ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

STL SYSTEMTECHNIKLABOR

Pi Retro Gaming mit HTML5

David Landry und Michael Reimsbach

Technical Report – STL-TR-2015-04 – ISSN 2364-7167

stl.htwsaar.de

Technische Berichte des Systemtechniklabors (STL) der htw saar
Technical Reports of the System Technology Lab (STL) at htw saar
ISSN 2364-7167

David Landry und Michael Reimsbach: Pi Retro Gaming mit HTML5
Technical report id: STL-TR-2015-04

First published: November 2015
Last revision: September 2015
Internal review: Thomas Kretschmer, Thomas Beckert

For the most recent version of this report see: https://stl.htwsaar.de/

Title image source: Michael Reimsbach

htw saar – Hochschule für Technik und Wirtschaft des Saarlandes (University of Applied Sciences)
Fakultät für Ingenieurwissenschaften (School of Engineering)
STL – Systemtechniklabor (System Technology Lab)
Prof. Dr.-Ing. André Miede (andre.miede@htwsaar.de)
Goebenstraße 40
66117 Saarbrücken, Germany
https://stl.htwsaar.de

ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

PI RETRO GAMING MIT HTML5

DAVID LANDRY UND MICHAEL REIMSBACH

Bachelor-Thesis
Studiengang: Praktische Informatik
Erstgutachter: Prof. Dr. Thomas Kretschmer
Einreichung: 30.09.2015

ZUSAMMENFASSUNG

In dieser Arbeit werden drei HTML5-Spiele mit Hilfe der JavaScript-Gaming-Engine **Phaser** entwickelt. Diese sollen auf dem Raspberry Pi der Firma Site Point betrieben werden.

Bei den Spielen wird jeweils auf die Konzepterstellung und auf die Implementierung eingegangen. Zudem wird erklärt, wie diese Spiele in einer rahmenlosen Anwendung auf einem Raspberry Pi mittels **NW.js** betrieben werden können. Es wird auch eine REST-Schnittstelle zur generischen Speicherung des Highscores entwickelt.

On my business card, I am a corporate president.
In my mind, I am a game developer.
But in my heart, I am a gamer.

— Satoru Iwata [75]

DANKSAGUNG

Unser Dank gilt Prof. Dr. Thomas Kretschmer und M.Sc. Thomas Beckert für die Betreuung, Prof. Dr.-Ing. André Miede für die Dokumentvorlage und Julia Bethscheider für die Grafiken in den Spielen.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

AKRONYME

API Application Programming Interface

HTML Hypertext Markup Language

XHTML Extensible Hypertext Markup Language

CSS Cascading Style Sheets

CMS Content-Management-System

MP3 MPEG-1/MPEG-2 Audio Layer III

WAV WAVE-Dateiformat

OGG OGG Vorbis Format

JSON JavaScript Object Notation

PNG Portable Network Graphics

UTF-8 8-Bit UCS (Universal Character Set) Transformation Format

MIT Massachusetts Institute of Technology

ARM Acorn RISC Machine

CPU Central Processing Unit

RAM Random Access Memory

MB Megabyte

GB Gigabyte

MHz Megahertz

GPU Graphics Processing Unit

REST Representational State Transfer

SQL Structured Query Language

WebGL Web Graphics Library

DOM Document Object Model

URI Uniform Resource Identifier

HTTP Hypertext Transfer Protocol

1 EINLEITUNG

1.1 ZIEL DER THESIS

Für den Arcade-Automaten der Firma Site Point [39], der mit einem Raspberry Pi [36] betrieben wird, sollen drei Spiele mittels Hypertext Markup Language (HTML) 5 [79] programmiert werden, die sich an drei bekannten Retro-Klassikern orientieren: Pac-Man [31], Space Invaders [41] und Light Cycles[47]. Zudem sollen sie möglichst einfach mit einem Launcher gestartet werden können.

Da die Spiele für eine Werbekampagne genutzt werden sollen, ist es wichtig, den Highscore abspeichern zu können, damit am Ende einer Werbe-Saison der Gewinner abgelesen werden kann.

1.2 MOTIVATION

1.2.1 *Der Arcade-Automat*

Die Firma Site Point ist im Besitz eines selbstgebauten Arcade-Automaten. Herzstück des Automaten ist ein Raspberry Pi, auf dem die Logik und später auch die Spiele laufen sollen. Für die Steuerung wird eine Einheit von X-Arcade verwendet [52]. Derzeit ist das RetroPie-Projekt[17] auf dem Raspberry Pi installiert, das Emulatoren für zahlreiche Konsolen bereitstellt und es ermöglicht, viele Spiele-Klassiker zu spielen. Im Rahmen dieser Thesis sollen eigene Spiele, die mit HTML5-Technologien entwickelt werden sollen, auf diesem Automaten gespielt werden können.

1.2.2 *Werbekampagne*

Die Spiele und der Arcade-Automat sollen Teil einer Werbekampagne werden, in der Kunden der Firma Site Point über einen bestimmten Zeitraum die Spiele spielen können, um sich einen Platz auf der Highscore-Liste zu sichern. Die besten Plätze sollen dann mit einem Preis belohnt werden.

1.3 AUFBAU DES DOKUMENTES

Im Rahmen einer Bachelor-Thesis wurde das Ziel aus Kapitel 1.1 im Zeitraum vom 01.07.2015-30.09.2015 umgesetzt. Das Dokument gliedert sich in fünf Kapitel:

(a) Von Vorne

(b) Von der Seite

(c) Die Steuerungseinheit

Abbildung 1: Der Arcade-Automat

auf Probleme und Erfahrungen eingegangen. Außerdem gibt es einen Ausblick auf weitere Features, die implementiert werden können.

1.4 AUFTEILUNG DER KAPITEL

DAVID LANDRY

- 1.1 Ziel der Thesis
- 2.1 Anforderungen (inkl. Unterkapitel)
- 2.6 Rahmenlose Anwendung
- 2.7 Game Design allgemein (inkl. Unterkapitel)
- 2.8 Retrospiele (inkl. Unterkapitel)
- 2.9 Spiel 1: Bugitar (inkl. Unterkapitel)
- 2.11.2 Spiel 3: Revenge of the Bugs: Konzept: Revenge of the Bugs
- 2.12 Die Spiele im Endlos-Konzept
- 3.1 Organisation (inkl. Unterkapitel)
- 3.2.1.1 Eingesetzte Technologien und Werkzeuge: Entwickler-Werkzeuge: Editoren
- 3.2.2.1 Eingesetzte Technologien und Werkzeuge: Der Spiele-Anteil: HTML5
- 3.2.2.2 Eingesetzte Technologien und Werkzeuge: Der Spiele-Anteil: JavaScript / ECMAScript
- 3.3.2 Allgemeine Architektur unserer Spiele: Grafiken, Animationen, Sounds
- 3.3.3 Allgemeine Architektur unserer Spiele: Tilemaps
- 3.4.2 Bugitar: Der Spieler
- 3.5.3 Site Point Arena: Spieler-Fähigkeiten
- 3.5.4 Site Point Arena: Portale
- 3.5.5 Site Point Arena: Pfeil-Felder
- 3.5.6 Site Point Arena: Geschwindigkeits-Booster
- 3.6.1 Revenge of the Bugs: Spieler
- 3.6.3 Revenge of the Bugs: Schutzwände
- 3.6.4 Revenge of the Bugs: Gegenstand: Honig

- 3.7.1 Launcher-Startskript in rahmenloser Anwendung: Launcher
- 4.2 Deployment (inkl. Unterkapitel)
- 5.1 Spieleentwicklung mit HTML5 und JavaScript (inkl. Unterkapitel)
- 5.2.1 Probleme: Debugging
- 5.3 Ausblick auf die Zukunft (inkl. Unterkapitel)

MICHAEL REIMSBACH

- 1.2 Motivation (inkl. Unterkapitel)
- 1.3 Aufbau des Dokumentes
- 2.2 Vorgehensweise (inkl. Unterkapitel)
- 2.3 Vergleich der Engines (inkl. Unterkapitel)
- 2.4 Raspberry Pi (inkl. Unterkapitel)
- 2.5 Anbindung zu Umbraco (inkl. Unterkapitel)
- 2.10 Spiel 2: Site Point Arena (inkl. Unterkapitel)
- 2.11.1 Spiel 3: Revenge of the Bugs: Analyse von Space Invaders
- 3.2.1.2 Eingesetzte Technologien und Werkzeuge: Entwickler-Werkzeuge: Lokaler Webserver XAMPP
- 3.2.2.3 Eingesetzte Technologien und Werkzeuge: Der Spiele-Anteil: Phaser Engine
- 3.2.3 Eingesetzte Technologien und Werkzeuge: Schnittstellen
- 3.3.1 Allgemeine Architektur unserer Spiele: Statemanager
- 3.3.4 Allgemeine Architektur unserer Spiele: Kollision
- 3.3.5 Allgemeine Architektur unserer Spiele: Highscore
- 3.4.1 Bugitar: Die Bugs / Käfer
- 3.4.3 Bugitar: Gegenstände
- 3.5.1 Site Point Arena: Zwei-Spieler-Modus
- 3.5.2 Site Point Arena: Schweife der Spieler
- 3.6.2 Revenge of the Bugs: Bugs / Käfer
- 3.7.2 Launcher-Startskript in rahmenloser Anwendung: Startskript

- 3.8 Datenbank-Anbindung mit Node.js (inkl. Unterkapitel)
- 4.1 Tests (inkl. Unterkapitel)
- 5.2.2 Probleme: Raspberry Pi und JavaScript
- 5.2.3 Probleme: Lernkurve bei der Entwicklung

2 ANALYSE UND KONZEPTION

2.1 ANFORDERUNGEN

Es sollen zwei oder drei Onlinespiele auf Basis von HTML5, JavaScript und Cascading Style Sheets (CSS) entwickelt werden, die per Konsolen-Aufruf auf einem Raspberry Pi gestartet werden können. Sie sollen dabei nicht in einem üblichen Web-Browser laufen, sondern in einer rahmenlosen Anwendung gespielt werden. Dennoch sollen sie in jedem Browser ohne zusätzliche Plugins lauffähig sein. Zusätzlich wird eine Schnittstelle zur Highscore-Speicherung benötigt, um beispielsweise über das Content-Management-System (CMS) **Umbraco** darauf zugreifen zu können.
Die Spiele sollen sich an drei alten Arcade-Klassikern orientieren:

1. Pac-Man
2. Space Invaders
3. Light Cycles (Zwei Spieler)

2.1.1 *Spieleentwicklung mit HTML5 und JavaScript*

Die Spiele sollen mit diesen Technologien realisiert werden, damit sie ohne weitere Plugins in jedem Browser lauffähig sind. Da HTML5 als Standard bereits am 28. Oktober 2014 vom World Wide Web Consortium W3C [79] vorgelegt wurde, bereitet dessen Verwendung in gängigen Browsern keine Probleme.
Auch JavaScript ist in jedem modernen Browser verfügbar und stellt somit kein Hindernis für die Lauffähigkeit der Spiele dar, sofern es beim Nutzer nicht deaktiviert wurde. Somit kann ohne Probleme eines der zahlreichen JavaScript-Frameworks zur Spieleentwicklung in HTML5 genutzt werden.

2.1.2 *Was ist mit CSS3?*

Da ein Spiel fast ausschließlich Bilder und Grafiken zur Darstellung des Geschehens nutzt, welche von der verwendeten JavaScript Engine gehandhabt werden, wird CSS3 innerhalb eines Spiels nicht gebraucht. Dennoch kann mit CSS3 sowohl der Spiele-Launcher als auch die Fläche außerhalb eines Spiels gestaltet werden. Dort befinden sich mitunter Informationen zur Bedienung eines Spiels.

2.1.3 *Bedeutung für die Thesis und Vorgehensweise*

Da die Spiele sich an den Arcade-Klassikern orientieren und den Retro-Look beibehalten sollen, muss analysiert werden, wie ein Spiel diesen Look erhält und welche Faktoren ein Retro-Spiel ausmachen. Außerdem dürfen sich die neuen Spiele weder zu sehr von den Klassikern unterscheiden noch als Kopie dieser angesehen werden.

Es muss eine Möglichkeit gefunden werden, auch auf einem Raspberry Pi eine Webanwendung ohne Browser starten zu können. Zudem muss sichergestellt werden, dass unsere Spiele auch auf einem Raspberry Pi spielbar sind (Siehe: 2.4).

Zusätzlich wird für die Highscore-Steuerung eine generische Datenbankanbindung die beste Lösung sein, um die Datenbank austauschbar zu halten.

2.2 VORGEHENSWEISE

Um unser Projekt erfolgreich umzusetzen, wählten wir zunächst ein passendes Vorgehensmodell aus der Softwaretechnik aus. Dadurch hatten wir stets einen Überblick über unseren Stand und konnten unsere Zeit gut für die einzelnen Teilprojekte einteilen.

2.2.1 *Welches Vorgehensmodell?*

In der Softwaretechnik gibt es einige Vorgehensmodelle, die sich teils stark unterscheiden, was die Frage aufwirft: „Welches Vorgehensmodell ist nun für unseren Zweck das Richtige?“. Um die Auswahl einzuengen, ist es sinnvoll, sich zunächst Gedanken über die eigenen Anforderungen an ein Vorgehensmodell zu machen. Unsere Hauptanforderung war, dass das Vorgehensmodell uns unterstützt und keine zusätzliche Arbeit erzeugt. Außerdem soll es für sehr kleine Teams anwendbar sein, in unserem Fall für ein Zwei-Mann-Team. Flexibilität ist ein weiterer Faktor, der uns wichtig ist, da wir dadurch leicht auf Änderungen reagieren können.

2.2.1.1 *Wasserfallmodell*

Das Wasserfallmodell ermöglicht eine gute Strukturierung, allerdings ist es für uns nicht flexibel genug, da die definierten Phasen vollständig durchzuführen sind. Hinzu kommt die Mehrarbeit durch das Erstellen eines Dokuments am Ende jeder Phase, weshalb das Wasserfallmodell auch „dokumentgetriebenes“ Modell genannt wird.

2.2.1.2 *Scrum*

Scrum hilft bei der agilen Softwareentwicklung und sieht vor, in kurzen Abständen Ergebnisse zu liefern. Dadurch erhöht sich die Qualität der Implementierung und es kann flexibel auf Änderungen eingegangen werden. Jedoch definiert Scrum auch viele Regeln und mehrere Rollen, weswegen es für ein kleines Team ungeeignet ist.

2.2.1.3 *Unser Vorgehen*

Die gängigen Vorgehensmodelle erschienen uns für ein so kleines Team ungeeignet, jedoch hielten wir es für unangebracht, einfach draufloszuentwickeln, deshalb entschlossen wir uns dazu, uns hauptsächlich an das Agile Manifest [1] zu halten. So haben wir in kurzen Abständen, in der Regel zwei Wochen, Ergebnisse präsentiert und diese mit dem Auftraggeber besprochen. So erhielten wir ständig Feedback und konnten auf Änderungswünsche problemlos eingehen. Ein weiterer Punkt des agilen Manifests ist, den Individuen ein angenehmes Arbeitsumfeld zu schaffen, was der Auftraggeber uns ermög-

lichte, indem wir unsere Arbeitszeiten frei einteilen konnten. Unsere Planung der Arbeitsschritte sah folgendermaßen aus:

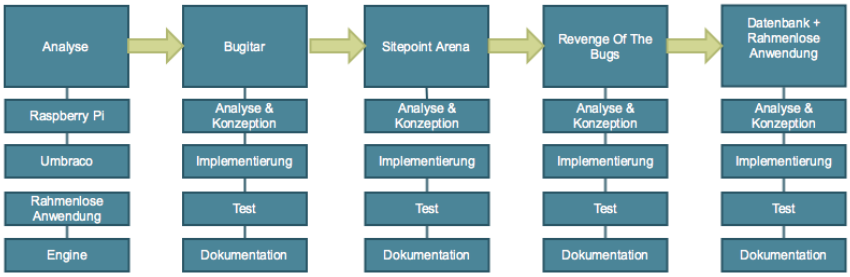


Abbildung 2: Projektplan

Wir haben das Projekt in fünf Meilensteine unterteilt: Analyse, Bugitar, Site Point Arena, Revenge Of The Bugs und Datenbank + Rahmenlose Anwendung. In der Analyse-Phase analysierten wir den Raspberry Pi 2.4, die Anbindung an Umbraco 2.5, die rahmenlose Anwendung 2.6 und die einzusetzende Engine 2.3. Die Spiele, Datenbank und rahmenlose Anwendung wurden jeweils in die Phasen Analyse & Konzeption, Implementierung, Test und Dokumentation unterteilt. In Kapitel 3.1 werden die Tools vorgestellt, die uns dabei unterstützten.

2.3 VERGLEICH DER ENGINES

Die Spiele wurden mit Hilfe einer JavaScript-Engine entwickelt, die die benötigten Kernfunktionen, wie z.B. die Physik, Bilder- und Soundverwaltung, etc. zur Verfügung stellt. Mittlerweile gibt es zahlreiche Engines, die es ermöglichen, Spiele für den Webbrowser zu entwickeln. Als Einstieg in die Spieleentwicklung mit HTML5 gab uns der Auftraggeber ein Video-Tutorial[45], das uns die Engine Lime.js vorstellte. Nachdem wir die ersten Erfahrungen sammelten, schauten wir nach weiteren Alternativen, die in Frage kommen. Wir fanden die Website „HTML5 Game Engines“[19], auf der HTML5-Engines nach ihrer Popularität aufgelistet werden.

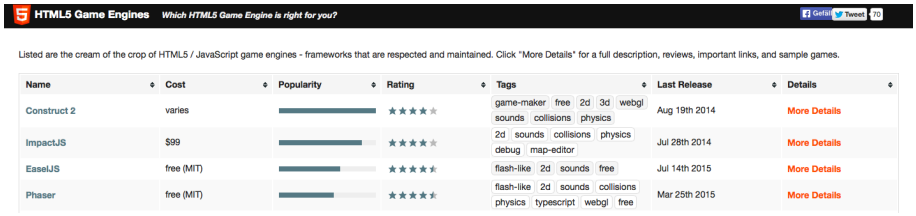


Abbildung 3: Rangliste der Engines (Abgerufen am 28.09.2015)

2.3.1 *Anforderungen*

Um die Auswahl einzugrenzen, legten wir zuerst unsere Hauptanforderungen, die wir an die Engine stellen, fest:

- Eine kostenfreie Nutzung der Engine
- Eine gute Dokumentation, die den Einstieg erleichtert

Die zu implementierenden Spiele sind allesamt zweidimensional und orientieren sich an Retro-Klassikern wie Pac-Man (Siehe Kapitel 2.9.1), Space Invaders (Siehe Kapitel 2.11.1) und Light Cycles (Siehe Kapitel 2.10.1). Deshalb muss die Engine folgende Features unterstützen:

- Physik für 2D-Spiele
- Multiplayer
- Tilemap-Unterstützung (Siehe Kapitel 3.3.3)

Nachdem wir unsere Anforderungen definiert hatten, schauten wir uns die populärsten Engines an.

2.3.2 *Construct 2 und ImpactJS*

Obwohl Construct 2 [8] und ImpactJS [21] die Rangliste anführen, entschieden wir uns gegen die Nutzung einer dieser Engines, da sie eine Lizenz benötigen.

2.3.3 *EaselJS*

EaselJS[11] befindet sich in der Rangliste auf Platz 3 und machte zunächst einen guten Eindruck. Allerdings ist die Dokumentation im Vergleich zu der von Phaser klein, weswegen wir uns nicht für EaselJS entschieden haben.

2.3.4 *LimeJS*

Auch wenn LimeJS[23] in der Rangliste im mittleren Feld gelistet ist, schauten wir uns die Engine genauer an. Doch die Dokumentation hielt uns, wie bei EaselJS, davon ab, die Engine zu nutzen.

2.3.5 *Die Phaser-Engine*

Unsere Wahl fiel auf die Phaser-Engine [34], denn sie ist eine Opensource-Engine und unterliegt der Massachusetts Institute of Technology (MIT)-Lizenz, die eine uneingeschränkte Nutzung ermöglicht. Somit gibt es keine Schwierigkeiten, die Spiele später zu veröffentlichen oder die Engine an die eigenen Bedürfnisse anzupassen. Die Engine ist umfassend dokumentiert, bietet viele Einführungs-Guides und gehört zu den derzeit populärsten JavaScript-Engines. Dadurch konnten wir uns zügig in ein fremdes Thema einarbeiten. Phaser stellt mehrere Arten von Spielephysiken zur Verfügung, darunter eine auf Arcade-Games spezialisierte, die für unsere Spiele perfekt geeignet ist. Phaser bietet die Möglichkeit, einen Online-Mehrspieler-Modus zu implementieren. Das übertrifft unsere Anforderung, einen lokalen Zwei-Spieler-Modus zu entwickeln, bei weitem. Eine große Anzahl an Retro-Spielen wird mit Hilfe von Tilemaps (Siehe Kapitel 3.3.3) entwickelt, weshalb wir uns ebenfalls dazu entschlossen haben, zwei unserer Spiele auf der Basis von Tilemaps zu entwickeln. Phaser bietet dazu zahlreiche Funktionen an, die eine Entwicklung mit Tilemaps unterstützen.

Sollte später der Bedarf bestehen, die entwickelten Spiele auf mobile Plattformen wie Smartphones oder Tablets zu veröffentlichen, steht die Phaser Engine nicht im Wege. Die Engine legt starken Fokus auf mobile Geräte, deshalb werden neue Funktionen nur in die Application Programming Interface (API) aufgenommen, wenn sie stabil auf mobilen Endgeräten laufen.

2.4 RASPBERRY PI

2.4.1 *Ausgangssituation*

Das Unternehmen Site Point besitzt einen selbstgebauten Arcade-Automaten, dessen Herzstück ein Raspberry Pi ist, auf dem die zu implementierenden Spiele gespielt werden sollen.

2.4.2 *Allgemeines zum Raspberry Pi*

Der Raspberry Pi ist ein Einplatinencomputer, der entwickelt wurde, um Studenten bzw. Interessierten einen kostengünstigen Einstieg in die Welt des Programmierens zu bieten. Der Raspberry Pi ist mit einem Acorn RISC Machine (ARM)-Mikroprozessor ausgestattet. Die ARM-Architektur eignet sich durch ihren effizienten Befehlssatz besonders für den Raspberry Pi Dembowski [63], jedoch ist sie auch der Grund, weshalb nicht jede Linux-Distribution bzw. Anwendung ausführbar ist, sondern an die Architektur angepasst werden muss. Wir nutzen das Betriebssystem Raspbian [38]. Daher mussten wir sicherstellen, dass unsere Technologien auf dem Raspberry Pi bzw. dem Betriebssystem unterstützt werden.

2.4.3 *Anforderungen an den Raspberry Pi*

2.4.3.1 *HTML5*

Die Spiele werden mit HTML5-Technologien entwickelt, deshalb ist die wichtigste Anforderung an den Raspberry Pi die Unterstüzung von HTML5-Technologien. Seit geraumer Zeit besitzt der Raspberry Pi mit dem Epiphany Webbrowser [15] HTML5-Unterstützung Upton [78]. Neben Epiphany gibt es noch weitere Webbrowser wie Chromium [7] oder Iceweasel[20], die HTML5 unterstützen.

2.4.3.2 *Testlauf auf dem Raspberry Pi A*

Nachdem wir die Sicherheit hatten, dass HTML5-Technologien auf dem Raspberry Pi unterstützt werden, testeten wir eine simple Version von Bugitar auf dem Raspberry Pi Modell A. Diese war jedoch nicht spielbar, da die begrenzten Ressourcen des Raspberry Pis sofort an ihr Limit stießen. Sowohl der Arbeitsspeicher als auch die Central Processing Unit (CPU) waren sofort komplett ausgelastet.

2.4.3.3 *Testlauf auf dem Raspberry Pi 2 Modell B*

Um das Ressourcenproblem zu umgehen, testeten wir die simple Version von Bugitar auf dem Raspberry Pi 2 Modell B, der eine deutlich stärkere Hardware besitzt als der Raspberry Pi Modell A.

RESSOURCE	PI 1 MODELL A	PI 2 MODEL B
RAM	256MB	1GB
CPU	700MHz ARM1176JZF-S	900 MHz Quad-Core ARM Cortex-A7

Tabelle 1: Vergleich Raspberry Pi Modell A und Raspberry Pi 2 Modell B

In Eben Uptons Blog-Beitrag Upton [78] wurde erwähnt, dass aufwendige JavaScript-Anwendungen auf dem Raspberry Pi nicht flüssig laufen. Der Blog-Beitrag erschien zwar im Jahr 2014, allerdings hat sich an dieser Situation nicht viel geändert. Obwohl unsere Spiele auf dem Raspberry Pi 2 gespielt werden können, laufen sie nicht so flüssig wie auf einem Desktop-Rechner. In den Raspberry Pi Foren[1] wird es damit begründet, dass auf dem Raspberry Pi keine Graphics Processing Unit (GPU) Unterstützung genutzt wird, dadurch übernimmt der Prozessor die ganze Arbeit, was dazu führt, dass die Anwendung langsam ist. Wir nutzten dennoch den Raspberry Pi 2.

2.4.3.4 *Datenbank*

Um den Highscore abspeichern zu können, benötigen wir eine relationale Datenbank, die auf dem Raspberry Pi laufen soll, deswegen recherchierten wir zunächst, ob und welche relationalen Datenbanken auf dem Raspberry Pi laufen. Einige sind:

- PostgreSQL [35]
- MySQL [26]
- SQLite [43]

Nachdem sichergestellt war, dass es eine Auswahl an relationalen Datenbanken gibt, konnten wir in Kapitel 2.5 die für unsere Zwecke geeignete Datenbank auswählen.

2.4.3.5 *Node.js*

Als Ausgangsbasis für die rahmenlose Anwendung diente der Artikel Kelleher [69] von Fionn Kelleher, in dem er schildert, wie rahmenlose Anwendungen in Node.js erstellt werden können. Außerdem nutzen wir Node.js auch für unsere Anbindung zu Umbraco2.5, deshalb mussten wir prüfen, ob Node.js auf dem Raspberry Pi unterstützt wird. Wir fanden einen ARM-Build zu Node.js[27] und konnten sichergehen, dass wir Node.js auf dem Raspberry Pi verwenden können.

