ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Konzeption und Realisierung einer mobilen Anwendung zur Untersuchung der akustischen Lokalisationsfähigkeit

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Felix Rottler
felix.rottler@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Marc Schickler

2017

Fassung 4. Mai 2017

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Durch die stetig zunehmende Verbreitung von mobilen Geräten steigt auch der Bedarf an mobilen Anwendungen. Ein wichtiger Anwendungsbereich für mobile Anwendungen findet sich in der Medizin, insbesondere bei der Behandlung von Tinnitus Betroffenen, wieder. Allerdings ist die Entwicklung mobiler Anwendungen für unterschiedliche Plattformen sehr aufwändig, da sich deren Entwicklung deutlich voneinander unterscheidet. Aus diesem Grund bietet eine plattformunabhängige Entwicklung Vorteile in Bezug auf Komplexität und Kostenfaktor.

Ziel dieser Arbeit ist die Konzeption und Realisierung einer hybriden Anwendung auf Basis des Frameworks Meteor, mit welcher die akustische Lokalisationsfähigkeit untersucht und verbessert werden soll. Um dies zu erreichen, wird mit Hilfe von WebGL eine 3D Szene erstellt. In dieser Szene werden Geräuschquellen mit der Web Audio API platziert und der Anwender muss diese Geräuschquellen lokalisieren. Weiterhin werden während des gesamten Interaktionszyklus Informationen aufgezeichnet, die für die Untersuchung der akustische Lokalisationsfähigkeit von großer Bedeutung sind.

Danksagung

Zunächst gebührt ein besonderer Dank Herrn Prof. Dr. Manfred Reichert für die Begutachtung dieser Arbeit.

Weiterhin möchte ich mich bei allen meinen Kommilitonen, Freunden und meiner Familie bedanken, die mich während der Erstellung dieser Arbeit mental unterstützt und motiviert haben.

Weiterer Dank gilt meinem Betreuer Marc Schickler, der meine Ideen für diese Arbeit immer unterstützt hat.

Inhaltsverzeichnis

1

Einleitung

In der heutigen Zeit sind mobile Anwendungen fast nicht mehr wegzudenken, denn mit der zunehmenden Verbreitung von mobilen Geräten steigt auch deren Bedarf und Anwendungsgebiete. In nahezu jedem Bereich werden diese eingesetzt und sollen den Nutzer unterstützen. Dazu gehören mobile Anwendungen, die es dem Nutzer ermöglichen ihren Tagesablauf leichter planen zu können, indem sie diesen an vorher festgelegte Termine erinnern. Allerdings auch Anwendungen, die dem Nutzer einen Trainings- und Ernährungsplan anbieten, um diesen gesundheitlich zu unterstützen. Außerdem werden auch mobile Anwendungen im Bereich der Medizin, insbesondere der Psychotherapie, eingesetzt. Diese dienen jedoch vielmehr einem wissenschaftlichen Kontext. Eine weitere Möglichkeit mobile Anwendungen in der Medizin einzusetzen, ist die Behandlung von Hörgeschädigten, insbesondere von Tinnitus Betroffenen [1]. Bei der Behandlung ebendieser ist es den Betroffenen möglich, durch gezieltes Training, wie zum Beispiel das Fokussieren und Lokalisieren von Geräuschquellen, den Tinnitus in bestimmtem Maße zu unterdrücken [2]. Unter dem Fachbegriff Lokalisation versteht man das Erkennen von Richtung und Entfernung einer Geräuschquelle im dreidimensionalen Raum. Um die Lokalisationsfähigkeit mit Hilfe einer mobilen Anwendung zu verbessern wird in der Arbeit von Schickler et al. [2] ein *Serious Game* eingeführt, bei der die Patienten spielerisch mit der Anwendung interagieren können, um so ihre Lokalisationsfähigkeit zu verbessern. Das hat den Vorteil, dass dadurch die Motivation der Betroffenen aufrechterhalten werden kann [3]. Die Behandlung von Tinnitus erweist sich allerdings als problematisch, da bei einem subjektiven Tinnitus nur die Betroffenen selbst hören können. Deswegen hängt die Behandlung der Patienten davon ab, dass sie ihren Zustand über eine mobile Anwendung festhalten können. Eine solche mobile

Anwendung wird von Pryss et al. [4] vorgestellt. Die Notwendigkeiten mobiler Anwendungen spiegelt sich auch deutlich in der Anzahl der bisher verfügbaren Apps wieder. Der Google Play Store bietet zum jetzigen Zeitpunkt etwa 2,3 Millionen Apps und auch der Apple Store hat ungefähr 2 Millionen Apps im Angebot [5]. Die Entwicklung solcher mobilen Anwendungen ist daher zu einem wichtigen Bestandteil für Software Unternehmen geworden und sollte sich im besten Fall kosteneffizient, schnell und einfach handhabbar gestalten.

1.1 Problemstellung

Die meisten mobilen Anwendungen laufen auf den Betriebssystemen Android, iOS und Windows Phone. Die Entwicklung von mobilen Anwendungen für diese Plattformen ist allerdings sehr unterschiedlich, denn sie setzen jeweils andere Programmiersprachen voraus. Dementsprechend ist es für Entwickler sehr aufwändig, dieselbe Anwendung für drei unterschiedliche Plattformen zu entwickeln, was die Kosten für die Entwicklung in beträchtlichem Maße erhöht. Allerdings bietet die native Entwicklung auf diesen Plattformen eine sehr ausgereifte API, mit der auf Hardware Komponenten, wie zum Beispiel Kamera oder interner und externer Speicher, zugegriffen werden kann. Benötigt dagegen eine Web-Applikation Zugriff auf Hardware Komponenten, kann es sein, dass dies aufgrund fehlender Unterstützung oder Sicherheitsrisiken nicht möglich ist. Eine weitere Möglichkeit sind sogenannte hybride Anwendungen, die auf Web-Anwendungen aufbauen, also mit modernen Web-Technologien wie HTML5, CSS und Javascript programmiert sind und daher auch nicht so nah am Betriebssystem liegen, wie es bei native Anwendungen der Fall ist. Diese hybriden Anwendungen benutzen eine sogenannte *WebView*, die von dem jeweiligen Betriebssystem abhängig ist und werden durch ein Framework ergänzt, das Funktionen vorgibt, mit denen Zugriffe auf Hardware Komponenten möglich sind. Da die Entwicklung dieser hybriden Anwendungen allerdings auch auf HTML5, Javascript und CSS beruht, fällt deren Entwicklung deutlich leichter und kostengünstiger aus, als bei nativen Anwendungen. Daher ist es wichtig zu wissen, in welchen Fällen eine hybride Anwendung Sinn ergibt und ob die Funktionalität und Leistungsfähigkeit für das vorgegebene Ziel, die Entwicklung einer mobilen Anwendung

zur Untersuchung der akustischen Lokalisationsfähigkeit, ausreicht. Weiterhin erweist sich die Behandlung von Tinnitus Patienten als schwierig, da es sich bei Tinnitus um einen subjektiven Zustand handelt. Sowohl Festhalten dieses Zustands, als auch die Möglichkeit Geräuschquellen zu lokalisieren [2], sind wichtige Aspekte die mobile Anwendungen zur Behandlung von Tinnitus erfüllen müssen.

1.2 Zielsetzung

Ziel dieser Arbeit soll nun die Konzeption und Realisierung einer hybriden Anwendung auf Basis von HTML5, Javascript und CSS sein, um die akustische Lokalisationsfähigkeit von Geräuschquellen im Raum zu untersuchen. Hierfür soll eine 3D Szene konzipiert werden, in der der Benutzer die Kamera steuern kann, um sich durch die Szene zu navigieren. Weiterhin sollen in dieser 3D Szene Geräuschquellen platziert werden, die durch den Benutzer lokalisiert werden müssen. Je nach Blickrichtung werden die Töne unterschiedlich wahrgenommen und anhand dieser Varianz muss der Benutzer die Richtung der Geräuschquellen erfassen. Dabei soll untersucht werden, ob die Web Audio API und WebGL unter Verwendung des hybriden Frameworks Meteor genug Funktionalität und Leistungsfähigkeit bietet, um alle in Kapitel 3 vorgestellten Anforderungen zu erfüllen. Besondere Bedeutung kommt dabei der Web Audio API zu, denn diese wird bisher von nicht allen Browsern unterstützt, was zu Problemen bezüglich unterschiedlicher *WebViews* führen kann.

1.3 Struktur der Arbeit

Im nachfolgenden Kapitel 2 werden zunächst die Grundlagen beschrieben, die benötigt werden, um eine hybride Anwendung auf der Basis von Meteor, WebGL und der Web Audio API zu entwickeln. Des Weiteren werden in Kapitel 3 funktionale und nichtfunktionale Anforderungen vorgestellt, die die Applikation beinhalten soll. Anschließend werden in Kapitel 4 die einzelnen Komponenten für die mobile Anwendung konzipiert und geeignet zusammengesetzt. Anknüpfend daran wird in Kapitel 5 die Implementierung und in

Kapitel 6 der Anforderungsabgleich der Anwendung vorgestellt. Abschließend wird in Kapitel 7 ein Fazit gegeben.

