Dokumentation

Mass Click Orchestra

Tobias Gärtner - 551145
Jonas Leitner - 551527

Dozent: Herr Tobias Hiep
Projektarbeit Medieninformatik HS Flensburg

PROJEKTIDEE

Ziel dieser Projektarbeit war, eine Webanwendung zu entwerfen, in welcher Nutzer in Echtzeit kooperativ musizieren können.
Es gibt verschiedene Räume, zwischen denen man sich frei entscheiden kann, und zu welchem man andere User einladen kann, in denen man ihnen die Raum-ID nennt oder die URL schickt.

INSTALLATION

Das ganze Projekt wird in der package.json definiert, und kann mit einem simplen „npm install“ und „npm start“ zum laufen gebracht werden. Danach ist es unter localhost:8084 erreichbar. Alternativ liegt der aktuelle Master live auf http://mco.jonasleitner.de, das Github-Projekt ist unter https://github.com/Gaertner12/MassClickOrchestra zu finden.

FRONTEND

Zunächst soll es die Möglichkeit geben, nach Betreten der Seite einen Raum zu finden. Auf der Startseite kann man daher die Raumnummer eingeben oder einem zufälligen Raum beitreten. Ist man nun in einem Raum, sieht man ein Rasterlayout (Grid), welches mit Blöcken gefüllt wird. Diese Blöcke sind Div-Elemente, von denen jedes einen Jquery-Clicklistener hat (klingt unperformant, macht aber tatsächlich keinerlei Probleme). Klickt man einen Block an, färbt sich dieser und wird undurchsichtig, und gilt solange als aktiv, bis er wieder abgewählt wird (oder von einem anderen User geändert wird). Die Farbe wird je nach UserID aus einem Array aller CSS-Farben ausgewählt, was die Einzigartigkeit der Farben garantiert. Sollten alle diese Farben verwendet sein, wird eine zufällige Farbe generiert.

Beim Abspielen werden nur alle aktiven Blöcke abgespielt. Das Abspielen wird durch einen „Play-Button“ verwaltet, welcher die Musik auch pausieren kann.
In der rechten oberen Ecke befindet sich eine Check-Box, mit welcher man eine Endlosschleife an- oder ausschalten kann. Der Slider-Thumb, der die Abspielposition anzeigt, kann zudem auch angefasst und verschoben werden. Der Slider ist ein Input vom Typ Range, welcher geschickt an das Grid angepasst wurde. Er wird für Nutzer deaktiviert, während Sounds abgespielt werden, und wird von der Abspiel-Funktion auf die aktuelle Position am Grid bewegt. Zudem kann (auch während das Stück abgespielt wird) das Instrument client-seitig geändert werden.

Funktionsweise Sounds:

Die Sounds wurden mit dem Programm Logic erstellt und durch virtuelle Instrumente mit Tonhöhen versehenen.

Bei Pageload wird ein HowlerJS-Objekt erstellt, welches für die Wiedergabe der Töne genutzt wird. HowlerJS lädt ein Soundfile jeweils als mp3 oder webm nach, welches die 9 verschiedenen Tonhöhen aller Instrumente nacheinander abgespielt beinhaltet (um die Pageload gering zu halten, wurde die wav-Datei herausgenommen). Nun werden die Töne durch ein wenig Javascript-Magie in Sprites unterteilt, welche parallel zueinander abgespielt werden können.

```
function setSprites() {
        let obj = {};

        instruments.forEach((ele, ind, arr) => {
                for (let i = 0; i <= toneAmount; i++) {
                        let begin = soundLength * i + (toneAmount * ind * soundLength);
                        let end = soundLength;
                        obj['pitch-' + (i + 1) +'-'+ele] = [begin, end];
                }
        });

        return obj;
}
```

HowlerJS erwartet für Sprites ein Array wie folgt: [[startTime, soundLength], ...]
Hier wird für jedes Instrument und jeden der neun Töne die Position im Soundfile ausgeschnitten

Zudem werden an das sound-Objekt von HowlerJS einige Flags und Funktionen angehängt, um den Abspielstatus zu kontrollieren, etwa sound.isPlayingFlag, sound.position, sound.instrument, sound.toggleTrack() (wird vom Play-Button aufgerufen), sound.playTrack(), sound.pauseTrack() (werden wiederum von toggleTrack() aufgerufen) und sound.fadeAfterTime() (um einen Sound erst nach einer gewissen Zeit zu faden, ist nicht per default in HowlerJS vorhanden).

