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: 7. 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 JavaScript

2 Backend

2.1 Kommunikation

2.1.1 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.

2.1.2 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'.

2.1.3 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'.

2.1.4 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.

2.1.5 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.

2.1.6 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.

3 Frontend

3.1 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.

3.1.1 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.

3.1.2 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

3.1.3 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:

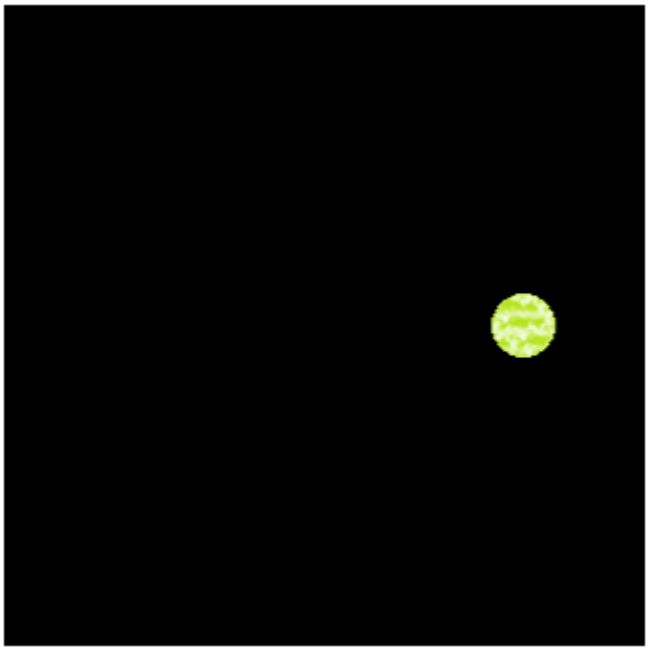

Abbildung 3.1: Erster Prototyp

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