Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Projektarbeit

Andreas Müller, Claus Torben Haug, Jan-Dennis Bartels, Marjan Bachtiari

MBC-Ping-Pong

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Andreas Müller, Claus Torben Haug, Jan-Dennis Bartels, Marjan Bachtiari

MBC-Ping-Pong

Projektarbeit eingereicht im Rahmen der Wahlpflichtfach

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke

Eingereicht am: 28. Dezember 2016

Andreas Müller, Claus Torben Haug, Jan-Dennis Bartels, Marjan Bachtiari

Thema der Arbeit
MBC-Ping-Pong

Stichworte
Ping-Pong, NodeJS, JavaScript, WebRTC

Kurzzusammenfassung
In diesem Dokument wird das Projekt im MBC-Ping-Pong, das im Rahmen des Wahlpflichtfaches Modernebrowserkommunikation an der HAW-Hamburg erstellt wird, behandelt. Hierbei handelt es sich um eine Pong Clone welcher auf einem zentralen Bildschirm spielbar ist und mnittels WebRTC gesteuert wird. Es wird auf die Architektur, JavaScript, Frontend und Backend eingegangen.
Andreas Müller, Claus Torben Haug, Jan-Dennis Bartels, Marjan Bachtiari

Title of the paper
MBC-Ping-Pong

Keywords
Ping-Pong, NodeJS, JavaScript, WebRTC

Abstract
This document is about the Project MBC-Ping-Pong, which is made for the elective course Modernebrowserkommunikation at HAW-Hamburg. Its about a Pong clone, played on a central screen nd controled via WebRTC. The architecture, JavaScript, frontend and backend are discussed.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Architektur

1.1 Arbeitsablauf zur Bearbeitung eines Issue

Um eine erfolgreiche Zusammenarbeit zu gewährleisten, sind allgemein gültige Regeln nötig. Insbesondere wird festgelegt, wie die einzelnen Arbeitsschritte ablaufen sollten, um ein Issue zu bearbeiten. Zudem werden weiterhin die Zuständigkeiten geregelt.

1.1.1 Verwaltung der zu bearbeitenden Issues

Die zu bearbeitenden Issues werden auf GitHub unter Issues (https://github.com/Transport-Protocol/MBC-Ping-Pong/issues) gepflegt. Um den Verlauf eines Issues darzustellen wird das Kanbanboard von GitHub (https://github.com/Transport-Protocol/MBC-Ping-Pong/projects) genutzt.

1.1.2 Erstellen der zu bearbeitenden Issues

Prinzipiell kann und darf jedes Projektmitglied zu jeder Zeit Issues erstellen. Gerade bei Bugs ist dies ein gewünschtes vorgehen. In der Regel sollten dies jedoch aus Gruppensitzungen hervorgehen und durch den Architekten ausformuliert werden.
Ein Issue besteht aus drei Absätzen:

- **Beschreibung**
  In der Beschreibung wird allgemein auf den Kontext des Issues eingegangen.
- **Anforderung**
  In Anforderung wird die Zielvision dargestellt.
- **Abnahmekriterien**
  In Abnahmekriterien werden alle Punkte aufgeführt, die notwendig sind, um das Issue als erfolgreich bearbeitet anzusehen.

1.1.3 Das Kanbanboard

Das Kanbanboard ist in fünf Abschnitte eingeteilt:

- **Selected for Development**
  Diese Spalte enthält alle Issues, die der Architekt zur Bearbeitung in nächster Zeit ausgewählt hat. Hier enthaltene Issues sind entweder durch den Architekten einem bestimmten Teammitglied zugeordnet. Diese sollten dann auch vorrangig bearbeitet werden. Oder (dies sollte der Normalfall sein) sie sind niemandem zugeordnet, dann kann sich jedes Teammitglied entscheiden, ob er das Issue bearbeitet. Gründe für das direkte zuweisen können unteranderem sein, dass es eine entsprechende vorhergehende Absprache gab, dass der Architekt das Issue speziell einem Bereich zugehörig sieht bzw. eine spezielle Paarung erreichen möchte, oder aber auch, weil ein Issue schon zu lange in SSelected for Development"verweilt. Hat sich ein Teammitglied für ein Issue entschieden, trägt er sich als Bearbeiter ein und zieht es in auf Ïn Development".

- **In Development**
  In dieser Spalte verweilen alle Issues, an denen gerade entwickelt wird. Wenn die Entwicklung an einem Issue abgeschlossen ist, zieht der Bearbeiter das Issue weiter auf "Needs Review".

- **Needs Review**
  Hier verweilen alle Issues, deren Entwicklung abgeschlossen ist, aber noch nicht geprüft wurde, ob die Abnahmebedingungen erfüllt sind. Normalerweise sollte die Abnahme durch den Architekten erfolgen. Issues, die der Architekt bearbeitet hat, muss das Review von einem anderen Teammitglied gemacht werden. Ein Issue bei dem das Review durchgeführt wird, wird in die Spalte Ïn Review"verschoben.

- **In Review**
  Hier sind alle Issues enthalten, die sich gerade im Review befinden. Sind alle Abnahmekriterien erfüllt, und sind durch die Bearbeitung des Issue keine neuen Probleme/Fehler hinzugekommen, wird es in die Spalte "Done"verschoben und das Issue geschlossen. Ist dies nicht der Fall, wird ein Entsprechender Kommentar mit einer möglichst detaillierten Beschreibung des Problems an das Issue angehängt, und es wieder auf Ïn Development"geschoben.

- **Done**
  Diese Spalte enthält alle abgeschlossenen Issues.

1.1.4 Git

Hier sind die Verhaltensweisen für die Nutzung von Git aufgeführt. Alles hier nicht aufgeführte kann von jedem Teammitglied nach eigenen ermessen gehandhabt werden.

- **Branches**
  Für jedes Issue wird ein Branch erstellt, außer es handelt sich um reine Dokumentation (im Ordner Docu). Ein Branchname folgt folgendem Muster: "#<IssueNummer> <KurzerName>". Dadurch lässt sich
- **Commits**
  Commits folgen folgendem Namensschema: "#<IssueNummer> <Beschreibung>".
- **Push und Pull**
  Es sollte möglichst häufig gepusht werden, um einen eventuellen Datenverlust zu vermeiden. Beim Pull sollte mit -rebase"gearbeitet werden, um die Historie möglichst sauber zu halten.
- **Merge und Pullrequest**
  Bevor ein Issue auf "Needs Review"geschoben wird, ist der Master in den Branch zu mergen und ein Pullrequest (https://github.com/Transport-Protocol/MBC-Ping-Pong/pulls) zu erstellen. Derjenige der das Issue reviewt hat, merget den Branch dann mithilfe des Pullrequests in den Master und löscht ihn.

1.2 Meilensteine

In diesem Abschnitt werden die Meilensteine festgelegt. Hierbei wird beschrieben, was wan erreicht sein sollte.

1.2.1 Projekt Aufsetzen

- **Beschreibung**
  Die Grundlegenden für die Entwicklung notwendigen anfangs Infrastrukturen sind aufgesetzt.
- **Kriterien**
  - **NodeJS-Server aufsetzen**
    Der NodeJS Server ist aufgesetzt und stellt eine statische Website zur Verfügung

  - **Docker**
    Eine einheitliche Umgebung wird durch Docker und Docker-Compose ermöglicht.

- **Beendet:** 25.11.2016

1.2.2 Prototyp (Technik)

- **Beschreibung**
  Um die identifizierten technischen Risiken schnellst möglich in den Griff zu bekommen, werden diese möglichst früh bearbeitet. In dem Prototyp (Technik) soll gezeigt werden, das die kritische Technik funktioniert. Dies wird anhand von kleine losgelösten Beispielen, die aber nahe der Zielarchitektur sind gezeigt.

- **Kriterien**
  - **Darstellung**
    Es wird gezeigt, das im Webbrowser eine Flüssige Darstellung möglich ist.
  - **Kollisionserkennung**
    Es wird gezeigt, dass eine Kollisionserkennung erreichbar ist.
  - **Kommunikation mittels WebRTC**
    Architektur bedingt ist die Nutzung von WebRTC unumgänglich. Es ist zu zeigen, dass eine Verbindung von mehreren Handys zum Darstellungsmedium möglich ist.
  - **Steuerung**
    Die Steuerung soll über den Touchscreen geschehen. Es ist zu zeigen, dass es möglich ist, die Position des Fingers auf dem Touchscreen im Browser auszulesen.
  - **Größen der Handys/Tabletts**
    Unterschiedliche Handys und Tabletts haben verschiedene Größen und Formen. Somit ist ein Konzept zu erarbeiten, welches diesem Problem bei der Steuerung gerecht wird.

- **Beendet:** 16.12.2016

1.2.3 Release 1.0 (Zwei Spieler)

- **Beschreibung**
  In diesem ersten Release ist eine Basisversion des Spieles implementiert. Es können zwei Spieler gegeneinander Spielen, indem sie ihre Schläger mit den Handys steuern. Gleichzeitig ist diese Version die minimal Version und enthält alle "MustFeatures.

- **Kriterien**
  - **Schläger**
    Für jeden Spieler existiert ein Schläger, der mit dem Handy Steuer
  - **Ball**
    Es gibt ein Ball, der sich über das Spielfeld bewegt. Kollidiert er mit einem Schläger oder der Wand, an der sich kein Schläger befindet, prallt er davon ab. Es gilt hierbei, dass der Einfallwinkel dem Ausfallwinkel entspricht. Zudem beschleunigt der Ball, wenn er mit einem Schläger kollidiert. Wenn der Ball mit der Wand hinter einem Schläger kollidiert, wir er in die Ausgangsposition versetzt und erhält die Ausgangsgeschwindigkeit.
  - **Punkte**
    Immer wenn der Ball mit der Wand hinter einem Schläger kollidiert, erhält der andere Spieler einen Punkt.
  - **Spielende**
    Das Spiel endet automatisch nach X Spielen (wobei gilt: $X \in \mathbb{N} \wedge X \mod 2 = 1$).
    Das genaue X ist noch zu definieren.
- **Beendet:** 13.01.2017

1.2.4 Release 1.X (Diverse Features)

- **Beschreibung**
  Basierend auf der Version 1.0 wird das Spiel weiterentwickelt. Jedoch sind alle Features die hier bearbeitet werden können "CanFeatures. Daher kann es sein, dass das Release 1.X äquivalent zu dem Release 1.0 ist. Zudem sind alle hier genannten möglichen Features noch nicht näher spezifiziert und auf ihre Machbarkeit geprüft. Es gilt jedoch, dass je Umgesetztes Feature die Versionsnummer im Minorbereich um Eins steigt.
- **mögliche Kriterien**
  - **N Spieler**
    Mehr als 2 Spieler
  - **Zusätzliche Hindernisse**
    Auf dem Spielfeld sind zusätzliche Hindernisse.
  - **Highscore-Liste**
    Es wird eine Highscore-Liste geführt und angezeigt.

  - **TBD**
    To be discussed.

