HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Bearbeitungsbeginn: [21.03.2019]

Vorgelegt am: [23.09.2019]

Thesis

zur Erlangung des Grades

Bachelor of Science

im Studiengang Medieninformatik

an der Fakultät Digitale Medien

Thomas Dorner

Matrikelnummer: 250355

Integration der Web Audio API in eine auf Webtechnologien basierenden Game Engine

Erstbetreuer: *Prof. Dr. Jirka Dell'Oro Friedl*

Zweitbetreuer: *Prof. Dr. Ruxandra Lasowski*

Abstract

Auf dem Markt befinden sich eine Vielzahl von Spiele-Engines welche sich in vielen Aspekten ähneln aber auch große Unterschiede aufweisen. Die Meisten davon finden sich in der kommerziellen Anwendung wieder und sind nicht für die Anwendung in der Lehre zugeschnitten.

FUDGE (Furtwangen University Didactic Game Editor) steht einem didaktischen Paradigma zugrunde, welcher aber trotzdem die grundlegendsten Funktionen von Engines wie Unity abbilden soll. FUDGE basiert auf Webtechnologien und wird damit ein plattformunabhängiger Spiele-Editor.

Diese Bachelorarbeit beschäftigt sich mit der Web Audio API und deren Integration in FUDGE. Zu Beginn werden die grundlegenden Webtechnologien, die für diese Arbeit relevant sind, aufgegriffen. Anschließend werden alle Komponenten der Integration erklärt und ihre Funktionen beschrieben.

There are multiple Game-Engines on the market which resemble each other in a lot of aspects but also show major diefferences. Most of them are commercial applications and are not tailored for application in teaching.

FUDGE (Furtwangen University Didactic Game Editor) is based on a didactic pardigm, which nevertheless should represent the most basic function of engines like Unity. FUDGE is based on web technologies and thus becomes a cross-platform compatible Game-Editor.

This bachelor thesis deals with the Web Audio API and its integration into FUDGE. At the beginning the basic web technologies that are relevant for this thesis are taken up. All components of the integration are then explained and their functions described.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

FUDGE - Furtwangen University Didactic Game Editor

HTML - Hypertext Markup Language

CSS - Cascading Style Sheets

JS - JavaScript

TS - TypeScript

UML - Unified Modeling Language

GUI - Graphical User Interface

WAA - Web Audio API

API - Application Programming Interface

XHR - XMLHTTPRequest

1. Einleitung

“Games sind das fortschrittlichste Medium unserer Zeit: Sie sind gleichermaßen Kulturgut, Innovationsmotor und Wirtschaftsfaktor. [1]“ Die Spielebranche ist eine der umsatzstärksten Branchen in Deutschland. Im ersten halben Jahr von 2019 wurden Deutschlandweit 2,8 Milliarden Euro durch Spiele erwirtschaftet. Und das Wachstum steigt rapide, denn im Gegensatz zum Vorjahr war dies bereits ein Anstieg von etwa 11% [2]. Deutschland zu den fünf Ländern mit dem größten Umsatz in der Spielebranche. Platz eins bis vier werden dabei von der USA, China, Japan und Korea gehalten [3]. Die Spielebranche generiert etwa 30.000 Stellen in Deutschland [1].

Allerdings bieten Spiele mehr als nur den wirtschaftlichen Faktor: “Sie vereinen nicht nur alle vorangegangenen Medienformen, Sprache, Text, Klang und Bewegtbild, sondern erweitern diese durch ihre spezifische Eigenschaften Interaktivität, dem Spielerischen und dem Sozialen zu einer einzigartigen Erfahrung. [1]“ Spiele gehören zu einer Kunstform, mit der sich Entwickler ausdrücken können und Spieler ihre eigenen Interessen innerhalb dieser Spiele nachgehen können. Videospiele werden seit Jahrzehnten von Nutzern auf der ganzen Welt gespielt. Sie sind auf den unterschiedlichsten Plattformen präsent, dazu zählen Desktop PC, Konsolen und seit kürzerer Zeit Mobile Geräte.

Viele dieser Anwendungen werden auf sogenannten Game-Engines erstellt. Die größten Spiele werden auf einer Basis einer Game-Engine entwickelt, die oftmals speziell für das Spiel angefertigt worden ist auf dessen Anforderungen zugeschnitten ist. Diese sind jedoch meistens nur innerhalb des Entwicklerstudios sichtbar. Einige kommerzielle Game-Engines sind öffentlich nutzbar und erleichtern die Entwicklung von Spielen. Unity und Unreal Engine gehören zu den am weitesten verbreiteten Game-Engines. Ältere Anwendungen wie Adobe Animate (Flash) können auch zur Entwicklung von Spielen genutzt werden. Unity, Unreal und Animate besitzen eine Benutzeroberfläche und vorgefertigte Werkzeuge, welche die Entwicklung vereinfachen und beschleunigen.

Jedoch stehen diese Game-Engines vor allem für erfahrene Entwickler und einer kommerziellen Nutzung zur Verfügung. Diese Programme sind jedoch nicht für didaktische Zwecke ausgelegt, was der ausschlaggebende Faktor für die Konzeptionierung und Entwicklung des Furtwangen University Didactic Game Editor, kurz FUDGE [4]. Hochschulen wie Furtwangen, bieten spezielle Vorlesungen an, um Studenten die Entwicklung von digitalen Spielen zu lehren. Für diese Aufgabe gibt es keine speziell angepasste Game-Engine. FUDGE soll ausschließlich für den Einsatz in der Lehre eingesetzt werden und richtet alle Design Entscheidungen danach aus. Er soll außerdem Werkzeuge liefern um zusätzliche Assets, das heißt Ressourcen, die in einer Form für ein Spiel genutzt werden können, innerhalb der Umgebung anfertigen zu können [5].

In dieser Arbeit wird eine Anwendung entwickelt welche FUDGE um eine Audio Komponente erweitern soll. Hierbei kommen Webtechnologien wie TypeScript, Fetch API und die Web Audio API zum Einsatz um diese Anwendung zu planen und erstellen. Im Laufe der Arbeit werden die Grundlagen dieser Technologien erläutert. Auf die Web Audio API und ihrer Funktionsweise wird explizit eingegangen. Anschließend werden die Anforderungen deklariert und in einem Konzept eingebracht. Alle Klassen werden daraufhin erklärt und deren Kommunikation dargelegt. Zum Schluss werden Beispielanwendungen in FUDGE beschrieben um den Umgang dieser Implementation zu verdeutlichen.

Damit eine einfache und leicht erweiterbare Anwendung zu schaffen, steht die Planung und Konzeptionierung im Vordergrund dieser Arbeit, welche in die Entwicklung einfließt.

2. Grundlagen der Audio Implementation

Die Implementation der räumlichen Audiowiedergabe in FUDGE inkludiert die Nutzung verschiedener Technologien. Zu diesen zählen Webtechnologien wie HTML, JavaScript, TypeScript und Fetch API sowie der Game Editor FUDGE. Diese Technologien bilden die Grundlage für die Integration und Umsetzung der Web Audio API (Kapitel 3) und werden aus diesem Grund im Folgenden vorgestellt.

2.1 Webtechnologien

Diese Webtechnologien sind mittlerweile ausgereift und stellen bereits gängige Standards dar. Darunter sind nicht nur die fundamentalen Technologien, wie HTML, CSS und JS, sondern auch eine Vielzahl an weiteren Technologien, die zum Teil auf diesen aufbauen, wie z.B. TS. Ein großer Vorteil der Nutzung dieser Webtechnologien besteht darin, dass die Grundlagen für die Entwicklung im Bereich der Webtechnologien in den meisten Fällen bereits existieren. Der Nutzer muss somit das Programm nicht von Grund auf entwickeln, sondern kann sich auf bereits existierende Ansätze stützen.

Auf diesen Ansätzen baut auch FUDGE auf, denn er ist ein für die Lehre ausgelegter Game Editor, mit dem Studenten die Möglichkeit bekommen sollen, ohne detaillierte Kenntnisse, Spiele oder ähnliche interaktive Software zu schreiben. FUDGE stellt ein Gerüst dar, das eine Vielzahl von Webtechnologien unter sich vereint, ohne dass der Nutzer die Übersicht verliert oder sich viele Vorkenntnisse aneignen muss. [5]

HTML

HTML (HyperText Markup Language, deutsch: Hypertext Auszeichnungssprache) ist die Basiskomponente, um Webseiten Struktur zu verleihen. Die Anzeige und Inhalte von Texten, Bildern und anderen Inhalten wird durch HTML auf einer Webseite beschrieben. Dafür nutzt HTML spezielle Elemente, die alle mit einer `<Tag></Tag>` Syntax erstellt werden.
Ein Beispiel dazu könnte wie folgt aussehen:

```
<html>
    <head>
    </head>
    <body>
        <p> Inhalt </p>
    </body>
</html>
```

Webseiten werden durch Links in HTML miteinander verbunden. Dabei ist es möglich sowohl mehrere Webseiten, als auch verschiedene Seiten einer Webseite zu verbinden.

Technologien wie CSS (Cascading Style Sheets) und JavaScript werden genutzt, um das Erscheinungsbild und die Funktionalität einer Webseite zu erweitern. [6]

JavaScript

ECMAScript wurde ursprünglich als eine Webbasiert Skriptsprache entwickelt, ist aber inzwischen eine universell nutzbare Programmiersprache.

Das ECMAScript ist eine Programmiersprache, die den Standard für JavaScript darstellt. Aktuell wird die 10. Edition genutzt, welche im Juni dieses Jahres aktualisiert wurde.

JavaScript ist eine Scriptsprache, die für ihre Anwendungen im Web Bereich bekannt ist. Sie wird allerdings auch in vielen Anwendungen ohne

Browserumgebung genutzt, wie beispielsweise in NodeJS oder Acrobat Reader.

Die clientseitige Ausführung von JavaScript wird dazu genutzt, das Verhalten sowie das Design von Webseiten ereignisgesteuert zu verändern. Somit nimmt JavaScript Einfluss auf HTML und CSS und kann diese verändern.

JavaScript ist eine schlanke und dynamisch typisierte Skriptsprache darstellt, die objektorientierte und prozedurale genutzt werden kann. [7]

TypeScript

TypeScript ist ein Superset, dass von Microsoft für JS entwickelt wurde und bietet dem Nutzer damit eine größere Menge an Funktionalitäten. Dazu gehören die Typsierung, welche statische und auch dynamische Typen enthalten kann sowie die Klassifizierung. Zusätzlich bietet TS Konstrukte, die über das Ausmaß von JS hinausgehen wie beispielsweise der Gebrauch von Interfaces. Ein weiterer Vorteil von TS ist, dass auch JS Bibliotheken problemlos angebunden und benutzt werden können.