1 Bsp: http://raspberrypi.stackexchange.com/questions/5096/where-is-the-bottleneck-of-web-browse-speed-on-raspberry-pi

2.5 ANBINDUNG ZU UMBRACO

2.5.1 *Was ist Umbraco?*

Umbraco ist ein Open Source Content-Management-System[48], das auf dem ASP.NET-Framework[2] von Microsoft basiert. Es wurde von Niels Hartvig entwickelt und eignet sich dazu, dynamische Webseiten zu erstellen.

2.5.2 *Wieso eine Anbindung zu Umbraco?*

Umbraco wird von der Firma Site Point eingesetzt und die Highscore der Spiele soll darüber verwaltet werden können. Zu Beginn einer neuen Werbekampagne soll die aktuelle Highscore-Tabelle zurückgesetzt werden und am Ende der Werbekampagne soll der Gewinner in das Umbraco-System eingetragen werden können. Um das zu ermöglichen, wird eine Schnittstelle zwischen Umbraco und Datenbank benötigt.

2.5.3 *Die Schnittstelle*

Wir entschieden uns für eine generische Schnittstelle, die nicht von Umbraco abhängig ist, sodass sie auch außerhalb von Umbraco genutzt werden kann. Sollte das Unternehmen Site Point zu einem anderen Content-Management-System wechseln, kann die Schnittstelle immer noch integriert werden. Um das zu erreichen soll ein Datenbank-Server erstellt werden, der mit Hilfe von Representational State Transfer (REST)-Abfragen angesprochen werden kann.

2.5.4 *Die Datenbank*

Um die Ressourcen des Raspberry Pis zu schonen, wählten wir als Datenbank SQLite [42]. Die leichtgewichtige relationale Datenbank SQLite besteht lediglich aus einer Datei, die unter 1 MegaByte groß ist und mit gängigen Structured Query Language (SQL)-Abfragen arbeitet. Datenbanksysteme wie beispielsweise MySQL [25] haben den großen Nachteil, dass der Arbeitsspeicher und der Prozessor des Raspberry Pis stark ausgelastet werden, da permanent der MySQL-Service aktiv ist. Für unsere Spiele, die bereits ressourcenintensiv sind, ist SQLite folglich die perfekte Wahl.

Um eine REST-API bereitzustellen, entschieden wir uns dazu, einen Web-Server mit Node.js[66] zu erstellen. Der Vorteil von Node.js ist, dass es ein schlankes und effizientes Framework ist und somit für den Raspberry Pi geeignet ist. Node.js ermöglicht den Einsatz von JavaScript auf der Serverseite, dadurch müssen wir keine Technologien vermischen, sondern können bei dem Haupteinsatz von JavaScript

bleiben. Für Node.js existiert das Web-Framework Express [77], das es ermöglicht, Routen zu definieren, die auf Anfragen des Clients reagieren können.

Nachdem wir uns auf die Datenbank und die Technologie festgelegt hatten, definierten wir unsere Datenbanktabellen sowie die jeweiligen Routen, die wir zur Interaktion benötigten.

2.5.4.1 *Die Tabellen*

Für zwei unserer Spiele benötigen wir jeweils eine Highscore- und eine Hall-of-Fame-Tabelle. In letzterer sollen die besten Spieler aller Werbekampagnen gespeichert werden. Der Aufbau der Tabellen ist identisch und sieht folgendermaßen aus:

Id	**User**	**Points**
1	Michael	420
2	David	404
3	Kek	1337

Tabelle 2: Beispiel Highscore

2.5.4.2 *Die Routen*

Für die Interaktion mit der Datenbank werden nun noch die entsprechenden Routen benötigt. Dabei haben wir uns an den CRUD-Operationen orientiert:

- Create, einen neuen Datensatz anlegen
- Read, einen Datensatz lesen
- Update, einen Datensatz aktualisieren
- Delete, einen Datensatz löschen

Unsere Routen sehen folgendermaßen aus:

HTTP Verb	**Route**	**Aktion**
GET	/TabellenName	Alle Datensätze des Spiels anzeigen
GET	/TabellenName/:id	Datensatz mit dieser Id anzeigen
POST	/TabellenName	Erzeugt einen neuen Datensatz
DELETE	/TabellenName/:id	Löscht Datensatz mit dieser Id
PUT	/TabellenName/:id	Aktualisiert Datensatz mit dieser Id
DELETE	/TabellenName/reset	Alle Datensätze löschen

Tabelle 3: Routen

Um eine Datenbank zurückzusetzen, wählten wir bewusst die Route /TabellenName/reset statt einem DELETE Request ohne Id, damit nicht versehentlich die komplette Datenbank zurückgesetzt wird, wenn die Id vergessen wird.

2.6 RAHMENLOSE ANWENDUNG

Eine der Anforderungen war es, die Spiele auch ohne Browser in einer rahmenlosen Anwendung spielbar zu machen. Bereits vor der Bachelorarbeit fiel uns beim Programmieren mit dem Editor **Atom**, der mittels **Electron** [14] als Desktop-Anwendung mit Web-Technologien funktioniert, auf, dass solche Desktop-Webanwendungen immer populärer werden. Weitere Beispiele für Werkzeuge, die die Verwendung von Web-Technologien für Desktopanwendungen erlauben, sind **Brackets Shell** [5] und **NW.js** [30].
Mit ihnen wurden unter anderem **Brackets**, **Powder Player**, **Messenger for Desktop** (Facebook), **Visual Studio Code**, **Slack** und noch viele weitere[2] entwickelt.

Die Brackets Shell von Adobe wurde hauptsächlich für den Brackets Editor entwickelt, statt als Basis für andere Anwendungen zu dienen, auch wenn dies prinzipiell möglich ist. Daher schränkten wir unsere Auswahl auf Electron und NW.js ein.

Electron von GitHub, früher als Atom-Shell bekannt, diente ursprünglich auch nur für den Atom Editor. Allerdings ist es möglich, auch andere Anwendungen mit dessen Hilfe zu erstellen, und es gibt bereits zahlreiche Beispiele dafür [14]. Es basiert auf Chromium und io.js und bedient sich dank GitHub einer großen Community.

NW.js, früher Node-Webkit, scheint die gängigste und auch simpelste der drei Varianten zu sein, wenn auch Electron dank des Atom Editors schnell an Popularität zunimmt. Es basiert auf Chromium und Node.js und erlaubt ein einfaches Starten einer HTML-Datei in einer eigenen Anwendung mittels Kommando.

Da unsere Spiele auf einem Raspberry Pi, also auf einem ARM-basierten System, laufen sollen, fiel unsere Wahl auf NW.js. Denn nur von NW.js gibt es eine Version, die auf einem ARM-Prozessor lauffähig ist [28]. Leider ist diese Version nicht offiziell, allerdings funktioniert sie unter Raspbian auf dem Raspberry Pi 2 des Site Point Automaten ohne Probleme.

2 Siehe Electron-Website [14] und List of Apps and Companies using nw.js [29]

2.7 GAME DESIGN ALLGEMEIN

2.7.1 *Was ist ein Spiel?*

Diese Frage scheint sehr einfach zu beantworten zu sein. Dennoch stellt sich heraus, dass es keine allgemein gültige Definition gibt. Laut Duden handelt es sich bei einem Spiel beispielsweise um eine

> „Tätigkeit, die ohne bewussten Zweck zum Vergnügen, zur Entspannung, aus Freude an ihr selbst und an ihrem Resultat ausgeübt wird.“[10]

Eine andere Definition lieferte Huizinga 1938/39 in seinem Werk Homo Ludens[68]:

> „Spiel ist eine freiwillige Handlung oder Beschäftigung, die innerhalb gewisser festgesetzter Grenzen von Zeit und Raum nach freiwillig angenommenen, aber unbedingt bindenden Regeln verrichtet wird, ihr Ziel in sich selber hat und begleitet wird von einem Gefühl der Spannung und Freude und einem Bewusstsein des ‚Andersseins‘ als das ‚gewöhnliche Leben‘.“

Es lassen sich jedoch Kernelemente festlegen, die zwar nicht zwingend bei jedem Spiel erforderlich, aber bei den meisten durchaus vorhanden sind: [72, S.73-75]:

- **Zielvorgaben**: Es gibt Vorgaben, die der Spieler tun muss, um Fortschritt im Spiel zu erlangen.
- **Ergebnis**: Ein Spiel hat verschiedene Endergebnisse, die das Spiel beenden und aus den Handlungen des Spielers resultieren. Die einfachsten Ergebnisse sind Sieg und Niederlage.
- **Ungewissheit**: Während im echten Leben die Unsicherheit oft negativ behaftet ist, ist sie in Spielen durchaus erwünscht. Wer möchte schon ein Spiel spielen, bei dem alles von vornherein bekannt ist?
- **Regeln**: Regeln bilden die Struktur eines jeden Spiels und Spieler müssen sich an diese Regeln halten.
- **Rahmen**: Spiele erschaffen temporäre Grenzen zur echten Welt, einen Spiele-Kontext.

2.7.2 *Schwierigkeitsgrad*

Es ist wichtig, einen guten Schwierigkeitsgrad im Spiel zu finden, durch den der Spieler weder frustriert noch gelangweilt wird [72, S. 125-126] . Das Prinzip des Flows von Mihály Csíkszentmihályi[57]

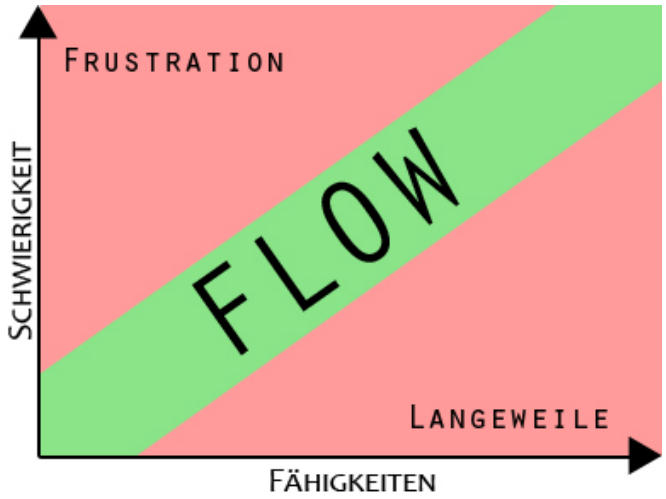


Abbildung 4: Balance zwischen Schwierigkeit und Fähigkeiten nach der Idee des *Flows* von Mihály Csíkszentmihályi

lässt sich gut auf den Schwierigkeitsgrad eines Spiels anwenden Adams[58, S.339].

Solange eine Balance zwischen Schwierigkeit und Fähigkeiten des Spielers gehalten wird, hat das Spiel einen angemessenen Schwierigkeitsgrad. Wir wollten dies in unseren Spielen damit erreichen, dass der Schwierigkeitsgrad der Spiele stetig und nicht zu schnell ansteigt.

2.8 RETROSPIELE

Unsere Spiele sollten sich an Retrospielen orientieren, deshalb mussten wir herausfinden, was ein Retrospiel ausmacht. Wir analysierten einige Retrospiele und fanden auch im Internet Beiträge, die dieses Thema behandelten.

2.8.1 *Was macht ein Retrospiel aus?*

Um zu klären, was ein Retrospiel ausmacht, muss zunächst geklärt werden, was Retro allgemein bedeutet. Laut Duden handelt es sich bei Retro um die

> „[bewusste] Nachahmung von Elementen früherer Stilrichtungen in Musik, Design o. Ä.“[9]

Wie lässt sich das auf Spiele übertragen? Der Artikel von Anderson[59] wirft viele interessante Sichtweisen auf dieses Thema auf:

2.8.1.1 *Das Alter*

Eine Möglichkeit, etwas als Retro zu klassifizieren, ist das Alter. Wir stimmen mit dieser Aussage jedoch nur zum Teil überein, denn wir sind der Meinung, dass es auch heute noch durchaus möglich ist, ein Retro-Design zu erschaffen.

2.8.1.2 *Design*

Dies scheint unter der Allgemeinheit der wichtigste Faktor zu sein. Auch wir sind der Meinung, dass ein Retro-Stil sich durch dessen Design definiert. Dabei gibt es wesentliche Aspekte (Peterson[71]), die beachtet werden müssen, um den typischen Retro-Look zu erreichen, und welche zum großen Teil auch auf Spiele angewandt werden können:

- **Farben:** Bei einem Retro-Look werden meist nur wenige verschiedene Farben verwendet, da früher Drucker-Farben gespart werden mussten.
- **Formen:** Oft werden auch nur simple Formen benutzt, wie Kreise oder Linien.
- **Schrift:** Ein weiterer Faktor ist die Schriftart. Oft lässt sich allein durch die verwendete Schriftart ein Retrostil erkennen. Vor allem in Spielen trifft dies zu: Die Texte alter Spiele haben meist ein verpixeltes Aussehen.
- **Ränder:** Retro-Designs besitzen häufig mehr Ränder und Konturen als moderne Designs. Diese sind oft nicht nur einfarbig, sondern auch verziert.

- **Texturen und Rauschen:** Vor allem Hintergründe werden oft mit Texturen oder Rauschen verziert.
- **Logos:** Die verwendeten Logos in einem Retro-Bild haben oft das Aussehen von Abzeichen.

2.8.2 *Subjektive Betrachtung und Nostalgie*

Für manche Menschen ist der Retro-Begriff einfach nur subjektiv: Für sie bedeutet Retro etwas anderes, als für junge Generationen, da sie die Ära selbst miterlebt haben. Oft gehen nostalgische Gefühle damit einher, wenn sie sich die Spiele aus ihrer Kindheit anschauen.

Auch wenn das Design für Spiele der ausschlaggebende Punkt zu sein scheint, ist das nicht alles. Bei unserer Analyse mehrerer alter Spiele stellte sich heraus, dass die meisten Retrospiele sehr simpel aufgebaut waren. Bewegungs- und Handlungsmöglichkeiten beschränkten sich auf einfachste Funktionen.

Beim Erstellen unserer Spiele legten wir also Wert auf folgende Punkte:

- Bei unseren Grafiken wollen wir eine beschränkte Anzahl an Farben verwenden.
- Außerdem sollen die Texturen nicht zu komplex aussehen.
- Die Möglichkeiten eines Spielers sollen sich auf einfache Funktionen beschränken.
- Die Schriftart soll eine pixelartige sein.

2.9 SPIEL 1: BUGITAR

2.9.1 *Analyse von Pac-Man*

2.9.1.1 *Vorgaben*

Für das erste Spiel hatten wir die Vorgabe, uns am Spiel „Pac-Man“ zu orientieren. Dieses wurde erstmals in Japan von **Namco** entwickelt und 1981 auch in den USA veröffentlicht Rabin [72, Seite 23]. Der Protagonist steuert durch ein Labyrinth, in welchem er Punkte auffressen muss und von Geistern verfolgt wird. Wird er von einem Geist getroffen, verliert er ein Leben. Allerdings kann er durch Aufsammeln eines der größeren Punkte die Geister auffressen, welche dann vor ihm fliehen. Schafft er es dennoch, verschwindet der gefressene Geist für bestimmte Zeit in den Käfig in der Mitte des Spiels.

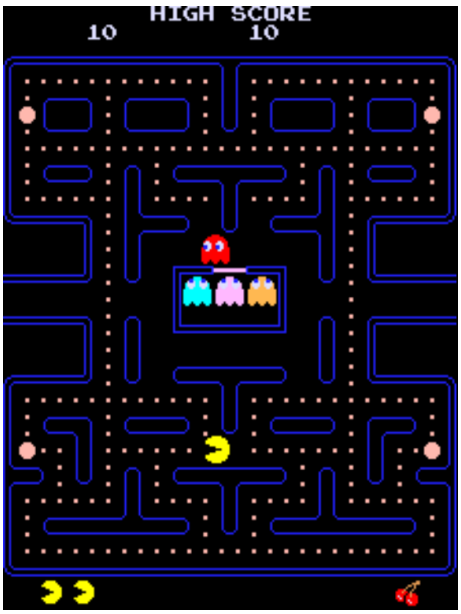


Abbildung 5: Der Arcade-Klassiker Pac-Man

Durch diese Vorgaben standen bereits einige Dinge für unser Spiel fest:

1. Der Protagonist wird in Vogelperspektive durch ein Labyrinth gesteuert.
2. Dabei sammelt er Objekte auf, welche den Punktestand erhöhen.
3. Er darf dabei von Gegnern, die sich ebenfalls durch das Labyrinth bewegen, nicht berührt werden.
4. Er kann seine Gegner jedoch kampfunfähig machen, wenn er ein spezielles Objekt aufsammelt.

Wir wollten natürlich keine Kopie des Klassikers erstellen und überlegten deshalb, wie wir dieses Spielekonzept etwas auffrischen konnten, ohne dabei zu sehr von „Pac-Man“ abzuweichen.

2.9.1.2 *Eigene Ideen*

Um etwas Spannung in das Spiel zu bringen, wollten wir, dass die sammelbaren Gegenstände, die den Punktestand erhöhen, vom Gegner fallen gelassen werden, statt von Anfang an im Labyrinth verteilt zu sein. Dadurch entsteht Dynamik, da man zwar die Gegenstände berühren muss, ihre Herkunft jedoch gar nicht berühren darf.

Den Gedanken, mehr Spannung in das Aufsammeln der Objekte zu bringen, wollten wir fortführen: Es sollte ein Nachteil entstehen, wenn der Protagonist ein Objekt zu lange nicht aufsammeln würde. Dabei dachten wir an tickende Bomben, die bei Explosion den Punktestand erniedrigen; oder, wie im Fall unseres Konzeptes, an Eier, aus denen weitere Gegner schlüpfen konnten.
Dadurch kann es natürlich zu einer viel zu hohen Anzahl an Gegnern kommen. Hier kommen die Gegenstände ins Spiel, mit deren Hilfe der Protagonist Gegner zerstören kann. Diese sollen in einem bestimmten Zeitintervall zufällig auf der Karte erscheinen. Hebt der Spieler einen solchen Gegenstand auf, kann er ihn als Projektil verwenden, um Gegner zu zerstören.

Zusätzlich wollten wir dem Spieler Hilfe in Form weiterer einsammelbarer Gegenstände geben: ein Gegenstand, der nach Einsammeln alle Gegner für kurze Zeit verlangsamt.

2.9.1.3 *Das Thema des Spiels*

Nun fehlte noch ein Thema für unser Spiel, eine kleine Hintergrund-Geschichte, die den Handlungen des Spiels einen Grund gab. Es lag nahe, sich dabei am Helden-Thema der Firma Site Point, für welche dieses Spiel entwickelt wird, zu orientieren. Deshalb sollte unser Protagonist ein Held sein, der vom Aussehen her zu diesem Schema passen würde.

Zudem sollte ein Bezug zur Informatik geschaffen werden, weshalb unsere Wahl für die Gegner auf Käfer fiel. Auf Englisch also „Bug“, ein Begriff, der in der Informatik für Softwarefehler verwendet wird.

2.9.2 *Konzept: Bugitar*

Der Protagonist, ein Site Point-Held namens Tommy Beckman, muss in einem Labyrinth gegen Käfer antreten. Diese lassen Eier fallen, welche vom Spieler für Punkte aufgesammelt werden können und müssen, da sonst nach einer bestimmten Zeit neue Käfer aus ihnen schlüpfen. Die Käfer werden mit der Zeit schneller, damit sich der

Abbildung 6: Das Helden-Thema der Firma Site Point

Schwierigkeitsgrad erhöht. Um dem Spieler zu helfen, erscheinen zufällig auf der Karte weitere Hilfsgegenstände: Schilde, die der Spieler abfeuern kann, um Eier und Käfer zu zerstören, und Honig, um alle Käfer für kurze Zeit zu verlangsamen. Außerdem verfügt der Held von Anfang an über 3 Insektensprays, die ihn jeweils einen Zusammenstoß mit einem Käfer überleben lassen. Stößt er ein weiteres Mal auf einen Käfer, ist das Spiel zu Ende. Damit das Spiel nicht durch Zerstören aller Käfer beendet wird, gibt es Käfer, die unzerstörbar sind.

Doch wo findet das ganze statt? Auf dem Planeten **Bugitar**.

2.10 SPIEL 2: SITE POINT ARENA

2.10.1 *Analyse von Light Cycles*

Im Gegensatz zu Bugitar soll Site Point Arena aus einem Zwei-Spieler-Modus bestehen, sodass zwei Spieler gegeneinander antreten und ihre Fähigkeiten messen können. Als Konzept-Vorlage dient das Spiel Light Cycles.

Midway Games veröffentlichte 1982 das Arcade-Spiel Tron, das an den gleichnamigen Film angelehnt ist. Das Spiel besteht aus den vier Spielmodi:

1. Grid Bugs
2. MCP Cone
3. Tanks
4. Light Cycles

Für uns ist lediglich der Modus Light Cycles interessant, weshalb wir diesen analysiert haben.

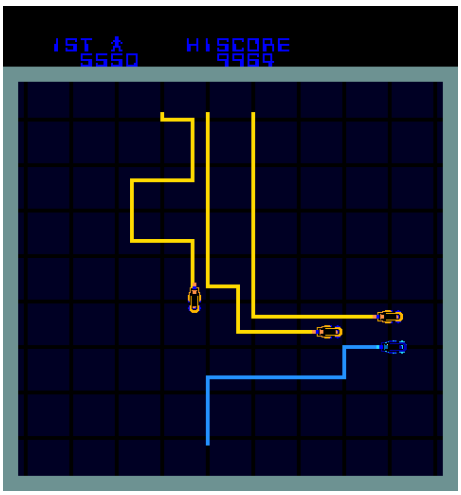


Abbildung 7: Der Arcade-Klassiker Light Cycles

1. Beide Spieler sehen die Spielwelt in einem gemeinsamen Fenster aus der Vogelperspektive.
2. Jeder Spieler steuert einen Charakter, der eine Wand hinterlässt.
3. Sobald ein Charakter mit seiner eigenen Wand, der Wand eines anderen Spielers oder mit der Wand der Spielwelt in Berührung kommt, ist das Spiel zu Ende.
4. Spieler können ihr Bewegungstempo kurzzeitig erhöhen.

2.10.1.1 *Eigene Ideen*

Da Light Cycles sehr simpel aufgebaut ist, haben wir das Konzept als Basis genommen und weitere eigene Ideen eingebracht.

Wie im ursprünglichen Spiel steuern die Spieler ihren Charakter, der in unserem Fall einen Site Point Helden darstellt, in der Vogelperspektive. Die Site Point Helden hinterlassen einen Superhelden-Schweif, der die Wand repräsentieren soll. Das Spiel ist vorbei, sobald ein Spieler mit einem Schweif in Berührung kommt oder die Spielwelt verlässt.
Im Gegensatz zu Light Cycles wird das Bewegungstempo eines Spielers erst nach dem Einsammeln eines bestimmten Gegenstands erhöht. Der Gegenstand erscheint in einem bestimmten Zeitintervall und wird als Site Point Energy-Drink in der Spielwelt dargestellt, der das Bewegungstempo beim Einsammeln kurzzeitig erhöht. Dadurch wird der Wettbewerb zwischen den Spielern erhöht und die Spielspannung steigt.
Beim Spielen fiel uns auf, dass die Wege des Mitspielers in manchen Fällen vorhersehbar waren und das Spiel dadurch an Spielspaß verlor. Deshalb überlegten wir uns Möglichkeiten, das Spielverhalten dynamischer zu gestalten. Wir entschieden uns für die Einführung von Portalen. Betritt ein Spieler Portal A, so wird er zu Portal B teleportiert. Ein weiteres Element, das die Dynamik des Spiels erhöht, sind Felder, die die Laufrichtung beim Darüberlaufen ändern.
Dadurch kann sich der Spieler ausgeklügelte Taktiken ausdenken, um seinem Kontrahenten den Weg abzuschneiden und somit das Spiel zu gewinnen.
Wir wollten die Fehlentscheidungen des Spielers nicht sofort bestrafen. Die Spieler sollen einen Notfallplan in der Hinterhand haben, der ihnen nochmal das Leben retten kann. Deshalb hat jeder Charakter eine besondere Heldenfähigkeit, die beim Aktivieren einen Schweif zerstört, sodass der Spieler hindurch laufen kann.

2.10.1.2 *Das Thema des Spiels*

Wie schon bei Bugitar überlegten wir uns auch für Site Point Arena eine kleine Hintergrund-Geschichte, damit das Spiel auch einen Sinn hat. Das Superhelden-Thema von Site Point konnte wieder genutzt werden, um die Spielcharaktere zu modellieren. Außerdem konnten wir mit dem Site Point Energy-Drink einen weiteren Bezug zum Unternehmen herstellen.

2.10.2 *Konzept: Site Point Arena*

In der Site Point Arena liefern sich die Site Point-Superhelden, Tommy Beckman und Romano Davinco, ein unerbittliches Duell. Die per-

Abbildung 8: Der Site Point Energy-Drink

sönliche Superheldenkraft kann dazu genutzt werden, in aussichtslosen Situationen nochmal einen Ausweg zu finden und dem Gegner einen Strich durch die Rechnung zu machen. Mit Hilfe von Portalen können unvorhersehbare Manöver durchgeführt werden, die den Kontrahenten staunen lassen. Doch die Helden müssen auf der Hut sein, denn manche Felder können ihren Weg umlenken, sodass in kurzer Zeit auf die neue Situation reagiert werden muss. Sollte einer der Helden eine Extraladung an Reserven brauchen, können sie den Site Point Energy-Drink nutzen und ihre letzten Kraftreserven sammeln, um einen kurzen Power-Sprint hinzulegen.