- **Beendet:** 24.02.2017

1.3 Highlevel View

In diesem Abschnitt wird die Grob-/Gesamtarchitektur betrachtet. Hierbei wird nicht nur auf wesentliche Schnittstellen und Komponenten eingegangen. Zur engeren Auswahl standen zwei mögliche Ansätze. Es werden beide betrachtet, und erläutert warum der Ansatz 2 umgesetzt werden wird.

1.3.1 Ansatz 1

Der Ansatz 1 verfolgt den Klassischen Client-Server-Ansatz. Hierbei dient der Server als zentrale Instanz, die alle signifikanten Logikoperationen übernimmt. Die Clients dienen lediglich zur Ein-/Ausgabe.

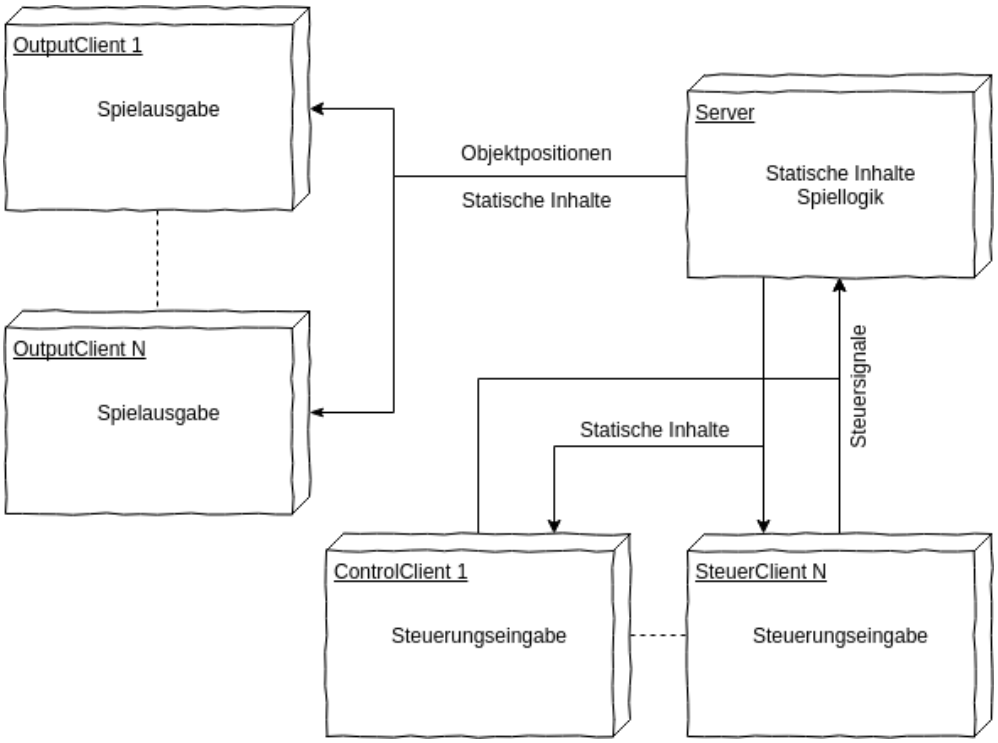

Abbildung 1.1: Highlevel Ansatz 1

Server

Der Server stellt die statischen Inhalte (HTML, CSS, JS) bereit. Zudem enthält er die gesamte Spiellogik. Der Server empfängt Steuerinformationen vom ControlClient, verarbeitet diese und sendet einen aktuellen Gamestate an den OutputClient.

ControlClient

Der ControlClient erfasst die Steuereingaben des Nutzers und sendet sie an den Server.

OutputClient

Der OutputClient empfängt den Gamestate vom Server und aktualisiert die Anzeige entsprechend.

Analyse

Einerseits ist dieser Ansatz architektonisch sehr einfach umzusetzen, da es eine zentrale Instanz gibt und Seperation of Concerns Architektur bedingt unterstützt wird. Zudem ist es möglich ein Spiel auf mehreren OutputClients darzustellen. Da sich die Clients mit dem Server verbinden, ist der Einsatz von WebSockets möglich. Mit socket.io gibt es eine sehr gute Abstraktionsschicht für WebSockets. Andererseits kann durch den Einsatz von WebSockets ein nicht zu vernachlässigendes Delay entstehen, da diese auf TCP basieren. Um diesem zu begegnen ist der Einsatz des WebRTC Protokollstacks notwendig. Zudem muss der Server die Zuordnung der ControlClients und OutputClients zu einem Spiel organisieren. Außerdem sind zwei Netzwerkübertragungen von Nöten, damit die Eingabe des Spielers auf dem OutputClient sichtbar wir. Dadurch wird das Netzwerk doppelt belastet, und es ist zweimal das Übertragungsdelay vorhanden.

1.3.2 Ansatz 2

Server

Der Server stellt die statischen Inhalte (HTML, CSS, JS) bereit. Zudem fungiert er als zentrale Instanz für den WebRTC-Protokollstack

ExternerServer

Der externe Server ist ein öffentlicher Stun-/Turn-/Ice-Server, der beispielsweise von Google bereitgestellt wird.

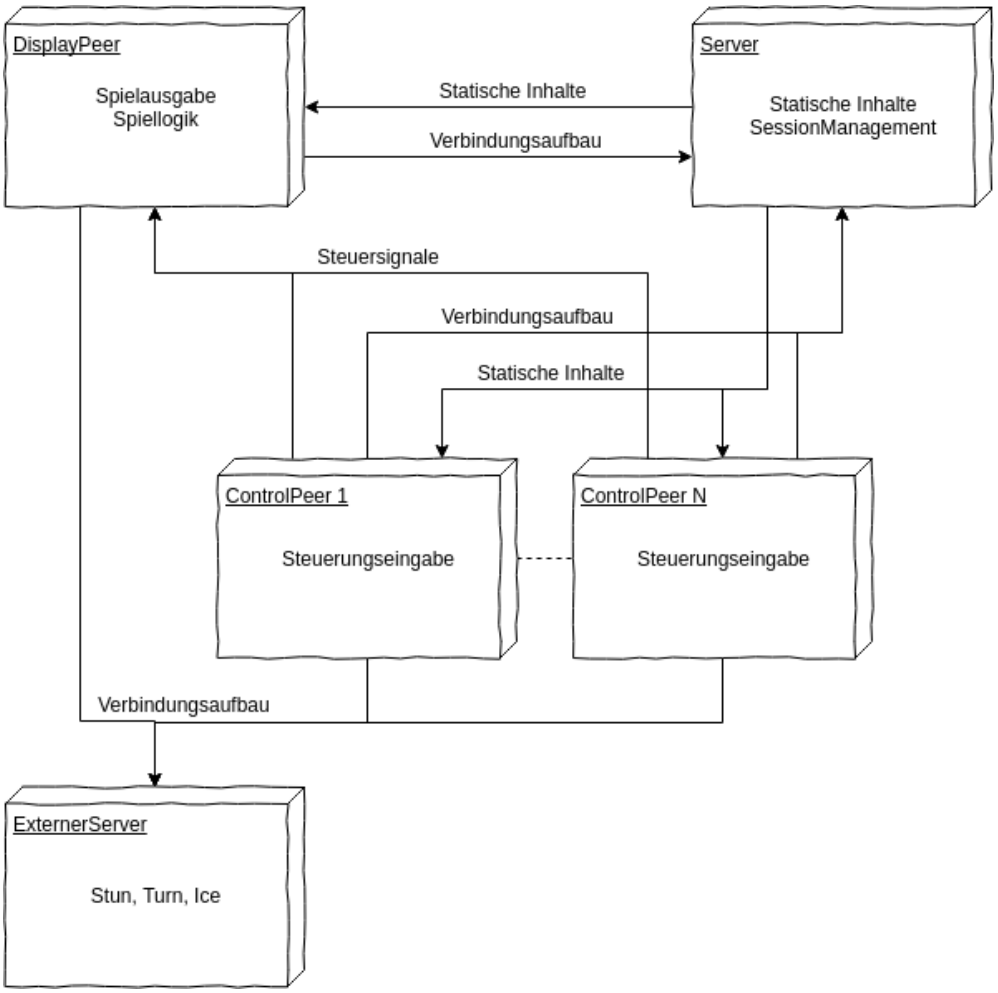

Abbildung 1.2: Highlevel Ansatz 2

ControlPeer

Der ControlPeer erfasst die Steuereingaben des Nutzers und sendet sie an den DisplayPeer.

DisplayPeer

Der DisplayPeer ist ein Fat-Client. Er enthält die gesamte Spiellogik und empfängt über den WebRTC-Protokollstack direkt die Steuereingaben des Spielers. Zudem zeigt er das Spiel an.

Analyse

Der Ansatz 2 ist architektonisch recht Komplex, da die Aufteilung auf Client und Server wegfällt, und somit die native Unterstützung von Seperation of Concern nicht gegeben ist. Es muss während der Entwicklung verstärkt darauf geachtet werden, das die Trennung von Spiellogik und Ausgabe eingehalten wird. Zu dem ist man auf den WebRTC-Protokollstack angewiesen, da die Peers direkt miteinander Kommunizieren müssen. Außerdem ist man auf einen einzelnen DisplayPeer beschränkt. Andererseits erfolgt die Übertragung der Steuerung direkt von dem ControlPeer an den DisplayPeer, dadurch ist das kleinst mögliche Delay zwischen Eingabe, Verarbeitung und Ausgabe gewährleistet. Außerdem ermöglicht der Einsatz von WebRTC den Einsatz von UDP als Transportprotokoll, wodurch das Delay weiter verringer werden kann, da UDP Verbindungslos ist.

1.3.3 Fazit

Auch wenn der Ansatz 2 zunächst komplexer erscheint und einen geringeren Funktionsumfang bietet, da nur ein AusgabeClient pro Spiel unterstützt wird und WebRTC eingesetzt werden muss, überwiegen doch die Vorteile dieses Ansatzes. Durch die fehlende zweite Netzwerkübertragung und der mögliche Einsatz von UDP, werden Delays minimiert. Gleichzeitig wird mehr Rechenleistung auf die Clients ausgelagert und eine aufwändige Verwaltung, welcher Spieler zu welchem Spiel gehört ist auch nicht notwendig.

1.4 Risiken

In diesem Kapitel wird auf die Risiken eingegangen, die zu Schwierigkeiten bei der Projektdurchführung führen können. Die Auswirkungen und Eintrittswahrscheinlichkeit werden in 3 Kategorien eingeteilt: "1:Gering, 2:Mittel, 3: Hoch". Das potenzielle Risiko ist das Produkt aus Auswirkungen und Eintrittswahrscheinlichkeit.

1.4.1 Technische Risiken

Zunächst wird auf die technischen Risiken eingegangen

Darstellung ist nicht möglich

- **Beschreibung:**
  Gerade bei Ping-Pong wird die Bewegung des Balls irgendwann sehr schnell. Dies muss trotzdem im Browser darstellbar sein, ohne das es ruckelt.