2

Grundlagen

Im folgenden Kapitel werden die Grundlagen beschrieben, die für die Umsetzung des Systems nötig sind. Dazu wird in Kapitel 2.1 zuerst das Framework Meteor genauer beschrieben, dass die Grundlage für die Applikation bildet und für das Kompilieren einer Web-Anwendung in eine *cross-platform*-Anwendung zuständig ist. Weiterhin widmet sich Kapitel 2.2 den mathematischen Grundlagen, die für die Darstellung von 3D Objekten notwendig sind. Danach wird in Kapitel 2.3 eine Einleitung zu WebGL gegeben. Abschließend wird die Web Audio API in Kapitel 2.4 genauer erklärt.

2.1 Meteor

Meteor ist eine Full-Stack Javascript Plattform, die auf **Node.js** basiert und als Back-End Datenbank **MongoDB** benutzt. Im Gegensatz zur klassischen (LAMP - Linux, Apache, MySQL, PHP) Web Architektur verwendet Meteor einen Ansatz, um Inhalte dynamisch zu generieren. Das bedeutet, dass die klassische Client-Server Kommunikation nicht mehr vorhanden ist. Zum Beispiel wird HTML nicht mehr über das Netzwerk gesendet, sondern komplett zum Client gesendet, welcher dann entscheidet, wie, wann und wo HTML angezeigt wird [6]. Dazu verwendet Meteor das HTML-Rendering-System **Blaze**, was für die Erstellung von reaktiven HTML-Templates zuständig ist.

2.1.1 Prinzipien

Um die Entwicklung mit Meteor zu vereinfachen, hat das Meteor Team folgende Prinzipien entworfen [6].

1. *one language*
   Meteor erlaubt die Entwicklung in einer einzigen Programmiersprache, nämlich *Javascript*, in allen Entwicklungsumgebungen.
2. *data on the wire*
   Es werden nur noch Daten vom Server verschickt und kein HTML. Das bedeutet, der Client entscheidet, was angezeigt wird.
3. *embraces the ecosystem*
   Meteor profitiert von einer aktiven *Javascript Community*, das bedeutet, andere Frameworks, wie zum Beispiel *Angular, React* sind leicht zu integrieren.
4. *full stack reactivity*
   Alle Meteor Komponenten repräsentieren den tatsächlichen Status und sind daher reaktiv.

2.1.2 Node.js

Node.js ist eine serverseitige Plattform zur Entwicklung von Netzwerkanwendungen. Im Speziellen werden damit Webserver realisiert. *Node.js* basiert auf der Laufzeitumgebung "V8", die von Google entwickelt worden ist. V8 und *Node.js* sind größtenteils in *C* und *C++* implementiert mit einem Fokus auf Performanz, Leistungsfähigkeit und der Reduzierung von Speicherverbrauch. Im Gegensatz zu den meisten anderen modernen Plattformen, ist ein *Node* Prozess nicht auf *Multithreading* angewiesen, um eine parallele Ausführung zu gewährleisten. Stattdessen benutzt *Node.js* eine asynchrone Bearbeitung von nicht blockierenden I/O Zugriffen, was eine hohe Geschwindigkeit zur Folge hat.

2.1.3 MongoDB

MongoDB bietet für Web-Anwendungen, in denen eine große Anzahl an Daten verarbeitet werden müssen, eine performante und dokumentenorientierte Lösung zur Speicherung von Daten. Diese Daten werden in MongoDB als Dokumente abgespeichert. Das bedeutet, dass MongoDB nicht, wie klassisch, auf einer relationalen Datenbank, sondern

auf *Key-Value*-Paaren beruht, was die dahinterliegende Datenstruktur dynamisch macht. Die Werte können einfache Datentypen oder aber auch andere Dokumente sein. Die Dokumente können beliebig ergänzt werden, auch wenn die Ergänzung nicht zum vorherigen Eintrag passt [7]. Um diese Dynamik zu gewährleisten, verwendet MongoDB das Format *BSON*, binäres *JSON* [8]. Ein Beispiel für das *JSON*-Format wird in Quelltext 2.1 gezeigt.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
```

Listing 2.1: Beispiel für JSON-Format

2.1.4 Mobil

Meteor integriert **Cordova**, ein Apache *open source* Framework, um mobile Anwendungen aus demselben Code zu generieren, der auch für reguläre Web-Anwendungen benutzt wird. Die Plattform spielt dabei keine Rolle, denn die **Cordova** Integration ermöglicht das Kompilieren auf iOS und Android [9].

Cordova

Apache Cordova ist ein *open source* Framework, mit dem durch die Verwendung von Web-Technologien wie HTML5, Javascript und CSS3, mobile Anwendungen entwickelt werden können. Diese Anwendungen werden innerhalb eines *WebViews* des jeweiligen Betriebssystems ausgeführt und haben Zugriff auf die vom Betriebssystem bereitgestellte API, für beispielsweise Sensoren, Speicher, Netzwerk und Batterie Status. Abbildung 2.1 soll den Aufbau einer Cordova Applikation verdeutlichen.

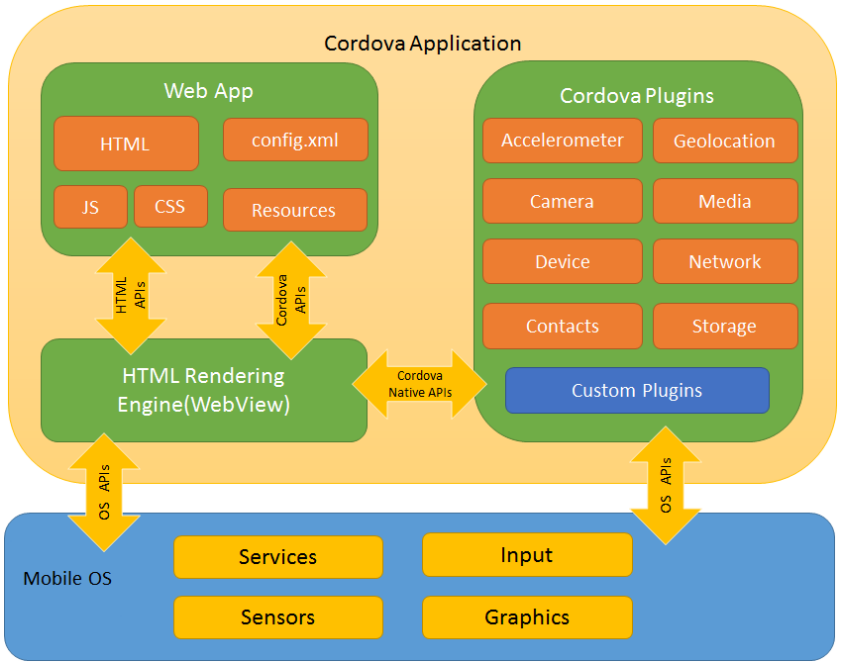


Abbildung 2.1: Aufbau einer Cordova Applikation [10]

Eine Cordova Applikation besteht aus einer *Web App*, der *WebView* und *Cordova Plugins*, die Schnittstellen für die *WebView* bereitstellen. Der Grund, warum sowohl die *HTML Rendering Engine* als auch die *Cordova Plugins* mit dem jeweiligen Betriebssystem kommunizieren können, liegt in der unterschiedlichen Leistungsfähigkeit der einzelnen APIs. Gerade für Hardware Zugriffe existieren noch nicht genug Schnittstellen für die *WebView*, dieses Problem löst Cordova über Plugins, die in der nativen API des jeweiligen Betriebssystems geschrieben sind und daher deutlich mächtiger sind. Die in der Spezifikation von Mozilla [11] vorgestellte Kamera API wird von keinem aktuellen Browser unterstützt, während in Meteor und Cordova das Benutzen der Kamera problemlos funktioniert. Mit Meteor ist es allerdings nicht notwendig Cordova selbst zu installieren, da bei dem *run and build* von Meteor auch die Erstellung einer Cordova Applikation mit inbegriffen ist.

2.1.5 Blaze

Blaze ist ein mächtiges Framework für die Erstellung von dynamischen *user-interface*-Elementen. Dabei setzt Blaze nicht auf die manuelle Aktualisierung von HTML-Elementen, wie es zum Beispiel bei *JQuery* der Fall ist, sondern automatisiert diesen Prozess. Das bedeutet, dass Blaze innerhalb der mobilen Anwendung auf Änderungen in der *DOM* reagiert, diese automatisch aktualisiert und die geänderten Elemente wieder in die *DOM* einträgt. In Quelltext 2.2 wird die Funktionsweise von Blaze und wie es in dieser Arbeit genutzt werden könnte, erklärt.