2.11 SPIEL 3: REVENGE OF THE BUGS

2.11.1 *Analyse von Space Invaders*

Das letzte Spiel orientiert sich an dem Arcade-Klassiker „Space Invaders“, das von Toshihiro Nishikado entworfen und programmiert und von der Firma Taito vertrieben wurde Rabin [72, Seite 22ff.]. Der Spieler steuert eine Kanone, die er entweder nach links oder rechts bewegen kann. Mit der Kanone schießt er auf Aliens, die vom oberen Bildschirmrand kommen und sich abwärts bewegen. Trifft der Spieler einen Alien, erhält er Punkte, die je nach Art des Aliens variieren. Die Aliens können ebenfalls schießen. Wird der Spieler von einem Schuss eines Aliens getroffen, verliert er eines seiner Leben. Nachdem er alle Leben verloren hat, ist das Spiel zu Ende. Der Spieler kann Schutz hinter den unteren Wänden suchen, die Schüsse abfangen, jedoch langsam zerbröckeln. Mit jedem getöteten Alien erhöht sich das Bewegungstempo der noch lebenden Aliens, so wird der Schwierigkeitsgrad stetig erhöht. Dadurch ergaben sich folgende Eigenschaften für

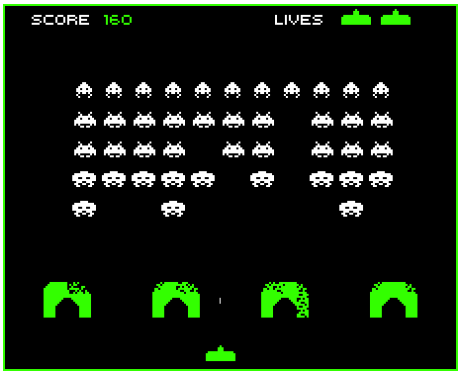


Abbildung 9: Der Arcade-Klassiker Space Invaders [3]

unser Spiel:

1. Der Spieler steuert einen Charakter, den er nur horizontal steuern kann
2. Es gibt Gegner, die sich vom oberen Bildschirmrand abwärts bewegen
3. Sowohl Spieler als auch Gegner können schießen
4. Der Spieler hat die Möglichkeit, Schutz hinter einer Wand zu suchen

3 „SpaceInvaders-Gameplay“ by Source. Licensed under Fair use via Wikipedia - https://en.wikipedia.org/wiki/File:SpaceInvaders-Gameplay.gif#/media/File:SpaceInvaders-Gameplay.gif

2.11.1.1 *Eigene Ideen*

Im Gegensatz zum herkömmlichen „Space Invaders“ erscheinen in „Revenge of the Bugs“ immer wieder neue Gegner, bis der Spieler schließlich keine Leben mehr hat.
Um dem Spieler mehr Möglichkeiten zu geben, hat er neben seiner gewöhnlichen Waffe, die herkömmliche Projektile abfeuert, noch seinen Schild, der mehrere Gegner auf einmal töten kann und an den Spielwänden abprallt, um so noch weitere Gegner zu töten.
In „Revenge of the Bugs“ gibt es zwei Arten von Gegnern:

1. Die Gewöhnlichen, die lediglich schießen.
2. Der Spezial-Gegner, der nur einmal pro Welle vorhanden ist. Solange dieser am Leben ist, ist der Spieler in einer anderen Realität. Das wird durch einen anderen Hintergrund und durch die Vertauschung der Steuerung hervorgehoben. Sobald dieser Gegner stirbt, lässt sich das Spiel wieder wie zuvor spielen.

In unserem Spiel erhöht sich das Tempo der Gegner mit jeder Welle, statt sich der Anzahl der lebenden Gegner anzupassen. Dadurch erreichen wir einen gleichmäßigen Anstieg der Schwierigkeit. Um dem Spieler eine Möglichkeit zu geben, die Schwierigkeit ein wenig zu reduzieren, erscheinen im Spiel Gegenstände, die beim Einsammeln das Tempo der Gegner für eine kurze Zeit reduzieren.

2.11.1.2 *Das Thema des Spiels*

Das Spiel ist die Fortsetzung zu unserem ersten Spiel Bugitar. Wir nutzen wie zuvor das Superhelden-Thema der Firma Site Point und wählten als Protagonist erneut Tommy Beckman, der gegen die Bugs antreten muss.

2.11.2 *Konzept: Revenge of the Bugs*

Im letzten Spiel, Revenge of the Bugs, tritt der Site Point-Held Tommy Beckmann erneut gegen die Käfer aus Bugitar an. Er selbst kann sich unten auf einer Horizontalen bewegen, während von oben Wellen von Käfern näher kommen. Tommy hat 2 Projektile, mit denen er die Käfer besiegen kann: ein normales und ein Spezial-Projektil, seinen Schild, welches mehrere Käfer auf einmal töten kann. Dieses steht ihm alle 20 Sekunden zur Verfügung. Schafft er es, eine komplette Welle zu beseitigen, erscheint eine neue Welle, die sich Tommy schneller nähert als die vorherige. Die Käfer selbst feuern auch in regelmäßigen Abständen Geschosse ab, welche Tommy, wenn er getroffen wird, eines seiner drei Leben abziehen. Schutz davor bieten mehrere kleine Wände zwischen dem Helden und den Käfern. Diese können jedoch nach und nach von den Projektilen der Käfer und

des Helden zerstört werden. Tommys Spezial-Projektil zerstört diese Wände nicht.

2.12 DIE SPIELE IM ENDLOS-KONZEPT

Normalerweise hat ein Spiel mehrere Level, die es zu bewältigen gilt. Nach Abschluss eines Levels kommt der nächste und der Schwierigkeitsgrad erhöht sich. So ist es zumindest bei den meisten Spielen.

Davon abgesehen, dass dieses Konzept bei unserem Zwei-Spieler-Spiel keinen Sinn machen würde, wollten wir aus mehreren Gründen auch bei den beiden anderen Spielen nicht diesem level-basierten Konzept folgen.

In der Bachelorarbeit wäre die Zeit gar nicht ausreichend gewesen, um verschiedene Level für jedes Spiel zu designen und zu implementieren, und die einzelnen Level nur durch den Schwierigkeitsgrad voneinander zu unterscheiden, erschien uns nicht richtig. Stattdessen entschieden wir uns dazu, den Schwierigkeitsgrad in den Spielen stetig ansteigen zu lassen.

Durch unser Konzept in Bugitar ließ sich diese Idee gut verwirklichen: Die Gegner-Käfer lassen Eier fallen, die Punkte bringen und aus welchen neue Käfer schlüpfen können, und manche Käfer sind unzerstörbar. So lässt sich das Spiel nicht durch Zerstören aller Käfer beenden und es gibt immer Eier, um Punkte zu sammeln.

Im Spiel The Revenge of the Bugs ist das Endlos-Konzept dadurch erreicht, dass stetig neue Käfer-Wellen erscheinen. Es entsteht durch die einzelnen Gegner-Wellen zwar der Eindruck, es handle sich um verschiedene Level, tatsächlich ist das Spiel jedoch endlos realisiert.

Da das Spiel Site Point Arena ein Zwei-Spieler-Spiel ist, welches nach Sieg eines Spielers beendet ist, war dort weder ein level-basiertes noch ein endloses Design sinnvoll.

3 IMPLEMENTIERUNG

3.1 ORGANISATION

Da eine gute Organisation Grundstein für das Gelingen eines Projektes ist, machten wir uns zu Beginn Gedanken darüber, wie eine gute Organisation stattfinden könnte, ohne dabei selbst zu viel unnötige Arbeit zu produzieren. Zum einen musste die Aufgabenverteilung organisiert werden, zum anderen das Programmieren.

3.1.1 *Organisation der Aufgabenverteilung*

Zur Unterstützung bei der Aufgabenverteilung verwendeten wir **Trello** [46], welches uns nicht nur einen guten Überblick über alle Aufgaben verschaffte, sondern aufgrund seiner Aufteilung in Listen auch ein einfaches Verwalten dieser Aufgaben ermöglichte. Wir entschieden uns für folgende Board-Struktur:

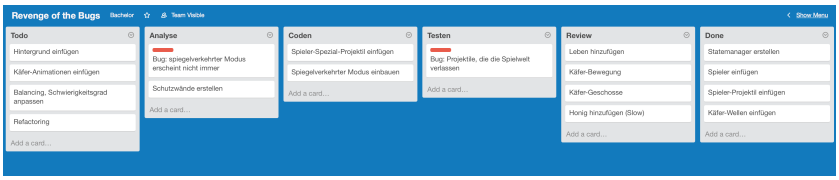


Abbildung 10: Das Board im dritten Spiel

3.1.1.1 *Todo*

Hier befinden sich Aufgaben, die zwar festgelegt, aber noch nicht ausreichend besprochen wurden.

3.1.1.2 *Analyse*

Die Aufgaben in dieser Spalte werden analysiert, um sie für die nächste Spalte „Coden“ vorzubereiten.

3.1.1.3 *Coden*

Dies ist die Implementierungsphase einer Aufgabe.

3.1.1.4 *Testen*

Falls möglich, werden die Aufgaben getestet. In unserem Fall bestanden diese Tests meistens aus Beta-Tests.

3.1.1.5 *Review*

Aufgaben, die sich in dieser Spalte befinden, wurden bereits implementiert und ggf. getestet. Nun müssen sie noch vom Auftraggeber abgenommen werden. Fehlt etwas oder ist etwas fehlerhaft, bleibt die Aufgabe in „Review“ und es wird eine neue Aufgabe dafür in „Analyse“ erstellt.

3.1.1.6 *Done*

Aufgaben in dieser Spalte sind komplett fertig.

3.1.1.7 *Sonstige Regeln*

Damit Komplikationen und Missverständnisse vermieden werden, darf keine Aufgabe in eine vorherige Spalte gezogen werden. Stattdessen wird, sollte eine solche Situation eintreten, eine neue Aufgabe in „Analyse“ erstellt, die das fehlende Feature oder den Bug repräsentiert.

Es ist möglich, eine Aufgabe einer bestimmten Kategorie zuzuordnen. Da wir jedoch nur zu zweit programmierten, hielten wir dies für einen zu großen Aufwand und beschränkten uns lediglich darauf, die Fehler-Aufgaben als solche mit einer Kategorie zu versehen.

3.1.2 *Organisation des Codes und Versionskontrolle*

3.1.2.1 *Git oder Subversion?*

Die Entscheidung zwischen Git und Subversion fiel relativ schnell: Da es das Arbeiten ohne Internetverbindung und die Erstellung von Branches ermöglicht, entscheiden wir uns für Git [16]. Zwar hatten wir beide durch unser Bachelor-Projekt mehr Erfahrung mit Subversion [44], allerdings wollten wir Git aufgrund der genannten Vorteile besser kennenlernen. Es zeigte sich schnell, dass dies die richtige Entscheidung war: die Verwendung von Branches erleichterte das Arbeiten im Team deutlich.

Für unser Repository entschieden wir uns für Bitbucket [4], da wir ein privates Repository bevorzugten. Github [18] erlaubt dies nur mit einem kostenpflichtigen Upgrade des Accounts.

3.1.2.2 *Verwaltung des Codes*

Für jedes Spiel legten wir ein eigenes Repository an, auf das beide vollen Zugriff hatten. Um das Arbeiten als Team zu erleichtern, arbeiteten wir viel mit Branches. Wir erstellten für fast jede Aufgabe einen eigenen Branch, um Konflikte bei einem Push zu vermeiden. Dies erwies sich als äußerst hilfreich, hatten wir doch in unserem Bachelorprojekt mit Subversion mehrmals die Erfahrung gemacht, dass bei fast jedem Commit schnell Konflikte auftreten.

3.2 EINGESETZTE TECHNOLOGIEN UND WERKZEUGE

3.2.1 *Entwickler-Werkzeuge*

3.2.1.1 *Editoren*

Zum Programmieren verwendeten wir simple Editoren wie Atom [3] und Brackets [6]. Um Syntaxfehler in JavaScript leichter zu finden, verwendeten wir teilweise auch Jetbrains' Entwicklungsumgebung Webstorm [49].

3.2.1.2 *Lokaler Webserver XAMPP*

Damit die Spiele im Webbrowser ausgeführt werden können, wird ein Webserver benötigt. Wir nutzten dazu den lokalen Webserver XAMPP [51] von Apache. Da wir sowohl auf Linux, OS X und Windows implementieren, benötigen wir einen lokalen Webserver, der auf diesen drei Betriebssystemen läuft.

3.2.2 *Der Spiele-Anteil*

3.2.2.1 *HTML 5*

HTML5 ist die Auszeichnungs-Sprache des Internets. Die erste Version wurde 1990 geschaffen, allerdings untersteht sie erst seit 1995 dem W3C. Im selben Jahr erschienen HTML 3.2 und HTML 4. Im folgenden Jahr beschloss das W3C, an Extensible Hypertext Markup Language (XHTML) weiterzuarbeiten, statt an HTML4.

Erst 2006 arbeitete der W3C zusammen mit Apple, Mozilla und Opera, welche zuvor schon als WHATWG (Web Hypertext Application Technology Working Group[50]) an eigenen Spezifikationen arbeiteten, an den Spezifikationen von HTML5. Da beide Gruppen jedoch unterschiedliche Interessen verfolgten, erstellte der W3C 2012 eine neue Editoren-Gruppe für HTML5. Seither fließen Patches mit Bugfixes von beiden Gruppen in den W3C-Standard ein [79].

Am 28. Oktober 2014 wurde HTML5 vom World Wide Web Consortium W3C [79] als Standard offiziell vorgelegt und wird seither von allen gängigen Browsern unterstützt.

Seit seiner Festlegung als Webstandard löst HTML5 Flash immer mehr ab, auch im Bereich der Bewegtbilder und Videos Rentz[73], in welchem der Flash Player lange Zeit der Standard war. Ein Grund dafür ist das neue Canvas-Element, das uns auch die Spieleentwicklung ermöglicht. Dieses erlaubt das Rendern von Graphen, Spiele-Grafiken, Kunst oder anderen Visuellen Objekten ohne große Vorbereitung [79]. Das heißt jedes unserer Spiele befindet sich komplett in einem solchen Canvas-Tag.

3.2.2.2 *JavaScript / ECMAScript*

JavaScript, eigentlich ECMAScript, war ursprünglich als Skript-Sprache gedacht, um andere bereits vorhandene Systeme zu manipulieren. Allerdings hat es sich mittlerweile zu einer interpretieren Programmiersprache entwickelt [65].

In unserer Arbeit verwendeten wir den ECMAScript 5 Standard [64], da er, vor allem im Hinblick auf die spätere Nutzung auf dem Raspberry Pi, besser unterstützt wird, als ECMAScript 6 (vgl. [12] und [13]).

Obwohl die Sprache funktional aufgebaut ist, besteht, abgesehen von primitiven Datentypen, alles darin aus einem Objekt. Ob man ECMAScript 5 objektorientiert nennen kann, ist ein umstrittenes Thema, generell sind die Kerneigenschaften Vererbung, Polymorphie und Datenkapselung damit aber möglich Crockford[62]. Anders sieht das bei ECMAScript 6 aus, welches aufgrund einiger Neuerungen, wie z.B. einer Klassen-Syntax, schon viel mehr als objektorientierte Sprache angesehen werden kann als sein Vorgänger.

3.2.2.3 *Phaser Engine*

In Kapitel 2.3 haben wir uns verschiedene JavaScript-Engines angeschaut und uns für die Phaser-Engine entschieden.

3.2.2.4 *Grundstruktur eines Spiels mit Phaser*

Um die Phaser-Engine verwenden zu können, muss die phaser.min.js in die html-Seite integriert werden:

Listing 1: Phaser in HTML einbinden

```
1  <html>
2    <head>
3      <title>Spieletitel</title>
4      <script type="text/javascript" src="js/phaser.min.js"></script>
5      ...
6    </head>
7    <body>
8      <div id="gameDiv"></div>
9    </body>
10 </html>
```

Die ID, welche man dem div-Element zuteilt, ist wichtig, um sie später bei der Erstellung des Spieles referenzieren zu können (siehe: Kapitel 3.3.1.1 S.40.

Die Grundstruktur eines einfachen Spiels sieht wie folgt aus:

Listing 2: Grundstruktur Phaser-Spiel

```
1  var game = new Phaser.Game(800, 600, Phaser.AUTO, '', { preload:
       preload, create: create, update: update });
2
3  function preload() {
4  }
5
6  function create() {
7  }
8
9  function update() {
10 }
```

Neues Spiel erstellen
In Zeile 1 wird ein neues Phaser-Spiel erstellt. Dazu wird zuerst die Größe des Fensters übergeben, 800 x 600 Pixel. Als nächstes wird angegeben, ob Canvas oder Web Graphics Library (WebGL) genutzt werden soll. In diesem Fall steht es auf Phaser.AUTO, d.h. es wird WebGL genutzt, sofern der Webbrowser WebGL unterstützt, ansonsten wird Canvas verwendet. Der vierte Parameter gibt das Document Object Model (DOM)-Element an, in dem das Spiel erstellt werden soll. Ein leerer String fügt es einfach in das Body-Element ein. Zuletzt werden die Grundfunktionen Preload, Create und Update registriert.

Preload
Die Preload-Funktion dient dazu, alle benötigten Assets wie Spritesheets, Images, Audio, etc. zu laden.
Create
In der Create-Funktion werden die eigentlichen Spielobjekte erzeugt. Hierzu gehören die Spielwelt, der Spieler, die Gegner und viele weitere Komponenten.
Update
In der Update-Funktion spielt sich das eigentliche Spiel ab. Sie wird in einer Endlosschleife aufgerufen und reagiert auf Spielereignisse und passt die Spielwelt dementsprechend an.
Statemanager
Für simple Spiele ist dieser Aufbau ausreichend, jedoch besitzen die meisten Spiele ein Menü, eine oder mehrere Spielszenen und einen Gewonnen bzw. Verloren Zustand. Diese Zustände können mit Hilfe des Phaser-Statemanagers modelliert werden (Siehe Kapitel 3.3.1).

3.2.3 *Schnittstellen*

3.2.3.1 *Node.js*

Node.js erschien 2009 und basiert auf Chromes JavaScript-Laufzeitumgebung V8 und ermöglicht den serverseitigen Einsatz von JavaScript. Dadurch ergibt sich für Entwickler die Möglichkeit, Web-Anwendungen in einer Sprache zu schreiben. Node.js nutzt einen eventbasierten Ansatz, wodurch es sehr ressourcenschonend ist und besonders für verteilte Echtzeitanwendungen geeignet ist Mike Cantelon [70].

3.2.3.2 *REST*

REST Bojinov [60] ist ein Konzept für verteilte Systeme. Um eine Webanwendung RESTful zu gestalten, müssen folgen Prinzipien eingehalten werden:

- Alles ist eine Ressource und kann mit Hilfe einer Uniform Resource Identifier (URI) identifiziert werden.
- REST verwendet die Hypertext Transfer Protocol (HTTP)-Verben POST, GET, PUT und DELETE
- Ressourcen können mehrere Darstellungen haben
- Die Kommunikation ist zustandslos.

3.3 ALLGEMEINE ARCHITEKTUR UNSERER SPIELE

3.3.1 *Statemanager*

Wir haben bereits die Grundstruktur eines Spiels mit der Phaser-Engine in Kapitel 3.2.2.4 erklärt, allerdings eignet sich diese Struktur nur für kleine Spiele. Um das Projekt übersichtlicher zu gestalten und den Aufbau besser zu struktieren, verfügt die Phaser-Engine über einen Statemanager, der das Wechseln zwischen den verschiedenen States ermöglicht. Ein State kann z.B. das Startmenü oder das Spiel selbst sein. Innerhalb eines States stehen dem Entwickler mehrere Methoden zur Verfügung, mit denen in den Lebenszyklus eingegriffen werden kann. Beispiel:

- create()
- pause()
- update()

Weitere Methoden können in der Phaser-Dokumentation gefunden werden [32].

Für unsere Spiele haben wir die folgenden States erstellt:

- Game
- Boot
- Load
- Menu
- Play
- Gameover

Dabei wurde jeder State in eine eigene Datei ausgelagert.

3.3.1.1 *Der Game-State*

Der Game-State ist kein richtiger State, da er dem Statemanager nicht hinzugefügt wird. Der Game-State erzeugt das Game-Objekt, fügt diesem die anderen States hinzu und startet anschließend den Boot-

State.

Listing 3: Game-State

```
global.game = new Phaser.Game(640, 480, Phaser.AUTO, 'gameDiv');

global.game.state.add('boot', bootState);
global.game.state.add('load', loadState);
global.game.state.add('menu', menuState);
global.game.state.add('play', playState);
global.game.state.add('gameover', gameoverState);

global.game.state.start('boot');
```

3.3.1.2 *Der Boot-State*

Im Boot-State wird der Ladebalken für den Load-State initialisiert und die Physik für das Spiel gesetzt.

Listing 4: Boot-State

```
var bootState = {
  preload: function(){
    global.game.load.image('loadingbar', 'assets/bar.png');
  },
  create: function(){
    //  Set physics to ARCADE
    global.game.physics.startSystem(Phaser.Physics.ARCADE);

    global.game.state.start('load');
  }
}
```

3.3.1.3 *Der Load-State*

Im Load-State werden alle Sounds und Grafiken geladen.

3.3.1.4 *Der Menu-State*

Im Menu-State wird das Startmenü des jeweiligen Spiels aufgebaut.

3.3.1.5 *Der Play-State*

Der Play-State ist der wichtigste Zustand, denn hier läuft das eigentliche Spiel ab.

Listing 5: Play-State

```
1  var playState = {
2    /*
3      creates all important objects to start the game
4    */
5    create: function(){
6
7    },
8    /*
9      The update function is called automatically and handles the game
           flow.
10   */
11   update: function(){
12
13   },
14   gameover: function(){
15     global.game.state.start('gameover');
16   }
17 }
```

Die Create-Methode erzeugt alle Spielelemente, sodass das Spiel gespielt werden kann. In der Update-Methode spielt sich das eigentliche Spiel ab. Hier wird das Spiel ständig aktualisiert und passt sich den Spieleraktionen an. Sobald das Spiel zu Ende ist, wird der Gameover-State durch die Gameover-Methode aufgerufen.

3.3.1.6 *Der Gameover-State*

Im Gameover-State hat der Spieler die Möglichkeit, sich in die Highscore-Tabelle einzutragen, vorausgesetzt, er hat die nötige Punktzahl erreicht. Außerdem kann er das Spiel neu starten, dazu wird wieder der Menu-State aufgerufen.

3.3.2 *Grafiken, Animationen, Sounds*

3.3.2.1 *Grafiken*

Eine Grafik muss zunächst in Phaser geladen werden:

Listing 6: Laden einer Grafik

```
1 game.load.image('shield', 'assets/bild.png');
```

Der erste Parameter ist der Name, unter dem die Grafik im Spiel referenziert werden kann, der zweite ist der Pfad zum Bild.

Ein solches Bild kann nun als sog. *Sprite* benutzt werden:

Listing 7: Verwenden einer Grafik als Sprite

```
1 var mySprite = game.add.sprite(x, y, 'shield');
```

Dieses Sprite wird nun mit dem Bild *shield* an den Koordinaten x und y des Spielfeldes geladen. Als Sprite hat es die Möglichkeit, mit einer Physik ausgestattet zu werden, um so beispielsweise auf Kollisionen zu reagieren.

Listing 8: Hinzufügen einer Physik zu einem Sprite

```
1 game.physics.arcade.enable(mySprite);
```

3.3.2.2 *Animationen*

Bei einer Animation werden gewöhnlich verschiedene Einzelbilder schnell genug nacheinander abgespielt, um die Illusion von bewegten Bildern zu erschaffen. So werden Animationen auch in Phaser realisiert.

Man benötigt ein sog. *Spritesheet*, eine Portable Network Graphics (PNG)-Bilddatei, die alle Einzelbilder der gewünschten Animation enthält. Dabei ist es wichtig, dass jedes Einzelbild dieselbe Größe hat, da Phaser zum Abspielen der Bilder das Spritesheet in gleichgroße Stücke aufteilt.

Abbildung 11: Ein Ausschnitt des Spritesheets vom Helden aus Bugitar

Zunächst muss ein solches Spritesheet in das Spiel geladen werden, was auf ähnliche Weise wie das Laden eines einfachen Bildes geschieht:

Listing 9: Laden eines Spritesheets

```
1 game.load.spritesheet('hero', 'assets/hero.png', 32, 32);
```

Wichtig ist dabei, dass zusätzlich zu Name und Bild-Pfad die Höhe und Breite eines **Einzel**bildes übergeben wird.

Jetzt kann es wie ein normales Bild als Sprite einer Variablen zugeordnet werden:

Listing 10: Verwenden eines Spritesheets als Sprite

```
1 var player = game.add.sprite(x,y, 'hero');
```

Allerdings enthält dieses Sprite alle Einzelbilder des Spritesheets. Um eine Animation hinzuzufügen, genügt folgender Aufruf:

Listing 11: Hinzufügen einer Animation

```
1 player.animations.add('left', [6, 7, 8], 10, true);
```

Da dies die Animation für das Laufen nach links wird, erhält sie als erstes Argument den Namen *left*. Das nächste Argument ist ein Feld, das die Einzelbilder, die nacheinander abgespielt werden sollen, enthält. Mit der darauffolgenden Zahl kann man die Frequenz, in der die Bilder (pro Sekunde) abgespielt werden, festlegen, und mit dem letzten Parameter gibt man an, ob die Animation in einer Schleife weiterlaufen soll. Übergibt man *false*, stoppt die Animation beim letzten Einzelbild.

Man kann einem Sprite mehrere Animationen hinzufügen:

Listing 12: Hinzufügen mehrerer Animationen für ein Sprite

```
1 player.animations.add('down', [0,1,2], 10 , true);
2 player.animations.add('up', [3,4,5], 10, true);
3 player.animations.add('left', [6, 7, 8], 10, true);
4 player.animations.add('right', [9, 10, 11], 10, true);
```

Eine Animation kann jetzt über folgende Zeile abgespielt werden:

Listing 13: Abspielen einer Animation

```
1 player.animations.play('up');
```

Nun werden in einer Schleife die drei Einzelbilder 3, 4 und 5 mit einer Frequenz von 10 Bildern pro Sekunde abgespielt.