- **Eintrittsgründe;**
  - Das gewählte Grafikframework ist nicht leistungsfähig genug.
  - Das gewählte Grafikframework wurde nicht richtig genutzt.
  - Das Zielsystem ist nicht Leistungsfähig genug.
- **Folgen:**
  - Das Spielen ist nicht möglich => Projektfehlschlag.
- **Eintrittswahrscheinlichkeit:** 2
- **Auswirkungen:** 3
- **Risiko:** 6
- **Maßnahmen:**
  - Bereits im Prototyp eine Beispielimplementierung durchführen.
  - Möglichst früh einen Test auf der Zielplattform absolvieren.

Kollisionserkennung funktioniert nicht

- **Beschreibung:**
  Durch die schnelle Ballbewegung bei Ping-Pong ist eine gute Kollisionserkennung notwendig.
- **Eintrittsgründe;**
  - Die gewählte Physicsengine ist nicht leistungsfähig genug.
  - Die gewählte Physicsengine wurde nicht richtig genutzt.
  - Das Zielsystem ist nicht Leistungsfähig genug.
- **Folgen:**
  - Das Spielen ist nicht möglich => Projektfehlschlag.
- **Eintrittswahrscheinlichkeit:** 2
- **Auswirkungen:** 3
- **Risiko:** 6
- **Maßnahmen:**

  - Bereits im Prototyp eine Beispielimplementierung durchführen.
  - Möglichst früh einen Test auf der Zielplattform absolvieren.

Kommunikation mittels WebRTC funktioniert nicht

- **Beschreibung:**
  Um die Übertragung in akzeptabler Geschwindigkeit zu gewährleisten, ist der Einsatz von WebRTC unausweichlich. WebRTC ist jedoch absolutes Neuland für das gesamte Team.
- **Eintrittsgründe;**
  - WebRTC wird nicht korrekt genutzt.
  - Im Zielnetzwerk wird die Verwendung durch Firewalls behindert.
- **Folgen:**
  - Die Architektur muss von Fat-Client Ansatz auf einen Serverzentrierten Ansatz umgestellt werden => Projektverzögerung.
- **Eintrittswahrscheinlichkeit:** 2
- **Auswirkungen:** 2
- **Risiko:** 4
- **Maßnahmen:**
  - Bereits im Prototyp eine Beispielimplementierung durchführen.
  - Möglichst früh einen Test auf der Zielplattform absolvieren.

Steuerung ist nicht möglich

- **Beschreibung:**
  Das auslesen der Position des Fingers auf dem Bildschirm über den Browser funktioniert nicht, bzw. nicht schnell genug.
- **Eintrittsgründe;**
  - Zu altes Handy verwendet (Browser unterstützt es nicht).
  - Schnittstelle nicht korrekt verwendet

- **Folgen:**
  - Das Spiel ist nicht Steuerbar => Projektfehlschlag.
- **Eintrittswahrscheinlichkeit:** 1
- **Auswirkungen:** 3
- **Risiko:** 3
- **Maßnahmen:**
  - Bereits im Prototyp eine Beispielimplementierung durchführen.

1.4.2 Konzeptuelle Risiken

Steuerung ist nicht Fair

- **Beschreibung:**
  Gerade mit dem Prinzip des Browser basierten Ansatzes werden sehr viele unterschiedliche Endgerätetypen angesprochen. Jedes Endgerät hat jedoch eine andere Bildschirmgröße und Pixeldichte.
- **Eintrittsgründe;**
  - Es möchten Personen mit unterschiedlichen Endgeräten gegeneinander antreten
- **Folgen:**
  - Das Spiel ist unfair => geringere Akzeptanz.
- **Eintrittswahrscheinlichkeit:** 3
- **Auswirkungen:** 1
- **Risiko:** 3
- **Maßnahmen:**
  - Das Risiko wird durch die weitergehende Analyse, und dem Entwurf eines möglichst Fairen Steuerungskonzeptes verringert.
  - Ggf. zum Teil ignorieren

1.4.3 Organisatorische Risiken

Temporärer Personalausfall

- **Beschreibung:**
  Es kann jederzeit zu einem temporären Personalausfall kommen.
- **Eintrittsgründe;**
  - Krankheit.
  - Klausuren.
  - Unmotiviertheit.
- **Folgen:**
  - Arbeitskraftmangel => Projektverzögerung.
  - Fachwissensmangel => Projektverzögerung.
- **Eintrittswahrscheinlichkeit:** 3
- **Auswirkungen:** 1
- **Risiko:** 3
- **Maßnahmen:**
  - Puffer einplanen.
  - Möglichst wenig Must-Anforderungen definieren.

Kompletter Personalausfall

- **Beschreibung:**
  Es kann jederzeit zu einem temporären Personalausfall kommen.
- **Eintrittsgründe;**
  - Krankheit.
  - Unmotiviertheit.
- **Folgen:**
  - Arbeitskraftverlust => Projektverzögerung.
  - Fachwissensverlust => Projektverzögerung.

- **Eintrittswahrscheinlichkeit:** 2
- **Auswirkungen:** 2
- **Risiko:** 4
- **Maßnahmen:**
  - Puffer einplanen.
  - Möglichst wenig Must-Anforderungen definieren.

1.5 Reviews

1.5.1 Auswahl einer geeigneten 2D-/Physiksengine

- **Datum:** 05.12.16
- **Prüfung der Anforderungen:**
  - ✓ Aufruf der Seite http://<IpDesNodeJsServers>:<HttpPortDesNodeJsServers>/test2DEngine.html zeigt die Testseite.
  - ✓ Ein quadratischer Bereich mit einer Form in der Mitte auf dem Bildschirm wird dargestellt.
  - ✓ In diesem Bereich bewegt sich eine Form (am besten eine Kugel) waagerecht von Links nach Rechts.
  - ✓ Kollidiert die Form mit der rechten Wand, bewegt sich die Form von Rechts nach Links.
  - ✓ Kollidiert die Form mit der linken Wand, bewegt sich die Form von Links nach Rechts.
- **Prüfung des Codes:**
  - Der Code ist strukturiert und lesbar.
  - Bis auf 2 Zeilen ist der Code verständlich:
    - this.ball.body.immovable = true;
      Ein besserer Kommentar, warum wäre gut gewesen. Nach einigem suchen, gehe ich davon aus, dass nur Objekte, die mit dem Ball kollidieren, beeinflusst werden, aber nicht der Ball selbst.

Dieses Verhalten ist zunächst OK, sollte jedoch bei späteren Features noch einmal geprüft werden.

* this.ball.body.bounce.set(1);
  Es funktioniert in der momentan verwendeten Version, daher ist es OK, sollte jedoch geprüft werden. Laut Dokumentation (http://phaser.io/docs/2.6.2/Phaser.Physics.Ninja.Body
  Sollte der Wert zwischen 0 und 1 liegen, also nicht 1.0 sein. Empfohlen wird ein wert von 0.999. Eine Prüfung ergab, das auch Werte oberhalb von 1.0 akzeptiert werden. Dies führt dann zu einer Beschleunigung des Balls.

1.6 Prototyp

In diesem Kapitel wird der Prototyp behandelt.

1.6.1 Schnittstellen

Dieses Diagramm (Abbildung 1.3) dient dem gemeinsamen Verständnis, welche Klasse welche Schnittstelle bietet.

1.6.2 Fachlicher Ablauf

Im fachlichen Ablauf (Abbildung 1.4) wird dargestellt, wie sich das Programm verhalten soll. Es wird jedoch nicht auf genaue Implementierungsdetails wert gelegt. Er dient dem allgemeine Verständnis, wie der Programmablauf sein soll. Auch die Funktionsnamen bei fremd-APIs dienen lediglich der Beschreibung, welches Verhalten von der API erwartet wird. **Allerdings sollte der Ablauf später im Programmcode ersichtlich sein.**

