType.APPLICATION_JSON_VALUE)
@ResponseBody
public String readingJSON() {

    // Get the content of the submit.json file
    ClassLoader classLoader = getClass().getClassLoader();
    File file = new File(classLoader.getResource("json/submit.json").getFile());

    // Read each line of the file and append the data into a StringBuilder
    StringBuilder result = new StringBuilder();
    try(Scanner scanner = new Scanner(file)){

        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            result.append(line).append("\n");
        }

        scanner.close();
    }catch(IOException e) {
        e.printStackTrace();
    }

    return result.toString();
}
```

Wie man an den Annotations der Cross-Origin erkennen kann wurde dafür gesorgt, dass Daten geliefert werden, auch wenn der GET Request von einer anderen Domain kommen sollte als die auf der das Spring MVC Programm deployed wurde.

Die Daten werden durch eine Java Klasse aus dem Eingabeformular gelesen, validiert und in das JSON Format formatiert.

Wenn Daten über das Eingabeformular übertragen werden, wird folgende Methode aufgerufen.

```
@RequestMapping(value = "/submit", method = RequestMethod.POST)
public String submit(@Valid @ModelAttribute("formModel") FormModel formModel,
        BindingResult result, ModelMap model) {
    if(result.hasErrors()) {
        return "form";
    }
    writingJSON(formModel.toString());
    return "result";
}
```

Diese mapped die eingegebenen Daten mit der Java Klasse FormModel und validiert diese. Sollten Fehler in der Eingabe vorhanden sein, zum Beispiel wenn ein Fakt gepflegt wurde aber keine Frage dazu, wird die Eingabeformularseite wieder aufgerufen, mit der Anmerkung an der jeweiligen Stelle welche Daten fehlen.
Sonst wird eine Methode zum Schreiben der eingegebenen Daten in das JSON aufgerufen und eine Erfolgsseite wird für den Admin/Sachbearbeiter zurückgegeben die indiziert, dass die Daten erfolgreich eingespielt wurden.

```
/**
 * Bean Model
 * Properties represent the fields declared in the formpage
 */
public class FormModel implements Serializable {


    private static final long serialVersionUID = 1L;

    private String subject;


    @NotEmpty(message = "Bitte ein Fakt einpflegen")
    private String fact;

    //@NotNull(message = "Bitte")
    @NotEmpty(message = "Bitte eine Frage einpflegen")
    private String question;

    //@NotNull(message = "Bitte")
    @NotEmpty(message = "Bitte eine Antwort einpflegen")
    private String correctAnswer;

    //@NotNull(message = "Bitte")
    @NotEmpty(message = "Bitte eine Antwort einpflegen")
    private String wrongAnswer1;

```

Anschließend werden die Daten durch das Plugin Jackson ObjectMapper aus einem Objekt zu einem JSON String formatiert und zur Weiterverarbeitung zur Verfügung gestellt.

```
    /**
     * Returns a String in JSON format
     * Manual adjustments were still required
     */
    public String toString() {
        try {
            return "\n" + new com.fasterxml.jackson.databind.ObjectMapper().writerWithDefaultPrettyPrinter().writeValueAsString(this);
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
        return "Please check the JSON Data";
    }
```

Das Eingabeformular wird mithilfe von JSP (JavaServer Pages) realisiert.
Es vereinfacht die Zusammenarbeit des Eingabeformulars mit der Datenmanipulation, Formatierung etc.

```
<form:form method="POST" commandName="formModel" action="/BuergermeisterRun/submit" modelAttribute="formModel">
    <table>
        <tr>
            <td><form:label path="subject">Thema auswählen</form:label></td>
            <td><form:select path="subject">
                <form:option value="russische-kirche">Russische Kirche</form:option>
                    <form:option value="neroberg">Neroberg</form:option>
                    <form:option value="warmer-damm">Warmer Damm</form:option>
                    <form:option value="kurhaus">Kurhaus</form:option>
                </form:select>
            </td>
        </tr>
        <tr>
            <td><form:label path="fact">Fakt eingeben</form:label></td>
            <td><form:input path="fact" /></td>
            <td><form:errors path="fact" class = "formValidationError" /></td>
        </tr>
        <tr>
            <td><form:label path="question">Frage eingeben</form:label></td>
            <td><form:input path="question" /></td>
            <td><form:errors path="question" class = "formValidationError" /></td>
        </tr>
```

Kleiner Ausschnitt der form.jsp Datei.

5 Installationsanleitung und Pflege

Installation Phaser Spiel

1. Für die Installation des Phaser Spiels wird ein Webserver benötigt.
   Die Wahl des Webservers ist nicht beschränkt.

   An sich ist die Installation recht einfach. Wenn ein Funktionsfähiger Webserver läuft, muss man lediglich den Inhalt des App Ordners in das Document Root des Webservers platzieren.

   Startpunkt des Spiels ist die index.html Datei in dem App Ordner.
   Da die meisten gängigen Webserver per default so vorkonfiguriert sind, dass immer nach einer index.html Datei gesucht wird, reicht es wenn das Document Root der App Ordner ist.

   Gegebenenfalls könnte man den App Ordner auch an einer anderen Stelle platzieren und einen Virtual Host konfigurieren der auf diesen Ordner zeigt.

   Wenn der App Ordner sich auf dem Zielserver befindet und der Webserver so konfiguriert ist, dass die index.html Seite aufgerufen wird, kann man mit dem nächsten Schritt beginnen.

2. Damit das Spiel sich die Daten für das Quiz und die Fakten in den jeweiligen Levels holen kann, muss man angeben auf welchem Server und wo genau sich das Spring MVC Programm befindet und die URL für die Ausgabe der JSON Datei angeben.

   Sobald man in Erfahrung gebracht hat wo das Spring MVC Programm installiert ist und wie die URL lautet für die Ausgabe der JSON Datei, muss man diese URL in der **app/js/states/LoadingState.js** in die Funktion **get_quiz_data()** angeben.

```
/**
 * Loads the quiz data from JAVA app with via GET request
 * @returns string[]
 */
BürgermeisterRun.LoadingState.prototype.get_quiz_data = function (){
    var result = "";
    $.ajax({
        url:"http://localhost:63342/OneNil/app/data.json",
        async: false,
        success:function(data) {
            result = data;
        }
    });
    return result;
};
```

Installation Spring MVC Programm

1. Für das Spring MVC Programm namens BuergerMeisteRun benötigt man einen Tomcat Server mit mindestens der Version 8.0

   Sobald man einen laufenden Tomcat Server aufgesetzt hat, muss man lediglich die BuergerMeisterRun/target/BuergermeisterRun.war Datei in den webapps Ordner des Tomcat-Servers platzieren.

   Der Tomcat Server würde dann basierend auf der BuergermeisterRun.war das Projekt installieren.

   Folgende URLs ergeben sich durch die Installation:

   - Eingabeformular Webseite: **<domain>/BuergermeisterRun/**

   - Abfrage der JSON Datei: **<domain>/BuergermeisterRun/json**
     (Diese URL muss in die get_quiz_data() Funktion des Phaser Spiels, Details oben)

Manuelle Pflege der JSON Datei

Wenn Bedarf besteht manuelle Änderungen an der JSON Datei vorzunehmen zum Beispiel um Daten zu löschen etc. geht das relativ einfach.

Dazu folgende Schritte durchführen:

1. Login auf dem Server auf dem das Spring MVC Programm deployed wurde
2. Navigation zum webapps Ordner der Tomcat Installation.
3. Datei **webapps/BuergermeisterRun/WEB-INF/classes/json/submit.json** öffnen und nach Bedarf bearbeiten.