TypeScript wird zur Laufzeit kompiliert und in JS übersetzt. Dabei verliert der daraus entstandene JS Code auch alle vorher deklarierten Typ-Notationen. Zudem ist TypeScript für den Nutzer nur während der Entwicklung sichtbar.

Beispiele von mit TS entwickelter Software sind React, Angular oder auch Visual Studio Code. [8]

Alle Codebeispiele in dieser Arbeit sowie die vollständige Implementierung der Prototypen werden in TypeScript dargelegt.

Promises

Ein `Promise` Objekt ist ein stellvertretender Wert, der bei Erstellung noch nicht bekannt sein muss, und stellt eine Asynchrone Operation dar. Ein `Promise` Objekt wird durch dessen Erfüllung mit einem resultierenden Ergebnis bzw.

einem Versagen dargestellt. Jedoch ist die Antwort eines `Promise` Objekts immer auch ein weiteres `Promise` Objekt.

Ein `Promise` Objekt kann drei Zustände besitzen:

1. `pending`: Ausgangszustand eines Promise Objektes, welcher darauf wartet seinen Zustand zu verändern.

2. `fulfilled`: Die erste Möglichkeit einer Antwort für ein Promise Objekt ist die Erfüllung der Nachfrage und erhält somit eine Antwort mit dem gewünschten Wert.

3. `rejected`: Die andere Möglichkeit einer Antwort ist die Ablehnung der Nachfrage, die meist durch einen Fehler entsteht, welcher direkt abgefangen werden kann.

Dieser Prozess wird auf der Abbildung 1 gezeigt.

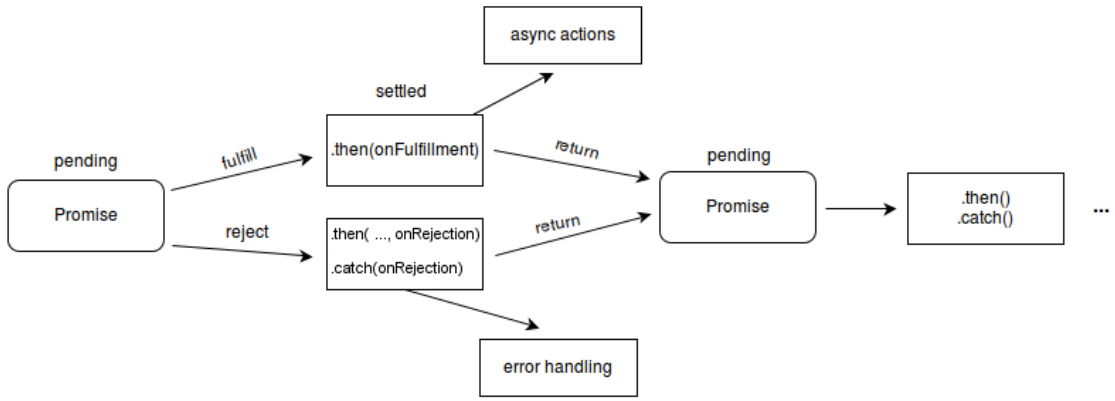

Abbildung 1: Allgemeine Darstellung einer Abwicklung von Promise Objekten [9]

`Promise.all` kann genutzt werden, falls eine große Menge an Promises verarbeitet werden muss. Man kann damit z.B. eine Liste an Sounds zum Start der Laufzeit auf einmal laden, allerdings soll für die spätere Integration nur in dem Moment ein Sound geladen werden, in dem dieser auch wirklich genutzt werden soll. [9]

Nachfolgend werden die Schlüsselworte `async` und `await` des `Promise` Objektes näher erläutert.

Async

`Async` definiert eineasynchone Funktion. Eine asynchrone Funktion ist eine Funktion welche asynchron operiert. Wenn eine Asynchrone Funktion aufgerufen wird, blockiert diese nicht den Ablauf des Programms.

Wenn man das Schlüsselwort `async` vor eine Funktion platziert, wird diese Funktion immer ein `Promise` zurückgeben. Für den Fall, dass der Rückgabewert der Funktion nicht den Typ `Promise` hat, wird dieser Wert automatisch in ein `Promise` umgewandelt. [10]

Dies sieht in der Umsetzung wie folgt aus:

```
 async function (): Promise<number> {
     let zahl: number = 10;
     return zahl;
}

async function (): Promise<number> {
     let zahl: number = Promise.resolve(10);
     return zahl;
}
```

Await

Das Schlüsselwort `await` kann nur innerhalb einer Funktion mit dem Schlüsselwort `async` aufgerufen werden. `Await` lässt unseren JS Code solange warten, bis dass `Promise` aufgelöst wurde und danach kann mit diesem Wert weitergearbeitet werden, falls möglich. Man kann auch mehrere `Promises` an aneinander ketten, so dass sie immer auf den Vorgänger warten bevor sie sich auflösen. Für den Fall, dass ein `Promise` einen Fehler ausgibt, kann man diesen mit einem `try..catch` Block abfangen. [10]

(Hier Beispielcode einfügen)

Diese Technologie wird in der später folgenden Umsetzung für die Asynchrone Audio Daten Speicherung genutzt. Diese soll unabhängig voneinander stattfinden könne sowie zeitlich flexibel angelegt sein.

Fetch API

Das Abrufen von Daten wird dem Nutzer durch die Fetch API zur Verfügung gestellt. Im Gegensatz zum XMLHTTPRequest, auf welchen im späteren Verlauf dieses Kapitels noch eingegangen wird, bietet die Schnittstelle der Fetch API einen mächtigeren und flexibleren Satz an Funktionen.

Das Konzept der Fetch API beschreibt generische Request und Response Objekte, die Netzwerkanfragen und -antworten darstellen. Diese können im weiteren Verlauf auch für andere Technologien eingesetzt werden, die dieses Verfahren nutzen, wie z.B. Service Worker oder die Cache API, falls das im späteren Verlauf für FUDGE benötigt wird.

Die Erstellung einer Anfrage zum Abruf von Ressourcen wird durch die fetch() Methode gestartet und ist in verschiedenen Schnittstellen implementiert. Da für diese Implementation `Window` relevant ist, ist es möglich die Ressourcen in fast allen Kontexten abzurufen.

Die Methode benötigt ein Argument, in Form einer URL als String zum Pfad der gewollten Datei, und ein optionales `Init` Objekt in der einige Einstellungen zur Übertragung beschrieben werden. Die Antwort darauf stellt ein Promise Objekt bereit, dass in ein Response Objekt aufgelöst wird. [11]

Um die Nutzung der Fetch API zu veranschaulichen, dient das folgende Beispiel. Hierbei wird ein Request erstellt welcher anhand der URL per GET eine Audio Ressource abfrägt:

```
let initObject: RequestInit = {
    method: "GET",
    mode: "same-origin",
    cache: "no-cache",
    headers: {
        "Content-Type": "audio/mpeg3"
    },
    redirect: "follow"
};

const response: Response = window.fetch(URL, initObject);
```

XMLHTTPRequest ist die Alternative zur Fetch API, welche grundsätzlich auch in Verbindung mit der Web Audio API genutzt werden kann. XHR ist ein Teil von JavaScript und wird vom W3C standardisiert.

Diese kann auch dazu genutzt werden um Daten über eine URL abzurufen, ohne eine vollständige Aktualisierung der Webseite durchzuführen. Dies bietet die Möglichkeit einen Teil der Webseite zu verändern, ohne damit die Interaktion des Nutzers zu stören.

Der Grund, weshalb die Implementierung die Fetch API anstatt des gängigen XHRs verwendet, ist das Paradigma von FUDGE nur die aktuellsten Webtechnologien zu verwenden. [12]

2.2 FUDGE

Der Furtwangen University Didactic Game Editor, oder kurz FUDGE, ist ein Projekt der Fakultät der Digitalen Medien an der Hochschule Furtwangen um eine Umgebung zu schaffen in dem Studenten die Grundlagen zu Game- und Interaktionsdesign kennenlernen. Die gängigen Werkzeuge sind bis zum jetzigen Zeitpunkt bisher Adobe Animate für zwei- und Unity für dreidimensionale Anwendungen. Beide Umgebungen liefern einen visuellen Editor mit dem Szenen aufgebaut, Assets erstellt und Skripte an Objekte in der Szene angehängt werden können. Dies erleichtert den Einstieg der Studenten in die Spieleentwicklung. Neben diesen zwei Game-Engines finden auch Unreal Engine, Phaser und FUSEE, die erste in der HFU erstellte Engine bei fortgeschrittenen Studenten einen Platz. Jedoch enthalten alle erwähnten Anwendungen große Lücken, wenn es um den didaktischen Bereich geht, was zur Erstellung von FUDGE geführt hat.

FUDGE ist ein kompakter Open-Source Game Editor, welcher in akademischen Bereichen Anwendung finden soll. Studenten sollen damit sowohl Prototyping durchführen als auch komplette audiovisuelle digitale Spiele im zwei- und dreidimensionalen Raum erstellen. FUDGE kann auf allen gängigen Desktop Geräten (Windows, Mac, Linux) und kann die Ergebnisse für Plattformen wie

Desktop, Mobilgeräten und im Web bereitstellen. Dies ist relevant damit die Anwendungen auf einfache Art und Weise, für Studenten oder Professoren, bereitgestellt werden können.

FUDGE soll keine Konkurrenz zu vollwertigen Game-Engines darstellen. Außerdem werden keine regelmäßigen Updates durchgeführt, was zu Inkompatibilitäten führen kann. Der Code ist komplett sichtbar, auch zur Laufzeit, was zu Problemen in der Sicherheit führen kann. Auch die Performanz spielt bei der Entwicklung von FUDGE keine große Rolle.

Die Hauptmerkmale von FUDGE sind folgende:

- Visuelle Editoren für Szenen, Meshes, Grafiken und Animationen um Assets innerhalb der Umgebung herstellen zu können. Davon sind Audiodateien ausgeschlossen.
- Dokumentformate, die für Menschen lesbar und verständlich sind. Das ermöglicht die Untersuchung und Manipulation von Datenstrukturen.
- Keine Trennung der Daten der Entwicklungszeit und der Laufzeit. Anwendungen können somit direkt aus dem Source-Repository ausgeführt werden.
- Unterstützt eine Versionskontrolle, Code Revisionen und eine Kommunikationsstruktur zwischen Studenten, Lehrkräften und Entwicklern.
- Client- und Serverkomponenten um Netzwerkbasierte Spiele zu ermöglichen.