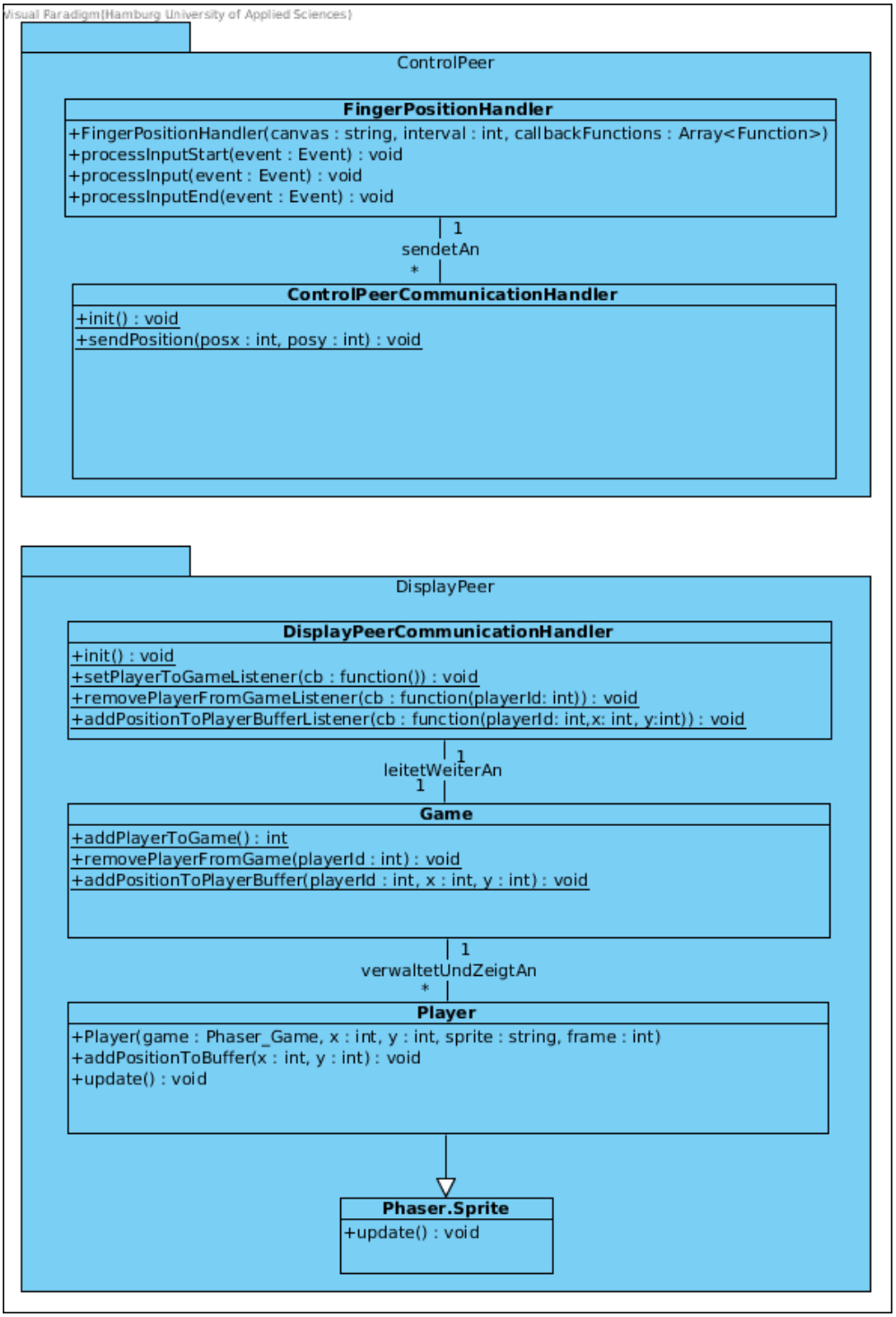

Abbildung 1.3: Prototyp Schnittstellen - Klassendiagram

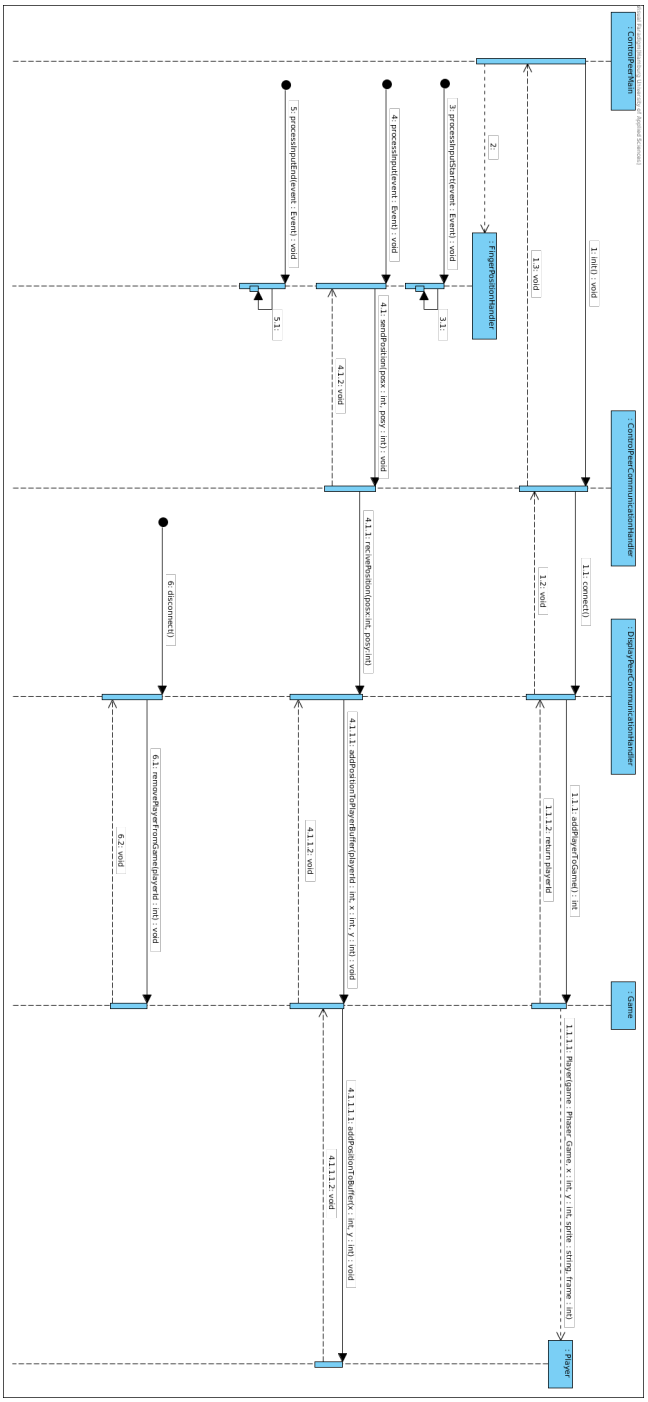

Abbildung 1.4: Prototyp fachlicher Ablauf - Sequenzdiagramm

1.7 Release 1.0

1.7.1 MookUps

Die Mookups (1.5 - 1.9) zeigen die Zustände des Spiels und die einzelnen Komponenten (in den MookUps durch grüne Namen gekennzeichnet) werden beschrieben. Im Abschnitt 1.7.2 wird gezeigt, wie die einzelnen States in einander Überführt werden.

Beschreibung der Komponenten

- **Spieler 1:** Spieler, der den Schläger auf der linken Spielfeldseite steuert.
- **Spieler 2:** Spieler, der den Schläger auf der rechten Spielfeldseite steuert.
- **Spielfeld:** Bereich, in dem das Spiel durch Phaser dargestellt wird.
- **Wand 1:** Wand hinter dem Schläger 1. Wird diese getroffen, erhält Spieler 2 einen Punkt.
- **Wand 2:** Wand hinter dem Schläger 2. Wird diese getroffen, erhält Spieler 1 einen Punkt.
- **Wand 3:** Seitenwände, die lediglich zur Spielfeldbegrenzung dienen.
- **Schläger 1:** Schläger der durch Spieler 1 gesteuert wird. Er kann sofort nach der Erstellung durch Spieler 1 bewegt werden.
- **Schläger 2:** Schläger der durch Spieler 2 gesteuert wird. Er kann sofort nach der Erstellung durch Spieler 2 bewegt werden.
- **Ball:** Der Ball bewegt sich über das Spielfeld, und muss mit den Schlägern davon abgehalten werden, die Wand hinter den Schlägern zu treffen. Trifft der Ball Wand 1 oder 2, wird er in die Ausgangsposition gesetzt und erhällt seine Ausgangsgeschwindigkeit, sowie eine zufällig Richtung. Trifft er Wand 3 oder einen Schläger, prallt er von diesem ab und wird beschleunigt.
- **Punkte:** Anzeige des aktuellen Spielstandes, im Format <Treffer auf Wand2> : <Treffer auf Wand 1>.
- **JoinGamePanel:** Hier werden alle Informationen angezeigt, die benötigt werden, um einem Spiel beizutreten.
  - **QR-Code:** URL 1 als QR-Code
  - **URL 1:** URL, um dem Spiel beizutreten.

- **Timer:** Restzeit bis zum Eintreten eines Events.

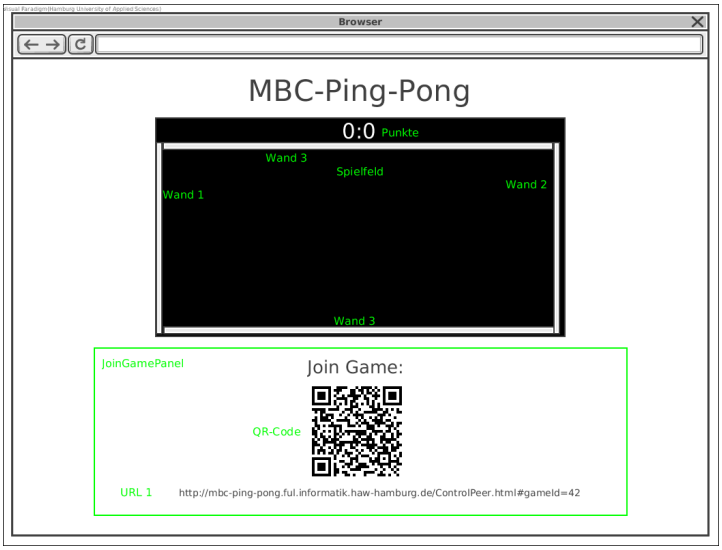

Abbildung 1.5: Mookup - Initial

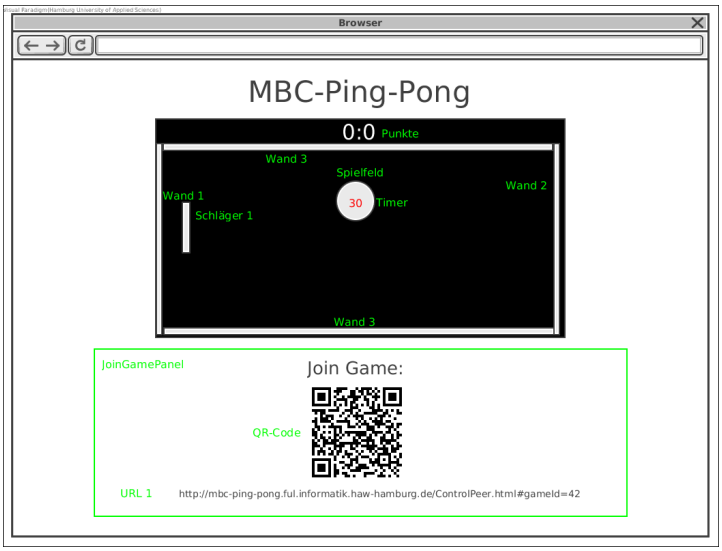

Abbildung 1.6: Mookup - FirstPlayerConnected

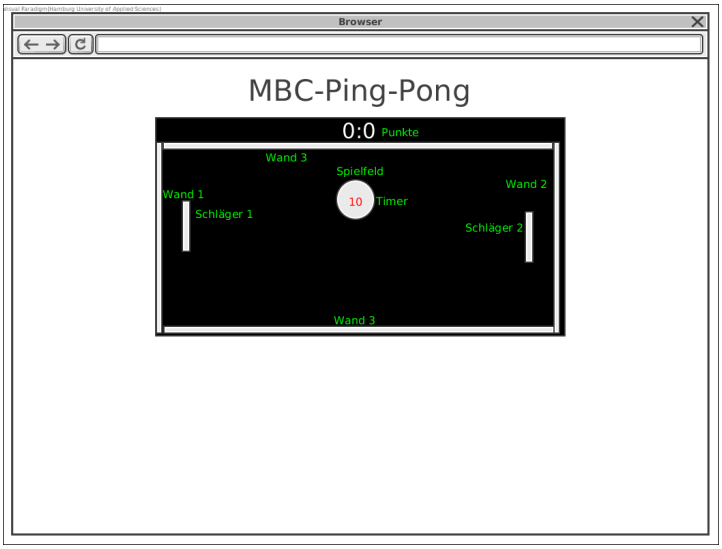

Abbildung 1.7: Mookup - SecondPlayerConnected

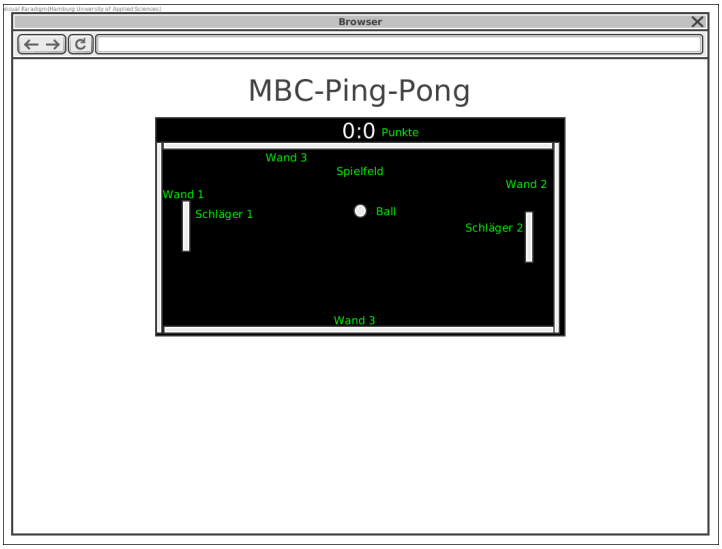

Abbildung 1.8: Mookup - GameRunning

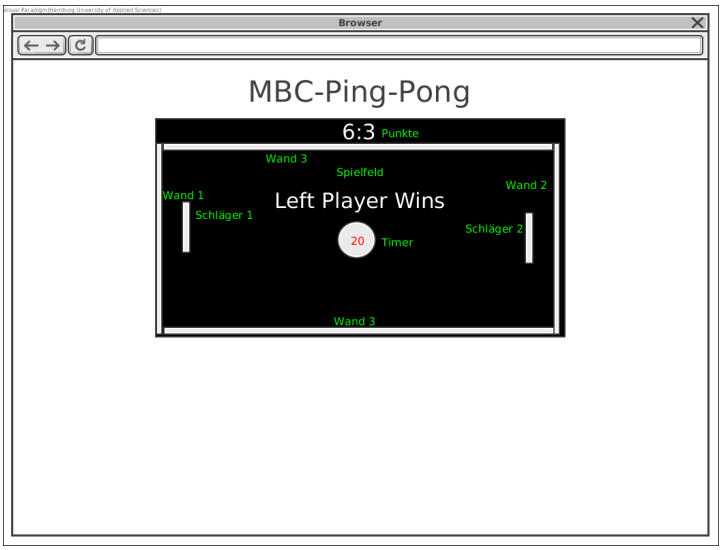

Abbildung 1.9: Mookup - GameEnded

1.7.2 States

Hier wird gezeigt, wie die States ineinander über gehen. Zu den grünen Zuständen gibt es Mookups in 1.7.1.