Drückt der User nun auf den Play-Button, wird eine window.SetTimeout-Funktion angestoßen, welche sich samt Timeout rekursiv selber aufruft, solange der Play-Button aktiv ist.
Innerhalb dieser Funktion werden überprüft, welche Töne in der nächsten Grid-Spalte aktiv sind, und für diese jeweils ein Ton im aktuell ausgewählten Instrument abgespielt. Jeder Ton wird mit einem Fade ein- und ausgeleitet. Die ID's der abgespielten Töne werden gespeichert, damit alle Töne per ID aufgerufen und pausiert werden können, sobald der User das Abspielen pausiert.

BACKEND

Das Backend wurde mit einem NodeJS-Server auf Basis des Express-Framework mit Handlebars als Templating-Engine umgesetzt. Handlebars wurde nicht genutzt, um Daten in die View zu rendern, sondern um die Views aufzuteilen in Layouts (welche die Scripts und Styles requiren), Partials (welche das Grid aufbauen) und die Haupt-Views login.handlebars und index.handlebars.

Es gibt drei Routen (zu finden in controller/index.js). Die „/„-Route ist für den Login zuständig, eine GET-Request auf / gibt die Loginseite zurück, welche ein Formular enthält, mit dem man eine POST-Request auf die Route /roomservice stellen kann. Die Request enthält entweder die ID des gewünschten Raums, an welchen dann man redirected wird, oder den String „random", welcher den Controller dazu veranlasst, den User in den Raum mit der letzten Änderung weiterzuleiten. Die Letzte Route /orchestra wird mit dem GET-Parameter room erreicht, und gibt die Seite mit dem Musik-Grid zurück.

Funktionsweise Multiplayer:

Die Echtzeit-Multiplayer-Logik wird über SocketIO umgesetzt, welches man sich als UDP-Stream für den Browser vorstellen kann (d.h. jede Änderung wird in Echtzeit von Client an Server und andersherum übertragen).

Beim Laden der Seite meldet sich der Client beim Server an und teilt ihm seine Raum-ID mit. Ist der Client der erste User im Raum, erstellt der Server ein Grid-Objekt und speichert es als globale Variable, in welcher alle Änderungen im Grid samt UserIDs und Timestamp der letzten Änderung gespeichert werden. Ist der Client dem Server noch unbekannt, wird seine ID in einem globalen Array gespeichert, um ihn später seinem Raum zuordnen zu können. Der Server antworten nun mit einer JSON dieses Grid-Objekts.

{ lastChangedTimestamp: 1499601875419,
userList: ['SryZW6ylrlVU6gJzAAAB'],
'0x9': { active: true, userId: 0, x: 0, y: 9 },
'1x8': { active: true, userId: 0, x: 1, y: 8 },
'2x7': { active: true, userId: 0, x: 2, y: 7 },
'3x6': { active: true, userId: 0, x: 3, y: 6 },
'4x5': { active: true, userId: 0, x: 4, y: 5 },
'5x4': { active: true, userId: 0, x: 5, y: 4 },
'7x2': { active: true, userId: 0, x: 7, y: 2 },
'8x1': { active: true, userId: 0, x: 8, y: 1 },
'6x3': { active: true, userId: 0, x: 6, y: 3 } }

Das Grid-Objekt, so wie es im Server gespeichert wird und bei der Initialisierung an die Clients gegeben wird.

Der Client passt nun sein Grid dem vom Server an und tritt dem SocketIO-Raum mit der entsprechenden ID bei. Jede Änderung irgendeines Clients am Grid wird nun an den Server übertragen, welcher seine eigene Grid-JSON anpasst, und nun die Änderung an alle Clients im selben Raum broadcastet, welche dann die Änderungen in ihrem Frontend-Grid übernehmen können. Die Änderung wird auch erst beim Client, welcher sie verursacht hat, übernommen, wenn der Broadcast angekommen ist. So wird ein einheitlicher Grid-State garantiert.

Anmerkung: Geplant war ein Cooldown-Feature, welches Spamming verhindert, und wurde auch halb umgesetzt, aber dann doch nicht eingebaut, da wir es nicht mehr als nötig empfunden haben. Spamming ist der halbe Spaß an der Sache und steigert die empfundene Responsivität der App.

QUELLEN

Bild1:

https://www.pexels.com/photo/macbook-pro-beside-black-headphones-on-gray-table-159376/

Bild2:

https://www.pexels.com/photo/headphones-technology-earphones-cable-30222/

Frameworks:

https://howlerjs.com/
https://socket.io/
http://handlebarsjs.com/
http://expressjs.com/de/
http://lesscss.org/
https://nodejs.org/en/