Eine Animation wird über folgenden Aufruf gestoppt:

Listing 14: Stoppen einer Animation

```
1 sprite.animations.stop();
```

Es ist zusätzlich möglich, explizit ein bestimmtes Einzelbild einzustellen:

Listing 15: Einstellen eines beliebigen Einzelbildes eines Spritesheets

```
1 this.sprite.frame = 1;
```

3.3.2.3 *Sounds*

Phaser erlaubt eine einfache Nutzung von Sounds. Ähnlich wie ein Bild wird eine Audio-Datei folgendermaßen geladen:

Listing 16: Laden einer Audio-Datei

```
1 game.load.audio('collectItem', 'assets/sounds/collectItem.ogg');
```

Abgespielt wird er dann mit diesem Aufruf:

Listing 17: Abspielen einer Audio-Datei

```
1 game.sound.play('collectItem');
```

Phaser bietet noch mehr Funktionen zum Handhaben von Sounds, diese findet man in der offiziellen Dokumentation [56]. Man sollte darauf achten, ein Dateiformat zu wählen, das vom Ziel-Browser unterstützt wird. Auf W3C Schools findet sich eine Tabelle, die die Browser-Unterstützung für die drei Formate MPEG-1/MPEG-2 Audio Layer III (MP3), WAVE-Dateiformat (WAV) und OGG Vorbis Format (OGG) auflistet[76].

Für unsere Spiele verwendeten wir das OGG-Format, da es ein freies Format ohne Patent ist, welches auf unserer Zielplattform, dem NW.js, lauffähig ist.

3.3.3 *Tilemaps*

3.3.3.1 *Was sind Tiles?*

Kurz gesagt handelt es sich bei Tiles, zu Deutsch Kacheln oder auch Spielsteine, um kleine Bilder, aus denen man größere zusammensetzen kann. Dies ist insbesondere dann nützlich, wenn man eine Spielkarte zusammensetzen möchte, ohne sehr große Bilder verwenden zu müssen.

Bei den Kacheln handelt es sich meist um quadratische Texturen von ca. 32x32 Pixeln. Aber auch andere Polygone und Größen sind zur Verwendung von Tiles geeignet.

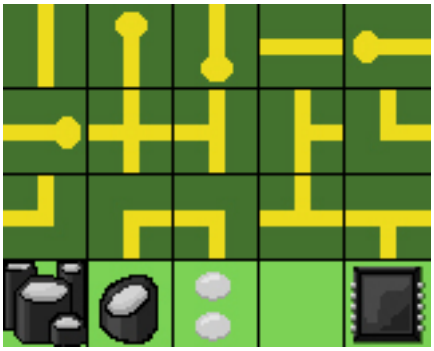


Abbildung 12: Das Tile-Muster von Bugitar mit Tiles von 32x32px

3.3.3.2 *Vorteile*

Das Erstellen unserer Spielkarten mit Hilfe von Kacheln war insbesondere deshalb ansprechend, da durch die Wiederholung von kleinen Texturen der Retro-Look gefördert wird. Allerdings besteht ein weiterer Vorteil in der Nutzung von Tiles: die Möglichkeit, eine Karte aus Tiles, eine Tilemap, als JavaScript Object Notation (JSON)-Dokument zu speichern.

In einem solchen JSON-Dokument werden Informationen wie die Größe der einzelnen Kacheln oder die Anzahl der Kacheln pro Zeile und Spalte gespeichert. Außerdem wird der Name der Bilddatei, welche alle Kacheln enthält, gespeichert.

Listing 18: JSON-Dokument für eine Tilemap

```
1  { "height":15,
2   "layers":[
3     {
4     "data":[20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
5       ...
6       20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20],
7      "height": 15,
8      "name": "Tile Layer 1",
9      "opacity": 1,
10     "type": "tilelayer",
11     "visible": true,
12     "width": 20,
13     "x": 0,
14     "y": 0
15    }
16   ],
17    "nextobjectid": 1,
18    "orientation": "orthogonal",
19    "properties": {
20    },
21    "renderorder": "right-down",
22    "tileheight": 32,
23    "tilesets": [
24      {
25        "firstgid": 1,
26        "image": "tiles.png",
27        "imageheight": 128,
28        "imagewidth": 160,
29        "margin": 0,
30        "name": "tiles",
31        "properties": {
32        },
33        "spacing": 0,
34        "tileheight": 32,
35        "tilewidth": 32
36      }
37    ],
38    "tilewidth": 32,
39    "version": 1,
40    "width": 20
41  }
```

Dies ist eine JSON-Datei, die die Spielkarte von Bugitar beschreibt. *Height* (Z. 1) und *width* (Z. 40) geben die Anzahl der Tiles in Höhe und Breite an. Das hinter *data* angegebene Feld beschreibt dabei, welche Kacheln welche Textur aus dem in *image* (Z. 26) angegebenen Kachelmuster (Abbildung 12) erhalten. Die Zahl 1 ist dabei die erste Kachel, 2 die zweite, usw. Damit die Kacheln aus dem Bild richtig herausgeschnitten werden, wird in den beiden Attributen *tileheight* und *tilewidth* (Z. 34, 35) die Größe einer Kachel in Pixeln angegeben.

Es ist möglich, einer Karte mehrere Ebenen zuzuweisen. Dazu kann man dem Array in *layers* eine weitere Ebene definieren. Wir verwendeten jedoch nur eine Ebene mit dem Namen *Tile Layer 1* (Z. 8).

Diese JSON-Datei wurde von dem Editor *Tiled* (s. unten) erstellt und enthält somit weitere Attribute, die bei Bedarf angepasst werden können. Damit Phaser ohne unerwartete Probleme mit dieser Karte arbeiten kann, sollte man keine Attribute löschen.

3.3.3.3 *Der Tiled Editor*

Um ein solches JSON-Dokument zu erstellen, eignet sich der Editor **Tiled** [24], der kostenlos verwendet werden darf. Mit ihm kann aus einem Kachelmuster eine Karte erstellt und als JSON-Datei exportiert werden.

Dadurch, dass das JSON-Dokument getrennt von der Kachel-Grafik gespeichert wird, lässt sich das eine ändern, ohne das andere zu beeinflussen. So lässt sich die Textur einer Kachel oder sogar die Struktur der Karte nachträglich ändern.

3.3.3.4 *Tiles und Tilemaps in Phaser*

Es müssen sowohl die JSON-Karte als auch das Kachelmuster in Phaser geladen werden. Beim Laden einer *Tilemap* muss der Name, der Dateipfad (oder optional ein JSON-Objekt oder String als dritter Parameter, bei uns null) und die Art der Tilemap übergeben werden. Da wir die Karte aus einer JSON-Datei generieren möchten, übergeben wir dazu Phaser.Tilemap.TILED_JSON. Ein Kachelmuster wird in Phaser wie ein normales Bild geladen.

```
1 game.load.tilemap('tileMap', 'assets/map.json', null,
      Phaser.Tilemap.TILED_JSON);
2 game.load.image('tiles', 'assets/tiles.png');
```

Um die Karte in Phaser zu verwenden, muss man dem Spiel eine *Tilemap* hinzufügen und diesem dann das Kachelmuster zuweisen. Die Ebene aus der JSON-Datei, die die Daten der Karte erhält, muss dann über dessen Name erstellt werden. Um die Ebene später noch benutzen zu können, beispielsweise um eine Kollision mit ihr zu integrieren, weisen wir die Ebene gleichzeitig einer eigenen Variablen zu.

```
1 var map = game.add.tilemap('tileMap');
2 map.addTilesetImage('tiles', 'tiles');
3 var layer = map.createLayer('Tile Layer 1');
```

Wenn man möchte, kann man die Größe des Spielfelds seiner Ebene anpassen:

```
1 layer.resizeWorld();
```

3.3.4 *Kollision*

In unseren Spielen haben wir zwei Arten von Kollisionen. Einmal die tatsächliche Kollision (Collision) und einmal die Überlappung (Overlap). Beides kann nur bei Sprites und Groups verwendet werden, denen eine Physik zugefügt wurde. Der Hauptunterschied zwischen der Kollision und Überlappung ist, dass die Kollision die Objekte vor einer Überlappung abhält.

3.3.4.1 *Collision*

Eine einfache Kollision zwischen zwei Sprites kann so erreicht werden:

```
1 function: update(){
2   game.physics.arcade.collide(playerSprite, enemySprite);
3 }
```

Hier wird auf eine einfache Kollision zwischen *playerSprite* und *enemySprite* geprüft. Das hat zur Folge, dass diese zwei Sprites sich nicht überlappen können. Findet eine Kollision statt, liefert die Methode als Rückgabewert **true**. Im Update-Zyklus des Spiels muss in jedem Durchlauf auf eine Kollision geprüft werden.

Es können noch weitere Parameter übergeben werden:

Listing 19: Collide-Konstruktor

```
1 game.physics.arcade.collide(playerSprite, enemySprite,
      collideCallback, processCallback, callbackContext);
```

Mit *collideCallback* wird eine Methode übergeben, die bei der Kollision aufgerufen wird, dadurch können weitere Aktionen ausgeführt werden. Mit *processCallback* kann eine weitere Funktion übergeben werden, die vor dem Aufruf von *collideCallback* aufgerufen wird und *collideCallback* nur aufruft, wenn *processCallback* **true** als Rückgabewert liefert. So können Bedingungen geprüft werden, die erfüllt sein müssen, damit *collideCallback* aufgerufen wird. Der *callbackContext* definiert den Kontext, in dem die Callback-Methoden ausgeführt werden.

3.3.4.2 *Overlap*

Eine einfache Überlappung zwischen zwei Sprites kann so erreicht werden:

Listing 20: Overlap-Konstruktor

```
1 game.physics.arcade.overlap(playerSprite, enemySprite,
      collideCallback, processCallback, callbackContext);
```

3.3.5 *Highscore*

Die Phaser-Engine bietet keine eigene Highscore-Verwaltung an, deshalb mussten wir uns dafür etwas Eigenes überlegen. Wir haben unsere Highscore in drei Komponenten unterteilt:

- Letter
- TextInput
- Highscore

Abbildung 13: Eingabe für den Highscore

3.3.5.1 *Letter-Komponente*

Die Letter-Komponente repräsentiert in der Abbildung 13 eine Stelle. Der Nutzer kann mit den Pfeiltasten den Wert einer Stelle ändern. Die Komponente besteht aus einem Textfeld, das den Buchstaben anzeigt.

Um den Wert einer Letter-Komponente zu ändern, gibt es die Methoden *incrementIndex* und *decrementIndex*. Diese werden in der TextInput-Komponente aufgerufen, wenn der Spieler die obere bzw. untere Pfeiltaste drückt.

Listing 21: Letter Index-Methoden

```
1   incrementIndex: function(){
2     this.charIndex++;
3     if(this.charIndex > this.characters.length - 1){
4       this.charIndex = this.characters.length - 1;
5     }
6     this.textField.text = this.characters[this.charIndex];
7   },
8   decrementIndex: function(){
9     this.charIndex--;
10    if(this.charIndex < 0){
11      this.charIndex = 0;
12    }
13    this.textField.text = this.characters[this.charIndex];
14  },
```

Damit der Spieler erkennt, dass die aktuelle Stelle markiert ist, gibt es noch eine Methode um eine Stelle hervorzuheben. Das wird durch Änderung der Schriftfarbe erreicht.

Listing 22: Letter Highlight-Methode

```
highlight: function(){
  this.textField.fill = '#00ff00';
},
```

3.3.5.2 *Die TextInput-Komponente*

Die TextInput-Komponente enthält mehrere Letter-Komponenten und repräsentiert in der Abbildung 13 die gesamte Eingabe. Die Anzahl der Stellen kann durch einen Parameter festgelegt werden. Der Spieler kann mit Hilfe der Pfeiltasten durch das Feld navigieren bzw. die Buchstabenwerte ändern.

3.3.5.3 *Die Highscore-Komponente*

Die Highscore-Komponente enthält die TextInput-Komponente und kümmert sich um die Verbindung zur Datenbank. Wenn der Spieler seinen Namen eingegeben hat und ihn abspeichert, werden die Daten zu unserem Datenbankserver geschickt:

Listing 23: Letter SavePoints-Methode

```
savePoints: function(callback){
  var playerScore = new Object();
  playerScore.user = this.userInput.getWord();
  playerScore.points  = global.score;
  var jsonString= JSON.stringify(playerScore);
  this.postRequest(this.url, jsonString, callback, this.x, this.y,
       "Highscore");
  this.postRequest(this.urlHof, jsonString, callback, this.x +
      200, this.y, "Hall of Fame");
},
```

Dazu wird zunächst ein JSON-String erzeugt, der den Namen und die Punktzahl enthält. Dann wird jeweils ein POST-Request an die Highscore- und Hall-of-Fame-Datenbank geschickt. Das Senden der Daten erfolgt asynchron mit Hilfe von jQuery.

Listing 24: Letter PostRequest-Methode

```
1   postRequest: function(url, jsonString, callback, tablex, tabley,
        title){
2     $.ajax({
3          url: url,
4          type: "POST",
5          data: jsonString,
6          dataType: "json",
7          contentType: "application/json"
8     }).done(function(res) {
9         callback.fetchData(this.url, callback, false, tablex, tabley
              , title);
10    }).fail(function(res) {
11        console.log("Invalid Response!");
12    });
13  }
```

Wenn der HTTP-Status in Ordnung ist, werden anschließend die Tabellen konstruiert und angezeigt.

3.4 BUGITAR

3.4.1 *Die Bugs / Käfer*

Die Datei *Bug.js* beinhaltet alle Funktionen, die von den Käfern benötigt werden. Die Hauptaufgaben der Käfer sind:

- Selbstständig durch das Labyrinth laufen
- Eier legen
- Form verändern

3.4.1.1 *KI der Käfer*

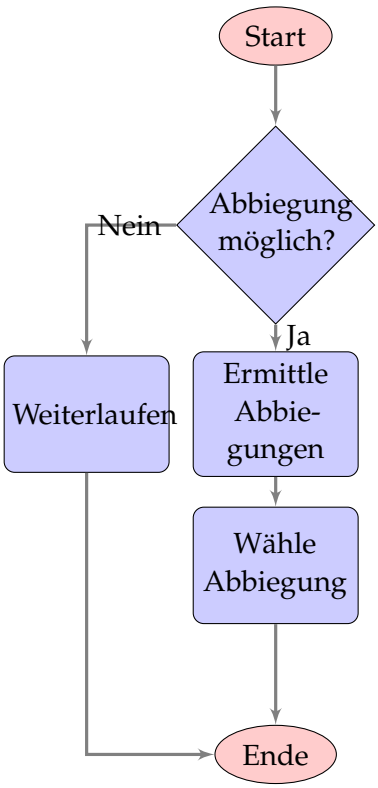
AUSGANGSSITUATION

Die Käfer sollen sich selbstständig durch das Labyrinth bewegen können. Dabei müssen folgende Punkte beachtet werden:

- Käfer dürfen nur auf den Wegen laufen
- Käfer dürfen nicht durch Wände laufen
- Käfer müssen Abbiegungen erkennen
- Käfer müssen Entscheidungen über das Abbiegen treffen

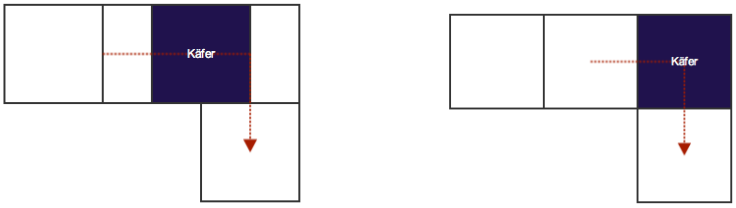
DER ALGORITHMUS

Der Abbiegeprozess wird mit Hilfe des dargestellten Algorithmus gelöst:



Abbiegevoraussetzung

Damit eine Abbiegung überhaupt durchgeführt werden kann, muss sich der Käfer zuerst einmal in der richtigen Position befinden.



(a) Abbiegung nicht möglich (b) Abbiegung möglich

Abbildung 14: Abbiegung eines Käfers

In der Update-Funktion wird die aktuelle Position der Käfer jedes Mal aktualisiert. Damit die Abbiegung möglich ist, muss sich der Käfer exakt auf dem aktuellen Tile befinden. Das wird in den seltensten Fällen der Fall sein, deswegen muss eine bestimmte Toleranz eingebaut werden. Die Phaser-Engine bietet eine Funktion, die überprüft, ob zwei Werte ungefähr gleich sind:

```
1 var equal = Phaser.Math.fuzzyEqual(a, b, toleranz);
```

Die fertige Überprüfung, ob ein Käfer abbiegen kann, sieht folgendermaßen aus:

Listing 25: Abbiegevoraussetzung

```
1  isInTurnPosition: function(){
2    //position to check, if the bug is exactly on a tile
3    var fx = Math.ceil(this.sprite.x) % global.GRIDSIZE;
4    var fy = Math.ceil(this.sprite.y) % global.GRIDSIZE;
5    var equalx = Phaser.Math.fuzzyEqual(fx, 0, this.threshold);
6    var equaly = Phaser.Math.fuzzyEqual(fy, 0, this.threshold);
7    this.tilex = Phaser.Math.snapToFloor(Math.ceil(this.sprite.x),
         global.GRIDSIZE) / global.GRIDSIZE;
8    this.tiley = Phaser.Math.snapToFloor(Math.ceil(this.sprite.y),
         global.GRIDSIZE) / global.GRIDSIZE;
9    return equalx && equaly;
10 },
```

Zeile 3 und 4 dienen dazu, die Position auf einem Tile zu ermitteln. Dazu werden folgende Informationen genutzt:

- ein Tile ist 32 x 32 Pixel groß
- das Spiel besteht aus 20 x 15 Tiles

Liefert die Modulo-Operation das Ergebnis 0 zurück, so befindet sich der Käfer exakt auf einem Tile. Da eine exakte Positionierung nur selten erreicht wird, prüfen die Zeilen 5 und 6, ob das Ergebnis ungefähr 0 ist. In den Zeilen 7 und 8 wird die Tile-Position des Käfers aktualisiert. Ist die horizontale (equalx) und vertikale (equaly) Position ungefähr 0, wird zurückgegeben, dass der Käfer in der richtigen Position ist, ansonsten nicht.

Abbiegungen ermitteln

Ein Käfer darf sich nur nach links, rechts, oben oder unten bewegen. Deswegen wird bei den nachfolgenden Tiles geprüft, ob es sich um einen Weg oder eine Wand handelt:

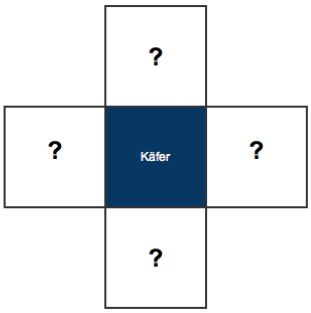


Abbildung 15: Nachbarfelder prüfen

Die Implementierung sieht folgendermaßen aus:

Listing 26: Mögliche Abbiegungen

```
1   checkPossibleDirections: function(){
2     //get surrounding tiles
3     var tileLeft = global.map.getTileLeft(global.layer.index,
          this.tilex, this.tiley).index;
4     var tileRight = global.map.getTileRight(global.layer.index,
          this.tilex, this.tiley).index;
5     var tileAbove = global.map.getTileAbove(global.layer.index,
          this.tilex, this.tiley).index;
6     var tileBelow = global.map.getTileBelow(global.layer.index,
          this.tilex, this.tiley).index;
7     //set possible directions with 0 or 1
8     this.possibleDirections[Phaser.LEFT] = util.isSafeTile(tileLeft)
          ;
9     this.possibleDirections[Phaser.RIGHT] = util.isSafeTile(
          tileRight);
10    this.possibleDirections[Phaser.UP] = util.isSafeTile(tileAbove);
11    this.possibleDirections[Phaser.DOWN] = util.isSafeTile(tileBelow
          );
12    //sets the direction, the bug comes from, to false. so it is not
           possible to move left-right-left-right-left-right...
13    this.possibleDirections[global.oppositeDirs[this.direction]] =
          0;
14  },
```

In den Zeilen 3-6 werden zuerst die zu prüfenden Tiles geholt. Anschließend werden in den Zeilen 8-12 die möglichen Abbiegerichtungen in einem Array als Boolean-Wert gespeichert. In Zeile 13 wird die Richtung, aus der der Käfer kommt auf false gesetzt. Ansonsten wäre es möglich, dass der Käfer ständig zwischen zwei Tiles hin und her läuft.

Abbiegungen wählen

Nachdem die möglichen Abbiegerichtungen ermittelt wurden, muss jetzt eine gültige Richtung ausgewählt werden.

Listing 27: Abbiegung wählen

```
1   takeTurn: function(){
2     var directionArray = new Array();
3     for(var i = 1; i <=4; i++){
4       if(this.possibleDirections[i]){
5         directionArray.push(i);
6       }
7     }
8     this.moveTo(this.chooseRandom(directionArray));
9
10  },
```

Dazu wird in Zeile 2 ein Array erzeugt, das die gültigen Richtungen speichert. In den Zeilen 3-7 werden die gültigen Positionen aus „possibleDirections“ gelesen und in „directionArray“ gespeichert. Das ermöglicht es, zufällig eine Richtung auszuwählen, ohne dabei die Gültigkeit der Richtung zu prüfen. In Zeile 8 wird die neue Richtung mit Hilfe der chooseRandom-Funktion ausgewählt und der moveTo-Funktion übergeben.

Listing 28: Random-Methode

```
1   chooseRandom: function(array){
2     return array[Math.floor(Math.random() * (array.length))];
3   },
```

Die chooseRandom-Funktion erzeugt einen zufälligen Array-Index und gibt den Wert an dieser Position zurück.

Listing 29: Laufrichtung ändern

```
1   moveTo: function(newDirection){
2     switch(newDirection){
3       case Phaser.UP:
4       this.direction = Phaser.UP;
5       this.setVelocity(0, -this.speed);
6       this.adjustAnimations('up', 'up2', 'up3');
7       break;
8       ...
9   },
```

In der moveTo-Funktion wird die Laufrichtung des Käfers schließlich je nach Richtung angepasst. Dabei wird die Laufrichtung, die Laufgeschwindigkeit und die Laufanimationen aktualisiert.

Kompletter Abbiegeprozess

Die einzelnen Schritte des Algorithmus werden in der folgenden Funktion zusammengefasst:

Listing 30: Kompletter Abbiegeprozess

```
1   checkTurnings: function(){
2     this.adjustStage();
3     if(this.isInTurnPosition() && this.isTileNew()){
4       this.lastTilex = this.tilex;
5       this.lastTiley = this.tiley;
6       var currentTileIndex = global.map.getTile(this.tilex,
            this.tiley).index;
7       this.setVelocity(0,0);
8       //place the "bug" exactly on the tile, so it is able to take
            turns
9       this.sprite.x = this.tilex * global.GRIDSIZE;
10      this.sprite.y = this.tiley * global.GRIDSIZE;
11      this.sprite.body.reset(this.sprite.x, this.sprite.y);
12
13      this.checkEggPlacing();
14      this.checkPossibleDirections();
15
16      if(this.isDeadEnd()){
17        this.moveTo(global.oppositeDirs[this.direction]);
18      }else{
19        this.takeTurn();
20      }
21    }
22  },
```

In Zeile 3 wird zunächst geprüft, ob der Käfer die Abbiegevoraussetzungen (Unterunterabschnitt 3.4.1.1) erfüllt:

- Befindet sich der Käfer auf einer gültigen Position?
- Wurde das Tile schon geprüft?

Der zweite Punkt ist besonders wichtig, da sonst eine Endlosschleife entsteht, in der ständig die Abbiegerichtungen ermittelt werden, ohne jemals abzubiegen.
In Zeile 7 wird die Laufgeschwindigkeit des Käfers auf 0 gesetzt, sodass der Käfer in den Zeilen 9-11 auf die exakte Position platziert werden kann, ohne dabei wieder zu einer falschen Position zu laufen.
In Zeile 14 werden die möglichen Abbiegerichtungen (3.4.1.1) ermittelt.
In Zeile 16 wird geprüft, ob der Käfer sich in einer Sackgasse befindet. Falls das der Fall ist, rennt er in die entgegengesetzte Richtung, ansonsten wird eine neue Richtung ausgewählt (**??**).

3.4.1.2 *Platzieren der Eier*

Sobald sich ein Käfer auf einem neuen Tile befindet, auf dem noch kein Ei liegt, soll er ein Ei auf diese Stelle platzieren. Diese Aufgabe übernimmt die *checkEggPlacing*-Methode, die nur aufgerufen wird, wenn der Käfer sich auf einem neuen Tile befindet.

Listing 31: Prüfung Ei-Platzierung

```
1   checkEggPlacing: function(){
2     if(global.eggArray[this.tilex][this.tiley] == null){
3       ++this.currentEgg;
4       if (this.currentEgg != global.BUG_BIG_EGG) {
5         global.eggArray[this.tilex][this.tiley] = new Egg(this.tilex
              , this.tiley, false);
6       } else {
7         global.eggArray[this.tilex][this.tiley] = new Egg(this.tilex
              , this.tiley, true);
8         this.currentEgg = 0;
9       }
10    }
11  },
```

Die Eier werden in einem zweidimensionalen Array gespeichert, das das Spielfeld repräsentiert. Zunächst wird geprüft, ob auf der aktuellen Position bereits ein Ei liegt. Ist dies nicht der Fall, wird in Zeile 4 geprüft, ob ein kleines oder großes Ei platziert werden soll. Der Unterschied zwischen kleinen und großen Eiern ist, dass aus den großen Eiern neue Käfer schlüpfen können.

3.4.1.3 *Verwandlung der Käfer*

Die Käfer sollen sich im Laufe des Spiels zu stärkeren Käfern entwickeln können. Erreicht ein Käfer seine letzte Form, kann er vom Spieler nicht mehr getötet werden. Das Entwickeln geschieht über das Lauftempo, das stetig erhöht wird. Übersteigt es eine gewisse Grenze, erreicht der Käfer die nächste Stufe.