Die Basis für FUDGE bilden moderne Webtechnologien, unter anderem gehören dazu HTML5, CSS, JavaScript, TypeScript auf welche im späteren Verlauf der Arbeit noch genauer eingegangen wird (Kapitel 3.1). Aber auch Technologien wie Node.js, WebGL, WebAssembly und Electron gehören zu dieser Basis.

Das Hauptziel von FUDGE ist es, Strukturen und Prozesse von Spielen und deren Entwicklung deutlich zu machen. Die Verwendung von FUDGE soll dabei so einfach wie möglich gehalten werden ohne dabei an Flexibilität der

Anwendung einzubüßen. Generische Anwendungen sollen ebenfalls erhalten bleiben. Alle Design Entscheidungen sollen dies Berücksichtigen, wobei Aspekte wie Performanz und visuelle Effekte eine geringere Priorität erhalten. [5]

3. Web Audio API

Im folgenden Kapitel beschäftigen wir uns mit der Web Audio API, welche zu den Webtechnologien gehört. Aufgrund ihrer Wichtigkeit für diese Arbeit, wird sie in einem eigenen Kapitel behandelt. Die Integration der Web Audio API in FUDGE, ist der Kern für den Ansatz dieser Arbeit, weswegen in diesem Kapitel die wichtigsten Elemente und Vorgehensweißen der WAA kurz erklärt werden.

3.1 Einleitung

Der erste Browser der eine Implementation für Audio im Web hatte, war der Internet Explorer, welcher mit dem <bgsound> Tag aufgerufen werden konnte. Dieser Tag wurde allerdings nie von anderen Browsern übernommen oder standardisiert. Anschließend wurde Flash genutzt um Audio auf allen damals gängigen Browsern im Web abzuspielen, welcher aber den Nachteil besaß ein Plug-In nutzen zu müssen. Mit dem Anbruch von HTML5 wurde allen modernen Browsern die Möglichkeit geliefert das <audio> Tag zu nutzen. Allerdings sind die größten Nachteile des <audio> Tags Folgende:

- Keine präzise Zeitsteuerung
- Limitierung an parallel abzuspielenden Sounds
- Keine Möglichkeit Echtzeit Effekte anzuwenden

Seitdem gab es einige Versuche eine stärkere API auf Basis des <audio> Tags zu erschaffen, wie z.B. die Audio Data API von Mozilla, welche aber inzwischen von der Web Audio API als Standard ersetzt wird. [13]

Die Web Audio API (WAA) liefert dem Nutzer ein mächtiges und vielseitig nutzbares System um Ton im Web zu kontrollieren. Der Entwickler erhält die Möglichkeit, Töne mit beliebigen Quellen zu erstellen und diesen eine unbegrenzte Anzahl an Effekten, wie Filter oder Lautstärken, hinzuzufügen. Zusätzlich bietet die WAA eine räumliche Darstellung aller Töne an, welche in den nächsten Kapiteln dieser Arbeit noch eine große Rolle spielen wird.

Die WAA ist mit allen gängigen Browsern kompatibel, vor allem mit den neusten Versionen von Chrome, Edge, Firefox und Safari. Diese Kompatibilität gilt auch für die Nutzung auf mobilen Geräten mit den gängigen Browsern. [14]

Zudem wird für die Nutzung der WAA keine zusätzlichen Abhängigkeiten benötigt, da diese alle standardmäßig in JS integriert sind.

Ein typischer Workflow der WAA sieht wie folgt aus:

1. `AudioContext` erstellen

2. Audioquellen erstellen, die innerhalb des `AudioContext` arbeiten (z.B. Audio Nodes, Oszillatoren oder Audio Streams).

3. EffektNodes erstellen (z.B. Filter, Panner, Kompressoren)

4. Den Bestimmungsort für den `AudioContext` festlegen, welcher in dieser Arbeit immer das Ausgabegerät des Systems, also die PC-Lautsprecher, sein werden.

5. Zum Schluss müssen alle Audioquellen mit den entsprechenden Effekten und dem Ausgabegerät verbunden werden. [14]

Abbildung 2: Web Audio API Workflow [14]

3.2 Schnittstellen

Dieses Kapitel dient dazu dem Leser einen groben Überblick über die WAA zu verschaffen. Da die WAA eine sehr mächtige API ist, wird nur auf die wichtigsten Schnittstellen, sowie Datentypen eingegangen und erklärt.

Zuerst werden wir auf den `AudioContext` eingehen und diesen beschreiben. Danach werden Audioquellen und deren Umwandlung in einen abspielbaren Datentyp. Zum Schluss wird darauf eingegangen, wie Audio lokalisiert werden kann, so dass dabei eine räumliche Wahrnehmung entsteht.

Audio Context

Die Erstellung eines `AudioContext` ist der erste Schritt des Nutzers zur Verarbeitung von Audio im Web. Dieser `AudioContext` muss erstellt werden, da die folgende Verarbeitung von Audio gänzlich in diesem `AudioContext` geschieht.

Der AudioContext stellt dem Nutzer einen Weg bereit, Audiodateien zu bearbeiten, als wäre diese ein Diagramm zur Audioverarbeitung. Audio Module werden dort miteinander verbunden und werden jeweils durch ein `AudioNode` dargestellt.

Dabei kontrolliert ein `AudioContext` sowohl die Erstellung dieser `AudioNodes`, als auch die Ausführung der Decodierung und des Verarbeitungsprozesses.

Der `AudioContext` besitzt auch eine zusätzliche zeitliche Eigenschaft, die wie eine Zeitleiste arbeitet und bei Start des `AudioContext` Objekts beginnt und dann kontinuierlich hochgezählt wird. Diese Zeit beginnt bei 0 und wird in Echtzeit erhöht, so dass mit ihr Ereignisse zeitlich festgelegt werden können. Durch die Eigenschaft `AudioContext.currentTime` kann die Zeit genau ausgelesen werden.

Die grundsätzliche Deklarierung eines Audio Kontextes in TS ist die Folgende:

```
let audioCtx: AudioContext = new AudioContext();
```

Bei der Erstellung eines `AudioContext` können optionale Parameter gesetzt werden, welche Optionen für die Latenz darstellen. Dabei ist der Standardwert von `latencyHint: “interactive”`, welcher den Browser dazu bringt den niedrigst möglichen Latenzwert anzuwenden, welcher dennoch zuverlässig arbeitet. SampleRate stellt die zweite Option dar. Hier wird standardmäßig ein

Wert von 44100 Hz genutzt. Im Normalfall sollte sich dieser zwischen 8000 Hz und 96000 Hz befinden. Dieser Wert stellt die Abtastrate dar und wird mit dem Typ `number` angegeben. Diese Latenz passt sich dem Ausgabegerät an, falls dieses den angegeben Wert nicht verarbeiten kann. [13][14][15]

Um damit den vorangegangen, vereinfachten Workflow zu vervollständigen, ist hier noch ein Beispiel in TS:

```
let oscNode: OscillatorNode = audioCtx.createOscillator();
let gain: gainNode = audioCtx.createGain();
let dest: AudioDestinationNode = audioCtx.destination;
oscNode.connect(gain);
gain.connect(dest);
```

Audio Nodes

Ein `AudioNode` repräsentiert eine Vielzahl von verschiedenen Audio Modulen. Darunter befinden sich vor Allem folgende:

- Audioquellen, z.B. `AudioBufferSourceNode`, `OscillatorNode` oder auch ein HTML `<audio>` Element
- Ausgabegeräte als `AudioDestinationNode`
- Audio Verarbeitungsmodule, wie z.B. ein Filter mit dem `BiquadFilterNode` oder das `ConvolverNode`
- Lautstärke Regelung mit z.B. dem `GainNode`

Jede `AudioNode` hat Eingaben sowie Ausgaben, die dem Nutzer die Möglichkeit geben eine Vielzahl dieser `AudioNodes` nach Belieben zu kombinieren. Jeder AudioNode hat seine eigene Art der Verarbeitung des Audios, gleichen sich aber dabei, dass jede die erhaltenen Daten auf eine eigene Weise verarbeitet und darauffolgend neue Werte weitergibt.

Je größer das komplette Gerüst des Audio Diagramms wird, desto höher wird die Latenz, bis diese abgespielt werden kann, da die Dauer der Verarbeitung damit zunimmt.

Microsoft Edge unterstützt den Gebrauch eines Konstruktors von AudioNodes nicht, weshalb man hier auf eine Methode ausweichen muss, bei der alle Parameter einzeln gesetzt werden müssen. [14][15]

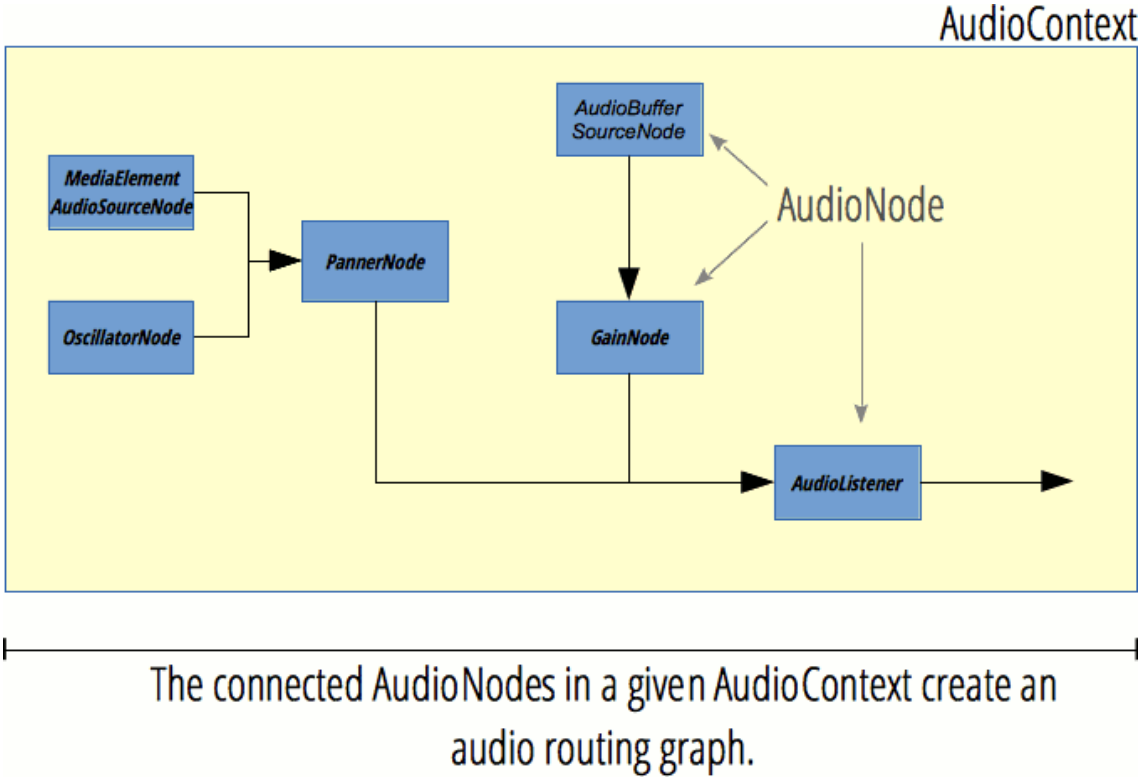

Abbildung 3: Ein Audioverarbeitungsdiagramm [14]