```
1 <template name="scenegraph">
2   <ol class="scene">
3     {{#each scenes}}
4       <li>{{name}}</li>
5       <li>{{position}}</li>
6     {{/each}}
7   </ol>
8 </template>
```

Listing 2.2: Ein Blaze-Template [12]

Ein Template wird durch einen einmaligen Namen identifiziert. Innerhalb dieses Templates stehen unterschiedliche Befehle zur Verfügung, um beispielsweise auf Datenbankeinträge zuzugreifen. Als Beispiel wird in Quelltext 2.2 über ein Objekt *scenes* iteriert, das alle in einer Datenbank gespeicherten Szenen zurückliefert. Innerhalb dieser Schleife kann auf die jeweiligen Attribute der einzelnen Szenen zugegriffen und in HTML dargestellt werden.

2.2 Mathematische Grundlagen

Die im Verlauf dieser Arbeit zum Verständnis notwendigen Konzepte der Computergrafik werden in diesem Kapitel erklärt. In Kapitel 2.2.1 werden zunächst der Vektor und Vektoroperationen erklärt. Weiterhin werden in Kapitel 2.2.2 die grundlegenden Transformationsmatrizen aufgezeigt, die für die Darstellung von 3D Objekten benötigt werden.

2.2.1 Vektoren

Fundamental für das Erstellen von 3D Objekten mit WebGL ist der sichere Umgang mit der dahinterliegenden Mathematik. Der Vektor spielt dabei eine zentrale Rolle, denn er kann dazu benutzt werden, um die Position eines Punktes in einem zwei- oder dreidimensionalen Raum zu beschreiben. Das bedeutet der Vektor definiert die Eckpunkte eines Objektes im dreidimensionalen Raum. Die folgende Gleichung zeigt den Aufbau eines Vektors x im n-dimensionalen Raum. Bei Vektoren wird zwischen einem Ortsvektor und einem Richtungsvektor unterschieden. Ein Ortsvektor beginnt im Ursprung des Koordinatensystems. Ein Richtungsvektor dagegen geht nicht vom Ursprung eines Koordinatensystems aus, sondern verbindet zwei Ortsvektoren. Gleichung 2.1 zeigt den Aufbau eines Vektors im n-dimensionalen Raum.

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \forall x_i \in \mathbb{R}, i = 1, \ldots, n \tag{2.1}$$

Addition und Subtraktion von Vektoren

Bei der Addition von zwei Vektoren x und y werden die einzelnen $x_i, i = 1, \ldots, n$ und $y_i, i = 1, \ldots, n$ miteinander addiert, wie es in Gleichung 2.2 gezeigt wird.

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{pmatrix} \tag{2.2}$$

Bei der Subtraktion von zwei Vektoren x und y werden die einzelnen $x_i, i = 1, \ldots, n$ und $y_i, i = 1, \ldots, n$ voneinander subtrahiert, siehe Gleichung 2.3.

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} - \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 - y_1 \\ x_2 - y_2 \\ \vdots \\ x_n - y_n \end{pmatrix} \tag{2.3}$$

Skalare Multiplikation

Bei der skalaren Multiplikation ist das erste Argument ein Skalar $a \in \mathbb{R}$, also eine reelle Zahl. In Gleichung 2.4 wird gezeigt, wie ein Vektor v mit einem Skalar a multipliziert wird.

Dabei wird jede Komponente des Vektors mit diesem Skalar multipliziert.

$$a * v = a * \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} a * x_1 \\ a * x_2 \\ \vdots \\ a * x_n \end{pmatrix} \tag{2.4}$$

2.2.2 Matrizen

Eine $m \times n-$Matrix ist ein mathematisches Konstrukt, das aus $m * n$ reellen Zahlen besteht.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \tag{2.5}$$

Matrizen werden in der Computergrafik genutzt, um Transformationen eines Punktes, der durch x, y, z-Koordinaten repräsentiert wird, zu speichern. Transformationen sind im Allgemeinen sehr wichtig, denn sie beschreiben eine Projektion, die notwendig ist, um 3D Objekte auf einen 2D Computerbildschirm darzustellen. Allgemein existieren drei einfache Transformationen, die im folgenden genauer erläutert werden.

Translation

Eine Translationsmatrix verschiebt die Punkte entlang einer oder mehrerer Achsen. Abbildung 2.2 stellt die Translation grafisch dar.

Als einfaches Beispiel sei ein Punkt $p = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 1 \end{pmatrix}$ gegeben. Um den Punkt p nun 5 Einheiten nach oben, also der $y-$Achse zu verschieben, wird eine Matrix gesucht, die folgende

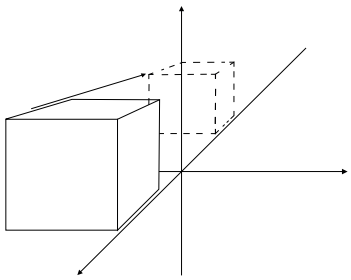


Abbildung 2.2: Translation der Eckpunkte eines Würfels

Gleichung erfüllt.

$$M \times p = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 7 \\ 2 \\ 1 \end{pmatrix} \tag{2.6}$$

Daraus ergibt sich eine, in Gleichung 2.7 dargestellte, Translationsmatrix.

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.7}$$

Das bedeutet, dass die letzte Spalte der Matrix die Translation speichert, nämlich

$$t = \begin{pmatrix} 0 \\ 5 \\ 0 \\ 1 \end{pmatrix}$$

Die allgemeine Form der Translationsmatrix wird in Gleichung 2.8 gezeigt.

$$M = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.8}$$

Rotation

Um ein Objekt, beziehungsweise Eckpunkte, die ein Objekt definieren, um eine oder mehrere Achsen zu rotieren, wird eine Rotationsmatrix benötigt. Abbildung 2.3 stellt die Rotation grafisch dar. Bei dieser Matrix ist zu beachten, dass für unterschiedliche Rotati-

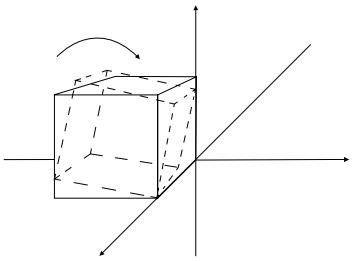


Abbildung 2.3: Rotation der Eckpunkte eines Würfels

onsachsen auch unterschiedliche Rotationsmatrizen benötigt werden. Das Grundgerüst der Rotationsmatrix ist die in Gleichung 2.9 gezeigte Matrix.

$$R = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \tag{2.9}$$

Je nachdem, um welche Achse rotiert wird, müssen die Werte der Matrix 2.9 dort in den Zeilen und Spalten eingetragen werden, die sich nicht mit der Rotationsachse überschneiden. Für die $x-$Achse ergibt sich folgende Matrix, wie Gleichung 2.10 zeigt.

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.10}$$

Dabei repräsentiert α den Winkel, mit dem um die $x-$Achse gedreht wird. Analog dazu lassen sich die Rotationsmatrizen für die $y-$ und $z-$Achse ableiten.

$$R_y = \begin{pmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.11}$$

$$R_z = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.12}$$

Skalierung

Die Skalierungsmatrix hat zur Folge, dass alle Punkte, auf die sie angewendet wird, auf einer oder mehreren Achsen skaliert werden können. Abbildung 2.4 stellt die Skalierung grafisch dar. Das bedeutet im Allgemeinen, dass dadurch Objekte größer oder kleiner

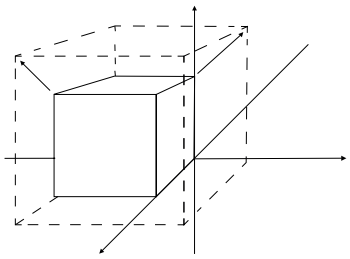


Abbildung 2.4: Skalierung der Eckpunkte eines Würfels

gemacht werden können, allerdings lassen sich damit auch Spiegelungen realisieren. Die Gleichung 2.13 veranschaulicht den Aufbau einer Skalierungsmatrix.

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.13}$$

Lookat Matrix

Für die Erstellung eines dreidimensionalen Raumes mit WebGL ist die Verwendung einer *Lookat*-Matrix, oder auch Betrachtungsmatrix, unumgänglich. Sie ermöglicht das Platzieren einer virtuellen Kamera im Raum, die auf einen vorher spezifizierten Punkt gerichtet wird. Um die richtige Orientierung dieser Kamera einzustellen, muss ein sogenannter Up-Vektor angegeben werden. Dieser ist in den meisten Fällen $\begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^T$. Das bedeutet, dass aus einem Ortsvektor, einem Richtungsvektor und dem Up-Vektor eine Matrix erstellt werden kann, die Räumlichkeit dadurch erzeugt, dass der Nutzer das Gefühl hat, durch eine Kamera zu blicken. Die in dieser Arbeit verwendete *Lookat*-

Matrix wird durch die folgende Funktion 2.3 definiert [13]. Im Verlauf dieser Arbeit wird die *LookAt*-Matrix als *View*-Matrix bezeichnet. Dabei ist *out* die Variable, in der das Resultat gespeichert wird, *eye* der Ursprungspunkt, *center* der Richtungspunkt und *up* der Up-Vektor, vgl. Shirley, P. (2015), S. 146 [14]

```
1 mat4.lookAt = function (out, eye, center, up)
```

Listing 2.3: lookAt-Funktion in der *glMatrix*-Bibliothek [15]

Perspektivische Projektion

Eine weitere Matrix, die genutzt wird um Räumlichkeit zu simulieren, ist die perspektivische Projektionsmatrix. Abbildung 2.5 stellt die perspektivische Projektion grafisch dar. Auf ein Objekt angewendet, erscheint dieses kleiner, falls es weiter weg ist oder

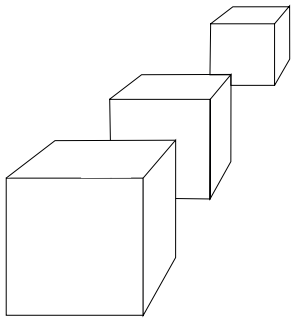


Abbildung 2.5: Perspektivische Projektion

größer, falls das Objekt näher an der Kamera 2.2.2 platziert ist. Die dazu benötigten Parameter sind die *near*- und *far*-Ebene, sowie die Eckpunkte *left, right, top, bottom* des

Sichtvolumens. Mit diesen Werten ergibt sich folgende Matrix.

$$S = \begin{pmatrix} \frac{2*near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2*near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2*near*far}{near-far} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{2.14}$$

Diese Matrix wird in *glMatrix*-Bibliothek [13] mit der Funktion 2.4 erzeugt. Dabei repräsentiert *out* die Variable, in der das Resultat abgespeichert wird. Die restlichen Parameter dieser Funktion sind zu einem *fovy*, dem *Field of View* und *aspect*, welches das Seitenverhältnis angibt und zum anderen die *near*- und *far*-Ebene. Eine genaue Herleitung wird von Shirley, P (2015), S. 151-156 [14] gegeben.