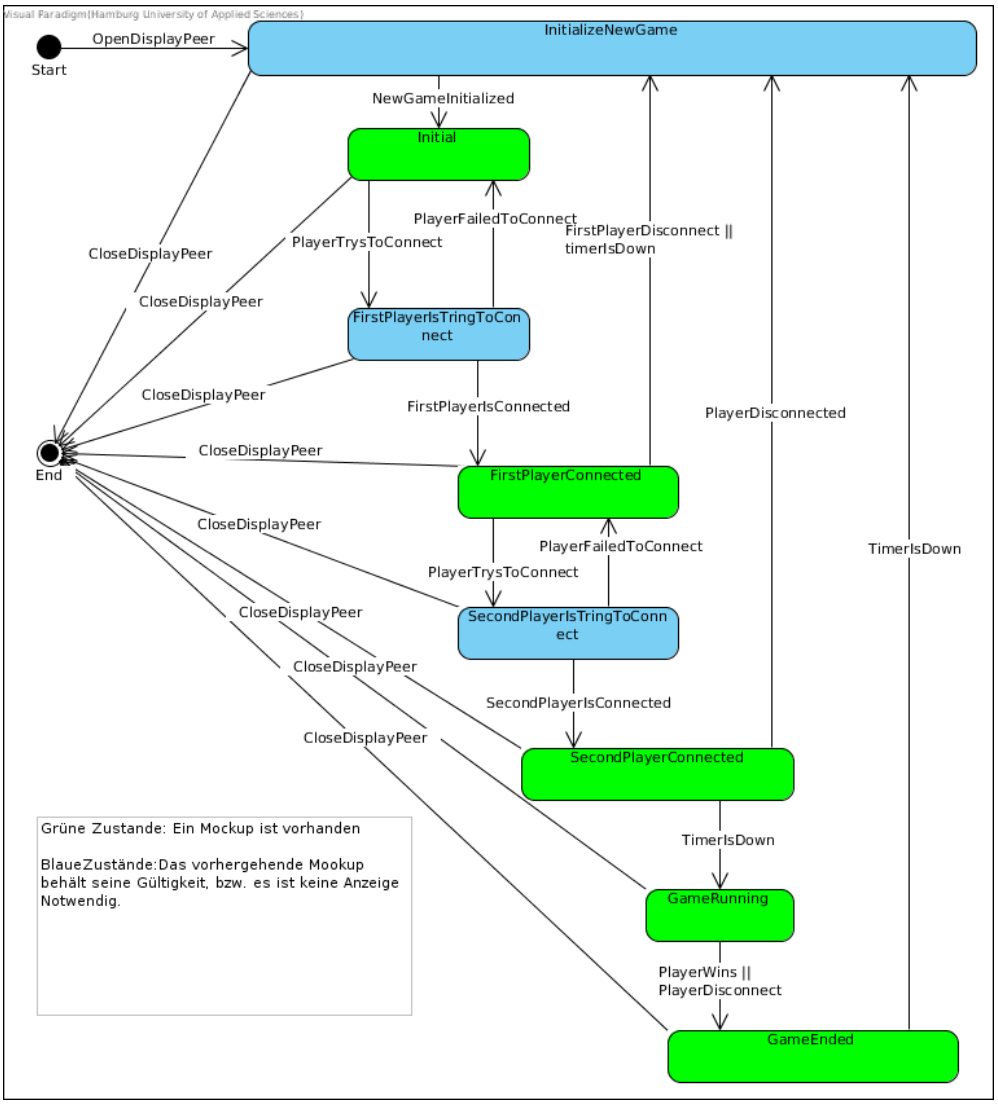

Abbildung 1.10: Statemachine Release 1.0

2 JavaScript

2.1 Auswahl einer Physics Engine

Die Darstellung auf dem Anzeigegerät ist über den DOM nicht möglich, da die Bewegung des Balls und der Schläger zu schnell werden. Zudem wird auch die Physik auf dem Anzeigegerät berechnet. Hierfür ist eine Kollisionserkennung erforderlich.

2.1.1 Anforderungen

- **JavaScript Game Engine**
  Da wir im Backend einen NodeJS Server stehen haben und im Frontend ebenfalls JavaScript nutzen, ergibt es sich von selbst, dass wir eine JavaScript Game Engine brauchen.
- **Unterstützung von 2D Graphics**
  Wir wollen unser Spiel in 2D umsetzen, insofern ist 3D-Unterstützung nicht notwendig.
- **Physik**
  Nicht jede Game Engine unterstützt auch Physik, in unserem Spiel sind allerdings physikalische Gegebenheiten zu beachten wie zB der richtige Ein- und Austrittswinkel des Balls. Da der thematische Schwerpunkt unseres Projektes auf den browserspezifischen Gegebenheiten liegt, wollen wir uns nicht mit umfangreicher Implementierung der Physik beschäftigen.
- **Kollisionserkennung**
  Ebensowenig unterstützt jede Physics Engine auch Kollisionserkennung. Diese ist allerdings ein zentraler Punkt des Spiels, sowohl die Kollisionen des Balls mit den Schlägern, als auch mit dem Spielfeldrand und später ggf. weiteren Bällen.
- **Opensource**
  Wir sind alle arme Studenten und wollen bzw können daher nicht Geld für unser Uniprojekt ausgeben. Hinzu kommt, dass man bei Opensource-Projekten den Sourcecode einsehen kann, was in vielen Situationen hilfreich ist.

- **Performance**
  Auch wenn wir nur ein sehr kleines Spiel bauen, so soll es doch so gut wie möglich in Echtzeit reagieren können, da Verzögerungen sofort auffallen. Die Physics Engine muss somit auch schnell die jeweiligen Positionen der Spielobjekte berechnen und darstellen können.

- **Dokumentation**
  Eine vorhandene und idealerweise auch gute Dokumentation lässt die Lernkurve zu Beginn steiler sein und ist auch bei später ggf. auftretenden Problemen wünschenswert.

- **Community**
  Wir wollen eine Engine aussuchen, die auch wirklich genutzt wird, und bei der es idealerweise auch einen entsprechenden Support aus der Community gibt. Dies ist ebenfalls nützlich im Hinblick auf zukünftige Schwierigkeiten.

2.1.2 Verglichene Engines

Die meisten Engines mussten nur kurz überflogen werden, da sie mindestens eine der Anforderungen nicht erfüllt haben. Es gab natürlich noch weitere, allerdings schien bei denen keine große Community dahinter zu stehen, was uns in vergangenen Projekten schon auf die Füße gefallen ist.

- **Construct 2**
  Construct 2 ist zwar vermeintlich kostenlos, dabei steht allerdings nur eine Demo zur Verfügung. Zusätzlich gefällt die Handhabung nicht.

- **ImpactJS**
  Kostet 99 USD und ist somit direkt raus.

- **EaselJS**
  Ist zwar kostenlos, bietet aber weder Physik- noch Kollisionsunterstützung an.

- **pixi.js**
  Ist zwar kostenlos, bietet aber weder Physik- noch Kollisionsunterstützung an.

- **Phaser**
  Erfüllt alle Anforderungen und hat zusätzlich zu der guten Dokumentation noch viele brauchbare Beispiele, an denen man sich orientieren kann. Hinzu kommt, dass sogar 3 verschiedene Physiksysteme unterstützt werden. Dabei ist Arcade Physics eine sehr

leichtgewichtige Variante, die auch auf ressourcenarmen Geräten läuft und für unser Spiel vollkommen ausreichend ist.

2.1.3 Entscheidung

Nach Rücksprache mit den Teammitgliedern ist die Auswahl entsprechend obigen Kriterien auf Phaser gefallen.

3 Backend

3.1 Kommunikation

3.1.1 Zeitkritische Informationen

Als zeitkritisch werden Informationen eingestuft, sofern sie die direkten Eingaben der ControlClients und eventuelles Feedback des OutputClient betreffen. Da es sich um ein Reaktionsspiel handel,t müssen diese Daten zeitnah von Sender zu Empfänger gelangen. Solch eine Relation ist nur über eine Peer-to-Peer Verbindung zu realisieren.

3.1.2 Möglichkeit 1: Predefined Packages 'rtc.io'

Der NodeJS Server wird als Verbindungsserver für die Etablierung einer Peer-to-Peer Verbindung zwischen ControlClients und OutputClient genutzt. Als Technik wird konret WebRTC eingesetzt. Auf dem NodeJS-Server wir das Package "rtc-switchboardëingesetzt, welches eine Grundlage für einen Signalisierungsserver ist. Die Clients nutzen "rtc-quickconnectüm neues Channels anzufordern und eine Peer-to-Peer Verbindung zu etablieren.

Vorteile:

- Leichtere Implementierung, da alle Ebenen der Kommunikation bereits abgedeckt wurden und nur noch semantisch auf Informationen eingegangen werden muss.

Nachteile:

- Durch Generalisierung ein deutlicher Overhead.
- Status ist offiziell noch 'unstable'.

3.1.3 Möglichkeit 2: Abstrakte WebRTC implementation 'WebRTC'

Wie auch bei Möglichkeit 1 wird der NodeJS-Server als Verbindungsserver genutzt. Jedoch muss auf Client-Seite, also für OutputClient und ControlClient, eine eigene Signalling Implementation stattfinden. Es wird eine viel abtraktere Schicht genutzt.

Vorteile:

- Da die genutzen Packages nur eine Grundlage bilden, ist eine spezialisierte Implementierung möglich.

Nachteile:

- Implementierungsaufwand deutlich höher.
- Spezialisierung für dieses Projekt eventuell nicht nötig.
- Status ist offiziell noch 'unstable'.