Audio Buffer

Ein `AudioBuffer` wird erstellt, indem eine Audio Datei mithilfe der Fetch API vom Server gezogen wird. Aus dieser Antwort wird der `ArrayBuffer` benutzt, welcher durch die Methode `AudioContext.decodeAudioData()` decodiert wird. Hier kann auch ein `AudioBuffer` verwendet werden, der selber durch Code erstellt wurde und diese mit der Methode `AudioContext.createBuffer()` zu einem `AudioBuffer` umwandeln. Anschließend kann der `AudioBuffer` weiter benutzt werden, z.B. in einem `AudioBufferSourceNode` mit welchem dann ein Ton abgespielt werden kann. Mit folgender Methodik kann eine Datei in einen `AudioBuffer` umgewandelt werden und in einem `AudioBufferSourceNode` angewendet zu werden:

```
let response: Response = window.fetch(URL);
let arrayBuffer: ArrayBuffer = response.arrayBuffer();
let decodedAudio: AudioBuffer =
    AudioContext.decodeAudioData(arrayBuffer);
let bufferSource: AudioBufferSourceNode =
    AudioContext.createBufferSource();
bufferSource.buffer = decodedAudio;
```

Das `AudioBufferSourceNode` ist dafür ausgelegt kurze Audio Daten wiederzugeben, meistens nicht länger als 45 Sekunden. Für längere Audio Daten gibt es die Möglichkeit, `MediaElementAudioSourceNode` zu nutzen. [14][15]

Audio Listener

Der `AudioListener` stellt die Position und Ausrichtung eines Zuhörers da, welcher in einer dreidimensionalen Szene steh. In jeder Szene gibt es nur einen einzigen `AudioListener`.

Der `AudioListener` benötigt somit einige positionsbezogene Parameter um zu erkennen, welche Position und Ausrichtung das Objekt, welches den `AudioListener` trägt, besitzt.

Der erste Parameter ist die Position in einem rechtshändigen kartesischen Koordinatensystem. Dafür wird ein Objekt vom Typ `Vector3` benötigt mit den gewünschten Werten der X, Y und Z Position.

Um den zweiten Parameter, die Orientierung festzulegen, werden zwei weitere Vector3 Objekte benötigt. Das erste Objekt ist das `forward` Objekt, welches eine nach vorne versetzte Variante der Ausgangsposition ist. Dieses gleicht der Nase des `AudioListeners`. Somit wird der erste `Vector3` als Standartwert um -1 in Richtung Z versetzt.

Das `up` Objekt ist ein weiteres `Vector3` Objekt, welches nun in einem 90 Grad Winkel nach oben von der Ausgangsposition versetzt wird. In diesem Fall wird der Wert in Richtung Y standardmäßig um 1 erhöht.

Die Methode `setValueAtTime(Pos, audioContext.currentTime)` gibt uns die Möglichkeit die Positionen zu einem bestimmten Zeitpunkt in Relation der Zeit des AudioContext Objekts zu verändern. [14][15]

Um auf diese positionsbezogenen Werte zuzugreifen kann man den `AudioListener` wie folgt per Code ansprechen:

```
audioContext.listener.positionX.setValueAtTime(xPosition,
      audioContext.currentTime);
audioContext.listener.forwardZ.setValueAtTime(zPosition,
      audioContext.currentTime);
audioContext.listener.upY.setValueAtTime(yPosition,
      audioContext.currentTime);
```

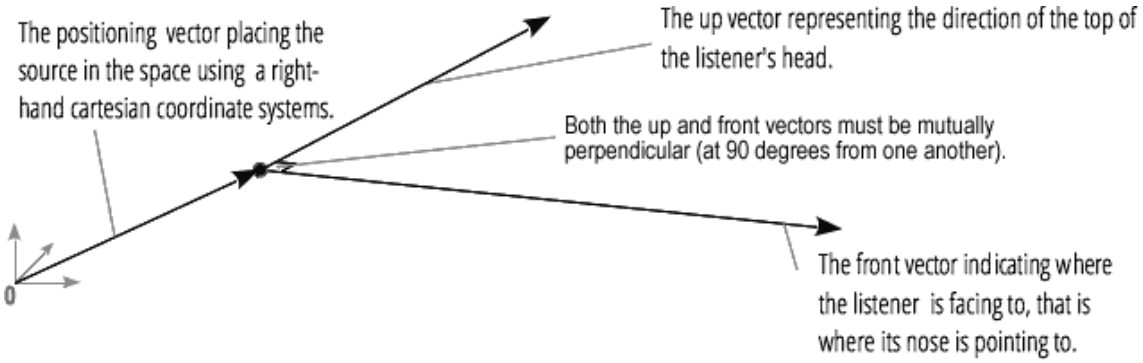

Abbildung 4: Funktionsweise eines Audio Listener Objekts der Web Audio API [14]

Panner Node

Ein `PannerNode` Objekt stellt die Position, die Orientierung und das Verhalten einer Audioquelle im dreidimensionalen Raum dar. Alle `PannerNode` Objekte verräumlichen Audio immer in Relation zum `AudioListener` Objekt.

Die Position des `PannerNode` Objekts wird, wie auch schon beim `AudioListener` durch ein Vector3 Objekt übergeben. Die Orientierung besteht beim `PannerNode` allerdings nur aus einer einzigen weiteren Position, in dessen Richtung die Audio Signale ausgestrahlt werden.

Für ein `PannerNode` Objekt werden zusätzlich zur Position und Orientation noch einige zusätzliche Parameter benötigt.

Die Distanz des `AudioListeners` zu einem `PannerNode` Objekt bestimmt die Lautstärke der Audioquelle. Die dazu benötigten Umrechnungsverfahren stecken im `panningModel` und die passende Berechnung für die Abnahme der Lautstärke wird durch das `distanceModel` beschrieben.

Zusätzliche Werte bestimmen die Distanz in dem der Abfall der Lautstärke beginnt und auch die Grenze, bei der die Lautstärke nicht mehr weiter abnehmen soll. Auch die Geschwindigkeit, mit der die Lautstärke abnimmt kann eingestellt werden.

Um den eigentlichen Weg des Audios wird ein Kegel gebildet, in dem die Töne zu hören sind. Die Größe dieses Kegels kann ebenfalls eingestellt werden indem man den inneren Winkel angibt. Gegenüber steht der äußere Winkel, in dem kein Kegel erstellt wird. Man kann die Lautstärke außerhalb des Kegels anpassen, wie es auch im inneren des Kegels möglich ist. [14][15]

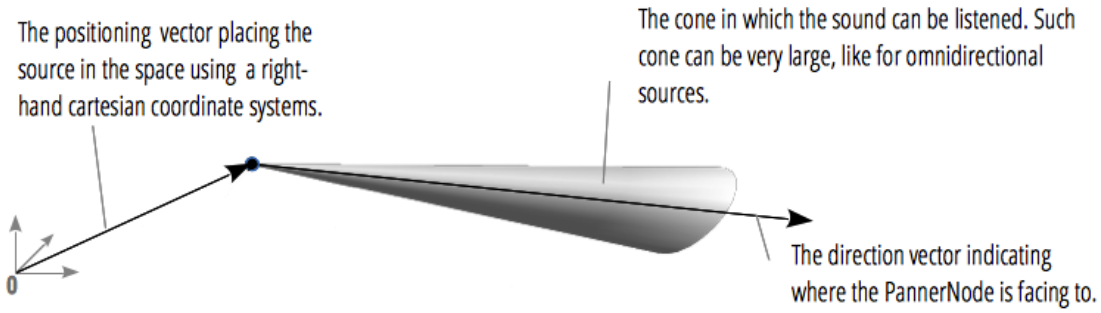

Abbildung 5: Funktionsweise eines Panner Nodes der Web Audio API [14]

Audio Destination Node

Der `AudioDestinationNode` zeigt das Ende eines Audio Graphen innerhalb eines AudioContext Objekts. Es wird als `AudioNode` gesehen aber besitzt keinen Output mehr, was bedeutet, dass keine weiteren `AudioNode` Objekte mehr hinzugefügt werden können. Ein `AudioDestinationNode` besitzt aber nur einen Input, was bedeutet, dass alle vorherigen `AudioNode` Objekte bereits

zusammengefasst sein müssen und anschließend mit dem `AudioDestinationNode` verbunden werden. [14][15]

Der Ausgang wird dem Ausgabegerät, in den meisten Fällen sind dies die Boxen des benutzten Gerätes, zugewiesen und darüber ausgegeben.

```
source.connect(audioContext.destination);
```

4. Umsetzung der Audio Implementation

In diesem Kapitel geht es um die Implementation der im letzten Kapitel angesprochenen Objekte der Web Audio API im Rahmen von FUDGE. Zuerst werden die Anforderungen an die Implementation angesprochen (Kapitel 4.1) und anschließend in 4.2 in einem Konzept zusammengefasst. Alle Klassen werden anschließend beschrieben und erklärt. In Kapitel 4.3 wird die Kommunikationsstruktur zwischen den Klassen erklärt und deren Interaktion nochmals verdeutlicht. Zum Schluss werden in Kapitel 4.4 einige Beispielszenen mit den wichtigsten Funktionalitäten erstellt und erklärt.

4.1 Anforderungen

Es gibt einige Vielzahl von Anforderungen, die die Web Audio API in FUDGE erledigen soll. Da FUDGE als Anwendung für die Hochschule gestaltet wird, muss eine einfache Anwendung der dargelegten Methoden des WAA gegeben sein. Diese müssen auch an die bereits aufgestellte Datenstruktur von FUDGE angepasst werden und mit ihr verwendbar sein. Da FUDGE eine komponentenbasierte Struktur, die auch in Unity (siehe Abbildung 6) oder der Unreal Engine genutzt werden, soll eine ähnliche Anwendungsform gegeben sein. Da FUDGE allerdings noch keine lauffähige GUI hat, werden diese zu einem späteren Zeitpunkt nutzbar sein. Vorerst werden die dargelegten Komponenten durch Code in einer Beispielszene untergebracht. Die Anwendung muss eine hohe Flexibilität besitzen, damit der Nutzer seine Anwendungen individuell gestalten kann. Deswegen muss ein hoher Grad an Einstellungsmöglichkeiten vorhanden sein.