```
1 mat4.perspective = function (out, fovy, aspect, near, far)
```

Listing 2.4: *perspective*-Funktion in der *glMatrix*-Bibliothek [15]

2.3 WebGL

WebGL [16] ist eine *cross-platform* API, die es ermöglicht, 3D Objekte im HTML *canvas*-Element zu erstellen. Die API ist sehr stark an die OpenGL ES 2.0 API angelehnt, bietet aber weniger Funktionalität, da auch hier wieder das Problem besteht, dass die HTML Rendering Engine nicht so leistungsstark ist. WebGL wird zum jetzigen Zeitpunkt von fast allen Browsern unterstützt. Im Folgenden werden die wichtigsten Konzepte der Computergrafik mit Hilfe von WebGL erläutert.

2.3.1 Einführung

Um WebGL im Browser nutzen zu können, wird das *canvas*-Element benötigt. Dieses zeichnet sich durch eine Breite und Höhe aus und wird je nach Orientierung des mobilen Gerätes angepasst. Der erste Schritt um 3D Objekte mit WebGL zu erstellen, ist das Erzeugen eines WebGL-Kontextes, der als *namespace* fungiert und Zugriff auf alle

WebGL-Funktionen erlaubt. In Quelltext 2.5 wird gezeigt, wie man einen solchen WebGl-Kontext erzeugt. Wenn dieser Befehl fehlschlägt, dann unterstützt der Browser kein WebGL oder es ist kein gültiges *canvas*-Element vorhanden.

```
1    var gl = canvas.getContext("webgl");
```

Listing 2.5: Erstellen eines WebGL-Kontextes

WebGL Shader

WebGL setzt, wie OpenGL, auf sogenannte **Shader**. Das sind kleine Programme, die zur Laufzeit auf der CPU kompiliert und anschließend auf der Grafikkarte ausgeführt werden. Diese Shader sind durch die **OpenGL ES Shading Language** spezifiziert, einer Sprache, die *C* sehr ähnlich ist. In WebGL existieren nur zwei verschiedene Arten von Shadern, Vertex- und Fragmentshader. Dem Vertexshader können Attribute, wie Positionskoordinaten und Texturkoordinaten übergeben werden. Mit Positionskoordinaten sind die in Kapitel 2.2.1 vorgestellten Vektoren gemeint. Im Fragmentshader werden die Vorläufer der Pixel generiert, sogenannte Fragmente. Die Aufgabe des Fragmentshaders ist es, jedem Fragment eine Farbe zu verleihen. Diese kann entweder statisch festgelegt werden oder aber durch den Entwickler bestimmt werden. Im Folgenden werden in Quelltext 2.6 und 2.7 jeweils ein Beispiel für einen Vertexshader und einen Fragmentshader gegeben. Eine präzise Erklärung wie Shader kompiliert und genutzt werden können, wird in [17] gegeben.

Der in Quelltext 2.6 gezeigte Vertexshader besitzt zwei Attribute, *aPosition* und *inColor*. Beide Attribute sind vom Typ *vec3*. In der *main*-Methode wird dem Wert *gl_Position* der Wert von *aPosition* zugewiesen. Die Variable *gl_Position* ist ein vordefinierter Output im Vertexshader und weist dem aktuellen Vertex die Position im *clip-space* zu. Die *varying*-Variable *outColor* bekommt den Wert von Attribut *inColor* zugewiesen. Das Schlüsselwort *varying* bedeutet, dass diese Variable sowohl im Fragment, als auch im Vertexshader verfügbar ist. Für den späteren Verlauf bedeutet das, dass *outColor* einen Wert zugewiesen bekommt und der Fragmentshader später diesen Wert verwenden kann.

```
attribute vec3 aPosition;
attribute vec3 inColor;
varying vec4 outColor;

void main(){
    gl_Positon = vec4(aPosition,1.0);
    outColor = vec4(inColor,1.0);
}
```

Listing 2.6: Beispiel für einen Vertexshader

```
varying vec4 outColor;

void main(){
   gl_FragColor = outColor;
}
```

Listing 2.7: Beispiel für einen Fragmentshader

Buffer

Das wichtigste Objekt in der WebGL API ist der Buffer. Er dient als Speicher für Daten, wie zum Beispiel Eckpunkte, gleichzustellen mit den in Kapitel 2.2.1 vorgestellten Vektoren, oder Werte für Farben, Texturkoordinaten, Indizes und Normalen. Wird der Buffer befüllt, können die in ihm gespeicherten Daten an Shader 2.3.1 weitergegeben werden, die diese Daten dann weiterverarbeiten. Im Folgenden soll erklärt werden, wie die Darstellung eines Dreiecks in WebGL funktioniert. Der erste Schritt dazu ist das Anlegen der Eckpunkte des Dreiecks. Jeder Eckpunkte besteht aus einer x-, y- und z-Koordinate und befindet sich zwischen den Werten $[-1; 1]$.

```
var data = new Float32Array([0, 0, 0, 1, 0, 0, 0, 1, 0]);
```

Listing 2.8: Anlegen der Eckpunkte des Dreiecks

Anschließend wird ein Buffer erzeugt, gebunden und mit den in Quelltext 2.8 erzeugten Eckpunkten befüllt. Ist der Buffer mit den Eckpunkten befüllt, so kann er danach wieder

entbunden werden, um zum Beispiel noch andere Buffer zu binden, um diese mit anderen Daten zu befüllen.

```
1 var buffer = gl.createBuffer();
2 gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
3 gl.bufferData(gl.ARRAY_BUFFER, data, gl.STATIC_DRAW);
4 gl.enableVertexAttribArray(0);
5 gl.bindBuffer(gl.ARRAY_BUFFER, null);
```

Listing 2.9: Der Buffer wird erzeugt, gebunden und anschließend mit Daten befüllt

Um den Buffer nun zu zeichnen muss zuerst ein WebGLProgram aktiviert werden. Dieses WebGLProgram ist ein vorher aus WebGLShader erstelltes Programm. Nach der Aktivierung des WebGLPrograms muss der in Quelltext 2.9 erzeugte Buffer wieder gebunden werden. Anschließend wird das Format der Eckpunkte spezifiziert und der *draw*-Aufruf gestartet.

```
1 gl.useProgram(program);
2 gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
3 var p = vertexPosition;
4 gl.vertexAttribPointer(p, 3, gl.FLOAT, false, 0, 0);
5 gl.drawArrays(gl.TRIANGLES, 0, 3);
6 gl.useProgram(null);
```

Listing 2.10: Aktivierung eines *WebGLPrograms* mit anschließendem *draw*-Aufruf

Anschließend veranschaulicht Abbildung 2.6 das Resultat.

Abbildung 2.6: Rendern eines Dreiecks mit WebGL

2.4 Web Audio API

Die Web Audio API ist eine mächtige API, die dazu genutzt wird, um Audio im Web nutzen zu können. Dazu gehört das Einbinden von Audiodateien, aber auch die Verwendung von räumlichen Audioeffekten, wie sie in dieser Arbeit genutzt werden. Der typische Ablauf, wie diese API genutzt wird, ist dabei folgender [18].

1. Ein Audio Kontext muss erzeugt werden.
2. Mit diesem Kontext werden Audiodateien bereitgestellt
3. Für diese Audiodateien können Effekte hinzufgefügt werden, zum Beispiel Räumlichkeit.
4. Die Ausgabe des Tons muss spezifiziert werden, in den meisten Fällen die Lautsprecher.

AudioContext

Der AudioContext ermöglicht das Verwenden aller Funktionen innerhalb dieses Kontextes. Er besteht aus mehreren miteinander verbundenen Audiomodulen, wobei es

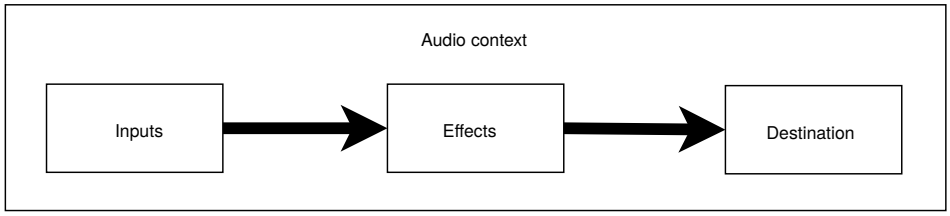


Abbildung 2.7: Funktionsweise innerhalb des AudioContextes [19]

sich bei jedem dieser Module um eine AudioNode handelt. Quelltext 2.11 zeigt, wie ein solcher AudioContext erstellt wird.