3.1.4 Möglichkeit 3: Socket.io P2P

Das Package Socket.io bietet auch selber eine Peer-to-Peer lösung auf WebRTC Basis. Hierbei wird eine Spezielle Art eines Sockets genutzt, welche wie ein normaler WebSocket agiert, bis ein Upgrade durchgeführt wird und die Clients nun direkt miteinander Kommunizieren.

Vorteile:

- Das Signalling und die P2P Kommunikation können mit einem Package geregelt werden.
- Variabel kann die Kommunikation über den Server, oder P2P ablaufen.
- Signalling events werden von Client-Server Kommunikation direkt zu P2P übernommen.

Nachteile:

- Wenig Einfluss auf die Upgrade-Mechanismen.

3.1.5 Statusinformationen

Für den Austausch nicht zeitkritischer Statusinformationen werden zwischen den Clienten und dem Server einfache WebSockets verwendet. Der Austausch von Zeitkritischen Statusinformationen muss über das Signalling des WebRTC durchgeführt werden.

3.1.6 Fazit

Die eigene Implementation auf Basis des WebRTC bietet die meisten Möglichkeiten, birgt jedoch auch enorme Risiken. RTC.io bietet eine Implementation die sehr gut für Prototypen geeignet ist, aber neben dem Zeitunkritischen Signalling eine eigene Einheit bietet, welche eine Generalisierung des WebRTC darstellt und somit viel Overhead besitzt. Die P2P Imeplementierung von Socket.io steht von der Implementierungs Komplexität in der Mitte. Es bietet viele

bereits bekannte Mechanismen des Client-Server Signalling über Events, welche mit einem Upgrade nun auch von Client zu Client funktionieren.

Socket.io P2P entspricht den Anforderungen und wird für den Prototypen verwendet.

4 Frontend

4.1 Ideensammlung

In diesem Teil finden alle Ideen und Gedanken zu dem Projekt Platz welche ich vor Projektbeginn hatte.

4.1.1 Allgemeine Ideen

Partikeleffekte

Bei Kollision des Balles mit den Wänden, den Schlägern oder gar anderen Bällen wäre ein Partikeleffekt sehr schön.
Ebenfalls könnte man mit Partikeleffekten den Ballverlust (Zerstörung des Balles in den "Toren") sehr schön grafisch unterstreichen.
Möglicherweise dafür geeignete Engine: Sketch.js

Multiplayer

Ideen zu Spielen mit mehr als 2 Spielern.

- Punkt bei Ballverlust erhält der letzte Spieler. Spiel merkt sich letzte 2 Ballkontakte sodass der Abschläger erkannt werden kann.
- Spieler Ball färben. Man könnte die Schläger Färben und die Bälle bei Ballkontakt der eigenen Farbe zuweisen. So erhalten die Spieler einen Visuellen Indikator wer Punkte bekommen kann.

Schläger

- **Bewegung des Schlägers** Bewegung kann Relativ oder Absolut erfolgen. Bei absoluter Bewegung wird der Schläger sehr stark springen und möglicherweise das Spiel zu einfach werden.
- **Geschwindigkeit** Man sollte die Geschwindigkeit des Schläger begrenzen sodass das Spiel schwieriger wird.
- **Begrenzung** Der Schläger braucht 2 Begrenzungen damit er das Spielfeld nicht verlässt.
- **Input der Controller** Der Input sollte so einfach wie möglich gestaltet werden. Am besten nur die Richtung und die Positionsänderung übertragen.

4.1.2 Spielfelder

Original Pong

Im Orinalen Pong betrugen die Abmaße der Komponenten folgende werte:

- Spielfeld Größe: 512*256px
- Ball 6*5px
- Schläger 2*28px
- Schläger Geschwindigkeit 4px pro Intervall

2 Spieler Normal

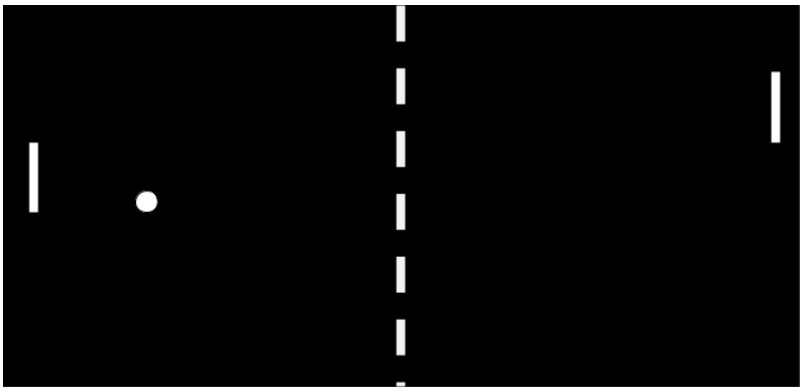

Abbildung 4.1: Mockup eines 2-Spieler Feldes

2 Spieler Breakout

Die Idee für dieses Feld war es 2 Klassiker miteinander zu verbinden. Die Spieler zerstören mit einem eigenen Ball die Felder. Das Spiel endet wenn alle Felder zerstört sind. Gewinner ist der Spieler welcher mehr Felder zerstört hat.
Die Spieler sollten nur den eigenen Ball beeinflussen können. Also muss für den jeweils anderen Spieler eine Barriere errichtet werden.

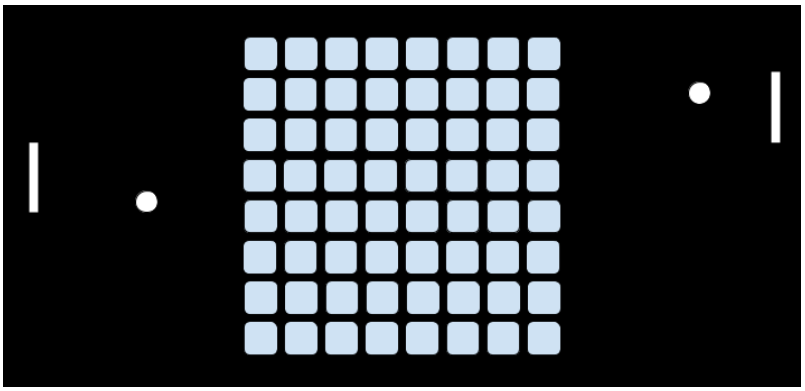

Abbildung 4.2: Mockup eines 2-Spieler Feldes im Spielmodus Breakout

3 Spieler

Als ich die Präsentation des Projektes sah war ich nicht mit den Ideen für einen Multispielermodus zufrieden. Sofort kam mir folgende Idee:
3 Spieler spielen in einem abgeschnittenem Dreieck, bzw eines gleichseitigen Sechseckes. Jeder kontrolliert eine Seite.

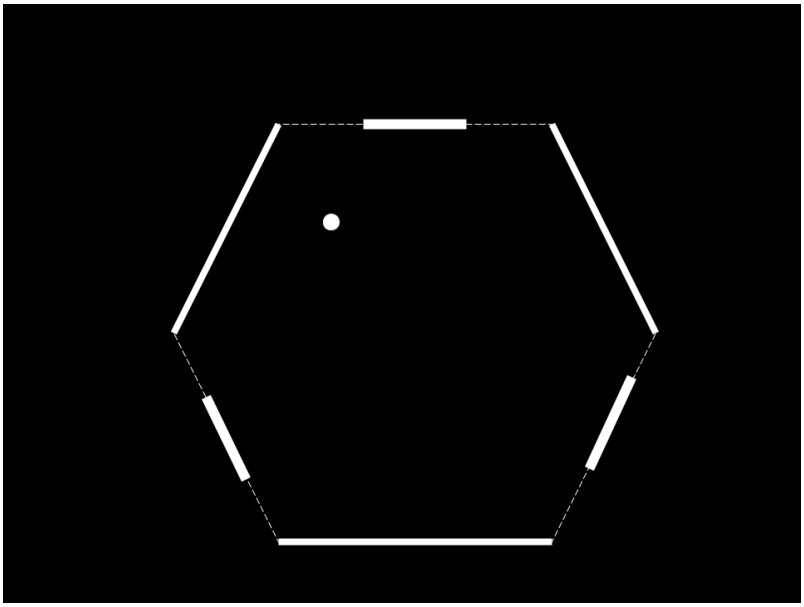

Abbildung 4.3: Mockup eines 3-Spieler Feldes

4 Spieler

Als weitere Idee kam mir ein Vier-Spieler Feld bei dem sich in einem Quadrat je 2 Spieler gegenüberstehen.
Mehr als 4 Spieler sind meiner Meinung nach nicht notwendig. Wenn man bedenkt das sich die Spieler vor einem Bildschirm im Flur sammeln.
Denkbar ist allerdings das man das Konzept für 3 und 4 Spieler für weitere Spieler weiterführt.

4.1.3 Power-Ups

Kurze Ideensammlung zu möglichen Verstärkungen.

- **Magnet:** Ball wird vom Schläger angezogen. Der Spieler erhält dann die Möglichkeit den Ball zu führen und gezielt abzustoßen.
- **Größerer / Kleinerer Ball:** Ball-Modifikator, denkbar auch eine pulsierende Variante wobei sich der Ball durchgehend in der Größe verändert.
- **Multi-Ball:** Ball wir mehrfach dupliziert. Die verschiedenen Bälle fliegen vom Startpunkt aus in alle Richtungen und zählen alle als normaler Ball. Bei Ballverlust tauchen diese nicht erneut auf.
- **Steuerung Invertieren:** Steuerung des Gegners wird invertiert.
- **Geschwindigkeitsmodifikation des Schlägers:** Eigener Schläger wird schneller oder gegnerischer wird langsamer.

4.1.4 Statistiken

Auflistung an Daten welche für eine Statistik relevant wären.

- Spieldauer
- Ballkontakte je Spieler
- Zurückgelegte Distanz des Balles
- Zurückgelegte Distanz je Spieler
- Genauigkeit der Schläger
  (Ballkontakte / Durchgelassene Bälle)

4.2 Auswahl einer Physics-Engine

> Auswahl einer geeigneten 2D-/Physiksengine #3
> Die Darstellung auf dem Anzeigegerät ist über den DOM nicht möglich, da die Bewegung des Balls und der Schläger zu schnell werden. Zudem wird auch die Physik auf dem Anzeigegeräts berechnet. Hierfür ist eine Kollisionserkennung erforderlich.

Für die physikalisch korrekte Kollision des Balles mit der Spielwelt und den Schlägern haben wir uns entschieden eine Physics-Engine zu verwenden.