In einer Szene soll eine Vielzahl an Objekten mit einer Audio Komponenten ausgestattet werden können, die interaktiv und zu jedem Zeitpunkt einen Ton abspielen können sollen. Dafür müssen Audio Dateien frei ausgewählt werden können. Diese Dateien müssen in den Audio Komponenten referenziert werden und decodiert werden um das Abspielen eines Sounds zu ermöglichen. Da in

einem dreidimensionalen Raum gearbeitet, müssen die Sounds ebenfalls diesen dreidimensionalen Raum wiedergeben können.

Sounds alleine reichen nicht um ein interaktives Erlebnis räumlich wirken zu lassen. Dafür brauchen diese Sounds zusätzliche Schichten um diese den gegebenen Umständen anzupassen. Damit werden Lautstärken, Filter und Verzögerungen zusätzlich ein relevanter Teil der Arbeit.

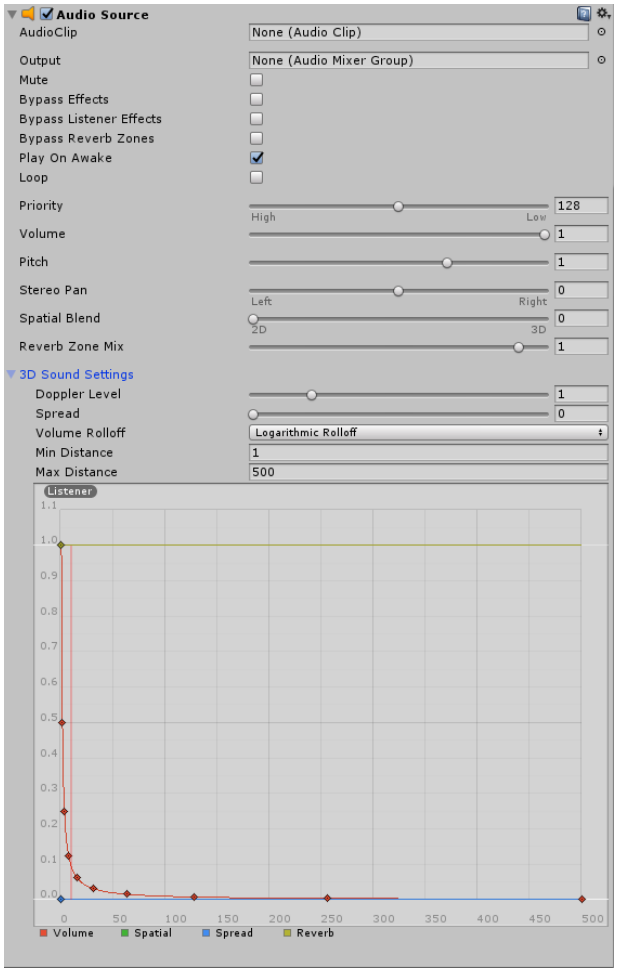

Abbildung 6: Audio Komponente aus Unity

4.2 Konzept

Der generelle Aufbau des Konzepts dieser Arbeit wird in drei Teilbereiche unterteilt:

1. Ein Globaler Bereich, in dem ein Audio Kontext Objekt erstellt wird, der später von allen Audioquellen und dessen Steuerung genutzt wird. Außerdem wird hier die oberste Schicht des Audioverarbeitung mit der globalen Lautstärke und dem darüber liegenden Kompressor erstellt. Dieser Bereich dient zusätzlich auch als Speicherort für alle decodierten Daten.
2. In diesem Bereich stehen die Komponenten, die in FUDGE genutzt werden können. Dabei dient die `AudioListener` Komponente als Zuhörer und die `Audio` Komponente dient als Audioquelle. Diese beiden Komponenten interagieren miteinander und werden beide benötigt um dreidimensionalen Sound zu erzeugen.
3. Der dritte Bereich dient dazu alle Daten, die die Audio Komponente benötigt, aus der Komponente selber auszulagern und diese aufzuteilen. Die Erstellung einer Komponente beinhaltet lediglich eine Audio Datei, kann jedoch aber um zusätzliche Elemente erweitert werden.

In den folgenden Kapiteln werden alle Klassen separat beschrieben und erklärt.

4.2.1 AudioSettings

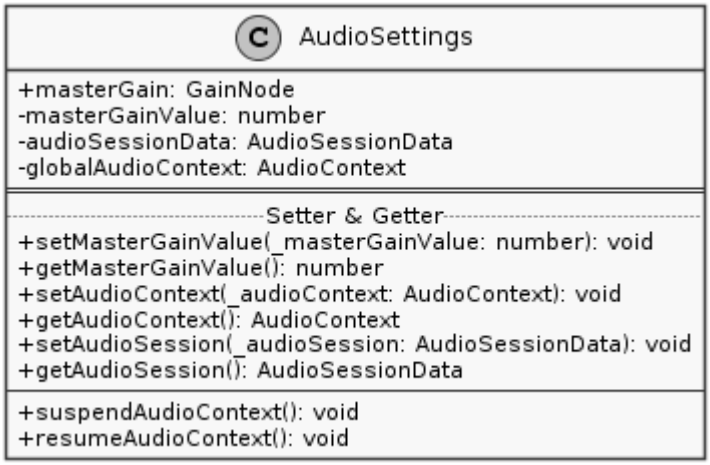

Abbildung 7: Klassendiagramm der AudioSettings Klasse

Die Klasse `AudioSettings` stellt eine Hauptkomponente des Audio Systems dar. In ihr wird ein `AudioContext` erstellt, welcher den Kernpunkt und Rahmen für alle zusätzlichen Instanzen des Audio Systems darstellt. Das heißt immer, wenn Audio Daten verarbeitet werden, muss diese Klasse als erstes angelegt werden. Im späteren Verlauf ist geplant, dass diese Klasse als ein Menüpunkt angelegt wird sobald FUDGE gestartet wird, damit diese Klasse frei zugänglich wird. Der `AudioContext` kann aus dieser Klasse auch angehalten und wieder gestartet werden, falls der Nutzer Sounds über einen bestimmten Zeitraum stoppen möchte.

In dieser Klasse wird auch ein `GainNode` erstellt, der als globale Lautstärke agiert. Diese Lautstärke Regulierung betrifft alle Audioquellen, und kann separat zu deren lokaler Lautstärke eingestellt werden. Alle Laustärken können mit 0 stummgeschaltet werden und bis zu 1 aufgedreht werden, welche die volle Lautstärke darstellt. Dies bezeichnet die oberste Ebene des Audiographen und verbindet den `GainNode` der globalen Lautstärke mit dem `AudioDestinationNode`.

Die `AudioSettings` Klasse erstellt sich zusätzlich noch ein `AudioSessionData` Objekt, welches sich um alle Audio Daten kümmert. `AudioSessionData` war in

der früheren Bearbeitungsphase von dieser Klasse getrennt, allerdings wird beides benötigt und es werden viele zusätzliche Referenzen auf diese beiden Klassen benötigt, weshalb diese nun vereint wurden.

4.2.2 AudioSessionData

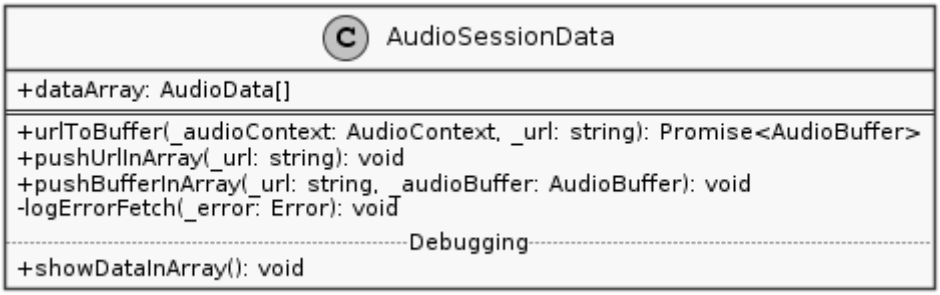

Abbildung 8: Klassendiagramm der AudioSessionData Klasse

Diese `AudioSessionData` Klasse dient als Speicherort für alle in einer Szene geladenen und gespeicherten Audio Daten. Diese Daten werden aus einer Referenz der `Audio` Klasse ausgelesen und bei dem Versuch die Audio Datei abzuspielen an das Objekt dieser Klasse weitergegeben. Dafür wird dafür ein Array erstellt, das mit Datenpaaren gefüllt wird die ausgelesen werden können. Diese Daten bestehen aus einer URL mit dem Typ `String` und einem in dieser Klasse decodierten `AudioBuffer`. Sobald der Nutzer also einen Ton z.B. durch einen Knopfdruck abspielen will, wird das Array nach einem Eintrag der URL durchsucht. Falls nicht vorhanden wird ein neuer Eintrag in angelegt, mit einem leeren Objekt als Platzhalter für den `AudioBuffer`. Der passende Buffer wird durch einen Asynchronen Aufruf der Fetch API abgeholt, decodiert und dem Eintrag des Arrays im Nachhinein hinzugefügt.

Diese Methodik wird genutzt, damit der User nicht alle Töne frühzeitig eintragen und laden muss, damit zu Beginn keine längere Wartezeit entsteht. Der Nachteil daran ist jedoch, dass die erste Nutzung des Tones möglicherweise eine kurze Verzögerung aufweist. Das Array wird allerdings angelegt, damit der Nutzer bereits genutzte Töne nochmals verwenden kann, ohne dass diese

komplett neu geladen werden müssen. Danach kann der `AudioBuffer` jederzeit nochmal genutzt werden, ohne weitere Verzögerungen. Somit kann der Nutzer alte Einträge auch wieder löschen, wenn sie nicht mehr benötigt werden. Da diese Anwendung Asynchron abläuft, können Einträge auch gleichzeitig angelegt und geladen werden.