```
1    var audioCtx = new AudioContext();
```

Listing 2.11: Erstellen eines AudioContext

AudioBuffer

Der AudioBuffer ist eine AudioNode, in welchem Audiodateien abgespeichert werden können. Dabei ist der AudioBuffer so gestaltet, dass er nur kleine Audiodateien speichern kann, typischerweise nicht länger als 45 Sekunden. Um die Audiodatei abzuspielen, wird der AudioBuffer einer AudioBufferSourceNode hinzugefügt. In Quelltext 2.12 und 2.13 wird gezeigt, mit welchen Funktionen ein AudioBuffer initialisiert werden kann.

```
1    var buffer = audioCtx.createBuffer(options);
```

Listing 2.12: Erstellen eines AudioBuffers, mit Hilfe des AudioContext aus Kapitel 2.4

```
1    var buffer = audioCtx.decodeAudioData(audioData).then(function(
         decodedData){
2      // use decoded data here
3    });
```

Listing 2.13: Erstellen eines AudioBuffers mit Hilfe der Funktion *decodeAudioData* [20]

AudioBufferSourceNode

Eine AudioBufferSourceNode repräsentiert eine Audioquelle einer im Speicher liegenden Audiodatei, wie es der in Kapitel 2.4 dargestellte AudioBuffer ist. Eine AudioBufferSourceNode hat genau einen Ausgang und keinen Eingang, wie es in Abbildung 2.8 gezeigt wird. Um einen AudioBuffer der AudioBufferSourceNode hinzuzufügen, wird der Wert

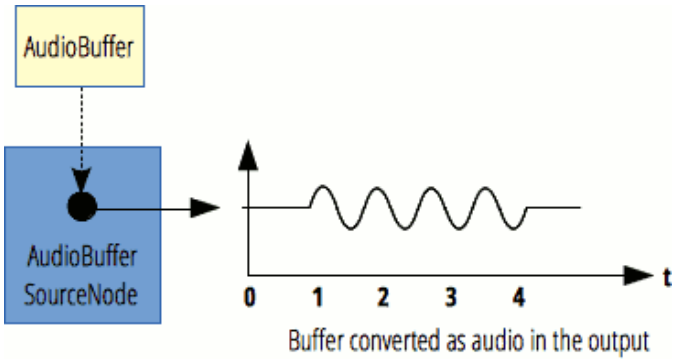


Abbildung 2.8: Ein *AudioBufferSourceNode*-Element [21]

des Attributes *buffer* der AudioBufferSourceNode mit dem AudioBuffer überschrieben, wie es in Quelltext 2.14 gezeigt wird.

```
1  var audioBufferSourceNode = audioCtx.createBufferSource();
2  audioBufferSourceNode.buffer = buffer;
3  audioBufferSourceNode.start();
```

Listing 2.14: Erstellen einer AudioBufferSourceNode und Hinzufügen eines AudioBuffers

Eine weitere Besonderheit der AudioBufferSourceNode ist, dass diese nur einmal abgespielt werden kann. Soll der Ton erneut abgespielt werden, muss dafür eine neue AudioBufferSourceNode erstellt werden.

AudioListener

Der AudioListener simuliert eine Person im Raum, die das Zentrum der zu empfangenden Tonsignale repräsentiert. Er wird durch einen Positions- und Orientierungsvektor dargestellt und für die Umsetzung von räumlichen Klang genutzt. Das bedeutet, dass

alle Geräuschquellen im aktuellen AudioContext auf einen festgelegten AudioListener abgestimmt sind. Der AudioListener kann nur einmal pro AudioContext festgelegt werden. Abbildung 2.9 zeigt den Aufbau des AudioListeners mit Hilfe von Vektoren.

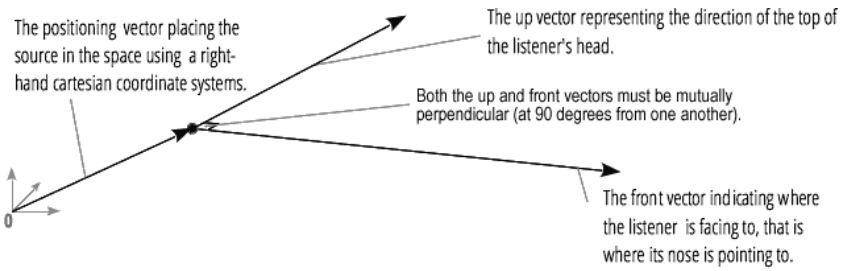


Abbildung 2.9: Darstellung des AudioListeners mit Hilfe von Vektoren [22]

PannerNode

Eine PannerNode repräsentiert die Position einer Audioquelle im Raum. Definiert wird eine PannerNode durch einen Positionsvektor, einen Orientierungsvektor und einen Kegel, der angibt in welchem Bereich der Ton gehört werden kann. Abbildung 2.10 stellt den Aufbau der PannerNode mit Hilfe von Vektoren grafisch dar. In folgendem

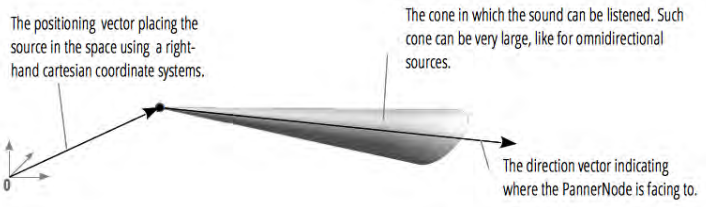


Abbildung 2.10: Darstellung einer PannerNode mit Hilfe von Vektoren [23]

Quelltext 2.15 wird gezeigt, wie eine räumliche Audioquelle erstellt und ihr eine Position zugewiesen wird.