Listing 32: Käferstufe anpassen

```
1   adjustStage: function() {
2     // check speed and adjust stage and new speed accordingly
3     if (this.speed >= global.BUG_SPEED_END_STAGE_2 && this.stage ==
          2) {
4       this.stage = 3;
5       this.speed = global.BUG_SPEED_STAGE_3;
6     } else if (this.speed >= global.BUG_SPEED_END_STAGE_1 &&
          this.stage == 1) {
7       this.stage = 2;
8       this.speed = global.BUG_SPEED_STAGE_2;
9     }
10    // increase speed steadily
11    if (this.speed <= global.BUG_SPEED_MAX && !this.isSlowed) {
12      this.speed += 0.005;
13    }
14  },
```

3.4.2 *Der Spieler*

Für den Spieler erstellten wir eine eigene Datei *Player.js*, die die komplette Funktionalität beinhaltet. Der Konstruktor erwartet zwei Argumente x und y, welche die Position des erstellten Spieler-Sprites bestimmen.

Listing 33: Player-Konstruktor Bugitar

```
function Player(x, y) {
  // Phaser-Objekt das Spritesheet und Physik zur Interaktion mit
      anderen Elementen erhält
  this.sprite = global.game.add.sprite(x,y, 'hero');
  global.game.physics.arcade.enable(this.sprite);
  this.PLAYER_IS_DEAD = false;
  // Anzahl noch verfügbarer Sprays (Leben)
  this.sprayCount = 3;
  // Bewegungsgeschwindigkeit des Spielers
  this.speed = global.PLAYER_SPEED;
  // Spieler Position in Tiles gemessen (beginnend bei 0)
  this.tilex = -1;
  this.tiley = -1;
  // Der Punkt auf einem Tile, auf dem es dem Spieler möglich ist,
      abzubiegen (die Mitte eines Tiles)
  this.turnPointx = -1;
  this.turnPointy = -1;
  // threshold for Phaser.Math.fuzzyEgal method
  this.threshold = 4;
  // Bewegungsrichtung des Spielers
  this.direction = Phaser.NONE;
  // Richtung, in die der Spieler abbiegen wird, falls möglich.
      Ansonsten: Phaser.NONE
  this.turnDir = Phaser.NONE;
  // Die den Spieler umgebenden Tiles, gescannt in getSurrounding
  this.possibleDirections = [ null, null, null, null, null];
  // Fügt dem Sprite alle Animationen hinzu
  this.addAllAnimations();
  // Anzahl an Schilden, die der Spieler schießen kann
  this.shieldCount = 0;
};
```

3.4.2.1 *Bewegung durch das Labyrinth*

In der Update-Funktion des Spiels wird mittels *getKey* permanent auf eine Eingabe des Spielers geprüft und entsprechend reagiert:

Listing 34: Prüfen auf Eingaben

```
1   getKey: function() {
2     if (global.cursors.left.isDown && this.direction !== Phaser.LEFT
          ) {
3       this.checkDir(Phaser.LEFT);
4
5     } else if (global.cursors.right.isDown && this.direction !==
          Phaser.RIGHT) {
6       this.checkDir(Phaser.RIGHT);
7
8     } else if (global.cursors.up.isDown && this.direction !==
          Phaser.UP) {
9       this.checkDir(Phaser.UP);
10
11    } else if (global.cursors.down.isDown && this.direction !==
          Phaser.DOWN) {
12      this.checkDir(Phaser.DOWN);
13
14    } else {
15      this.turnDir = Phaser.NONE;
16    }
17
18
19
20    // Stop the animations if the player is not moving
21    if (this.sprite.body.velocity.x == 0 &&
          this.sprite.body.velocity.y == 0) {
22      this.sprite.animations.stop();
23      this.sprite.frame = 1;
```

Wird eine der vier Richtungstasten gedrückt, wird mit der Funktion *checkDir* die entsprechende Richtung überprüft, um festzustellen, ob der Spieler abbiegen kann oder eine Wand im Weg ist.

Listing 35: Prüfen einer Richtung

```
1   checkDir: function(newDirection) {
2     this.getSurrounding();
3     if ( this.turnDir == newDirection
4           || !util.isSafeTile(this.possibleDirections[newDirection])
                ) {
5       return false;
6     }
7
8     if (this.direction == global.oppositeDirs[newDirection]) {
9       this.moveTo(newDirection);
10    } else {
11      this.turnDir = newDirection;
12      this.turnPointx = this.tilex * global.GRIDSIZE;
13      this.turnPointy = this.tiley * global.GRIDSIZE;
14
15    }
16  },
```

Dazu werden zunächst mit *getSurrounding* die vier den Spieler umgebenden Tiles gescannt und in *possibleDirections* gespeichert (s. unten).
Die darauffolgende if-Abfrage prüft, ob momentan schon in die gedrückte Richtung abgebogen wird oder ob sich in dieser Richtung eine Wand befindet. Trifft eines davon zu, wird die Funktion mit *return false* beendet.

Die nächste if-Abfrage prüft, ob es sich bei der gedrückten Richtung um die entgegengesetzte Spieler-Richtung handelt, in welchem Fall der Spieler einfach umkehren und die Funktion beendet werden kann. Ansonsten wird der Abbiegeprozess eingeleitet: in *turnDir* wird die gewünschte Richtung gespeichert und in Zeile 12 und 13 wird der Abbiegepunkt gespeichert.

Listing 36: Scannen der umliegenden Tiles

```
1   getSurrounding: function() {
2     this.tilex = Phaser.Math.snapTo(Math.floor(this.sprite.x),
3         global.GRIDSIZE) / global.GRIDSIZE;
4     this.tiley = Phaser.Math.snapTo(Math.floor(this.sprite.y),
5         global.GRIDSIZE) / global.GRIDSIZE;
6     var i = global.layer.index;
7
8     this.possibleDirections[Phaser.LEFT] = global.map.getTileLeft(i,
9           this.tilex, this.tiley).index;
10    this.possibleDirections[Phaser.RIGHT] = global.map.getTileRight(
          i,
11          this.tilex, this.tiley).index;
12    this.possibleDirections[Phaser.UP] = global.map.getTileAbove(i,
13          this.tilex, this.tiley).index;
14    this.possibleDirections[Phaser.DOWN] = global.map.getTileBelow(i
          ,
15          this.tilex, this.tiley).index;
16
17  },
```

Sie legt zunächst die Position des Spielers, gemessen in Tiles, fest. In possibleDirections werden dann die 4 umherliegenden Tiles gespeichert, indem mit map.getTileLeft (/Right/Above/Below) die entsprechenden Tiles gescannt werden. Die Richtungs-Konstanten von Phaser, LEFT, RIGHT, UP und DOWN (intern nur Integer-Werte von 1 bis 4) ermöglichen einen einheitlichen Array-Zugriff per Index: array[Phaser.UP].
In possibleDirections sind nun also die 4 umherliegenden Tiles gespeichert und mit dem Rest von *checkDir* kann fortgefahren werden.

Als nächstes folgt, sofern *turnDir* nicht Phaser.NONE (intern ein Integer-Wert 0) ist, die Funktion takeTurn:

Listing 37: Einleiten des Abbiegeprozesses

```
1   takeTurn: function() {
2     var cx = this.sprite.x;
3     var cy = this.sprite.y;
4
5     if (!Phaser.Math.fuzzyEqual(cx, this.turnPointx, this.threshold)
6         || !Phaser.Math.fuzzyEqual(cy, this.turnPointy,
              this.threshold)) {
7       return false;
8     }
9
10    this.sprite.x = this.turnPointx;
11    this.sprite.y = this.turnPointy;
12    this.sprite.body.reset(this.turnPointx, this.turnPointy);
13    this.moveTo(this.turnDir);
14    this.turnDir = Phaser.NONE;
15    return true;
16  },
```

Diese Funktion leitet letztendlich das eigentliche Abbiegen ein. Da die Spieler-Grafik genau gleich groß wie ein Tile ist (32x32px), kann der Spieler theoretisch nur dann abbiegen, wenn er exakt auf der Mitte eines Tiles liegt (Vergleiche Abbildung 14, S. 54). Allerdings ist dies praktisch nahezu unmöglich, weil der Spieler ständig in Bewegung ist: Er überquert diesen Punkt zu schnell, als dass darauf reagiert werden könnte.

Abhilfe schafft auch hier die Methode fuzzyEqual: Sie prüft die Gleichheit zweier Werte mit einer festgelegten Fehler-Toleranz. Somit wird geprüft, ob der Spieler gerade *ungefähr* auf einem Abbiegepunkt ist. Falls nicht, wird mit *return false* die Funktion verlassen und der Abbiegeprozess abgebrochen. Ansonsten wird mit den Zeilen 10 - 12 die Position exakt auf den Abbiegepunkt gelegt, der Spieler mit *moveTo* in die entsprechende Richtung bewegt und *turnDir* wieder auf Phaser.NONE gelegt; der Abbiegeprozess ist beendet.

Ein Ausschnitt aus der Funktion moveTo:

Listing 38: Richtungsänderung

```
1   moveTo: function(newDirection){
2     switch(newDirection){
3       case Phaser.UP:
4         this.direction = Phaser.UP;
5         this.setVelocity(0, -this.speed);
6         this.adjustShieldAnimations('up', 'shieldUp');
7         break;
8       case Phaser.LEFT:
9         ...
10        ...
11      case Phaser.NONE:
12        this.direction = Phaser.NONE;
13        this.setVelocity(0,0);
14        this.sprite.animations.stop();
15        this.sprite.frame = 1;
16      break;
17    }
18  },
```

Sie aktualisiert je nach Richtung die Variable *direction*, setzt mittels der Helfer-Funktion *setVelocity(x, y)* entsprechend die Geschwindigkeit (engl. velocity) der X- und Y-Achsen und stellt für das Sprite die richtigen Animationen ein. Wird die Richtung *Phaser.NONE* übergeben, soll der Spieler stehen bleiben. Folglich wird die Geschwindigkeit auf 0 gesetzt, werden alle Animationen gestoppt und das Sprite auf Frame 1 gestellt. Da der Held einen Schild tragen kann, haben wir eine Helferfunktion zum Einstellen der richtigen Animation, je nachdem ob er ihn gerade trägt oder nicht, geschrieben.

Listing 39: Anpassung der Animation

```
1   adjustShieldAnimations: function(animationNoShield,
        animationWithShield) {
2     if (this.shieldCount > 0) {
3       this.sprite.animations.play(animationWithShield);
4     } else {
5       this.sprite.animations.play(animationNoShield);
6     }
7   },
```

3.4.3 *Gegenstände*

Im Laufe des Spiels erscheinen Gegenstände, die dem Spieler einen Vorteil verschaffen. Das ist einmal der Honig, der beim Einsammeln alle Käfer für eine kurze Zeit verlangsamt, und einmal der Schild, der beim Werfen Eier und Käfer töten kann.

Für das Erscheinen der Gegenstände wird ein Timer verwendet. Der Timer für den Honig sieht so aus:

Listing 40: Honig-Timer

```
1 game.time.events.loop(HONEY_INTERVAL, spawnRandomHoney,
      callbackContext);
```

Dem Timer wird das Intervall übergeben, in dem er aufgerufen wird, die Callback-Methode, die aufgerufen werden soll, und der callback-Context, in dem die Callback-Methode aufgerufen wird.

3.4.3.1 *Erscheinen der Gegenstände*

Damit ein Gegenstand im Spiel erscheint, wird ein zufälliges Tile ausgewählt, auf dem der Gegenstand schließlich platziert wird.

Listing 41: Erscheinen der Gegenstände

```
1   spawnRandomItem: function(item) {
2     var randomPoint = global.safeTiles[Math.floor(Math.random() *
          global.safeTiles.length)];
3     var randomX = randomPoint.x * global.GRIDSIZE;
4     var randomY = randomPoint.y * global.GRIDSIZE;
5
6     switch(item){
7       case global.HONEY_NAME:
8         global.honeyGroup.create(randomX, randomY ,
              global.HONEY_NAME);
9         break;
10      case global.SHIELD_NAME:
11        global.shieldGroup.create(randomX, randomY,
              global.SHIELD_NAME);
12        break;
13    }
14  },
```

3.4.3.2 *Der Schild*

Ein Spieler kann bis zu 3 Schilde einsammeln, die er dann werfen kann, um Eier bzw. Käfer zu töten.

Listing 42: Schildwurf

```
1   throwShield: function(){
2     if(global.player.shieldCount > 0){
3       global.game.sound.play('throwShield');
4       global.player.shieldCount--;
5       global.shieldTxt.text = global.player.shieldCount + "/3";
6       global.shield = global.game.add.sprite(
            global.player.sprite.body.x, global.player.sprite.body.y,
            'shield');
7       global.game.physics.arcade.enable(global.shield);
8       var x;
9       var y;
10      switch(global.player.direction){
11        case Phaser.UP:
12          x=0;
13          y=-global.SHIELD_SPEED;
14          global.player.adjustShieldAnimations('up', 'shieldUp');
15          break;
16        case Phaser.DOWN:
17          x=0;
18          y=global.SHIELD_SPEED;
19          global.player.adjustShieldAnimations('down', 'shieldDown')
                ;
20          break;
21        case Phaser.LEFT:
22          x=-global.SHIELD_SPEED;
23          y=0;
24          global.player.adjustShieldAnimations('left', 'shieldLeft')
                ;
25          break;
26        case Phaser.RIGHT:
27          x=global.SHIELD_SPEED;
28          y=0;
29          global.player.adjustShieldAnimations('right', 'shieldRight
                ');
30          break;
31      }
32      global.shield.body.velocity.x = x;
33      global.shield.body.velocity.y = y;
34    }
35  },
```

Zu Beginn wird der Wurf-Sound abgespielt, anschließend wird die richtige Anzahl Schilder angezeigt. In Zeile 7 wird die Physik für den Schild aktiviert, damit er mit den Käfern bzw. Eiern kollidieren kann. Danach wird der Schild in die aktuelle Laufrichtung des Spielers geworfen, indem die **Velocity** für x und y des Sprite-Bodys entsprechend gesetzt wird.

3.4.3.3 *Der Honig*

Läuft der Spieler über einen Honig, wird die *slowBugs*-Methode in *Util.js* aufgerufen, in der alle Käfer verlangsamt werden. Nach einem bestimmten Zeitintervall soll die Verlangsamung wieder aufgehoben werden.

Listing 43: Käfer verlangsamen

```
1    slowBugs: function(player, honey){
2     honey.kill();
3     global.game.sound.play('collectItem');
4     global.game.time.events.add(4000, util.restoreSpeed, this);
5     // store count of bugs in array so that newly spawned bugs won't
            get
6     // the additional speed
7     if(global.bugs != null ){
8       global.slowedBugsCount = global.bugs.length;
9     }else{
10      global.slowedBugsCount = -1;
11    }
12    for(var i = 0; i < global.slowedBugsCount; i++ ){
13      global.bugs[i].slow();
14    }
15  },
```

Um die Verlangsamung aufzuheben, wird ein Timer-Event erstellt. Hierbei muss beachtet werden, dass die aktuelle Anzahl an Käfern gesichert wird, damit neu geschlüpfte Käfer später nicht beschleunigt werden, obwohl sie zuvor nicht verlangsamt wurden. In der For-Schleife werden alle Käfer verlangsamt.

3.5 SITE POINT ARENA

3.5.1 *Zwei-Spieler-Modus*

Für die Steuerung in Site Point Arena benötigt ein Spieler für folgende Aktionen eine Eingabemöglichkeit:

- Heldenfähigkeit nutzen
- Charakter steuern

Für die Steuerung stellt die Phaser-Engine die *Keyboard.createCursorKeys*-Methode zur Verfügung, die ein Objekt mit den Attributen: UP, DOWN, LEFT und RIGHT zurückgibt, die den Pfeiltasten zugeordnet werden. So kann die Steuerung des Charakters für einen Spieler leicht definiert werden. Um ein einheitliches Konzept beizubehalten, erzeugen wir für den zweiten Spieler ein eigenes Cursor-Objekt, das dieselben Eigenschaften besitzt:

Listing 44: Cursor

```
this.cursors = {
      up: global.game.input.keyboard.addKey(Phaser.Keyboard.W),
      down: global.game.input.keyboard.addKey(Phaser.Keyboard.S),
      left: global.game.input.keyboard.addKey(Phaser.Keyboard.A),
      right: global.game.input.keyboard.addKey(Phaser.Keyboard.D),
    };
```

Hier werden die Tasten: W, A, S und D zugewiesen, es können jedoch auch andere Tasten zugewiesen werden. Für die Heldenfähigkeit, wird für jeden Spieler eine neue Taste, die dem Konstruktor übergeben wird, registriert:

Listing 45: Schieß-Button

```
var btn = global.game.input.keyboard.addKey(shootBtn);
btn.onDown.add(this.shoot, this);
```

Wird die Taste gedrückt, wird die *shoot*-Methode aufgerufen.

3.5.2 *Schweife der Spieler*

Die Charaktere sollen beim Laufen einen Schweif hinterlassen, der bei Berührung mit einem Charakter diesen tötet. Das Spielfeld ist in Tiles unterteilt, was bedeutet, dass der Schweif erweitert werden muss, sobald der Charakter ein neues Tile betritt. Dabei darf der Schweif nicht auf dem aktuellen Tile platziert werden, da ansonsten der Charakter überdeckt wird und nicht mehr sichtbar für den Spieler ist.

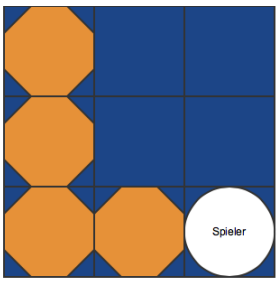


Abbildung 16: Spieler mit einem Schweif

Damit der Schweif auf das vorherige Tile platziert werden kann, muss die Position bekannt sein, deswegen speichert jeder Charakter die letzte Tile-Position. Allerdings darf der Schweif nicht auf einem Portal bzw. Pfeil-Tile platziert werden, da diese sonst nicht mehr für die Spieler sichtbar sind.

Listing 46: Schweif der Spieler

```
1   placeTail: function(){
2     if (!this.isOnArrow) {
3       var currentx = Phaser.Math.snapTo(this.sprite.x,
            global.GRIDSIZE) / global.GRIDSIZE;
4       var currenty = Phaser.Math.snapTo(this.sprite.y,
            global.GRIDSIZE) / global.GRIDSIZE;
5       if(currentx != this.lastPositionx || currenty !=
            this.lastPositiony){
6         if(global.playerTail[this.lastPositionx][this.lastPositiony]
              == null){
7           global.playerTail[this.lastPositionx][this.lastPositiony]
                =
8               global.game.add.sprite(this.lastPositionx *
                    global.GRIDSIZE,
9               this.lastPositiony * global.GRIDSIZE, this.tail);
10        }
11        this.lastPositionx = currentx;
12        this.lastPositiony = currenty;
13      }
14    }
15  },
```

3.5.3 *Spieler-Fähigkeiten*

Jeder Spieler hat die Fähigkeit, einmal ein Projektil abzufeuern, um einen Teil eines Schweifs, durch den er normalerweise verloren hätte, zu zerstören.

Info: Die Spielkarte basiert, wie in Bugitar, auf einer Tilemap. Sowohl ein Teil eines Schweifs als auch ein Spieler hat die Größe einer Kachel und jeder Spieler kann sich nur auf dem Raster der Kacheln bewegen. Somit passt ein Spieler nach Zerstörung eines Stücks vom Schweif genau durch den zerstörten Teil hindurch.

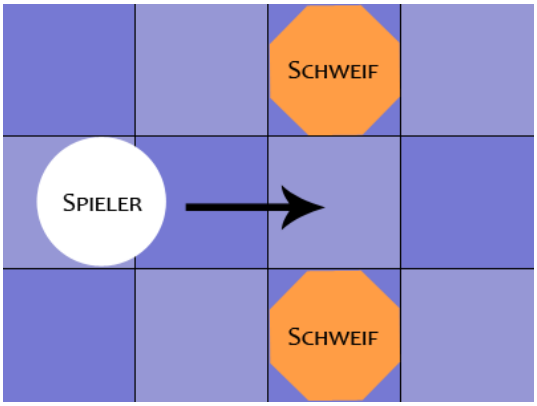


Abbildung 17: Der Spieler passt nach Zerstörung genau hindurch

Für das Projektil wird ein Bild verwendet.

```
1 global.game.load.image('fire', 'assets/fire.png');
```

Im Konstruktor des Spielers sind folgende Werte, die bei der Erstellung übergeben werden, von Bedeutung:

Listing 47: Player-Konstruktor Site Point Arena

```
1  function Player( ... , direction, ... , projectileImg, shootBtn) {
2    ...
3    this.direction = direction;
4    ...
5    this.canShoot = true;
6
7    var btn = global.game.input.keyboard.addKey(shootBtn);
8    btn.onDown.add(this.shoot, this);
9
10   this.projectile = '';
11   this.projectileImg = projectileImg;
12 };
```

Direction bestimmt die Bewegungsrichtung, die der Spieler nach Erstellung hat. Diese wird später zur Bestimmung der Feuer-Richtung

des Projektils benötigt. Mit *canShoot* kann später geprüft werden, ob er schießen kann, oder ob er sein Projektil bereits abgefeuert hat. Da es in diesem Spiel zwei Spieler gibt, wird dem Konstruktor ein eigener Button übergeben, damit für jeden Spieler ein eigener definiert werden kann. Dasselbe gilt auch für das Bild des Projektils. Wenn der Button gedrückt wird, wird die Methode *shoot* aufgerufen:

Listing 48: Abfeuern eines Projektils

```
1   shoot: function() {
2     if (this.canShoot) {
3
4       this.canShoot = false;
5       this.projectile = global.game.add.sprite(this.sprite.body.x,
6         this.sprite.body.y, 'fire');
7       global.game.physics.arcade.enable(this.projectile);
8       this.projectile.checkWorldBounds = true;
9       this.projectile.outOfBoundsKill = true;
10
11      var x;
12      var y;
13      switch(this.direction){
14        case Phaser.UP:
15          x=0;
16          y=-global.PROJECTILE_SPEED;
17          break;
18        case Phaser.DOWN:
19          x=0;
20          y=global.PROJECTILE_SPEED;
21          break;
22        case Phaser.LEFT:
23          x=-global.PROJECTILE_SPEED;
24          y=0;
25          break;
26        case Phaser.RIGHT:
27          x=global.PROJECTILE_SPEED;
28          y=0;
29          break;
30      }
31      this.projectile.body.velocity.x = x;
32      this.projectile.body.velocity.y = y;
33
34    }
35  },
```

Falls der Spieler schießen darf (Z. 2), wird *canShoot* auf false gesetzt und ein Sprite mitsamt Physik für das Projektil an der Stelle des Spielers erstellt (Z. 5, 6, 7). Mit *checkWorldBounds* und *outOfBoundsKill* auf true gesetzt wird sichergestellt, dass ein Sprite zerstört wird, sobald es die Spielwelt verlässt.

Der darauffolgende Switch-Case-Block dient dazu, die korrekte Bewegungsrichtung für das Projektil zu bestimmen. Es soll in dieselbe Richtung abgefeuert werden, in die der Spieler sich bewegt, was mit

den Zeilen 31 und 32 festgelegt wird.

Der Spieler ist nun also in der Lage, ein Projektil in seine Bewegungsrichtung abzufeuern. Damit das Projektil etwas bewirkt, muss eine Kollision hinzugefügt werden. Es ist möglich, eine Kollision in Phaser auch ohne Hilfe der in 3.3.4 (S. 49) besprochenen beiden Methoden *collision* und *overlap* zu implementieren. Um alles ausprobiert zu haben, taten wir dies für die Projektile:

Listing 49: Spieler-Kollision mit Schweif

```
1  checkProjectileCollision: function() {
2    if (this.projectile.body != null) {
3      var x = Math.floor(this.projectile.body.x / global.GRIDSIZE);
4      var y = Math.floor(this.projectile.body.y / global.GRIDSIZE);
5
6      if (global.playerTail[x] != null && global.playerTail[x][y] !=
           null) {
7        global.playerTail[x][y].kill();
8        global.playerTail[x][y] = null;
9        this.projectile.kill();
10       this.projectile.body = null;
11     }
12   }
13 }
```

Diese Funktion muss, genau wie *collision* und *overlap*, im Update-Zyklus auftauchen, damit eine Kollision erkannt wird, weshalb durch die selbstimplementierte Kollisions-Erkennung keine Performanz-Nachteile entstehen. Die Funktion prüft, ob für die Koordinaten des Projektils ein Schweif-Objekt in playerTail[][] enthalten ist. Falls ja, wird dieses Objekt (also der Schweif an dieser Stelle auf dem Spielfeld) und das Projektil zerstört.

3.5.4 *Portale*

Die Portale sollten aus allen Richtungen nutzbar sein und eine beliebige Größe besitzen können.

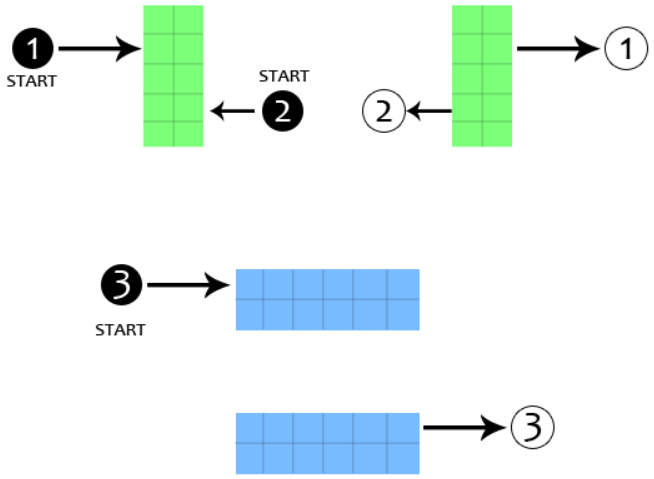


Abbildung 18: Die Nutzung der Portale veranschaulicht

Ein Portal hat bei uns die Größe einer Spielfeld-Kachel (16x16px). Da wir jedoch beliebige Größen verwenden wollen, setzen wir ein größeres aus mehreren kleinen zusammen. Das bedeutet, dass jedes einzelne die Größe des gesamten Portals speichern muss, um dem Spieler, der teleportiert werden soll, das richtige Ergebnis-Portal liefern zu können.