4.2.3 ComponentAudioListener

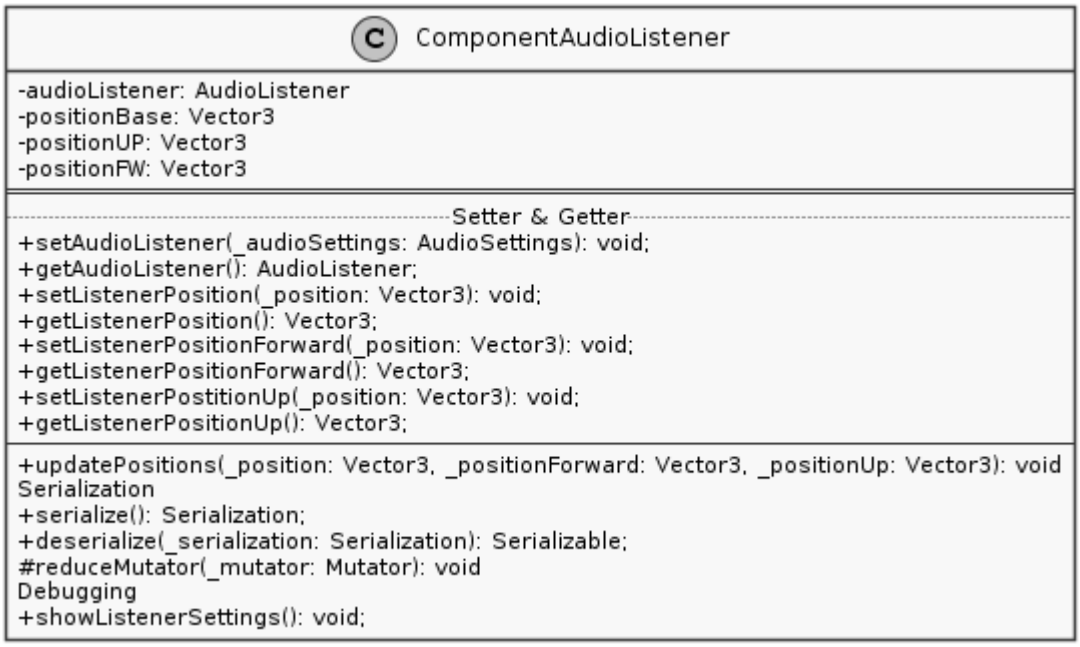

Abbildung 9: Klassendiagramm der ComponentAudioListener Klasse

Die erste Komponente der Arbeit verfügt über ein `AudioListener` Objekt. Dieses Objekt wird erstellt und dem Audio Kontext hinzugefügt, welcher diesen AudioListener dann als Standard `AudioListener` nutzt. Alle Audio Komponenten werden dann von diesem `AudioListener` wahrgenommen. In jeder Szene kann es nur einen aktiven `AudioListener` geben, nämlich den der im Audio Kontext referenziert ist. Man kann falls gewollt mehrere `AudioListener` Komponenten anlegen und diese dann an den `AudioContext` anschließen.

Die AudioListener Komponente besteht aus drei voneinander abhängenden Positionen. Eine Position wird von dem jeweiligen Objekt bestimmt an dem die Komponente angehängt wird. Es gibt zusätzlich noch eine nach vorne gerichtete Position und eine nach oben gerichtete Position. Es spielt keine Rolle wie weit diese Positionen von der Basis Position entfernt sind, allerdings müssen die anderen zwei immer in einem rechten Winkel zueinanderstehen. Damit wird ermöglicht die Töne richtig darzustellen.

4.2.4 ComponentAudio

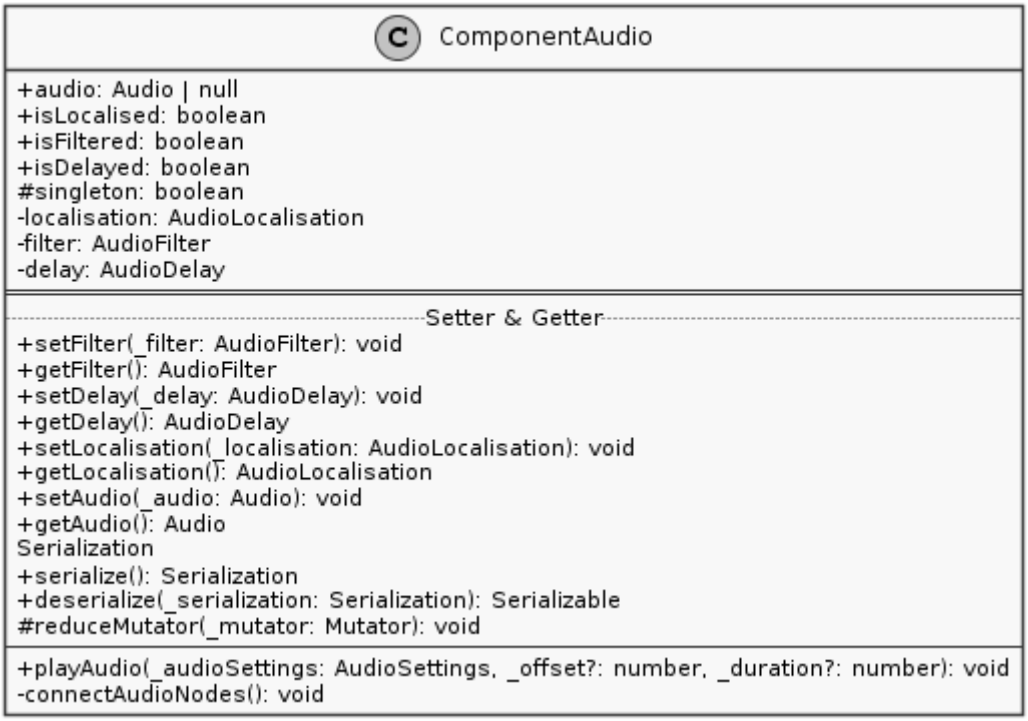

Abbildung 10: Klassendiagramm der ComponentAudio Klasse

Die zweite Komponente, die in FUDGE angewendet werden soll, ist die `ComponentAudio`. Sie dient als Gerüst, in dem die Daten aus den folgenden Kapiteln zusammengesetzt werden. Als Komponente kann dieses Objekt an FUDGE Nodes gehängt werden. Es ist hier auch möglich mehrere Audio Komponenten an dieselbe Node zu hängen, da verschiedene Töne vom selben Objekt ausgehen können. Das ist wichtig, da es nur möglich ist, eine einzelne

Audioquelle in dieser Komponente zu lagern. Dasselbe gilt auch für Filter, Verzögerungen und die Lokalisierung. Diese Entscheidung wurde getroffen, da die späteren Nutzer von den vielen Möglichkeiten der WAA und ihrer Kombinatorik nicht überfordert werden, sondern eine vereinfachte Anwendung erhalten um die grundlegenden Prinzipien zu erlernen.

Wie gerade erwähnt besteht die Komponente aus einem Audio Objekt, welcher die Audioquelle enthält. Zusätzlich gibt es drei weitere Schichten mit dem Filter, dem Delay und der Lokalisierung. Diese werden nur einmal hinzugefügt. Hier könnte ein Array an Filtern angelegt werden und dieses hintereinander koppeln falls gewollt, für die anderen Objekte ist eine Vielzahl nicht sinnvoll bzw. nicht möglich.

Die `Audio` Komponente hat noch eine wichtige Aufgabe, welche das Abspielen der Töne darstellt. Diese Töne werden hier auch aneinandergereiht, damit alle Schichten genutzt werden können.

4.2.5 Audio

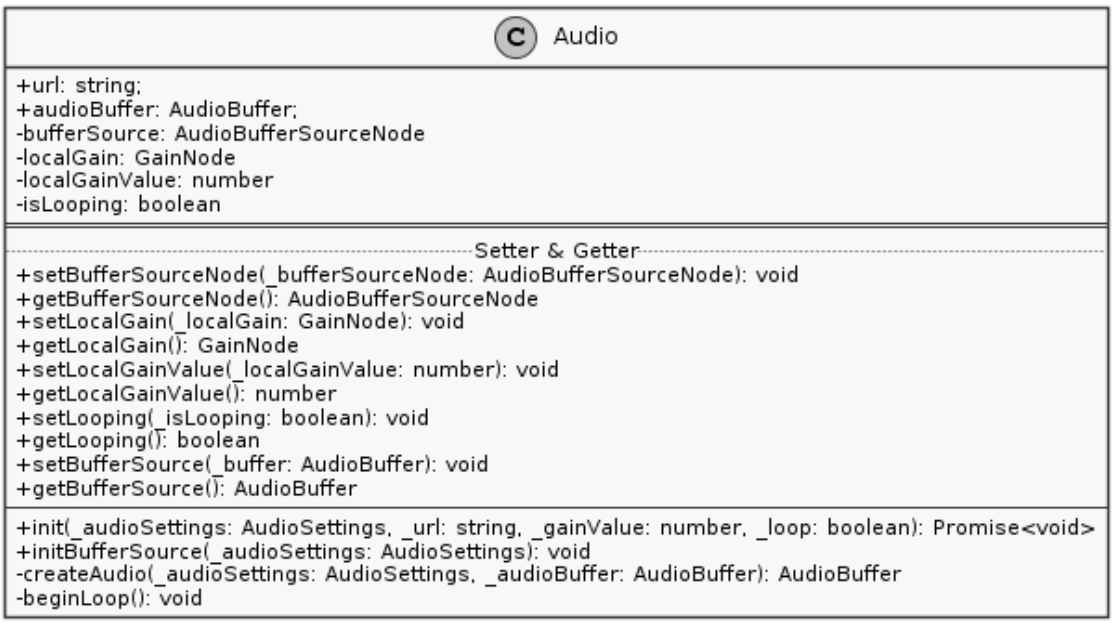

Figure 1: Klassendiagramm der Audio Klasse [16]

Diese Klasse stellt den Kern der Audio Komponente dar. Hier werden Referenzen zu Audio Daten aufgenommen und mithilfe der `AudioSessionData` decodiert. Auch diese Decodierten Daten lassen sich hier nochmals auffinden. Aus diesem `AudioBuffer` wird bei gebrauch bei Laufzeit ein `AudioBufferSourceNode` Objekt generiert, welche benötigt wird um den gewünschten Ton abspielen zu können. Darin wird der im Vorhinein geholte `AudioBuffer` eingefügt. Die Audio Klasse bietet dem Nutzer eine lokale Einstellung der Lautstärke, damit nicht alle Töne nur durch die globale Lautstärke in den `AudioSettings` eingestellt werden können. Zusätzlich hat die Klasse noch die Möglichkeit eine Schleife aus der Audio Quelle zu erschaffen, das heißt die Quelle wird ohne Pause immer wiederholt.

Diese Klasse dient auch dazu die ersten beiden Schichten eines Audioverarbeitungsgraphs miteinander zu verbinden, welche hier die `AudioBufferSourceNode` in die lokale `GainNode` anschließt.