```
1 var panner = audioCtx.createPanner();
2 panner.setPosition(0,0,0);
```

Listing 2.15: Erstellen einer PannerNode

3

Anforderungsanalyse

In diesem Kapitel werden die Anforderung für die Anwendungen spezifiziert, bewertet und anschließend erläutert. Dazu gehören funktionale und nichtfunktionale Anforderungen. Diese Anforderungen werden nach ihrer Relevanz bewertet und dienen als tragfähige Grundlage der anschließenden Systementwicklung.

3.1 Funktionale Anforderungen

Die in diesem Kapitel vorgestellten Anforderungen beschreiben die gewünschten Funktionen der Anwendungen, die Interaktion zwischen der Anwendung und dem Benutzer, als auch die Kommunikation zwischen Server und Client. In der folgenden Tabelle 3.1 sind die einzelnen funktionalen Anforderungen aufgelistet und nach ihrer Priorität bezüglich der Wichtigkeit für das System bewertet.

FA01 Dokumentdatenmodell für eine Szene

Eine Szene ist eine komplexe Struktur, die aus vielen Objekten besteht. Daher ist es wichtig, nur die notwendigen Informationen aus einer Szene zu extrahieren, um sie anschließend wieder zu rekonstruieren. Dafür wird ein JSON benötigt, das anschließend als Dokument in der Datenbank gespeichert wird.

Nr.	Anforderung	Priorität
FA01	Dokumentdatenmodell für eine Szene	unverzichtbar
FA02	Laden einer Szene aus der Datenbank	unverzichtbar
FA03	Speichern einer Szene	sehr wichtig
FA04	Textur der Szene ändern	nicht wichtig
FA05	Neue Sounds der Szene hinzufügen	wichtig
FA06	Vorhandene Sounds bearbeiten	nicht wichtig
FA07	Unterscheidung zwischen Bearbeitung und Benutzung	nicht wichtig
FA08	Kamerasteuerung	unverzichtbar
FA09	Szenengraph	wichtig
FA010	Visuelles Feedback	wichtig
FA011	Informationen über Nutzungsverhalten	unverzichtbar

Tabelle 3.1: Funktionale Anforderungen

FA02 Laden einer Szene aus der Datenbank

Bei Start der Anwendung soll dem Benutzer die Möglichkeit gegeben werden, Szenen aus der Datenbank zu laden. Dazu werden alle Szenen in einem Auswahlmenü mit ihrem Namen angezeigt.

FA03 Speichern einer Szene

Der Nutzer soll eine vorher aktivierte Szene abspeichern können. Dazu soll das in Kapitel 3.1 entwickelte Dokumentdatenmodell verwendet werden.

FA04 Textur der Szene ändern

Um die Anwendung dynamischer zu gestalten, soll es für den Bearbeiter der Szene die Möglichkeit geben, die Textur dieser zu verändern.

FA05 Neue Sounds der Szene hinzufügen

Der Benutzer soll aus einer Liste von vorgegebenen Audiodateien einen Sound auswählen und diesen in der Szene über ein Eigenschaftenmenü platzieren können.

FA06 Vorhandene Sounds bearbeiten

Falls der Benutzer seinen hinzugefügten Sound ändern möchte, so soll dieser über einen Szenengraphen die Möglichkeit haben, die Position und die Audioquelle des Sounds zu verändern oder zu löschen.

FA07 Unterscheidung zwischen Bearbeitung und Benutzung

Um Bearbeitung und Benutzung der mobilen Anwendung zu trennen, hat der Benutzer die Möglichkeit zwischen diesen beiden Optionen zu wechseln. Dabei heißt Bearbeitung, dass über einen Szenengraphen Einstellungen an der Szene vorgenommen werden können und Töne visualisiert werden, während bei der Benutzung alle Töne nur hörbar sind und keine Einstellungen an der Szene vorgenommen werden können.

FA08 Kamerasteuerung

Ein wichtiger Aspekt der zu entwickelnden Anwendung ist die Kamerasteuerung. Der Nutzer soll sowohl über den Browser mit der Maus sich im Raum bewegen können, als auch auf mobilen Geräten mit Touch-Input.

FA09 Szenengraph

Die Szene soll im Bearbeitungsmodus dem Nutzer zeigen, welche Objekte vorhanden sind und an welcher Stelle sie sich befinden. Des Weiteren soll der Anwender in diesem Auswahlmenü die Möglichkeit haben, selbst Änderungen vorzunehmen.

FA010 Visuelles Feedback

Wenn der Nutzer sich nicht im Bearbeitungsmodus befindet, wird der Cursor durch eine Raute ersetzt. Ist der Nutzer der Meinung, dass an einer bestimmten Stelle der Ursprung eines Sounds ist, wird nach einem Klick die Farbe innerhalb der Raute auf Grün gesetzt, falls die Vermutung des Nutzers richtig ist, und auf Rot, falls die Vermutung des Nutzers falsch ist.

FA011 Informationen über Nutzungsverhalten

Relevante Informationen, die das Nutzerverhalten beschreiben sollen abgespeichert werden. Dazu gehören die Zeit, die der Nutzer aufwenden muss, um eine Audioquelle zu lokalisieren, und die Anzahl der Fehler, falls der Nutzer die Audioquelle falsch lokalisiert.

3.2 Nichtfunktionale Anforderungen

Die Anwendung muss nicht nur die in Kapitel 3.1 definierten funktionalen Anforderungen einhalten, sondern auch nichtfunktionale Anforderungen erfüllen. Die folgenden nichtfunktionalen Anforderungen bestimmen die Qualitätseigenschaften der Anwendung, die im Architekturentwurf berücksichtigt werden müssen.

Nr.	Anforderung	Priorität
NFA01	Funktionalität	unverzichtbar
NFA02	Benutzerfreundlichkeit	sehr wichtig
NFA03	Zuverlässigkeit	sehr wichtig
NFA04	Effizienz	sehr wichtig
NFA05	Wartbarkeit	wichtig
NFA06	Wiederverwendbarkeit	wichtig

Tabelle 3.2: Nichtfunktionale Anforderungen

NFA01 Funktionalität

Bei der Entwicklung von Software nimmt die Qualität eine zentrale Rolle ein, denn gerade fehlerhafte Merkmale einer Software fallen dem Nutzer besonders auf und verleiten diesen dazu, sich nicht weiter mit der Anwendung zu beschäftigen. Daher soll das System die in Kapitel 3.1 definierten Funktionen je nach ihrer Prioritätsstufe fehlerfrei ausführen können, um einen störungsfreien Ablauf zu gewährleisten.

NFA02 Benutzerfreundlichkeit

Die vorhandenen Funktionen sollen für den Benutzer verständlich sein, das heißt der Nutzer soll das Systemkonzept verstehen und damit umgehen können. Des Weiteren soll der Anwender die Aspekte der Anwendung, wie zum Beispiel die Bedienung, leicht erlernen können. Außerdem soll durch ein einheitliches Erscheinungsbild dem Nutzer ein konsistentes System vermittelt werden.

NFA03 Zuverlässigkeit

Die Anwendung soll eine möglichst geringe Fehleranfälligkeit besitzen und auch durch ungewollte Benutzereingaben nicht abstürzen. Dazu gehört auch die Wiederherstellung des ursprünglichen Systems, um eine Persistenz der Daten zu gewährleisten.

NFA04 Effizienz

Die im System benötigten Funktionen zur Kommunikation mit dem Server sollen eine angemessene Verarbeitungszeit aufweisen. Des Weiteren sollen die für die Anwendung benötigten statischen Dateien möglichst wenig Speicherplatz verbrauchen, um mobile Geräte nicht unnötig zu belasten.

NFA05 Wartbarkeit

Die Architektur des Systems soll so aufgebaut sein, dass alle benötigten Bausteine modular aufgebaut und einfach auszutauschen beziehungsweise zu erweitern sind. Außerdem soll das System die Möglichkeit bereitstellen, Mängel oder Ursachen von Versagen zu diagnostizieren.

NFA06 Wiederverwendbarkeit

Durch einen modularen Aufbau sollen auch Einzelkomponenten in anderen Systemen wieder verwendet werden können.