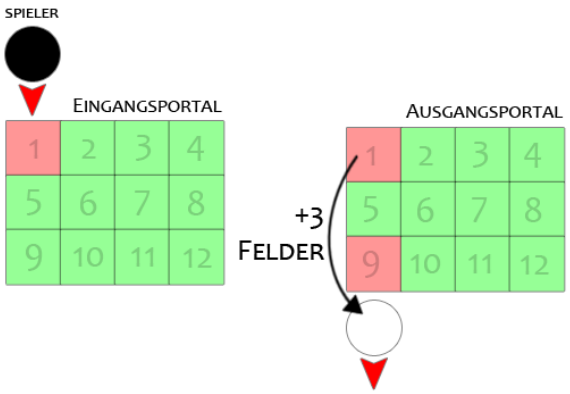


Abbildung 19: Nutzung der Portalgröße (hier die Höhe) zur Bestimmung des Zielorts

Der Konstruktor eines Portal-Paares sieht folgendermaßen aus:

Listing 50: Portal-Konstruktor

```
1 function Portal(x1, y1, x2, y2, imgName, portalSizeX, portalSizeY) {
2   this.portal1 = new PortalComp(x1, y1, imgName, portalSizeX,
        portalSizeY);
3   this.portal2 = new PortalComp(x2, y2, imgName, portalSizeX,
        portalSizeY);
4 };
```

Die Parameter:

- x1, y1: Die Koordinaten der ersten Portal-Komponente
- x2, y2: Die Koordinaten der zweiten Portal-Komponente
- imgName: Das verwendete Bild für jede Portalkomponente
- portalSizeX: Die horizontale Größe in Kacheln
- portalSizeY: Die vertikale Größe in Kacheln

Es ist wichtig, zu verstehen, dass die beiden Argumente *portalSizeX* und *portalSizeY* erst im Zusammenspiel mehrerer Portale, die zusammen ein großes bilden, Sinn ergeben.

Es werden zwei Portal-Komponenten erstellt (Z. 2, 3), welche zusammen ein Portal-Paar bilden. Eine Portal-Komponente hat folgenden Konstruktor, welche als Parameter jeweils x1, y1 und x2, y2 zur Positionierung erhalten. Die übrigen Parameter sind identisch zu denen des Portals.

Listing 51: eine Portal-Komponente

```
1  function PortalComp(x, y, imgName, portalSizeX, portalSizeY) {
2    this.portalSizeX = portalSizeX;
3    this.portalSizeY = portalSizeY;
4    var xCoord = x * global.GRIDSIZE;
5    var yCoord = y * global.GRIDSIZE;
6    this.sprite = global.game.add.sprite(xCoord, yCoord, imgName);
7    this.tilex = x;
8    this.tiley = y;
9
10   global.portalChecker[x][y] = true;
11 };
```

Es werden sowohl die Größe des Portals, als auch die Position in Kacheln als Variablen gespeichert. Zudem wird ein Sprite mit dem übergebenen Bild an der richtigen Stelle erstellt.

Eine Portal-Komponente hat die Methode *getOutComeTile*, welche nach dem in Abbildung 19 gezeigten Schema die richtige Spielerposition

nach einer Teleportation berechnet:

Listing 52: Berechnung der Ergebnis-Position

```
getOutComeTile: function(direction) {
  var newX = this.tilex;
  var newY = this.tiley;
  if (direction === Phaser.LEFT) {
    newX -= this.portalSizeX;
  } else if (direction === Phaser.RIGHT) {
    newX += this.portalSizeX;
  } else if (direction === Phaser.UP) {
    newY -= this.portalSizeY;
  } else if (direction === Phaser.DOWN) {
    newY += this.portalSizeY;
  }
  return {
    newX : newX,
    newY : newY
  }
}
```

Diese Funktion wird auf die Ziel-Portalkomponente angewandt (nicht auf die, auf die der Spieler trifft).

Es fehlt also nur noch die Kollision, die eine solche Teleportation auslöst. Dafür wird eine Methode im Update-Zyklus benötigt, die prüft, ob der Spieler ein Portal berührt:

Listing 53: Treffen auf ein Portal

```
checkForPortal: function() {
  var currentXTile = Phaser.Math.snapTo(this.sprite.x,
      global.GRIDSIZE) / global.GRIDSIZE;
  var currentYTile = Phaser.Math.snapTo(this.sprite.y,
      global.GRIDSIZE) / global.GRIDSIZE;


  for (i = 0; i < global.portal.length; ++i) {
    if (currentXTile === global.portal[i].portal1.tilex
        && currentYTile === global.portal[i].portal1.tiley) {

      this.teleport(global.portal[i].portal2);
    } else if (currentXTile === global.portal[i].portal2.tilex
        && currentYTile === global.portal[i].portal2.tiley) {
      this.teleport(global.portal[i].portal1);
    }
  }
},
```

Sie berechnet zunächst die Tile-Position des Spielers und prüft dann, ob an dieser Stelle ein Portal vorhanden ist. Falls ja, wird, je nachdem ob er eine erste oder zweite Portalkomponente berührt, die entspre-

chende *teleport*-Funktion aufruft:

Listing 54: Teleportation eines Spielers

```
1   teleport: function(portalComp) {
2     var newX = portalComp.getOutComeTile(this.direction).newX *
          global.GRIDSIZE;
3     var newY = portalComp.getOutComeTile(this.direction).newY *
          global.GRIDSIZE;
4
5     this.sprite.x = newX;
6     this.sprite.y = newY;
7     this.sprite.body.reset(newX, newY);
8     this.moveTo(this.direction);
9   },
```

Diese verwendet die Funktion *getOutcomeTile* der Ziel-Portalkomponente (Z. 2, 3) und setzt den Spieler an die richtige Position (Z. 5-8).

Um das Erstellen größerer Portale zu vereinfachen, lohnt es sich, Helfer-Methoden wie folgende zu schreiben:

Listing 55: Helfer-Funktion zur Portal-Erstellung

```
1 createPortal: function(x1, y1, x2, y2, imgName, portalSizeX,
      portalSizeY) {
2   for (i = 0; i < portalSizeX; ++i) {
3     for (j= 0; j < portalSizeY; ++j) {
4       global.portal.push(new Portal(x1+i, y1+j, x2+i, y2+j, imgName,
5           portalSizeX, portalSizeY));
6     }
7   }
8 },
```

Man übergibt zwei Koordinaten-Paare, von denen aus die Portale mit dem übergebenen Bild und der gewünschten Größe erstellt werden.

Dies ist beispielsweise die Positionierung der zwei Portale nach folgendem Aufruf:

```
1 createPortal(1, 2, 8, 2, 'rotesPortal', 4, 2);
```

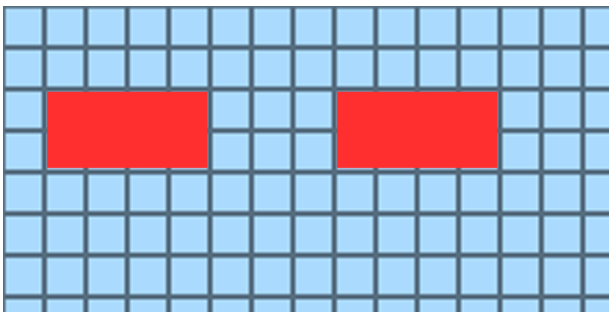


Abbildung 20: Beispiel-Positionierung der Portale

Um es noch weiter zu vereinfachen, haben wir eine weitere Methode geschrieben, welcher man nur die Position des ersten Portals übergibt. Das zweite Portal wird automatisch horizontal an der Mittelachse des Spielfelds gespiegelt erstellt:

Listing 56: Weitere Helfer-Funktion zur Portal-Erstellung

```
1  createPortalMirroredX: function(x, y, imgName, portalSizeX,
       portalSizeY) {
2    var xMirror = global.map.width - 1;
3
4    this.createPortal(x, y,
5      xMirror - x - portalSizeX + 1, y,
6      imgName, portalSizeX, portalSizeY);
7  },
```

Wir haben zusätzlich noch die Funktionen *createPortalMirroredY* (automatische vertikale Spiegelung des zweiten Portals) und *createPortalMirroredXY* (Spiegelung erfolgt horizontal und vertikal) erstellt.

3.5.5 *Pfeil-Felder*

Diese Felder sollen, wenn ein Spieler sie berührt, die Bewegungsrichtung vorgeben und eine Richtungsänderung seitens des Spielers blockieren. Der Konstruktor eines Pfeils sieht so aus:

Listing 57: Pfeil-Feld-Konstruktor

```
function Arrow(direction, tilex, tiley) {
  this.direction = direction;
  this.tilex = tilex;
  this.tiley = tiley;

  var xCoord = tilex * global.GRIDSIZE + global.GRIDSIZE/2;
  var yCoord = tiley * global.GRIDSIZE + global.GRIDSIZE/2;
  this.sprite = global.game.add.sprite(xCoord, yCoord, 'arrow');
  this.sprite.anchor.setTo(0.5, 0.5);
  this.setAngle();
};
```

Als Parameter erhält er eine Richtung und die Position in Tiles. Es wird ein Sprite mit einem Pfeil-Bild erstellt und entsprechend der Richtung rotiert:

Listing 58: Richtiges Rotieren eines Pfeil-Feldes

```
  setAngle: function() {
    if (this.direction == Phaser.LEFT) {
      return;
    } else if (this.direction == Phaser.RIGHT) {
      this.sprite.angle = 180;
    } else if (this.direction == Phaser.UP) {
      this.sprite.angle = 90;
    } else if (this.direction == Phaser.DOWN) {
      this.sprite.angle = -90;
    }
  }
```

Zur Erstellung haben wir eine Funktion geschrieben, mit der man eine Linie von mehreren Pfeilen auf einmal aneinandergereiht positionieren kann:

Listing 59: Helfer-Funktion zur Erstellung von mehreren-Pfeil-Feldern

```
1   drawArrowLine: function(tilex, tiley, direction, drawDirection,
        length) {
2     var targetx = tilex;
3     var targety = tiley;
4     if (drawDirection == Phaser.RIGHT) {
5       targetx += length - 1;
6     } else if ( drawDirection == Phaser.LEFT) {
7       targetx -= length - 1;
8     } else if (drawDirection == Phaser.UP) {
9       targety -= length - 1;
10    } else if (drawDirection == Phaser.DOWN) {
11      targety += length - 1;
12    }
13
14    if (drawDirection == Phaser.RIGHT || drawDirection ==
          Phaser.DOWN) {
15      for (i = tilex; i <= targetx; ++i) {
16        for (j = tiley; j <= targety; ++j) {
17          global.arrows[i][j] = new Arrow(direction, i, j);
18        }
19      }
20    } else {
21      for (i = tilex; i >= targetx; --i) {
22        for (j = tiley; j >= targety; --j) {
23          global.arrows[i][j] = new Arrow(direction, i, j);
24        }
25      }
26    }
27  },
```

Die entstehende Linie beginnt bei den Koordinaten *tilex*, *tiley* und wird in die in *drawDirection* angegebene Richtung erstellt. Die Variable *direction* gibt die Richtung, die die Pfeile selbst dabei haben (also in welche Richtung ein Spieler bei Berührung gezwungen wird), an. Mit *length* kann angegeben werden, wie viele Pfeile erstellt werden sollen.

Jetzt muss auf die Pfeile bei Berührung reagiert werden. Bei den Pfeilen verzichteten wir wie bei den Portalen schon auf die typischen *collide*- und *overlap*-Funktionen von Phaser, stattdessen implementierten wir die folgende Funktion innerhalb des Update-Zyklus':

Listing 60: Treffen auf ein Pfeil-Feld

```
1   checkForArrow: function() {
2     var currentXTile = Phaser.Math.snapTo(this.sprite.x,
          global.GRIDSIZE) / global.GRIDSIZE;
3     var currentYTile = Phaser.Math.snapTo(this.sprite.y,
          global.GRIDSIZE) / global.GRIDSIZE;
4
5     if (global.arrows[currentXTile][currentYTile] !== null) {
6       this.takeArrowTurn(global.arrows[currentXTile][currentYTile].
            direction);
7       this.isOnArrow = true;
8     } else {
9       this.isOnArrow = false;
10    }
11
12  },
```

Diese Methode prüft für einen Spieler, ob er sich auf einem Pfeil befindet und ruft ggf. *takeArrowTurn* mit der Richtung des Pfeils auf. Zusätzlich wird die Variable *isOnArrow* auf true gesetzt, um eine Änderung der Bewegungsrichtung durch den Spieler selbst zu blockieren. Verlässt der Spieler ein Pfeil-Feld, wird durch den *else*-Zweig gewährleistet, dass *isOnArrow* auf false zurückgesetzt wird und eigene Richtungsänderungen wieder möglich werden.

Listing 61: Richtungsänderung auf einem Pfeil

```
1  takeArrowTurn: function(direction) {
2      if (direction === Phaser.LEFT && this.direction !== Phaser.LEFT)
           {
3        this.wantsTurn = Phaser.LEFT;
4
5      } if (direction === Phaser.RIGHT && this.direction !==
           Phaser.RIGHT) {
6        this.wantsTurn = Phaser.RIGHT;
7
8      } if (direction === Phaser.UP && this.direction !== Phaser.UP) {
9        this.wantsTurn = Phaser.UP;
10
11     } if (direction === Phaser.DOWN && this.direction !==
           Phaser.DOWN) {
12       this.wantsTurn = Phaser.DOWN;
13
14     }
15     this.turn();
16   },
```

Diese Funktion erhält als Argument die Richtung des Pfeils, auf dem sich der jeweilige Spieler befindet. Falls es sich dabei nicht um eine Richtung handelt, in die der Spieler sich gerade schon bewegt, wird *wantsTurn* auf diese Richtung gesetzt, um ein Abbiegen zum nächstmöglichen Zeitpunkt (also wenn der Spieler das nächste Mal genau auf einer Kachel ist) einzuleiten.

3.5.6 *Geschwindigkeits-Booster*

Auf dem Spielfeld sollen regelmäßig an zufälligen Orten Geschwindigkeits-Booster erscheinen, welche das Tempo eines Spielers kurzzeitig erhöhen. Dazu haben wir in der *create*-Methode eine Gruppe erstellt und mit einer Physik ausgestattet:

Listing 62: Erstellung der Booster-Gruppe mit Timer

```
1  create: function() {
2    ...
3    global.boosters = global.game.add.group();
4    global.boosters.enableBody = true;
5
6    global.game.time.events.loop(global.BOOSTER_SPAWN_INTERVAL,
7        this.spawnBooster, this);
8  },
```

Wie man sieht, erstellten wir einen Timer, der in einem bestimmten Intervall immer wieder die Funktion *spawnBooster* aufruft.

Listing 63: Erscheinen der Booster

```
1    spawnBooster: function() {
2      var randomXTile = Math.floor(Math.random() * global.map.width);
3      var randomYTile = Math.floor(Math.random() * global.map.height);
4
5      if (  global.portalChecker[randomXTile][randomYTile] ||
6            global.arrows[randomXTile][randomYTile] != null ) {
7      } else {
8        global.boosters.create(randomXTile * global.GRIDSIZE,
9                randomYTile * global.GRIDSIZE, 'booster');
10     }
11
12   }
```

Es werden 2 zufällige Koordinaten erstellt und geprüft, ob sich an dieser Position bereits ein Portal oder ein Pfeil befindet. In diesem Fall wird kein Booster erstellt. Ansonsten wird der Gruppe *boosters* an dieser Position ein Sprite mit dem *booster*-Bild hinzugefügt.

Um bei Kollision eine Geschwindigkeits-Erhöhung auszulösen, fügten wir der Booster-Gruppe und den beiden Spielern *Overlaps* hinzu:

Listing 64: Hinzufügen der Kollision mit Booster-Gruppe

```
// Spieler 1
global.game.physics.arcade.overlap(global.playerOne.sprite,
        global.boosters, this.boostPOne, null, this);
// Spieler 2
global.game.physics.arcade.overlap(global.playerTwo.sprite,
        global.boosters, this.boostPTwo, null, this);
```

Bei einem Overlap wird also folgende Funktion aufgerufen (für Spieler 2 *boostPTwo*):

Listing 65: Spieler-Kollision mit einem Booster

```
  boostPOne: function(playerSprite, booster) {
    global.playerOne.speedUp();
    booster.kill();
  },
```

Der betroffene Spieler erhält mit *speedUp* einen Geschwindigkeitsschub:

Listing 66: Geschwindigkeits-Boost

```
  speedUp: function() {
    this.speed = global.BOOSTER_SPEED;
  },
```

Da auf diese Weise der Geschwindigkeitsschub auf Dauer wäre (die Geschwindigkeit wird momentan noch nicht zurückgesetzt), bauten wir folgende Funktion in den Update-Zyklus mit ein:

Listing 67: Sanftes Zurücksetzen der Geschwindigkeit

```
  adjustVelocity: function() {
    if (this.speed >= global.STANDARD_SPEED) {
      this.speed -= 6;
    }
    if (this.direction === Phaser.LEFT) {
      this.setVelocity(- this.speed, 0);
    } else if (this.direction === Phaser.RIGHT) {
      this.setVelocity(this.speed, 0);
    } else if (this.direction === Phaser.UP) {
      this.setVelocity(0, -this.speed);
    } else if (this.direction === Phaser.DOWN) {
      this.setVelocity(0, this.speed);
    }
  },
```

Diese prüft lediglich, ob die aktuelle Geschwindigkeit des Spielers schneller als gewöhnlich ist und reduziert sie dann stückweise (mit jedem Update-Aufruf) um einen bestimmten Betrag, hier 6. Mit setVelocity wird die Geschwindigkeit in der entsprechenden Richtung dann angepasst.

Durch diese stückweise Reduzierung pro Update-Durchlauf wird kein Timer benötigt und die Geschwindigkeit geht sanft auf die Standard-Geschwindigkeit zurück, statt abrupt zurückgesetzt zu werden.

3.6 REVENGE OF THE BUGS

Auf der offiziellen Phaser-Seite gibt es eine Demo zu einer simplen Realisierung vom klassischen Space Invaders [40]. Wir erachteten Teile dieser Implementierung bis auf Kleinigkeiten als die beste Lösung und verwendeten sie daher als Basis für manche unserer Funkionen, wie z.B. die Käferbewegung mittels eines ***Tweens*** *oder die Verwendung einer festen Anzahl an Projektilen für den Spieler.*

3.6.1 *Spieler*

3.6.1.1 *Bewegung*

Im dritten Spiel benötigten wir einen Spieler, der sich am unteren Rand des Spielfelds nur auf einer horizontalen Ebene bewegen durfte.

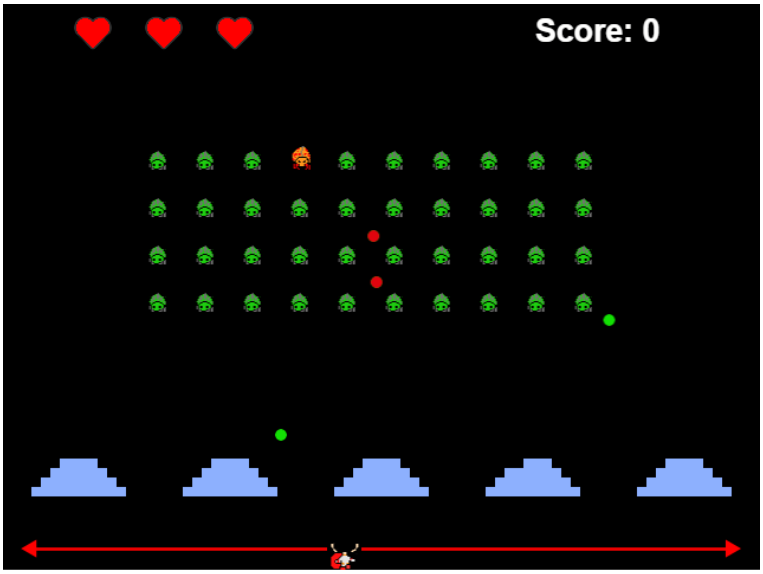


Abbildung 21: Spieler-Bewegung in Revenge of the Bugs

Dazu erstellten wir das Sprite zu Beginn am unteren Ende und ließen es nur auf zwei Tastenschläge, links und rechts, reagieren.

Listing 68: Links- und Rechts-Bewegung des Spielers

```
getKey: function () {
    if (global.cursors.left.isDown) {
      this.sprite.body.velocity.x = -150;
      this.adjustShieldAnimations('left', 'shieldLeft');

    } else if (global.cursors.right.isDown) {
      this.sprite.body.velocity.x = 150;
      this.adjustShieldAnimations('right', 'shieldRight');

    } else {
      this.sprite.body.velocity.x = 0;
      this.adjustShieldAnimations('up', 'shieldUp');

    }

    ...

}
```

Wird keine Taste gedrückt, wird nur eine Laufanimation abgespielt, die so aussieht, als laufe der Spieler nach oben.

3.6.1.2 *Normales Projektil*

Der Spieler kann zu jeder Zeit Projektile abfeuern, um die Käfer zu zerstören. Dazu erstellten wir eine Gruppe mit Physik, die bereits 30 vorgefertigte Sprites enthält:

Listing 69: Projektil-Gruppe für den Spieler

```
  createPlayerBullets: function () {
    global.playerBullets = global.game.add.group();
    global.playerBullets.enableBody = true;
    global.playerBullets.physicsBodyType = Phaser.Physics.ARCADE;
    global.playerBullets.createMultiple(30, 'fire');
    global.playerBullets.setAll('anchor.x', -0.5);
    global.playerBullets.setAll('anchor.y', 1);
    global.playerBullets.setAll('outOfBoundsKill', true);
    global.playerBullets.setAll('checkWorldBounds', true);
  },
```

Phaser liefert praktische Methoden zum Arbeiten mit mehreren Gruppen-Elementen: *createMultiple*, welches mehrere Gruppen-Sprites auf einmal erstellt, und *setAll*, welche Änderungen an einer Eigenschaft aller Mitglieder ermöglicht. Dazu wird die zu ändernde Eigenschaft als String und der gewünschte Wert übergeben.

Mit der Funktion *shoot*, die nach Drücken des *fireBtn*'s abgefeuert wird, wird eines der Projektile abgefeuert:

Listing 70: Festlegen des Feuer-Buttons

```
1 function Player(x, y) {
2   ...
3   this.fireBtn = global.game.input.keyboard.addKey(
        Phaser.Keyboard.SPACEBAR);
4     ...
5 },
```

Listing 71: Reagieren auf den Feuer-Button

```
1 getKey: function () {
2
3     ...
4
5     if (this.fireBtn.isDown) {
6       this.shoot();
7     }
8
9     ...
10  },
```

Listing 72: Abfeuern eines Projektils

```
1   shoot: function () {
2     if (global.game.time.now > this.bulletTime) {
3       var bullet = global.playerBullets.getFirstExists(false);
4       if (bullet) {
5         bullet.reset(this.sprite.x, this.sprite.y + 8);
6         bullet.body.velocity.y = -400;
7         this.bulletTime = global.game.time.now + 200;
8       }
9     }
10  },
```

Durch den ersten if-Befehl wird gewährleistet, dass die Projektile nicht zu schnell hintereinander abgefeuert werden können. Der Wert von *bulletTime* wird nach jedem Abfeuern um 200 (ms) erhöht. Es kann also höchstens einmal alle 200ms ein Projektil abgefeuert werden.

Mit *playerBullets.getFirstExists(false)* (Z. 3) erhält man das erste der zuvor 30 erstellten Projektil-Sprites, das nicht existiert (anders gesagt: ein momentan ungenutztes Projektil). Die Position dieses Projektils wird dann auf die des Spieler-Sprites gesetzt, außerdem erhält es eine Geschwindigkeit, die es nach oben fliegen lässt (Z. 5, 6).

Wie gewohnt erhalten die Projektil- und Käfer-Gruppen ein *Overlap*:

Listing 73: Hinzufügen der Käfer-Kollision

```
1 global.game.physics.arcade.overlap(global.playerBullets, global.bugs
      , this.hitBug, null, this);
```

Listing 74: Kollision mit einem Käfer

```
1   hitBug: function (bullet, bug) {
2     global.score += 10;
3     global.scoreTxt.text = "Score: " + global.score;
4     bullet.kill();
5     bug.kill();
6   },
```

Wird ein Käfer getroffen, erhöht sich der Score und das Projektil und der getroffene Käfer werden zerstört.

Auf dieselbe Weise, wie die Spieler-Projektile Käfer zerstören, zerstören sie auch die Schutzwände.

3.6.1.3 *Der Schild, das Spezial-Projektil*

Der Spieler sollte ein Spezial-Projektil, seinen Schild, abfeuern können, welches mehrere Käfer auf einmal zerstören und von der Spielwelt abprallen kann, um auf dem Rückweg noch mehr Käfer zerstören zu können. Im Gegensatz zum normalen Projektil zerstört der Schild keine Schutzwände.

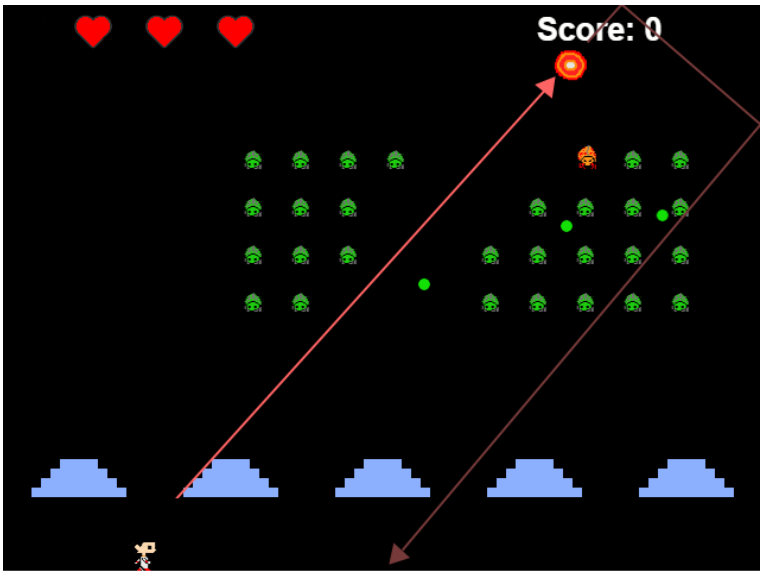


Abbildung 22: Der Schild wird in Spieler-Richtung abgefeuert

Hat er es abgefeuert, steht es ihm nach einem bestimmten Zeitintervall, bei uns 20 Sekunden, wieder zur Verfügung.