4.2.6 AudioLocalisation

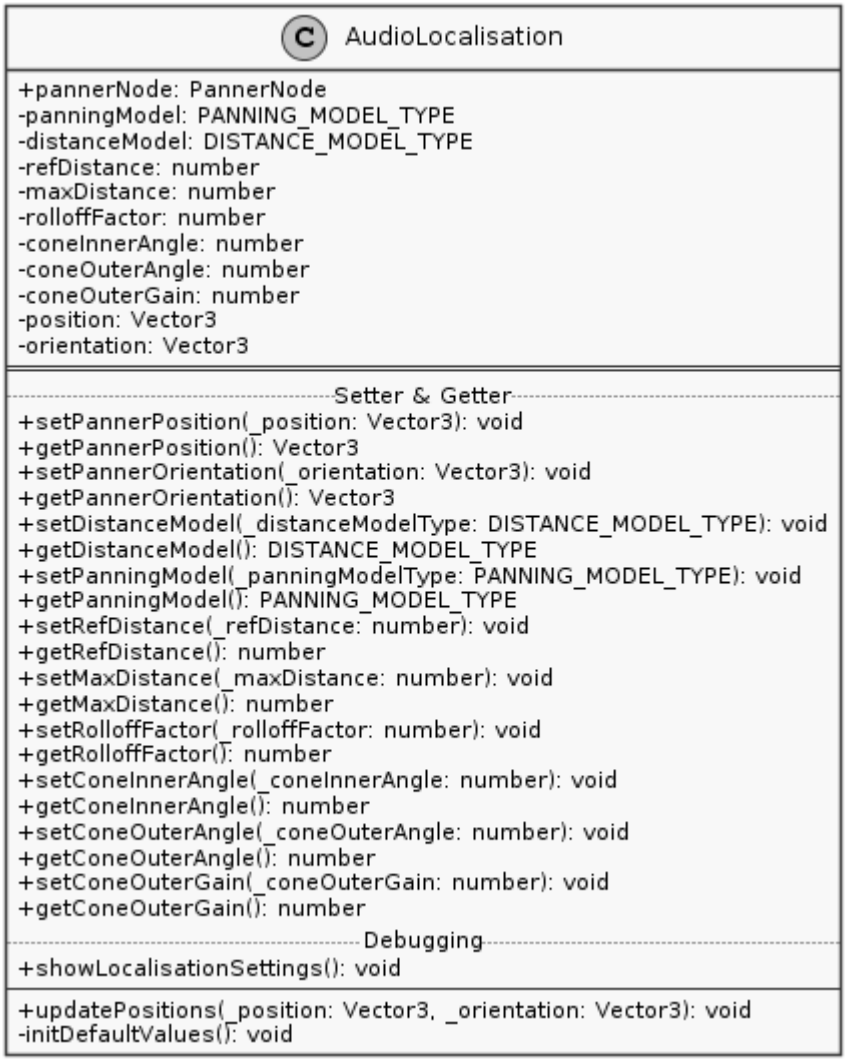

Abbildung 11: Klassendiagramm der AudioLocalisation Klasse

Die `AudioLocalisation` Klasse ist das Kernstück der räumlichen Wiedergabe der Audio Quellen. Hier werden alle dafür benötigten Einstellungen gehalten und können beliebig verändert werden. Die wichtigsten beiden Elemente sind dabei die Position des Objekts, auf dem die Audio Komponente liegt und die Position der Orientierung, das heißt die Position des Ziels in dessen Richtung die Audio Quelle abgefeuert werden soll. Diese beiden Einstellungen brauchen eine Angabe und können auch zu Laufzeit verändert werden, solange die beiden Methoden verändert werden. Dazu kann man diese in der Update

Methode verändern um eine räumliche Veränderung in Echtzeit zu erhalten. Da die Audio Quelle die Töne in einem Kegel um die festgelegten Punkte ausgibt, kann dieser natürlich auch verändert werden. Der innere Winkel besagt um wieviel Grad der Kegel ausgebreitet werden soll, dieser Wert wird in alle Richtungen gleichzeitig ausgebreitet, also wird der Wert pro Seite halbiert. Stellt der Wert 360 Grad da, wird um das komplette Objekt eine Audio Ausgabe durchgeführt. Der äußere Winkel ist abhängig vom inneren Winkel und besagt welche Regionen nicht von der Audioquelle beeinflusst werden. Zusätzlich kann man noch den Abfall der Lautstärke in den äußeren Bereichen festlegen und den Typ der dazugehörigen Berechnungen. Außerdem können noch Einstellung zur Distanz von und bis zu einer hörbaren Audioquelle getroffen werden.

4.2.7 AudioFilter

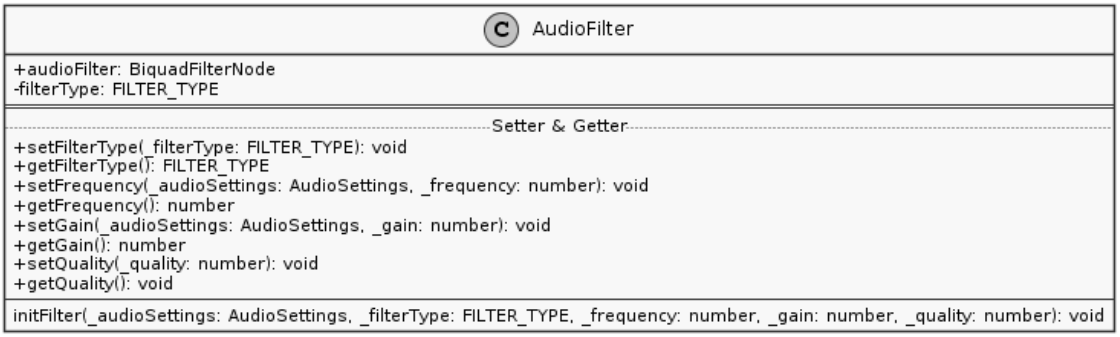

Abbildung 12: Klassendiagramm der AudioFilter Klasse

Die Klasse `AudioFilter` legt, wie der Name schon sag einen Filter vom Typ `BiquadFilterNode` an. Dieser Filter Typ kann durch den Enumerator `FILTER_TYPE` gesetzt werden und dieser wird dann an die `ComponentAudio` Klasse weitergegeben.

Die einzelnen Filtertypen nutzen und mischen die Attribute Frequenz, Qualität und Gain individuell zusammen.

Durch den Einsatz von Filtern kann die räumliche Wahrnehmung von Audio besser gestaltet werden und an die gewünschte Raumstruktur angepasst werden.

4.2.8 AudioDelay

Abbildung 13: Klassendiagramm der AudioDelay Klasse

`AudioDelay` erstellt ein Objekt vom Typ `DelayNode`, welche dafür sorgt, dass eine Soundquelle zeitlich nach hinten verschoben wird. Um diese Soundquelle zeitlich zu versetzten benötigt man lediglich ein einzelnes Attribut des Typen `number`. Dieses Attribut beschreibt die Zeit in Sekunden.

Hier könnte man zusätzlich noch dem `DelayNode` eine minimale und maximale Dauer der zeitlichen Verschiebung definieren. Die minimale Versetzung startet bei 0 und kann nicht in den negativen Bereich fallen.

Durch den Einsatz von einem Delay können nicht nur einfache Verzögerungen dargestellt werden, sondern es können z.B. auch Echo Effekte realisiert werden.

4.2.9 AudioOscillator

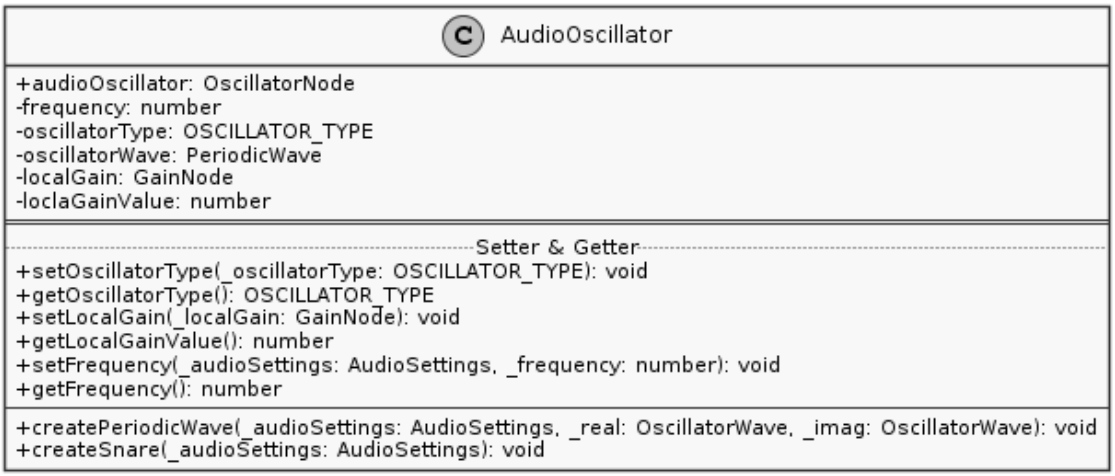

Abbildung 14: Klassendiagramm der AudioOscillator Klasse

Der `AudioOscillator` erstellt einen Oszillator, welcher in einen `AudioBuffer` decodiert wird. Dieser kann ebenfalls an ein `Audio` Objekt angehängt werden. Man kann hier wieder einen der im Enumerator gekennzeichneten Typen auswählen und diese mit einer Frequenz erweitern.

Zusätzlich werden dieser Klasse noch einige Methoden zur Verfügung gestellt, welche eine Reihe an Beispielen für bestimmte Töne abbilden, welche direkt ausgewählt werden können.