4

Konzept und Entwurf

Die in diesem Kapitel vorgestellten Konzepte und Entwürfe dienen als Grundlage für die in Kapitel 5 beschriebene Implementierung der Anwendung. In Kapitel 4.1 wird anschaulich die Benutzeroberfläche mit Hilfe von Mockups dargestellt und erklärt. Weiterhin werden in Kapitel 4.2 Dokumentdatenmodelle anhand einer Datenanalyse festgelegt und anschließend definiert. Anschließend wird in Kapitel 4.3 eine Software-Architektur vorgestellt, ihre einzelnen Komponenten erklärt und die Integration dieser Komponenten in das System beschrieben.

4.1 Mockups

Ein Teilziel dieser Arbeit ist die Darstellung von $360°$-Panoramabildern in Kombination mit räumlichen Geräuschquellen [2]. Grundsätzlich besteht die dazu benötigte Benutzeroberfläche nur aus einer View, die aus einem Quadrat besteht, dessen sechs Seiten texturiert sind. Ziel von Mockups ist die Darstellung der Interaktionsmöglichkeiten des Nutzers mit der Anwendung. Dabei soll unter Berücksichtigung der in Kapitel 3.2 vorgestellten nichtfunktionalen Anforderungen, im speziellen die Anforderung Benutzerfreundlichkeit 3.2, ein Layout für die Anwendung konstruiert werden, das dem Nutzer eine einfache und intuitive Bedienung ermöglichen soll.

Abbildung 4.1 stellt den Startbildschirm der Anwendung dar. Da der Nutzer im späteren Verlauf eigene Szenen erstellen und speichern kann, soll der Nutzer im Startbildschirm nur die Möglichkeit haben eine vorhandene Szene zu laden oder eine neue Szene zu erstellen. Wenn der Nutzer sich dazu entscheidet eine Szene zu laden, werden dem

Nutzer die vorhandenen Szenen in Form einer Liste angezeigt, aus welcher dann eine Szene gewählt werden kann. Bei der Auswahl einer Szene, wie es in Abbildung 4.1

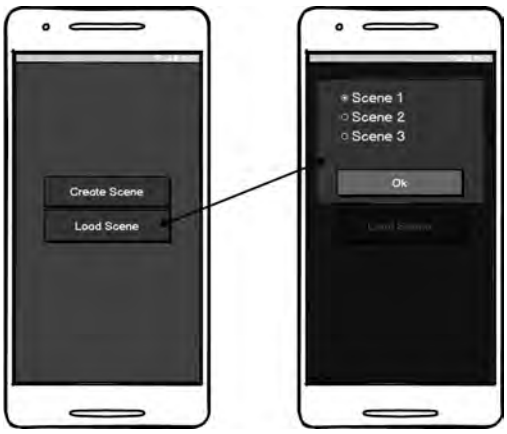


Abbildung 4.1: Der Startscreen der Applikation

gezeigt wird, ist zu beachten, dass alle in der Datenbank gespeicherten Szenen geladen werden können, jedoch nur eine Szene maximal aktiv sein kann. Hat der Benutzer eine

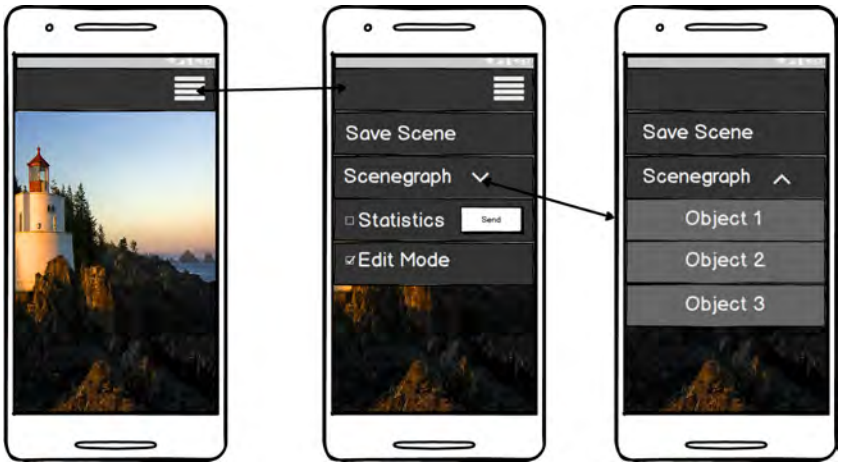


Abbildung 4.2: Übersicht der Anwendung und Ausfahren der Navigation

Szene ausgewählt, wird diese geladen und der Benutzer zum Hauptbildschirm weitergeleitet. Dieser besteht aus einer Navigation und einer View, die das $360°$-Panoramabild beinhaltet. In Abbildung 4.2 ist das Menü zu sehen und zeigt wie der Nutzer mit der Anwendung interagieren kann. Zunächst kann der Nutzer sich dazu entscheiden seine

Szene zu speichern, siehe Abbildung 4.3. Die gespeicherten Szenen sollen über ihren Namen identifiziert werden, deshalb wird der Nutzer dazu aufgefordert einen Namen für die Szene anzugeben. Eine weitere Möglichkeit, wie der Nutzer mit der Applikation interagieren kann, ist das Laden des Szenengraphen. Darin werden alle in der Szene vorhanden Objekte aufgelistet. Weiterhin kann der Anwender das automatische Senden von gespeicherten Nutzerdaten an- und ausschalten. Dabei ist zu beachten, dass das automatische Senden von Nutzerinformationen erst dann erfolgt, wenn der Nutzer die Anwendung beendet. Ist dagegen das automatische Senden deaktiviert, wird der *Send*-Button aktiv, mit dem die Daten manuell übertragen werden können. Die Aktivierung des letzten Menüpunktes hat zur Folge, dass die Szene in den Bearbeitungsmodus versetzt wird, was zur Folge hat, dass *Soundobjects* visualisiert werden. Ist der Szenengraph ge-

Abbildung 4.3: Beim Speichern einer Szene muss der Nutzer einen Namen angeben

laden, hat der Benutzer die Möglichkeit, wie in Abbildung 4.4 zu sehen, die Attribute der vorhandenen Objekte in der Szene zu editieren. Dazu sollen Attribute, wie die Position der Objekte oder der Pfad der dahinterliegenden Audiodateien oder Texturen, gehören.

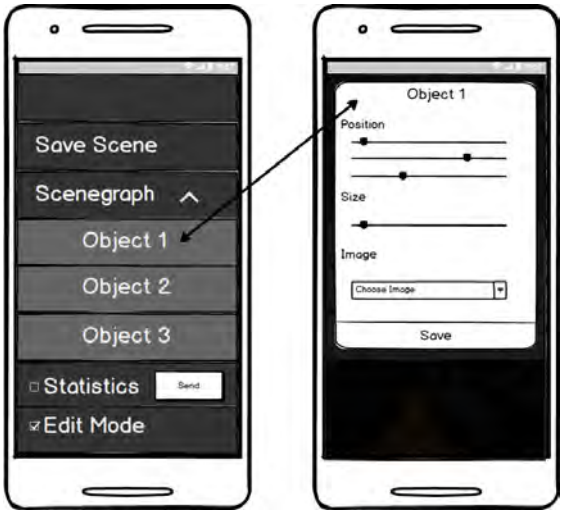


Abbildung 4.4: Eigenschaften eines Objektes können zur Laufzeit geändert werden

4.2 Datenbankentwurf

In diesem Kapitel werden Datenbankmodelle für die in Kapitel 3.1 vorgestellten funktionalen Anforderungen entworfen. Ziel dieses Kapitels ist die Ermittlung und Festlegung der zu verwaltenden Informationen mit Hilfe eines konzeptionellen Modells. Hierfür werden in Kapitel 4.2.1 die dazu benötigten Daten ermittelt und definiert. Weiterhin soll in Kapitel 4.2.2 durch ein konzeptionelles Modell verdeutlicht werden, wie und in welchem Zusammenhang die einzelnen Komponenten miteinander interagieren.

4.2.1 Datenanalyse

Szene

Die Szene bildet das Grundgerüst des Systems und verwaltet alle Objekte, die nachher auf dem Bildschirm zu sehen sind. Daher ist die Überführung der Szenenobjekte in ein Dokumentdatenmodell unumgänglich. Ziel des Dokumentdatenmodells für die Szene ist es, dass innerhalb eines Datenbankeintrages beliebig viele Objekte der Szene hinzugefügt, verändert oder gelöscht werden können.

Szenenobjekte

Szenenobjekte definieren den eigentlichen Inhalt der Szene, denn diese müssen sowohl Position, Skalierung, als auch Pfade zu statischen Inhalten, wie Texturen oder Audiodateien beinhalten. Grundsätzlich wird dabei zwischen zwei verschiedenen Szenenobjekten unterschieden, nämlich das *CubeObject* und das *SoundObject*. Im Datenmodell der Szene sollen die unterschiedlichen Szenenobjekte jeweils getrennt abgespeichert werden.

Kamera

Die Kamera ist ein essentieller Bestandteil des Systems und der Szene, allerdings sollte der Nutzer nicht die Möglichkeit haben diese nachträglich zu ändern. Aus diesem Grund soll die Kamera nicht mit in das Datenmodell der Szene einbezogen werden.

Log

Wie in Anforderung 3.1 vorgestellt, sollen bei der Benutzung der Applikation Nutzerdaten gespeichert werden. Dazu zählen sowohl die Zeit, die der Nutzer braucht, um eine Geräuschquelle zu lokalisieren, als auch die Zeit, die der Nutzer insgesamt für die Applikation aufwendet. Oder auch Statistiken, wie oft der Anwender erfolgreich eine Geräuschquelle lokalisiert hat.

4.2.2 Konzeptionelles Modell

Um die Datenmodellierung, der in Kapitel 4.2.1 vorgestellten Komponenten auszuarbeiten, soll mit Hilfe des *Entity-Relationship*-Diagramms 4.5 deren Kommunikation und Datenverwaltung genauer spezifiziert werden.

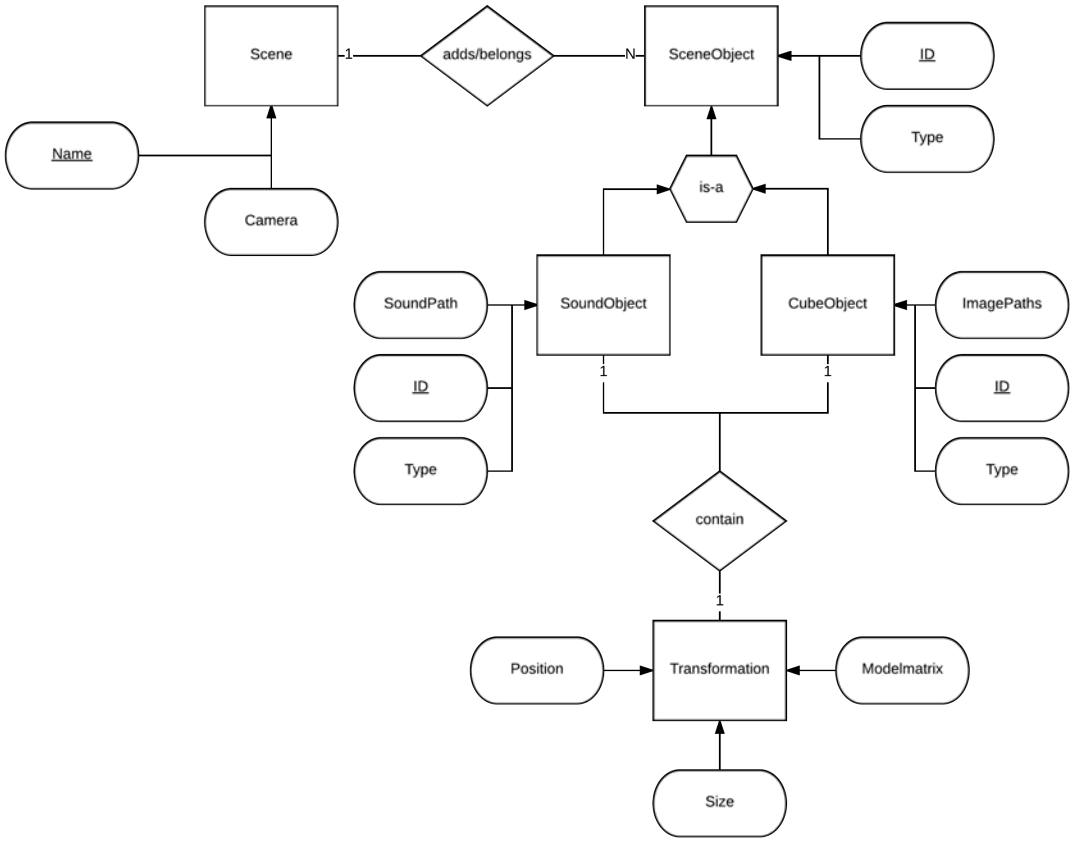


Abbildung 4.5: Das hier gezeigte ER-Diagramm visualisiert die Datenbeziehungen zwischen den Szenen und den Objekten, die sie verwaltet

CubeObject

Das *CubeObject* stellt den Prototyp der Cubemap dar und ist eine Unterklasse von *SceneObject*. Das *CubeObject* besitzt als Klassenvariable das Objekt *Transformation*, in welchem die Position und die Skalierung des *CubeObjects* gespeichert werden. Weiterhin besitzt das *CubeObject* eine eindeutige *ID*, einen *Type* und optional auch Pfade, die auf die Texturen verweisen. Daraus ergibt sich das in Quelltext 4.1 gezeigte Dokumentdatenmodell für das *CubeObject*.