Listing 75: Festlegen des Schild-Buttons

```
1  function Player(x, y) {
2      ...
3      this.fireSpecialBtn = global.game.input.keyboard.addKey(
           Phaser.Keyboard.V);
4      ...
5  },
```

Listing 76: Reagieren auf Schild-Button

```
1    getKey: function () {
2
3      ...
4
5      if (this.fireSpecialBtn.isDown) {
6        this.shootSpecial();
7      }
8
9      ...
10   },
```

Listing 77: Abfeuern des Spezialprojektils

```
1    shootSpecial: function () {
2      if (this.canShoot > 0) {
3        --this.canShoot;
4        global.shield = global.game.add.sprite(
             global.player.sprite.body.x, global.player.sprite.body.y,
             'shield');
5        global.game.physics.arcade.enable(global.shield);
6        global.shield.body.collideWorldBounds = true;
7        global.shield.body.bounce.setTo(0.8, 0.8);
8        global.shield.body.velocity.x = this.sprite.body.velocity.x *
             2;
9        global.shield.body.velocity.y = -300;
10     }
11   }
```

Besitzt der Spieler einen Schild, wird ein Sprite an seiner Position erstellt (Z. 4). Mit *shield.body.collideWorldBounds* und *shield.body.bounce.setTo(0.8, 0.8)* wird erreicht, dass der Schild an den Spielfeld-Wänden abprallt und bei jedem Aufprall seine X- und Y-Geschwindigkeit um 20% verringert. In Zeile 8 wird sichergestellt, dass die Flugrichtung von der Bewegung des Spielers abhängig ist (siehe Abbildung 22 S.88).

Damit der Schild nicht für immer durch die Spielwelt fliegt, sondern zerstört wird, sobald er einmal von oben zurückgekehrt ist, gibt es die Methode *handleShield*:

Listing 78: Regelung der Schild-Lebensdauer

```
1   getKey: function () {
2
3     ...
4
5     if (global.shield.body != null) {
6       this.handleShield();
7     }
8   },
```

Listing 79: Schild-Lebensdauer

```
1   handleShield: function () {
2     // collision with bugs
3     global.game.physics.arcade.overlap(global.shield, global.bugs,
          this.hitBug, null, this);
4     // Destroy shield when specific speed is reached (it looses
          speed with every bounce)
5     if (Math.abs(global.shield.body.velocity.y) < 220) {
6       global.shield.kill();
7     }
8   },
```

Diese sorgt auch für die Kollision mit den Käfern:

Listing 80: Käfer-Kollision

```
1   hitBug: function (shield, bug) {
2     bug.kill();
3   },
```

Die Funktion *handleShield* prüft nur die Y-, also die vertikale, Geschwindigkeit. Somit hat nur der Aufprall mit der oberen und unteren Wand Auswirkung auf dessen Zerstörung. Dadurch wird gewährleistet, dass der Schild durch Aufprall an der linken und rechten Außenwand nicht zu früh zerstört wird und mindestens einmal von oben zurückkehrt. Nach Zurückkehren von oben erreicht der Schild den Boden, dessen Aufprall seine Y-Geschwindigkeit wieder um 20% verringert. Dadurch unterschreitet er den Schwellenwert 220 und wird zerstört.

3.6.2 *Bugs / Käfer*

Im Spiel „Revenge of the Bugs“ erscheinen immer wieder neue Käfer-Wellen, die sich sowohl horizontal als auch vertikal bewegen. Um das zu erreichen, haben wir einen Tween verwendet.

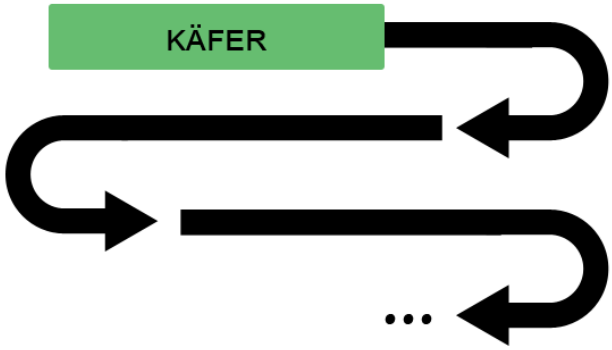


Abbildung 23: Bewegungsablauf der Käfer

3.6.2.1 *Tweens*

Tweens ermöglichen es, Eigenschaften eines Objekts über eine bestimmte Zeitspanne zu verändern. So kann beispielsweise ein Objekt über 10 Sekunden vergrößert werden oder langsam ausgeblendet werden [33].
Alle Käfer befinden sich in einer Gruppe, dadurch können wir einen einzigen Tween auf die Gruppe anlegen, statt für jeden einzelnen Käfer einen Tween zu erzeugen. Der Tween soll in einer Endlosschleife laufen und nach jedem Durchgang die Gruppe vertikal nach unten bewegen. Um das ganze umzusetzen, nutzen wir die *to*-Methode eines Tweens:

```
1  myTween.to(properties, duration, ease, autoStart, delay, repeat,
       yoyo)
```

In *properties* können Eigenschaften definiert werden, die durch den Tween erreicht werden sollen. In unserem Fall übergeben wir eine X-Koordinate, die erreicht werden soll, damit die horizontale Bewegung entsteht. Für *duration* wird die Dauer des Tweens in Millisekunden übergeben. *Ease* beschreibt wie sich der Wert aus *properties* verändern soll, in unserem Fall linear. Mit *autoStart* kann definiert werden, ob der Tween automatisch starten oder manuell aktiviert werden soll.

Durch einen *delay* wird der Tween verzögert gestartet. Soll der Tween wiederholt werden, kann für *repeat* ein ganzzahliger Wert übergeben werden, mit -1 wiederholt sich der Tween immer. Übergibt man für *yoyo* **true**, setzt sich der Tween selbst zurück und spielt sich rückwärts ab. In unserem Spiel werden diese Parameter übergeben:

Listing 81: Tween-Konstruktor

```
1 global.game.add.tween(global.bugs).to({ x: 150}, 2000,
      Phaser.Easing.Linear.None, true, 0, 1000, true);
```

Damit immer wieder neue Wellen erscheinen, wird in der *Update*-Methode geprüft, ob in der Käfer-Gruppe noch Käfer am Leben sind. Falls keine Käfer mehr leben, erscheint eine neue Welle.

Jetzt fehlt nur noch die vertikale Bewegung. Wir haben die *yoyo*-Eigenschaft aktiviert, die nach jedem Durchlauf ein onLoop-Event sendet, auf das wir reagieren können. Sobald der Tween einmal durchlaufen wurde, wollen wir die Käfer-Gruppe nach unten verschieben:

```
1 global.tween.onLoop.add(this.descend, this);
```

Wir übergeben dazu unsere Callback-Methode und den dazugehörigen Kontext. In der *descend*-Methode findet dann tatsächlich die Verschiebung statt:

```
1   descend: function () {
2     global.bugs.y += global.descendValue;
3   },
```

Damit sich die ganze Gruppe ein Stück nach unten bewegt, addieren wir einen Wert zu der Y-Koordinate hinzu. Der *descendValue* wird im Laufe des Spiels immer erhöht, sodass die Käfer-Gruppe immer schneller wird und der Schwierigkeitsgrad sich dadurch erhöht.

3.6.2.2 *Die Fähigkeiten*

Die Bewegung der Käfer ist nun abgeschlossen, fehlen noch die Fähigkeiten. Dazu zählt das Schießen der Käfer und der Spezial-Käfer.

SCHIESSEN

Listing 82: Schießen der Käfer

```
1   fire: function () {
2     if (global.game.time.now > global.nextFire) {
3       global.nextFire = global.game.time.now + global.fireRate;
4       var livingEnemies = new Array();
5       global.bugs.forEachAlive(function (aBug) {
6         livingEnemies.push(aBug);
7       });
8       if (livingEnemies.length > 0) {
9         var bug = livingEnemies[Math.floor(Math.random() *
              livingEnemies.length)];
10        if (bug.frame > 10) {
11          var bullet = global.enemyBullets.create(bug.body.x + 10,
                bug.body.y, 'red');
12        } else {
13          var bullet = global.enemyBullets.create(bug.body.x + 10,
                bug.body.y, 'green');
14        }
15
16        bullet.enableBody = true;
17        bullet.checkWorldBounds = true;
18        bullet.outOfBoundsKill = true;
19        bullet.reset(bug.body.x + 10, bug.body.y + 10);
20        global.game.physics.arcade.moveToXY(bullet, bullet.x, 600);
21      }
22
23    }
24  },
```

In der *nextFire*-Variable steht der frühestmögliche Zeitpunkt zum Schießen, so kann die Feuerrate kontrolliert werden. Falls der Zeitpunkt erreicht wurde, wird zuerst der nächste Zeitpunkt gesetzt. Als nächstes soll ein zufälliger Käfer gewählt werden, der den Schuss abfeuert. Alle Käfer befinden sich in der Käfer-Gruppe. Phaser bietet zwar eine *getRandom*-Methode, die einen zufälligen Käfer aus der Gruppe zurückgibt, allerdings könnten auch bereits tote Käfer ausgewählt werden. Deswegen werden alle Käfer in einem Array gespeichert, aus dem in Zeile 9 ein zufälliger Käfer gewählt wird. In den Zeilen 10-14 wird die Projektilfarbe an die Käferfarbe angepasst. Anschließend werden die Eigenschaften für ein Projektil in Zeile 16-18 gesetzt, sodass es sich selbstständig zerstört, sobald es außerhalb des Spielfelds ist. In den Zeilen 19-20 wird die Position des Projektils auf die des Käfers gesetzt und die Schussrichtung nach unten festgelegt.

DER SPEZIAL-KÄFER

Ab der 2. Welle erscheint in jeder Welle ein Spezial-Käfer. Dieser unterscheidet sich optisch durch eine andere Farbe von den anderen Käfer. Solange er am Leben ist, wird im Hintergrund ein buntes Farbmuster abgespielt und die Steuerung für den Spieler invertiert. Stirbt dieser Käfer, verschwindet das Farbmuster und die Steuerung ist wieder wie gewohnt.

Für den Hintergrund erstellen wir ein neues Sprite und fügen einen Phaser-Filter hinzu:

Listing 83: Filter

```
1  global.rainbow = global.game.add.sprite(0, 0);
2  global.rainbow.width = 800;
3  global.rainbow.height = 600;
4
5  global.filter = global.game.add.filter('Plasma', 640, 480);
6  global.rainbow.filters = [global.filter];
```

Der Filter kann über die folgenden Attribute konfiguriert werden:

Listing 84: Filter-Konfiguration

```
1   global.filter.size = 0.08;
2   global.filter.redShift = 0.8;
3   global.filter.greenShift = 0.2;
4   global.filter.blueShift = 0.4;
```

Abbildung 24: Der Filter

Für die Invertierung der Steuerung, gibt es in Player.js ein Attribut, das einfach auf **true** gesetzt werden kann, um die Steuerung zu invertieren.

```
1  global.player.revertMovement = true;
```

In Player.js wird in der *getKey*-Methode auf das revertMovement-Attribut geachtet:

Listing 85: Invertierte Steuerung

```
1     if (global.cursors.left.isDown) {
2       if(this.revertMovement){
3         this.moveRight();
4       }else{
5         this.moveLeft();
6       }
7     } else if (global.cursors.right.isDown) {
8       if(this.revertMovement){
9         this.moveLeft();
10      }else{
11        this.moveRight();
12      }
13    }
```

3.6.3 *Schutzwände*

Für die Schutzwände verwendeten wir kleine Bilder (8x8px), aus denen wir beliebig große Wände zusammensetzen können. Es handelt sich bei ihnen lediglich um eine Gruppe aus Sprites, die durch Kollision mit Spieler- und Käfer-Projektilen zerstört werden. Dadurch, dass wir eine Wand aus mehreren kleinen zusammensetzen, war die Realisierung, eine Wand nur nach und nach zu zerstören, sehr einfach.

Listing 86: Erstellung der Schutzwände

```
1   createWalls: function () {
2     global.wallGroup = this.createGroup();
3     var y = 410;
4     this.createEqualSpaceWalls(5, 80, y);
5     this.createEqualSpaceWalls(5, 64, y-8);
6     this.createEqualSpaceWalls(5, 48, y-16);
7     this.createEqualSpaceWalls(5, 32, y-24);
8
9   },
```

Die Methode *createGroup* ist eine Hilfsmethode zur Erstellung einer Phaser-Gruppe mit Physik:

Listing 87: Helfer-Methode zur Erstellung einer Gruppe

```
1    createGroup: function () {
2      var group = global.game.add.group();
3      group.enableBody = true;
4      group.physicsBodyType = Phaser.Physics.ARCADE;
5      return group;
6    },
```

Bei der Funktion *createEqualSpaceWalls* handelt es sich um eine Hilfsmethode, die wir schrieben, um eine beliebige Anzahl an Wänden gleichmäßig auf die Spielfeldbreite zu verteilen. Man kann ihr die Anzahl an gewünschten Wänden sowie deren jeweilige Breite und Position übergeben.

Listing 88: Helfer-Funktion zur Erstellung von Wänden

```
createEqualSpaceWalls: function(count, singleWidth, y) {
  var width = global.game.width;
  var emptySpace = width - count * singleWidth;
  var spaceBetween = emptySpace / (count);
  console.log(emptySpace);
  console.log(spaceBetween);

  var begin = spaceBetween/2;
  for (var i = 0; i < count; ++i) {
    if (i > 0) {
      begin = begin + singleWidth + spaceBetween;
    }
    this.createWall(begin, y, singleWidth);
  }
},
createWall: function (x, y, length) {
  var singleLength = 8;
  for (var i = x; i < x + length; i = i + singleLength) {
    global.wallGroup.create(i, y, 'wall');
  }
},
```

Der Algorithmus schneidet die Spielfeld-Breite in gleichgroße Stücke, berechnet den korrekten Abstand zwischen den einzelnen Wänden, und platziert entsprechend die Sprites.

Dies ist beispielsweise der Aufbau der Wände nach folgenden vier Aufrufen:

```
this.createEqualSpaceWalls(3, 140, 300);
this.createEqualSpaceWalls(8, 20, 340);
this.createEqualSpaceWalls(15, 20, 380);
this.createEqualSpaceWalls(1, 50, 420);
```

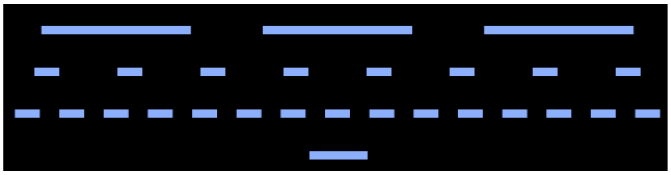


Abbildung 25: Beispiel-Aufbau von Schutzwänden

3.6.4 *Gegenstand: Honig*

Die Honig-Elemente sind Teil einer Gruppe (*slowers*):

```
1 global.slowers = this.createGroup();
```

CreateGroup ist die auf S.96 erwähnte Hilfsmethode zur Erstellung einer Gruppe mit Physik. Wie in den anderen Spielen auch, wird die Erstellung des Honigs über einen Timer in Intervallen geregelt:

```
1 global.game.time.events.loop(global.HONEY_SPAWN_INTERVAL,
      this.spawnRandomHoney, this);
```

Die Methode *spawnRandomHoney* erstellt auf der Höhe des Spielers an zufälliger Position ein Honig-Sprite:

Listing 89: Erstellung der Honig-Objekte

```
1   spawnRandomHoney: function () {
2     var randomX = Math.floor(Math.random() * (global.game.width - 32
          + 1));
3     global.slowers.create(randomX, global.game.height - 32, 'honey')
          ;
4
5   },
```

Das Einsammeln ist über ein Overlap von Spieler und Honig-Gruppe geregelt:

```
1 global.game.physics.arcade.overlap(global.player.sprite,
      global.slowers, this.hitHoney, null, this);
```

Listing 90: Einsammeln von Honig

```
1   hitHoney: function (player, honey) {
2     honey.kill();
3     global.descendValue = 0;
4     global.game.time.events.add(Phaser.Timer.SECOND * 3,
          this.resetDescend, this);
5   },
```

Durch Einsammeln wird der in Kapitel 3.6.2.1 (S. 91) erklärte *descendValue* auf 0 gesetzt, was das Voranschreiten der Käfer verhindert, und ein Timer gestartet, der nach 3 Sekunden diesen Wert wieder auf seinen ursprünglichen Wert zurücksetzt:

Listing 91: Rücksetzen des descendValues

```
1   resetDescend: function () {
2     global.descendValue = global.DESCEND_STANDARD;
3   }
```

3.7 LAUNCHER-STARTSKRIPT IN RAHMENLOSER ANWENDUNG

3.7.1 *Launcher*

Für den Launcher erstellten wir eine HTML-Seite, die eine Liste mit Links zu den drei Spielen enthält.

Listing 92: Launcher: Liste mit Links für die Spiele

```
...
<h1>Wähle ein Spiel</h1>

<div class="games">
  <ul>
   <li><a href="bugitar/index.html">Bugitar</a></li>
   <li><a href="racing/index.html">Site Point Arena</a></li>
   <li><a href="revenge/index.html">The Revenge of the Bugs</a></li>
  </ul>
</div>
...
```

Falls in Zukunft ein weiteres Spiel hinzugefügt werden sollte, kann dieses sehr einfach als weiterer Listeneintrag hinzugefügt werden.

Damit die Umlaute richtig dargestellt werden, muss mit dem *meta*-Tag die UTF-8-Zeichenkodierung angegeben werden.

Um dem ganzen ein besseres Aussehen zu verleihen, erstellten wir ein CSS-Sheet.

Listing 93: Launcher: Inkludierung des CSS-Sheets

```
...
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" type="text/css" href="style.css">
  ...
</head>
...
```

Listing 94: Launcher: Das CSS-Sheet für den Launcher

```
body {
  background: #333;
}
h1 {
  text-align: center;
  color: whitesmoke;
  font-family: impact;
}

.games {
  font-family: impact;
  width: 700px;
  margin: 0 auto;
}

.games a {
  display: table-cell;
  vertical-align: middle;
  text-align: center;
  height: 200px;
  width: 200px;
  text-decoration: none;
  color: darkslategray;
  font-size: 1.5em;
}

li {
  display: inline-block;
  border: 2px solid rgba(0,0,0,0);
  background: rgba(213, 255, 158, 0.6);
}

li.selected {
  border: 2px solid greenyellow;
}

li.selected a {
  color: greenyellow;
}
```

Dadurch wird die Liste blockweise dargestellt und ein selektieres Element durch die Klassen *li.selected* und *li.selected a* hervorgehoben.

Da der Launcher auch ohne Tastatur und Maus bedienbar sein muss, verwendeten wir jQuery [22], um auf Tastatureingaben flexibel reagieren zu können[1].

1 Als Hilfe diente dieser Beitrag: http://stackoverflow.com/a/8902976

Abbildung 26: Der gestylte Launcher

Listing 95: Inkludierung von jQuery

```
...
<head>
  ...
  <script src="jquery-2.1.4.min.js"></script>
</head>
...
```

Listing 96: Das Skript für die Tastatureingaben

```
1  ...
2  <script>
3    var li = $('li');
4    var liSelected = li.eq(0).addClass('selected');
5
6    $(window).keydown(function (key) {
7      if (key.which === 39) {
8        if (liSelected) {
9          liSelected.removeClass('selected');
10         next = liSelected.next();
11         if (next.length > 0) {
12           liSelected = next.addClass('selected');
13         } else {
14           liSelected = li.eq(0).addClass('selected');
15         }
16       } else {
17         liSelected = li.eq(0).addClass('selected');
18       }
19     } else if (key.which === 37) {
20       if (liSelected) {
21         liSelected.removeClass('selected');
22         next = liSelected.prev();
23         if (next.length > 0) {
24           liSelected = next.addClass('selected');
25         } else {
26           liSelected = li.last().addClass('selected');
27         }
28       } else {
29         liSelected = li.last().addClass('selected');
30       }
31     } else if (key.which === 13) {
32       if ( ($(".selected > a").attr('href')) !== undefined ) {
33         window.location.href = $(".selected > a").attr('href');
34       }
35     }
36   });
37 </script>
38 ...
```

Zunächst wird das erste Listenelement mit der CSS-Klasse *selected* versehen (Z. 4). Somit gilt bei Start der Anwendung das erste Spiel als momentan ausgewählt. Es wird mit der Funktion *keydown* auf die Pfeiltasten Links und Rechts mittels der entsprechenden Keycodes (39 und 37) reagiert (Z. 8 und 20). Dabei werden die List-Elemente nacheinander mit der CSS-Klasse *selected* versehen, um ein Navigieren durch die Liste mit den Pfeiltasten zu erlauben. Dem jeweils zuvor ausgewählten Element wird die CSS-Klasse *selected* wieder genommen.

Mit der Enter-Taste (Keycode 13) wird geprüft, ob ein Spiel selektiert ist (Z. 33), und, falls ja, zu diesem gewechselt (Z. 34).

Natürlich muss auch aus den jeweiligen Spielen ohne Tastatur oder

Maus wieder zurück in den Launcher gewechselt werden können. Dazu haben wir auch in den einzelnen Spielen jQuery verwendet und folgendes Skript eingefügt:

Listing 97: Das Skript zum Zurückwechseln in den Launcher

```
...
<script>
  $(window).keydown(function (e) {
    if (e.which === 27) {
      window.location.href = "../index.html";
    }
  });
</script>
...
```

Der Keycode 27 entspricht der Escape-Taste. Der Vorteil bei der Nutzung von Keycodes ist das leichte Austauschen zu jeder beliebigen Taste durch Ändern des Keycodes, was eine schnelle Anpassung an die Tasten des Automaten erlaubt.

3.7.2 *Startskript*

Der Launcher soll beim Start des Systems automatisch geöffnet werden. Das Problem hierbei ist, dass der Raspberry Pi zunächst ohne grafische Oberfläche gestartet wird, wir aber eine grafische Anwendung starten wollen. Da wir nicht die ersten waren, die vor dieser Aufgabe standen, fanden wir im Raspberry Pi Forum eine Anleitung dazu [37].

Zuerst erstellen wir einen Eintrag in */etc/rc.local*. Alle Einträge werden automatisch beim Systemstart geladen.

Listing 98: Eintrag /etc/rc.local

```
su -s /bin/bash -c startx pi&
```

Mit diesem Kommando wird eine Session für den Nutzer *pi* erstellt, in der die Bash-Shell gestartet und dort das Kommando *startx* ausgeführt wird, was eine neue X-Session im X-Window-System erzeugt. Es ist wichtig, dass das Kommando als Hintergrund-Prozess ausgeführt wird, da sonst der Boot-Prozess nicht abgeschlossen werden kann. Deswegen steht das *&* hinter dem Befehl.

Standardmäßig darf der Boot-Prozess keinen X-Server, der zur Erstellung einer grafischen Benutzeroberfläche benötigt wird, erstellen. Um es zu erlauben, muss man folgenden Befehl ausführen:

Listing 99: x11-common

```
dpkg-reconfigure x11-common
```

Danach öffnet sich ein Fenster, in dem man die Option *Anybody* wählt.

Sobald der X-Server startet, lädt er die Konfigurationsdatei *xinitrc*, die sich im Home-Verzeichnis des jeweiligen Nutzers befindet. In dieser Datei können nun unsere Befehle eingetragen werden, um unseren Datenbank-Server und den Launcher zu starten:

Listing 100: xinitrc

```
1 ...
2 sudo node /home/pi/retropi/DatabaseServer/Main.js &
3 exec /home/pi/retropi/nwjs/nw /home/pi/retropi
```

Der erste Befehl startet den Datenbank-Server und der zweite schließlich den Launcher mit Hilfe von *Nw.js*.

3.8 DATENBANK-ANBINDUNG MIT NODE.JS

3.8.1 *Erstellen der Datenbank*

Für jedes Spiel gibt es in der Datenbank eine Highscore-Tabelle und eine Hall-of-Fame-Tabelle. Die werden wie folgt erstellt:

Listing 101: Erstellen der Tabellen

```
1 create: function() {
2   self.createTables = function(){
3     self.db.serialize(function(){
4       self.db.run("CREATE TABLE IF NOT EXISTS " + name + " (id
            INTEGER PRIMARY KEY   AUTOINCREMENT, user TEXT, points
            INTEGER)");
5       self.db.run("CREATE TABLE IF NOT EXISTS " + name + "_Hof (id
            INTEGER PRIMARY KEY   AUTOINCREMENT, user TEXT, points
            INTEGER)");
6     });
7   };
8 },
```

3.8.2 *Routen*

Die Routen werden mit Hilfe von *Express.js* erstellt:

Listing 102: Konfigurieren von Express

```
1 self.restapi = self.express();
2 self.bodyParser = require('body-parser');
3 self.restapi.use(self.bodyParser.json());
4 self.restapi.use(self.cors());
```

Der *bodyParser* stellt einen JSON-Parser für den Body bereit, um so einfach auf den Request-Body zuzugreifen. Da unser Spiel unter der Domain A läuft und für die Datenbank auf Domain B zugreift, entsteht der folgende Fehler:

```
1 XMLHttpRequest cannot load http://localhost:3000/bugitar. Origin
      http://localhost is not allowed by Access-Control-Allow-Origin.
```

Dieses Vorhaben wird aus Sicherheitsgründen verhindert, um es dennoch zu erlauben verwenden wir *Cors*.

Eine neue GET-Route kann so angelegt werden:

Listing 103: GET-Route

```
1  self.restapi.get(route, function(req, res){
2         self.db.all("SELECT * FROM " + routeName + " ORDER BY points
               DESC", function(err, result){
3             res.json(result);
4         });
5     });
```

Wird die Route aufgerufen, wird die Methode ausgeführt und der Datenbankinhalt wird gesendet. Um andere HTTP-Verben zu nutzen, ersetzt man *restapi.get* durch das entsprechende Verb.