4.3 Kommunikation aller Komponenten

Die Kommunikation aller vorliegenden Klassen entspricht der flachen Hierarchie. Die `ComponentAudio` Klasse vereint Objekte aller zugehöriger Klassen und lagert somit größte Teile an Daten aus. Auch die Lagerung von der Audioquellen finden nicht in den Komponenten statt, sondern wird an die `AudioSettings` und `AudioSessionData` Klassen weitergegeben. Die Klasse `AudioSettings` wird von jeder anderen Klasse benötigt, da sie den `AudioContext` hält. Eine komplette Übersicht aller Klassen folgt:

Abbildung 15: UML Diagramm der Komponenten und ihrer Hilfsklassen

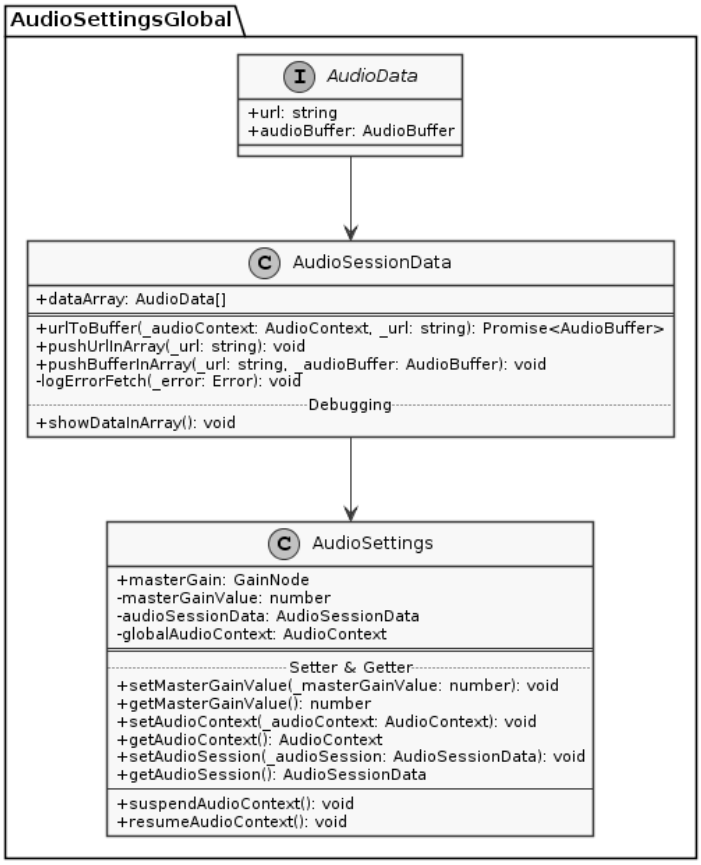

Abbildung 16: UML Diagramm der GlobalAudioSettings

4.4 Anwendung in FUDGE

In diesem Kapitel werden zwei Szenen beschrieben und erklärt, die auf FUDGE basieren und die wichtigsten Funktionen der Arbeit darstellen. Diese Szenen werden ebenfalls abgegeben und sind auf dem Datenträger verfügbar. Die in diesen Experimenten genutzten Variablen wurden vorher angelegt und werden nur beschrieben.

Experiment 1

In diesem Beispiel werden die Basisfunktionen der Audio Komponente dargestellt. Dafür wird als erstes ein Würfel Objekt angelegt. Für dieses Beispiel wird `ComponentAudio` Objekte erstellt und an diesen Würfel angehängt. Die

`ComponentAudio` Objekt erhalten jeweils ein zusätzliches `Audio` Objekt, dass eine URL zu einem mitgelieferten Ton enthält.

```
audioSettings = new ƒ.AudioSettings();

componentAudioOne = new ƒ.ComponentAudio(audioOne);
audioOne = new ƒ.Audio(audioSettings, audioFileOne, 1, false);
filterOne = new ƒ.AudioFilter(audioSettings, "highpass",
     100, 100, 1);
delayOne = new ƒ.AudioDelay(audioSettings, 0);
componentAudioOne.setFilter(filterOne);
componentAudioOne.setDelay(delayOne);
body.addComponent(componentAudioOne);

componentAudioOne.playAudio(audioSettings);
```

Dieses Experiment zeigt die Kombination aus `AudioFilter` und `AudioDelay` in der FUDGE Umgebung. Die Parameter können nach belieben eingestellt werden, damit eine gewünschte Varianz entsteht.

Nachdem ein `AudioSettings` Objekt angelegt wurde,

Experiment 2

Das letzte Beispiel soll den wichtigsten Bereich des Audio Systems zeigen, welcher den dreidimensionalen Sound darstellt. Dazu werden wir als erstes eine `AudioListener` Komponente anlegen und diese der Kamera zuweisen. Anschließend wird wieder eine Audio Komponente erstellt. Der Komponente wird zusätzlich ein `AudioLocalisation` Attribut geliefert, so dass die Räumlichkeit dargestellt wird.

```
audioSettings = new ƒ.AudioSettings();

audio = new ƒ.Audio(audioSettings, audioSource, .5, true);
componentAudio = new ƒ.ComponentAudio(audio);
audioLocalisation = new ƒ.AudioLocalisation(audioSettings);
componentAudio.setLocalisation(audioLocalisation);
body.addComponent(componentAudio);

componentAudioListener = new ƒ.ComponentAudioListener(audioSettings);
camera.addComponent(componentAudioListener);

componentAudioListener.updatePositions(mtxCamera.translation);
componentAudio.getLocalisation().updatePositions(
  _body.cmpTransform.local.translation, mtxCamera.translation);
```

Der Beginn dieses Experimentes ist derselbe wie zuvor, jedoch wird nun ein `AudioLocalisation` Objekt erstellt und mit der `ComponentAudio` verbunden. Zusätzlich wird ein ComponentAudioListener in der Szene erstellt. Dieser wird in diesem Experiment mit der Kamera verknüpft. Jedoch kann dafür auch jedes andere beliebige Objekt genutzt werden. Der Unterschied in diesem experiment ist, die Aktualisierung der Positionen der Komponenten zur Laufzeit. Damit können Bewegungen der Audioquellen imitiert werden.

5. Fazit und Ausblick

FUDGE wurde im Rahmen dieser Arbeit um den Prototypen für die Audio Wiedergabe erweitert. Dafür wurden Strukturen und Methoden entwickelt um diesen Prototyp umsetzbar zu machen. Nachdem die Planung und Vorarbeiten erledigt waren, wurden Klassenstrukturen aufgebaut und definiert.

Dieser Prototyp kann, wie in den Experimenten bereits gezeigt, zu teilen in FUDGE genutzt werden, kann aber noch um weitere Aspekte erweitert werden. Man könnte Beispielsweise die Möglichkeit anbieten, verschiedene Kanäle einzubauen, die unabhängig voneinander agieren bis sie schließlich mit den anderen zusammenlaufen. Der Oszillator hat auch noch großes Potential erweitert zu werden, eventuell sogar mit einem eigenen Editor, um den Nutzern eine visuelle Möglichkeit zu geben Töne zu erstellen. Außerdem gibt es auch Möglichkeiten Audio in Form eines Visualizers oder ähnlichen Anwendungen visuell darzustellen. Dafür müssten die gewünschten Audio Nodes zusätzlich noch durch einen `AnalyserNode` geführt werden um die Daten auszulesen. Teilstrukturen für die Serialisierung der Komponenten wurden bereits angelegt, müssen aber noch erweitert und funktionsfähig gemacht werden.

Um eine bessere räumliche Wahrnehmung erstellen zu können, müssen die Transformation und Rotationen im späteren Verlauf noch an die Transformationen der Objekte selber gekoppelt werden. Eine eigene Berechnung aller Rotationen macht wenig Sinn, wenn diese bereits stattfinden.

Kleine Anwendungen innerhalb von FUDGE sind bereits möglich, aber in naher Zukunft wird FUDGE noch um einige zusätzliche Aspekte erweitert, wie ein Serversystem, die Physik und ein GUI in dem alle Technologien für den Nutzer greifbar gemacht werden sollen.

Bis zum kommenden Semester soll FUDGE soweit fertiggestellt werden, dass es in der Lehre angewendet werden kann. Auch ohne ein GUI ist FUDGE nutzbar, jedoch wird es dadurch notwendig, große Teile selbst zu coden. Dies erschwert den Einstieg, bietet aber auch die Chance, die Grundlagen des Lernenden zu stärken.

Literaturverzeichnis

[1] Verband der deutsch Games-Branche: Überblick Games-Branche in Deutschland [Aufgerufen am 15.09.2019] Verfügbar unter: https://www.game.de/games-branche-in-deutschland/ueberblick/

[2] GamesWirtschaft: Halbjahresbilanz 2019 [Aufgerufen am 15.09.2019] Verfügbar unter: https://www.gameswirtschaft.de/wirtschaft/games-umsatz-halbjahr-2019-deutschland/

[3] Newzoo: Top 10 Countries/Markets by Game Revenues [Aufgerufen am 15.09.2019] Verfügbar unter: https://newzoo.com/insights/rankings/top-10-countries-by-game-revenues/

[4] J. Dell'Oro-Friedl: FUDGE. Repository [Aufgerufen am 15.09.2019] Verfügbar unter: https://github.com/JirkaDellOro/FUDGE

[5] J. Dell'Oro-Friedl: FUDGE. Wiki [Aufgerufen am 15.09.2019] Verfügbar unter: https://github.com/JirkaDellOro/FUDGE/wiki

[6] Mozilla Foundation: MDN Web Docs. HTML [Aufgerufen am 15.09.2019] Verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/HTML

[7] ECMA International: ECMAScript® 2020 Language Specification (2019) [Aufgerufen am 15.09.2019] Verfügbar unter: https://tc39.es/ecma262/

[8] Microsoft: Typescript [Aufgerufen am 15.09.2019] Verfügbar unter: http://www.typescriptlang.org/

[9] Mozilla Foundation: MDN Web Docs. Promises [Aufgerufen am 15.09.2019] Verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[10] Ilya Kantor: The JavaScript language. Async /await (2007) [Aufgerufen am 15.09.2019] Verfügbar unter: https://javascript.info/async-await

[11] Mozilla Foundation: MDN Web Docs. Fetch API [Aufgerufen am 15.09.2019] Verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

[12] Mozilla Foundation: MDN Web Docs. XMLHTTPRequest [Aufgerufen am 15.09.2019] Verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

[13] Smus, B.: Web Audio API. Advanced sound for games and interactive apps, 1st edn. O'Reilly, Beijing (2013)

[14] Mozilla Foundation: MDN Web Docs. Web Audio API [Aufgerufen am 15.09.2019] Verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API

[15] P. Adenot et al.: Web Audio API. W3C Candidate Recommendation [Aufgerufen am 15.09.2019] Verfügbar unter: https://www.w3.org/TR/webaudio/

Eidesstattliche Erklärung

Ich, Thomas Dorner, versichere, dass ich die vorliegende Arbeit selbstständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

_____ Furtwangen, den 23.09.2019

Thomas Dorner

Anhang

Alle die mit dieser Abschlussarbeit zusammenhängende Bearbeitungsunterlagen befinden sich im Anhang, der auf dem beigefügten Datenträger zu finden ist. Darunter finden Sie folgendes:

1. Eine Version von FUDGE mit der die Arbeit geendet hat
2. Alle im Rahmen dieser Arbeit angefertigten Diagramme
3. Jede Datei, die durch mich bearbeitet wurde
4. Eine Kopie dieser Arbeit in PDF Format