4.2.1 Matter.JS

Matter js ist eine sehr mächtige Engine. Sie unterstützt viele Formen und Physikalische Eigenschaften wie zum Beispiel Masse und wirkende Kräfte auf die jeweiligen Objekte. Es besteht die Möglichkeit physikalische Objekte zusammen zusetzen und sogar diese Elastisch erscheinen zu lassen, so ist es beispielsweise möglich Stoff oder schwingende Seile zu erstellen. Nach mehrstündiger Einarbeitung kam ich zu dem Schluss das diese Engine für unser Projekt nicht geeignet ist. Gründe hierfür sind:

- Zu Komplex: Die Einrichtung des Spielfeldes erwies sich als überaus schwierig. Gute Anleitungen für einfache Szenarien fehlten, die verfügbaren Anleitungen sind zu grundlegend beschrieben. Ebenfalls half die Anleitung der Engine nicht bei der Verwendung der einzelnen Komponenten.
- Die Positionierung der Objekte bezog sich immer auf den Mittelpunkt des Objektes, man kann beispielsweise kein Rechteck von (x,y) nach (x1,y1) erstellen sondern muss den Mittelpunkt und die Abmaße des Objektes angeben.
- Physik nicht immer korrekt. Die Engine sollte den Ball richtig von einer Ebene abprallen lassen, diese Engine allerdings lies den Ball teilweise an Plattformen abrollen obwohl keine Schwerkraft vorhanden war. Ich schließe darauf das die Engine Reibungskräfte und vielleicht sogar Anziehungskräfte zwischen den Objekten herstellt. Für unser Projekt ist dies aber nicht zu gebrauchen.
- Schwer zu debuggen. Während meiner Versuche bin ich immer wieder auf Probleme gestoßen. Einige der Debugausgaben ließen sich gut ableiten und waren hilfreich. Allerdings bin ich auch auf einige Probleme gestoßen welche nicht in den Debugausgaben

behandelt wurde. Meine letzten Versuche endeten alle darin das der Browser gecrasht ist aufgrund eines Memory-leaks.

4.2.2 Phaser.io

Phaser.io beschreibt sich selber als html5 Game Framework. Es wurde nach dem mobile-first Prinzip entwickelt und ist opensource. Die Entwicklung des gewünschten Prototypen erwies sich als sehr leicht, da es viele gute Beispiele gibt. Die Engine unterstützt von Haus aus eine Arcade-Physic, diese ist perfekt für unser Projekt. Sie beinhaltet Kollisions und Bewegungsfunktionen für den 2-Dimensionalen Raum

4.2.3 Erstellung eines Prototypen

In Phaser erstellt man ein Spiel über den Aufruf 'new Phaser.Game(...) die ersten Beiden Argumente geben die Dimensionen des Spielfeldes an, also die Weite und Höhe in Pixeln.
Der Nächste Parameter bestimmt die Render-Engine. Mögliche Werte sind 'Phaser.CANVAS', 'Phaser.WEBGL' oder 'Phaser.AUTO', wenn 'Phaser.AUTO' verwendet wird so probiert die Engine erst WebGL aus und für den Fall das der Browser WebGL nicht unterstützt wird Canvas verwendet.
Das nächste Argument gibt das Ziel im DOM an, wenn man diesen Parameter nicht setzt wird das Spiel einfach im Body angehängt.
Man kann als 5. Argument ein Startzustand angeben. Zustände kann man sich wie Spielszenen vorstellen. Ich habe mich dafür entschieden die Spielszene erst später hinzuzufügen, das bietet mir den Vorteil das ich diesem Zustand einen Namen geben kann und diesen Später erneut verwenden könnte.

Eine Szene bzw. einen Spielzustand kann man per game.state.add('name',State) vobei 'game' die Instanz des Spiels darstellt. Gestartet wird der Zustand per: 'game.state.start('name)' Ein Zustand ist wie folgt aufgebaut:

```
{
    preload: function () {

    },

    create: function () {

    },

    update: function () {

    },
};
```

Zu den einzelnen Funktionen:

- **preload:** Diese Funktion ist für das Vorladen von Assets gedacht. Beispielsweise Sprites oder Sounds werden hier vorgeladen damit während das Spiel läuft ohne Ladezeit zur Verfügung stehen.
- **create:** Hier werden alle Objekte erstellt die mit dem Beginn des Spielzustandes vorhanden sein sollen. Hier kann man auch Starteigenschafen wie Geschwindigkeit, Schwerkraft oder Ausrichtung setzen.
- **update:** Die update Funktion wird für die Berechnung jedes Frames aufgerufen. In dieser Funktion werden beispielsweise Kollisionen überprüft und darauf reagiert.

Für den Ball des Ping Pong Prototypen habe ich diese Funktionen wie folgt erstellt:

- **preload:**
  Laden des Sprites in den Namen 'ball':

```
game.load.image('ball','assets/testBall.png');
```

  Bekanntmachung der Ball Variable: this.ball

- **create:**
  Erstellen des Balls und einstellen des Ausrichtungspunktes in die Mitte des Sprites:

```
this.ball =
 game.add.sprite(game.world.centerX, game.world.centerY, 'ball');
this.ball.anchor.set(0.5, 0.5);
```

  Aktivieren der Physik für den Ball:

```
game.physics.startSystem(Phaser.Physics.ARCADE);
game.physics.enable(this.ball, Phaser.Physics.ARCADE);
```

  Als nächstes hab ich den Ball so konfiguriert das er mit den Spielfeldrändern kollidiert:

```
this.ball.checkWorldBounds = true;
this.ball.body.collideWorldBounds = true;
```

  Damit der Ball keine zusätzliche Geschwindigkeit beim Kollidieren mit einem anderem sich bewegendem Objekt erhält habe ich ihn 'unbeweglich' gemacht, damit erhält der Ball keine zusätzlichen Impulse von anderen Objekten.

```
this.ball.body.immovable = true;
```

  Und das er keine Geschwindigkeit verliert beim Abprallen:

```
this.ball.body.bounce.set(1);
```

  Als letztes musste ich dem Ball nur noch einen Startimpuls geben:

```
this.ball.body.velocity.setTo(200,0);
```

- **update:**
  Eine Update Funktion war nicht notwendig da der Ball im Prototypen nur mit dem Spielfeld kollidieren soll. Für die späteren Versionen muss hier die Kollision mit den Schlägern und anderen Objekten definiert werden.

Der fertige Prototyp sieht so aus:

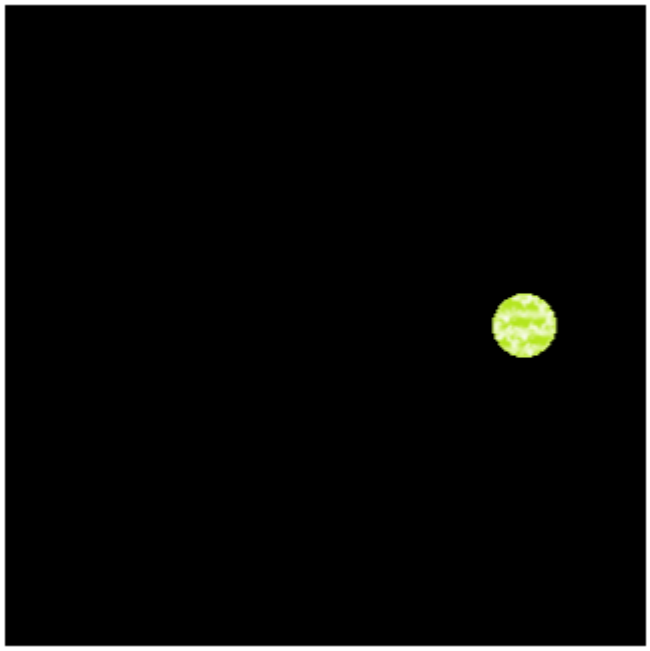

Abbildung 4.4: Erster Prototyp

4.3 Analyse - Steuerung

> Analyse - Steuerung #11
> *Die Steuerung des Spiels erfolgt über das Handydisplay. Jedes Handy ist unterschiedlich groß, zudem sollte der Schläger nicht springen können.*

Es zu analysieren wie sich verschiedene Auflösungen und verschiedene Pixeldichten auf das Spielgefühl auswirken können. Denkbar ist das Geräte mit einer geringen Pixeldichte gegenüber Spielern mit einer hohen Pixeldichte im Vorteil sind da aus Geräten mit einer hohen Pixeldichte sehr viel kleinere Gesten zur Steuerung ausreichen.
Zielsetzung dieser Analyse

1. **Feststellung** Es soll festgestellt werden in wie weit sich ein Unterschied der Pixeldichte auf die Spielweise auswirken kann.

2. **Behandlung** Es soll geprüft werden ob sich eine mögliche Unfairness beheben lässt und ein mögliches Konzept erstellt werden.

4.3.1 Feststellung der Auswirkung verschiedener Pixeldichten

Je nachdem wie die Steuerung implementiert wird die Pixeldichte eines Steuergerätes mehr oder weniger Einfluss auf das Spiel haben. Das birgt die Gefahr das Spieler mit einem Gerät mit einer geringen Pixeldichte gegenüber einem Spieler mit einem Gerät mit hoher Pixeldichte im Nachteil ist da er wesentlich stärkere Bewegungen auf der Touchfläche ausführen muss. Theoretisch kann dies dazu führen das Spieler auf einem Tablet mit niedriger Auflösung immer im Nachteil zu Hochauflösenden Smartphones sind. Laut der Android Spezifikation muss die Pixeldichte von Android Geräten mindestens 100 dpi betragen. Ferner sind in den Spezifikationen Pixeldichten bis zu 640dpi (xxxhdpi) definiert. Geräte mit 640dpi wären Geräten mit 100dpi um 640% überlegen.

Angenommen die Schläger würden 1:1 um den Pixelabstand der Controller bewegt werden und das Spielfeld habe eine Höhe von 400 Pixeln. Auf einem Gerät mit 100dpi müsste der Spieler über eine Stecke von 4 cm streichen müssen um den Schläger von einem Ende zum anderen zu bewegen. Auf einem Gerät mit 640dpi müsste der Spieler nur über eine Stecke von unter einem Zentimeter streichen.

4.3.2 Behandeln verschiedener Pixeldichten

Ermitteln von der DPI im Browser

Es wird nach vorhanden Lösungen zum erkennen der Pixeldichte in PPI bzw DPI gesucht. Die gefundenen Beispiele werden dann mit verschiedenen Geräten getestet und die Ergebnisse mit den tatsächlichen DPI verglichen.

Die Testgeräte

1.

Name	PC Monitor
Diagonale	23.6zoll
Auflösung	1920*1080
Pixeldichte	93 DPI

2.

Name	Samsung Galaxy S4
Diagonale	4.99zoll
Auflösung	1920*1080
Pixeldichte	441 DPI

3.

Name	Samsung Galaxy Tab A
Diagonale	9.7zoll
Auflösung	1024*768
Pixeldichte	132 DPI

- **http://www.infobyip.com/detectmonitordpi.php** Eine Website zur Ermittlung der DPI. Es wird ein Quadrat per CSS auf eine bestimmte Größe, zum Beispiel 1 Zoll, formatiert und dann die Größe in Pixeln ausgelesen. Auf diese Weise lässt sich ein DPI wert berechnen.

Tests:

	Erwarteter Wert	Ermittelter Wert
PC Monitor 1		
Pixeldichte	93 DPI	96 DPI
Größe des Feldes	8 cm	8.2 cm
Samsung Galaxy S4		
Pixeldichte	441 DPI	288 DPI
Größe des Feldes	8 cm	1.8 cm
Samsung Galaxy Tab A		
Pixeldichte	132 DPI	96 DPI
Größe des Feldes	8 cm	5.7 cm

Getestet wurde jeweils mit Firefox & Chrome, der PC Monitor wurde auch noch mit dem Edge Browser getestet. Die Ergebnisse waren allesamt für das Gerät identisch.
Ergebnis: Die DPI wurden nicht korrekt erkannt. Die Werte weichen auf dem Mobiltelefon um bis zu 77.5% ab.