4 TESTS UND DEPLOYMENT

4.1 TESTS

In Zeiten von Test-Driven-Development stehen Tests an oberster Stelle. Code soll niemals ungetestet in das Live-System integriert werden. Deswegen analysierten wir, wie unsere Spiele am sinnvollsten getestet werden können.

4.1.1 *Keine herkömmlichen Unit-Tests?*

Das Phaser-Framework stellt keine Testsuite zur Verfügung, deshalb mussten wir nach Alternativen suchen. Bis auf sehr wenige Versuche aus der Community, eine minimale Testsuite zur Verfügung zu stellen, fanden wir nichts. Diese wurden entweder nicht weiterentwickelt oder wurden ohne Dokumentation veröffentlicht. In manchen Fällen trafen beide Punkte zu. Um jedoch Fehler in unserem Spiel finden zu können, benötigten wir eine Alternative. Wir entschieden uns dazu, wie in Spielen üblich, eine Alpha- und Beta-Phase zu starten. Dadurch entstand ein weiterer großer Vorteil: Nutzer informierten uns nicht nur über Fehler, sondern gaben uns auch Rückmeldung zur Umsetzung. So konnte der Schwierigkeitsgrad exakter angepasst werden und Verbesserungsvorschläge umgesetzt werden.

ALPHA-PHASE

Die Test-Phase startete mit der Alpha-Phase, in der die Entwickler, also wir, die Spiele testeten. Das Ziel der Phase ist es, grobe Fehler zu beseitigen, sodass das Spiel vom Nutzer gespielt werden kann.

BETA-PHASE

In der Beta-Phase wurde das Spiel schließlich von anderen Nutzern getestet. Da jeder Spieler ein Spiel auf eine andere Art und Weise spielt, können nicht alle Fehler von einer oder zwei Personen entdeckt werden. Durch die Beta-Phase konnten einige Fehler entfernt werden und der Schwierigkeitsgrad angepasst werden. Einige Fehler, die durch die Beta-Phase gefunden wurden:

- Das Spiel Bugitar stürzt ab, nachdem ein Käfer von seinem Weg abkommt
- Die Restart-Funktion von Bugitar funktioniert manchmal nicht
- In Site Point Arena kann man auf einem Pfeil-Feld stehen bleiben, wenn man in die entgegengesetzte Richtungs-Taste drückt

- Die Portale in Site Point Arena teleportieren manchmal falsch
- Der Spezial-Modus in Revenge of the Bugs startet manchmal nicht

Einige Verbesserungsvorschläge:

- In Bugitar ist es möglich alle Käfer zu töten, wodurch das Spiel nicht mehr weitergespielt werden kann
- Mehrere Leben für Bugitar einfügen
- Eine größere Spielfläche für Site Point Arena

Die meisten Fehler traten in unserem ersten Spiel Bugitar auf, was daran lag, dass wir noch wenig Erfahrung mit dem Framework und Spieleentwicklung im Allgemeinen hatten. Mit jedem Spiel wurden wir sicherer und konnten im Vorfeld Fehler vermeiden, indem wir mögliche Fehlerursachen vermieden.

4.2 DEPLOYMENT

4.2.1 *Development-Phase*

4.2.1.1 *Warum wird ein Webserver benötigt?*

Der Grund, warum man die HTML-Seite eines Spiels nicht einfach im Browser öffnen und dann spielen kann, geht aus der Browser-Sicherheit hervor. Denn in jedem Browser herrscht die *Same-origin policy*[54], die besagt, wie ein Dokument oder Skript mit Ressourcen aus einer anderen Quelle interagieren darf. Somit wird JavaScript die Möglichkeit verwehrt, auf Ressourcen anderer Quellen zuzugreifen.

4.2.1.2 *Wann sind zwei Quellen gleich?*

Zwei Quellen sind dann gleich, wenn sie das gleiche Protokoll, den gleichen Port und den gleichen Host besitzen [54].

4.2.1.3 *Was hat dies mit Phaser zu tun?*

Würde man die HTML-Seite eines Spiels einfach im Browser öffnen, wäre sie unter dem **file://** Protokoll geladen. Browser haben sehr hohe Sicherheits-Standards unter diesem Protokoll. Da Phaser jedoch viele Ressourcen, wie Bilder, Sounds oder JSON-Dateien, laden muss, muss es ungehindert durch die starken Sicherheitsmaßnahmen der Browser agieren können. Dies ist nur unter dem **http://** Protokoll möglich, weshalb ein lokaler Webserver benötigt wird.

4.2.1.4 *Unterschiede bei verschiedenen Browsern*

Bei der Entwicklung fiel uns auf, dass wir die Spiele durchaus durch normales Öffnen im Browser spielen konnten. Dies beschränkte sich jedoch nur auf die beiden Browser Firefox und Safari. Google Chrome ließ das aufgrund der *Same-origin policy* nicht zu:

Abbildung 27: Chrome verhindert aus Sicherheitsgründen das Laden aus dem Dateisystem

Das liegt daran, dass jeder Browser die Sicherheits-Vorkehrungen zur Same-origin policy anders löst [53]. Um nicht auf unerwartete Fehler zu stoßen, entwickelten wir dennoch, auch bei Nutzung von

Firefox oder Safari, mit dem lokalen Webserver XAMPP (siehe 3.2.1.2 S.35).

4.2.2 *Produktiv-Phase*

In der Produktiv-Phase kommt die rahmenlose Anwendung mit Hilfe von NW.js (2.6) zur Geltung. Die Phaser-Engine funktioniert reibungslos in einer NW.js-Applikation[61] und das Starten ist zudem erstaunlich einfach.

Da NW.js intern einen eigenen Server nutzt, benötigt man keinen weiteren Server wie XAMPP, wenn man die Spiele in der rahmenlosen Anwendung spielen möchte.

5 FAZIT

5.1 SPIELEENTWICKLUNG MIT HTML5 UND JAVASCRIPT

Jetzt, da alle drei Spiele fertig implementiert sind, sehen wir uns in der Lage, ein Fazit zu ziehen.

5.1.1 *Warum sollte man ein Spiel in HTML5 entwickeln?*

Kurz gesagt: Weil es gut ist. Wir waren erstaunt darüber, wie reibungslos HTML5-Spiele funktionieren. Dies ist wohl der Tatsache, dass HTML5 zusammen mit JavaScript plattformunabhängig in fast allen Browsern funktioniert, zu verdanken. Wir entwickelten auf drei verschiedenen Betriebssystemen (OS X, Ubuntu 14.04, Windows 10) und in allen drei gab es keine Komplikationen.

Was uns außerdem besonders gefällt: Kein Flash-Player mehr. Er hat sich in der vergangenen Zeit nicht nur durch Sicherheitslücken immer unbeliebter gemacht, sondern auch durch seine ständigen Versuche, bei jedem Update weitere Software zu installieren. Sogar Werbebranchen erkennen den Trend zu HTML5 und steigen langsam aber sicher darauf um[73]. Zudem ist mit HTML5 im Gegensatz zu Flash die Unterstützung mobiler Endgeräte gewährleistet[55].

Außerdem gefällt uns die Möglichkeit, ein solches Spiel mit Hilfe von *NW.js* oder *Electron* als Desktopanwendung zu starten.

Wir sind gespannt, was die HTML5-Spieleentwicklung in Zukunft noch bringen wird. Sicher werden sogar 3D-Spiele an Beliebtheit gewinnen. Diese sind dank WebGL zwar heute schon möglich, allerdings gibt es derzeit nur wenige Beispiele für 3D-HTML5-Spiele. Es gibt allerdings Beispiele, die das Potenzial von HTML5-Spielen deutlich machen [67].

5.2 PROBLEME

5.2.1 *Debugging*

Um Fehlersuchen möglichst kurz halten zu können, ist es immer von Vorteil, einen Debugger bereit zu haben. Obwohl Phaser einen Debugger bereitstellt, benutzten wir hauptsächlich den normalen Debugger von JavaScript, den man mittels **debugger;** an eine beliebige Stelle im Code setzen kann, oder verwendeten simple Textausgaben im Spiel selbst, um uns bestimmte Informationen während des Spielverlaufs anzeigen lassen zu können.

Der Debugger von JavaScript bietet den Vorteil, das Spiel an einem beliebigen Ereignis anhalten zu können, um sich dann alle nötigen Informationen über Variablen und Objekte in der Browser-Konsole anschauen zu können.

5.2.2 *Raspberry Pi und JavaScript*

In unserer Analyse-Phase (Siehe Kapitel 2.4.3.3 haben wir bereits auf das Problem mit JavaScript hingewiesen. Wir hoffen, dass die nächsten Webbrowser-Releases die GPU nutzen und den Einsatz von aufwendigen JavaScript-Applikationen ermöglichen. Da JavaScript immer populärer wird und nicht mehr wegzudenken ist [74], sind wir optimistisch und denken, dass es nur noch eine Frage der Zeit ist.

5.2.3 *Lernkurve bei der Entwicklung*

Neue Technologien lernt man erst, wenn man sie nutzt. So waren wir zu Beginn unserer Entwicklung mit dem Phaser-Framework noch unerfahren und nutzten nicht das volle Potential des Frameworks. Mit jedem weiteren Spiel lernten wir das Framework besser kennen und konnten unsere Anforderungen mit den von Phaser bereitgestellten Funktionen umsetzen. Dadurch verkürzte sich nicht nur unser Entwicklungszyklus, sondern die Anzahl der Fehler verringerte sich.

5.3 AUSBLICK AUF DIE ZUKUNFT

5.3.1 *Einbindung weiterer Spiele*

Es ist denkbar, dass die Firma Site Point in Zukunft weitere Spiele einbinden möchte. Dies ist kein Problem, da jedes Spiel eine eigene Komponente ist, die ohne weiteres dem Launcher hinzugefügt werden kann (Siehe 3.7.1 S.99).

5.3.2 *Erweiterung vorhandener Spiele*

Die Erweiterung oder Änderung eines vorhandenen Spiels gestaltet sich nicht ganz so einfach, da ein Spiel prinzipiell ein geschlossenes System zusammenhängender Elemente ist.

Allerdings haben wir bei der Implementierung stets darauf geachtet, so zu programmieren, dass Änderungen bezüglich Leveldesign mit nur wenig Aufwand möglich sind. So haben wir beispielsweise Helfer-Funktionen (z.B. zur Portal-Erstellung Seite 77) zur Level-Gestaltung geschrieben, die es erlauben, die Struktur des Spiels durch einfache Änderungen im Code an eigene Wünsche anzupassen. In Bugitar kann beispielsweise auch eine andere Karte, also ein anderes JSON-Dokument, verwendet werden.

ANHANG

A

SHOWCASE

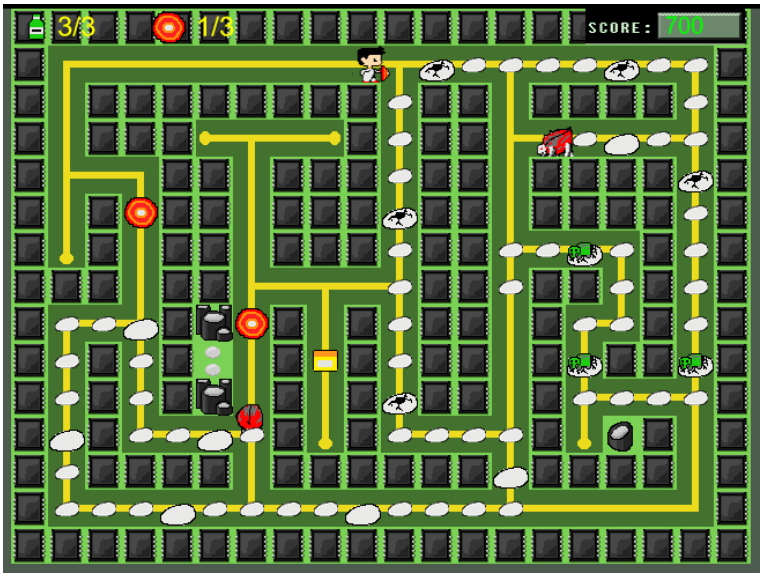


Abbildung 28: Bugitar Showcase

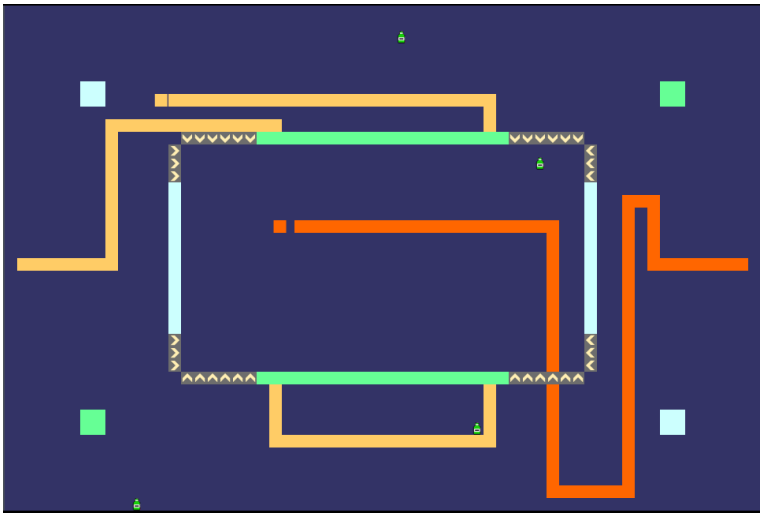


Abbildung 29: Site Point Arena Showcase

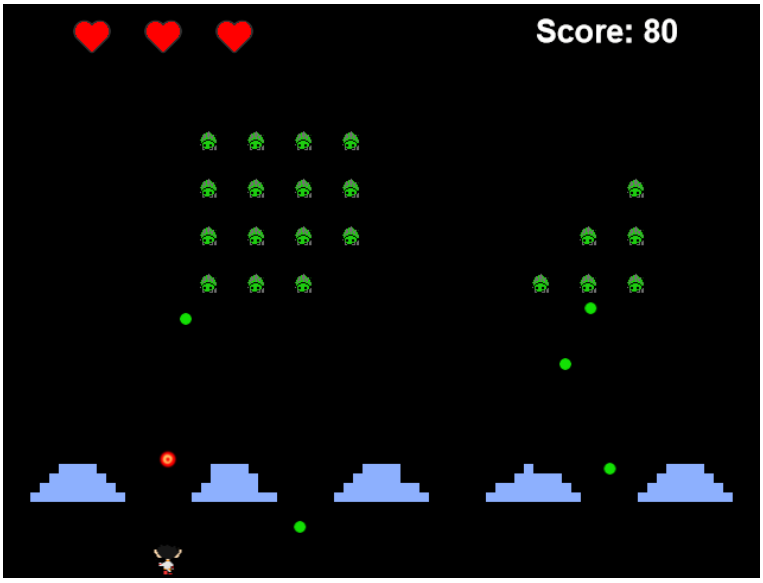


Abbildung 30: Revenge Of The Bugs Showcase

B

INHALT DER CD

Auf der beigelegten CD befindet sich folgender Inhalt:

- Quellcode zu Bugitar
- Quellcode zu Site Point Arena
- Quellcode zu Revenge Of The Bugs
- Quellcode zu Datenbank-Server
- Quellcode zu Launcher

LITERATURVERZEICHNIS

[1] Manifest für agile Softwareentwicklung. http://agilemanifesto.org/iso/de/ (Abgerufen am 19.09.2015.

[2] Microsoft ASP.NET Website. www.asp.net/web-pages (Abgerufen am 22.09.15).

[3] Atom Website. https://atom.io/ (Abgerufen am 24.09.15).

[4] Bitbucket Website. https://bitbucket.org (Abgerufen am 24.09.15).

[5] Brackets Project. . https://github.com/adobe/brackets-shell (Abgerufen am 19.09.15).

[6] Brackets Website. . http://brackets.io (Abgerufen am 24.09.15).

[7] The Chromium Projects. https://www.chromium.org (Abgerufen am 21.09.15).

[8] Scirra Construct2. https://www.scirra.com/construct2 (Abgerufen am 21.09.15).

[9] Duden Online, Stichwort: Retro. . http://www.duden.de/rechtschreibung/Retro_Nachahmung_Reproduktion (Abgerufen am 24.09.15).

[10] Duden Online, Stichwort: Spiel. . http://www.duden.de/rechtschreibung/Spiel (Abgerufen am 24.09.15).

[11] Createjs EaselJs. http://createjs.com/easeljs (Abgerufen am 21.09.15).

[12] ECMAScript 5 compatibility table. . http://kangax.github.io/compat-table/es5/ (Abgerufen am 21.09.2015).

[13] ECMAScript 6 compatibility table. . http://kangax.github.io/compat-table/es6/ (Abgerufen am 21.09.2015).

[14] Electron Website. http://electron.atom.io (Abgerufen am 19.09.15).

[15] Epiphany Webbrowser. https://wiki.gnome.org/action/show/Apps/Web?action=show&redirect=Apps%2FEpiphany (Abgerufen am 21.09.15).

[16] Git Website. . https://git-scm.com (Abgerufen am 24.09.15).

[17] RetroPie-Setup. *RetroPie*, . https://github.com/RetroPie/RetroPie-Setup/wiki (Abgerufen am 19.09.2015).

[18] Github Website. . https://github.com (Abgerufen am 24.09.15).

[19] Which HTML5 Game Engine is right for you? *HTML5 Game Engines*. http://html5gameengine.com/ (Abgerufen am 19.09.2015).

[20] Debian Mozilla team. http://mozilla.debian.net (Abgerufen am 21.09.15).

[21] ImpactJs. http://impactjs.com (Abgerufen am 21.09.15).

[22] Jquery Website. http://jquery.com (Abgerufen am 24.09.15).

[23] LimeJs. http://www.limejs.com (Abgerufen am 21.09.15).

[24] Mapeditor Website. http://www.mapeditor.org/ (Abgerufen am 24.09.15).

[25] MYSQL Website. . https://www.mysql.de (Abgerufen am 22.09.15).

[26] MySQL Download. . http://dev.mysql.com/downloads/repo/apt/ (Abgerufen am 21.09.15).

[27] Node.js Builds. https://nodejs.org/dist/v0.10.28/ (Abgerufen am 21.09.15).

[28] Arm build NW.js. . https://github.com/nwjs/nw.js/issues/1151#issuecomment-78030735 (Abgerufen am 19.09.15).

[29] List of apps and companies using nw.js. . https://github.com/nwjs/nw.js/wiki/List-of-apps-and-companies-using-nw.js (Abgerufen am 19.09.15).

[30] NW.js Website. . http://nwjs.io (Abgerufen am 19.09.15).

[31] Pacman, Original Pac Man. http://pacman.com/ (Abgerufen am 21.09.2015).

[32] Phaser State Methods. . http://phaser.io/docs/2.3/Phaser.State.html#methods (Abgerufen am 24.09.15).

[33] Phaser Tween API. . http://phaser.io/docs/2.4.3/Phaser.Tween.html (Abgerufen am 24.09.15).

[34] Phaser. . http://phaser.io (Abgerufen am 21.09.15).

[35] Raspberry PG. http://raspberrypg.org (Abgerufen am 21.09.15).

[36] Raspberry Pi - Teach, Learn, and Make with Raspberry Pi. . https://www.raspberrypi.org/ (Abgerufen am 21.09.2015).

[37] Graphical Application without Desktop Environment. . https://www.raspberrypi.org/forums/viewtopic.php?p=344408 (Abgerufen am 24.09.15).

[38] Raspbian. . https://www.raspbian.org (Abgerufen am 21.09.15).

[39] umbraco Webdesign Webseiten Homepage - Site Point Saarbrücken. http://www.sitepoint.de/ (Abgerufen am 21.09.2015).

[40] Space Invaders Example. . http://phaser.io/examples/v2/games/invaders (Abgerufen am 24.09.15).

[41] Space Invaders. . http://spaceinvaders.net/ (Abgerufen am 21.09.2015).

[42] SQLite Website. . https://www.sqlite.org (Abgerufen am 22.09.15).

[43] Debian Wheezy SQLite. . https://packages.debian.org/de/wheezy/sqlite3 (Abgerufen am 21.09.15).

[44] Subversion Website. http://subversion.apache.org (Abgerufen am 24.09.15).

[45] Coding your first HTML5 Game. *Library*. http://teamtreehouse.com/library/coding-your-first-html5-game (Abgerufen am 19.09.2015).

[46] Trello Website. https://trello.com (Abgerufen am 24.09.15).

[47] Fltron - Light Cycle and Tron Games. http://www.fltron.com/ (Abgerufen am 21.09.2015).

[48] Umbraco Website. www.umbraco.com (Abgerufen am 22.09.15).

[49] Webstorm Website. https://www.jetbrains.com/webstorm/ (Abgerufen am 24.09.15).

[50] Web Hypertext Application Technology Working Group. https://whatwg.org/ (Abgerufen am 21.09.2015).

[51] XAMPP Website. https://www.apachefriends.org/de/index.html (Abgerufen am 24.09.15).

[52] X_arcade. *Arcade Joysticks*. http://shop.xgaming.com (Abgerufen am 19.09.2015).

[53] Security in depth: Local web pages. *Chromium Blog*, 04.12.2008. http://blog.chromium.org/2008/12/security-in-depth-local-web-pages.html (Abgerufen am 26.09.2015).

[54] Same-origin policy. *Web Security | MDN*, 07.07.2015. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy (Abgerufen am 26.09.2015).

[55] Adobe beendet flash-entwicklung für smartphones. *Spiegel*, 2012. http://www.spiegel.de/netzwelt/gadgets/adobe-stellt-flash-entwicklung-fuer-smartphones-ein-a-850332.html (Abgerufen am 26.09.2015).

[56] Phaser 2.4.3 api docs. *Phaser - Learn*, 2015. http://phaser.io/docs (Abgerufen am 23.09.2015).

[57] Mihaly Csikszentmihalyi über Flow. *TED Talks*, Februar 2014. https://www.ted.com/talks/mihaly_csikszentmihalyi_on_flow?language=de (Abgerufen am 19.09.2015).

[58] Ernest Adams. *Fundamentals of Game Design*. New Riders, Berkely, CA 94710, 2. edition, 2010.

[59] Rain Anderson. What makes a game retro? *thatvideogameblog*, 20.04.2008. http://www.thatvideogameblog.com/2008/04/20/what-makes-a-game-retro/ (Abgerufen am 21.09.2015).

[60] Valentin Bojinov. *RESTful Web API Design with Node.js*. Packt Publishing, 2015.

[61] Cheng Zhao. node-webkit: app runtime based on Chromium and node.js. https://speakerdeck.com/zcbenz/node-webkit-app-runtime-based-on-chromium-and-node-dot-js (Abgerufen am 24.09.15).

[62] Douglas Crockford. Javascript: The world's most misunderstood programming language. *Crockfords JavaScript Blog*, 2001. http://javascript.crockford.com/javascript.html (Abgerufen am 22.09.2015).

[63] Klaus Dembowski. *Raspberry Pi - Das technische Handbuch*. Springer Vieweg, Hamburg, Deutschland, 2. edition, 2015.

[64] ECMA. Ecmascript 5.1 language specification. *Standard ECMA-262*, Juni 2011. http://www.ecma-international.org/ecma-262/5.1/ (Abgerufen am 21.09.2015).

[65] ECMA. Ecmascript 2015 (6th edition) language specification. *Standard ECMA-262*, Juni 2015. http://www.ecma-international.org/publications/standards/Ecma-262.htm (Abgerufen am 22.09.2015).

[66] Node.js Foundation. Node.js Website. https://nodejs.org/ (Abgerufen am 22.09.15).

[67] Gavin. 25 Best HTML5 Games for 2014. https://codegeekz.com/25-best-html5-games-for-2014/ (Abgerufen am 24.09.15).

[68] Johan Huizinga. *Homo Ludens*. 1938/39.

[69] Fionn Kelleher. The state of desktop applications in node.js. *NodeSource*, 15.10.2014. https://nodesource.com/blog/node-desktop-applications (abgerufen am 20.09.2015).

[70] TJ Holowaychuk und Nathan Rajlicj Mike Cantelon. *Node.js in Action*. Manning, 2013.

[71] Roxanne Peterson. What makes a design retro? *ProductiveDreams*, 15.07.2011. http://www.productivedreams.com/what-makes-a-design-retro-vintage/#comments (Abgerufen am 21.09.2015).

[72] Steve Rabin. *Introduction to Game Development*. Charles River Media, Boston, MA02210, USA, 2. edition, 2010.

[73] Ingo Rentz. Bye bye flash: Warum html5 zum standard in der online-werbung wird. *Horizont Online*, 04.08.2015. http://www.horizont.net/marketing/nachrichten/Bye-Bye-Flash-Warum-HTML5-zum-Standard-in-der-Werbemitteltechnologie-wird-135632 (Abgerufen am 22.09.2015).

[74] Rob Beschizza. Javascript is the most popular language at GitHub. https://boingboing.net/2015/04/21/javascript-is-the-most-popular.html (Abgerufen am 24.09.15).

[75] Satoru Iwata. Game Developers Conference. 2005. http://gdcvault.com/play/1014847 (Abgerufen am 21.09.15).

[76] W3C Schools. Html5 audio. *HTML5 Tutorial*, 2015. http://www.w3schools.com/html/html5_audio.asp (Abgerufen am 23.09.2015).

[77] StrongLoop. Express.js Website. http://expressjs.com (Abgerufen am 22.09.15).

[78] Eben Upton. Web browser released! *Official Raspberry Pi Blog*, 01.09.2014. https://www.raspberrypi.org/blog/web-browser-released/ (abgerufen am 19.09.2015).

[79] W3C. Html5. *All Standards and Drafts*, 28.10.2014. https://www.w3.org/TR/html5/ (Abgerufen am 21.09.2015).

KOLOPHON

Dieses Dokument wurde mit der LaTeX -Vorlage `classicthesis` von André Miede angefertigt.

`https://bitbucket.org/amiede/classicthesis/wiki/Home`

Zur Bearbeitung wurde der Online-Dokumentbearbeiter *„Overleaf“* verwendet.

`https://www.overleaf.com/`

SELBSTÄNDIGKEITSERKLÄRUNG

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde. Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“ zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, September 2015

David Landry

Michael Reimsbach