Verwenden von 'window.devicePixelRatio'

Die Eigenschaft *window.devicePixelRatio* gibt das Verhältnis der Größe der physikalischen Pixel des aktuellen Displays zu der Größe der Geräteunabhängigen-Pixel (*device independent pixels(dips)*) wieder.

$$window.devicePixelRatio = physicalpixels/dips$$

Es wird vor allem dafür genutzt um eine Einheitliche Darstellung von Webinhalten auf verschiedenen Displaygrößen zu erreichen.
Die Methode klingt sehr vielversprechend, also entschied ich einen Test zu erstellen. Meine Idee ist es ein Quadrat zeichnen zu lassen bei dem die Größe abhängig von dem devicePixelRatio ist und dann die Seitenlänge auf dem Display zu messen. Mein verwendeter Code:

```
1 var c=document.getElementById("myCanvas");
2 var ctx=c.getContext("2d");
3 var w = 100*window.devicePixelRatio;
4 ctx.rect(20,20,w,w);
5 ctx.fillText(w,30,30);
6 ctx.stroke();
```

Tests:

Gerät	DevicePixelRatio	Größe TestQuadrat
PC Monitor 23.6"	1.0	2.7 cm
Samsung Galaxy S4	3.0	1.9 cm
Samsung Galaxy Tab A	1.0	1.5 cm
Samsung Galaxy S3 Neo	2.0	1.3 cm
Samsung Galaxy S5 mini	2.0	1.1 cm
Samsung Galaxy S7 edge	4.0	>4 cm

Ergebnis:

Das Ergebnis der Tests ist leider ernüchternd. Die ersten Geräte zeigten das Quardrat allesamt in einer ähnlichen Größe. Doch auf weiteren Testgeräten zeigte sich das die Größe sehr stark variierte. Auf dem Samsung Galaxy S7 edge war das Quadrat sogar sehr viel größer als der vordefinierte Bereich.

Suche nach Alternativen zu DevicePixelRatio

An vielen Stellen habe ich gelesen das der DevicePixelRatio sehr häufig gerundet wird. Dies fiel ebenfalls bei den Testgeräten auf.
Also kam mir die Idee eine eigene Version der Funktion zu schreiben. In Javascript kann man mit folgender Funktion testen ob ein bestimmter DevicePixelRatio unterstützt wird.

```
1 if (window.matchMedia('(-webkit-min-device-pixel-ratio: 1)').matches)
```

Meine Idee war es den gewünschten Wert der Funktion schrittweise zu erhöhen und den zuletzt akzeptierten Wert zurückzugeben.
Daraus ergab sich folgende Funktion:

```
1 function getDPR() {
2     var numb = 1.0;
3     while(window.matchMedia(
4         '(-webkit-min-device-pixel-ratio: ' + numb + ')'
5         ).matches) {
6                 numb += 0.1;
7     }
8     return numb - 0.1;
9 }
```

Leider ergaben die Test mit dieser Funktion keine nennenswerte Ergebnisse weswegen ich auf eine Auflistung und Auswertung der Ergebnisse in diesem Falle verzichte.

Verwendung der CSS3 MediaQuery Eigenschaften

Mit CSS3 lassen sich verschiedene Fälle für verschiedene Auflösungen definieren.

```
1 @media (resolution: 96dpi) { /* Exakt 96 Bildpunkte pro Zoll */ }
2 @media (min-resolution: 200dpcm) { /* Mindestens 200 Punkte pro cm */ }
3 @media (max-resolution: 300dpi) { /* Maximal 300 Punkte pro Zoll */ }
```

Abbildung 4.5: Quelle: https:wiki.selfhtml.orgwikiCSSMedia_Queries

Hiermit ist es möglich für die verschiedenen Pixeldichten die Kontrollfelder anzupassen. Allerdings werden die Pixeldichten der Geräte nicht immer richtig erkannt. Es

4.3.3 Ergebnis der Analyse

Es gibt leider keine Möglichkeit eine Webapp mit exakten physikalischen Größen zu gestalten.

Gestaltung per CSS Einheiten In CSS besteht die Möglichkeit Größen in CM oder Zoll zu definieren. Leider entsprechen die Werte auf Mobilen Endgeräten nicht den gewünschten Werten.

Gestaltung mit Device Pixel Ratio Die Gestaltung per DevicePixelRatio erscheint wesentlich genauer als die Gestaltung mit CSS Längenangaben allerdings ist auch diese Lösung sehr ungenau und lässt sich nicht einheitlich nutzen.

Gestaltung der Steuerung Man könnte die Steuerung so definieren das ein Vorteil der Pixeldichte sich nicht zu sehr auf das Spiel auswirkt. Beispielsweise Kann man eine Maximalgeschwindigkeit der Schläger so definieren das sie auch auf Geräten mit geringer Pixeldichte leicht erreicht werden kann. Dadurch sind allerdings möglicherweise Geräte mit hoher Pixeldichte im Nachteil da man für kurze Bewegungen nur noch minimale Bewegungen auf dem Bildschirm durchführen dürfte.
Denkbar wäre auch eine Steuerung mit fester Bewegungsgeschwindigkeit sodass nur noch die Richtung der Schläger durch die Controller geregelt wird. Diese Art Steuerung verspricht allerdings nicht viel Spaß oder Reaktionsmöglichkeiten der Spieler.

Lösungsmöglichkeit:

Verrechnung der Information zur Pixeldichte
Mit den CSS3 Media Queries ist es zumindest ansatzweise Möglich die Pixeldichte der Geräte zu erkennen. Allerdings sich auch sie sehr ungenau und keinesfalls verlässlich.
Zur fairen Gestaltung der Touchfelder sollten wir die Media Queries bzw den DevicePixelRatio verwenden um zwischen Geräten mit hoher und geringerer Pixeldichte unterscheiden.
Beispielsweise indem wir den PixelRatio mit der zurückgelegten Stecke in Pixeln verrechnen.

```
1 movement = screenDistance / window.devicePixelRatio;
```

Somit werden Geräte mit einem PixelRatio von 1 nicht benachteiligt. Außerdem hat dies zur Folge das Spieler auf einem Gerät mit einem hohen PixelRatio ihre Schläger nicht unnötig schnell bewegen wenn sie zum Beispiel nur kurze Bewegungen erreichen wollen.
Ein Problem bleibt jedoch, der PixelRatio ist nicht sehr genau. Um ein Gefühl für die Auswirkungen zu bekommen müssen wir mehrere Testgeräte für die Steuerung verwenden.

4.3.4 Flüssige Bewegung

Um eine flüssige Bewegung zu garantieren sollten wir uns auf einige Eigenschaften der Steuerung festlegen.

Anforderung: Bildschirmposition != Schlägerposition
Die Schläger sollen in Abhängigkeit einer Bewegung, also einer Positionsänderung auf dem Bildschirm bewegt werden. Das bedeutet für uns das wir die Information der Controller nicht als Absolute Positionen sehen sondern eher die Änderung während einer Geste betrachten. Hierfür sei folgendes Festgelegt:

- **Beginn der Geste:** Eine Geste beginnt mit dem Berühren des Touchfeldes. Die Startposition wird ermittelt und in einer Variable gespeichert.
- **Während der Geste:** Während der Finger sich auf dem Touchfeld bewegt wird die neue Position des Fingers ausgewertet. Die neue Position wird von der alten Position abgezogen wodurch sich eine Differenz ergibt. Diese wird dann mit dem PixelRatio verrechnet und an das Spiel (DisplayPeer) geschickt.
  Der ganze Prozess sollte alle 40ms neu gestartet werden.
- **Ende einer Geste:** Eine Geste endet sobald das Touchfeld nicht mehr berührt wird. Die aktuelle Änderung der position wird auf 0 gesetzt.

Wenn der Controller nicht mehr berührt wird so darf sich der Schläger nicht weiter bewegen. Der Controller braucht in diesem Zustand dem Spiel keine Infos übermitteln.
Moderne Mobiltelefone haben wesentlich höhere Auflösungen als unser Spielfeld groß sein wird. Aus diesem Grund sollte die Bewegung auf einem Maximumwert begrenzt werden. Maximal sollte sich ein Schläger über die gesamte Spielfläche in einem Zyklus bewegen können. Es ist denkbar das wir den Wert später nach unten korrigieren müssen um ein gutes Spielgefühl schaffen zu können.

4.4 Erstellung des Technischen Demonstrators

Für die Präsentation des Projektstandes war unsere Zielsetzung alle Technischen Hürden überwunden zu haben und einen Funktionsfähigen Prototypen zeigen zu können.
Dieser Prototyp sollte vor allem den Technischen Durchstich erreicht haben, konkret waren das folgende Anforderungen:

- Initialisieren der Lobby durch den Server
- Kommunikation der Peers via WebRTC
- Einlesen von User Eingaben am ControlPeer
- Anzeige der Schläger in Abhängigkeit der Usereingaben am DisplayPeer

4.4.1 Realisierung des ControllPeers

Meine Aufgabe war es die Darstellung auf dem DisplayPeer, nach den gegebenen Anforderungen, zu realisieren.

Es sollten nur die Schläger angezeigt und bewegt werden können. Damit jeder Schläger von den anderen Unterscheidbar bleibt hab ich mich dafür entschieden verschiedene Schläger-Sprites in einem Spritesheet zu speichern zu jedem Spieler eine Sprite ID mit zu geben. Das hat den Vorteil das die Sprites in einer einzigen Datei lagern und leicht getauscht werden können.
Laden des Spritesheets (Argumente:Name, Pfad, Weite, Höhe, Anzahl):

```
1 game.load.spritesheet('paddle', 'assets/testPaddles.png', 9, 100, 6);
```

Die Spieler erhalten ihren Sprite in Abhängigkeit zu ihrer ID:

```
1 function assignPlayerSprite(player) {
2 player.sprite = game.add.sprite(player.x, player.y, 'paddle');
3 player.sprite.frame = player.id; }
```

Die Methode wurde in einer Methode verbaut welche einen neuen Spieler erzeugt. Dabei wurde auch geprüft ob die maximale Anzahl an Spielern überschritten wurde.
Das Spielerobjekt selber bekam eine Funktion um die neuen Eingaben zu seinem Buffer hinzuzufügen sowie eine Update Methode welche vom Spiel aufgerufen wurde um die Position zu aktualisieren.
In der Update Methode wurden über alle Positionen im Buffer iteriert. Das hat den Hintergrund das später nur Positionsänderungen übertragen werden und dabei alle berücksichtigt werden sollen.

See also One und Two (2010).

Literaturverzeichnis

[One und Two 2010] One, Author ; Two, Author: A Sample Publication. (2010)

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 28. Dezember 2016 Andreas Müller, Claus Torben Haug, Jan-Dennis Bartels, Marjan Bachtiari