```
{
  "id": id,
  "position" : transformation.position,
  "size" : transformation.size,
  "texturePaths" : {
        "img1" : texturePaths[0],
        "img2" : texturePaths[1],
        "img3" : texturePaths[2],
        "img4" : texturePaths[3],
        "img5" : texturePaths[4],
        "img6" : texturePaths[5]
  }
}
```

Listing 4.1: Dokumentdatenmodell für das CubeObject

SoundObject

Das *SoundObject* repräsentiert nicht nur ein *CubeObject*, sondern auch eine Geräuschquelle im Raum. Daher sind hier dieselben Attribute wie bei einem *CubeObject* abzuspeichern. Allerdings wird anstelle der *texturePaths* jedoch ein Pfad angegeben, der auf die Audiodatei des *SoundObjects* verweist.

```
{
  "id": id,
  "position" : transformation.position,
  "size" : transformation.size,
  "soundPath" : soundPath
}
```

Listing 4.2: Dokumentdatenmodell für das *SoundObject*

Scene

Für die Szene werden nun alle vorhandenen Objekte in die in Kapitel 4.2.2 vorgestellte Notation für *SoundObjects* und *CubeObjects* gebracht. Zusätzlich wird noch der Name der Szene mit abgespeichert, um diese zu identifizieren. Daraus ergibt sich das in Quelltext 4.3 dargestellte Format.

```
{
  "name": name,
  "sceneObjects" : {
        "cubes" : [
            "id": id,
            "position" : transformation.position,
            "size" : transformation.size,
            "texturePaths" : {
                "img1" : texturePaths[0],
                "img2" : texturePaths[1],
                "img3" : texturePaths[2],
                "img4" : texturePaths[3],
                "img5" : texturePaths[4],
                "img6" : texturePaths[5]
            ]

        },
        "sounds" : [
            "id": id,
            "position" : transformation.position,
            "size" : transformation.size,
            "soundPath" : soundPath
        ]
  }
}
```

Listing 4.3: Dokumentdatenmodell für die Szene

Log

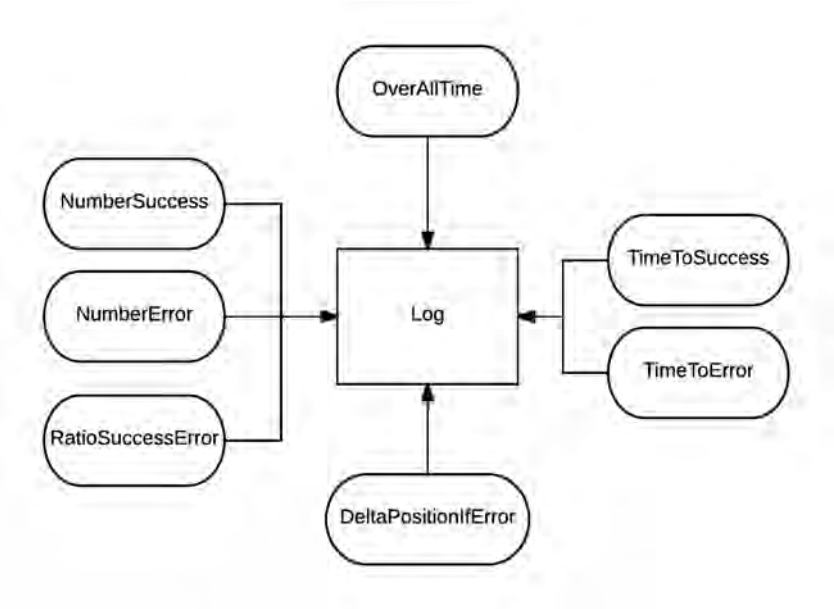


Abbildung 4.6: ER-Diagramm für den Log

Das Log Objekt dient zur Aufzeichnung von Nutzerdaten während der Interaktion. Bei jeder Benutzung der Applikation durch den Anwender sollen diese Nutzerdaten in einer Datenbank gespeichert und anschließend ausgewertet werden. Das ER-Diagramm 4.6 zeigt den Aufbau und die zu verwaltenden Informationen dieser Klasse. Relevante Informationen, die der Log abspeichern soll, sind zu einem die Zeit, die der Nutzer insgesamt für die Benutzung der Applikation aufwendet, die Anzahl der Erfolge und Fehler bei der Lokalisierung von Geräuschquellen. Zusätzlich soll noch die Zeit, die bis zu einem Erfolg oder Fehler verstrichen ist, und die Fehlerdifferenz im Raum, falls der Nutzer die Geräuschquelle falsch lokalisiert hat, mit aufgezeichnet werden. Quelltext 4.4 verdeutlicht das Dokumentdatenmodell des Logs auf Basis des in Abbildung 4.6 gezeigten ER-Diagramms.

```
1 {
2   "id": id,
3   "overAllTime" : overAllTime,
4   "timeToSuccess" : timeToSuccess,
```

```
5     "timeToError" : timeToError,
6     "deltaPositionError" : [
7           "x" : delta[0],
8           "y" : delta[1],
9           "z" : delta[2]
10    ],
11    "ratioSuccessError" : ratio,
12    "numberSuccess" : numberSuccess,
13    "numberError" : numberError
14 }
```

Listing 4.4: Dokumentdatenmodell für den Log

4.3 Architektur

Ein wichtiger Bestandteil der Softwareentwicklung ist das Erstellen einer geeigneten Software-Architektur. Sie beschreibt die einzelnen Komponenten und deren Verbindungen untereinander. Allerdings ist die genaue Ausarbeitung einer Software-Architektur nicht immer von Vorteil, denn oft werden dadurch Implementierungsaspekte eingeschränkt. Daher sollte die Software-Architektur noch nicht den finalen Entwurf darlegen, sondern vielmehr die Kommunikation zwischen den einzelnen Komponenten beschreiben.

4.3.1 Überblick

Abbildung 4.7 zeigt einen Überblick über die einzelnen Komponenten der Anwendung und deren Verbindungen untereinander. Im Mittelpunkt der Anwendung steht die *Application*. Sie stellt Methoden zur Verfügung, um eine *Scene* aus der Datenbank zu laden oder zu speichern, welche über die Benutzeroberfläche ansprechbar sind. Außerdem ist die *Application* dafür zuständig, dass die aktuelle *Scene* mit den richtigen Parametern gerendert wird. Notwendig dafür sind *ShaderPrograms*, deren Quellcodes statisch in die Meteor Templates eingebunden sind und eine *Camera*. Die beiden letzteren sind

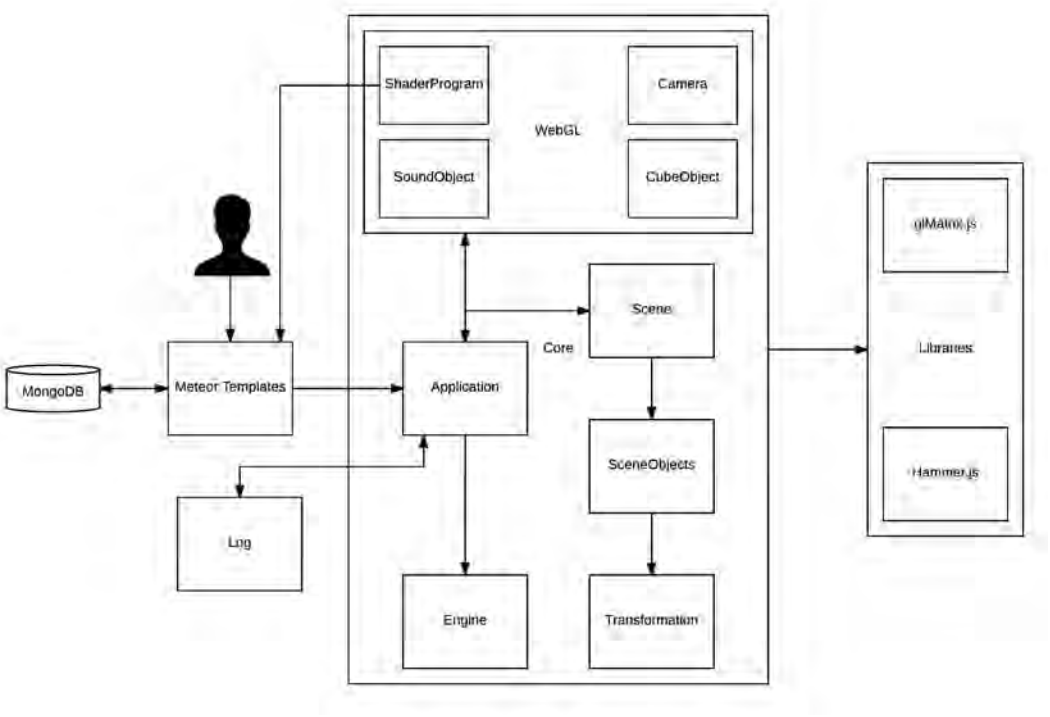


Abbildung 4.7: Überblick über die einzelnen Komponenten der mobilen Anwendung und deren Verbindungen

Komponenten die Zugriffe auf einen *WebGLContext* brauchen, den der *Core* bereitstellt. Dieser ist als *namespace* zu verstehen, der wichtige Funktionen und Variablen, die für WebGL Zugriffe benötigt werden, bereitstellt. Weitere Komponenten, die WebGL Zugriffe benötigen, sind jeweils das *SoundObject* und das *CubeObject*, welche beide vom Typ *SceneObject* sind. Diese werden in der *Application* mit dem jeweiligen *Shaderprogram* erstellt und der *Scene* hinzugefügt. Interagiert der Anwender mit der Benutzeroberfläche, so kann er die *Transformation* der einzelnen *SceneObjects* ansprechen und diese zur Laufzeit verändern. Da für die Kamerabewegung auch eine geeignete *FPS* nötig ist, dient die *Engine* zur Erzeugung einer Render Loop, die eine gleichbleibende *FPS* gewährleistet.

4.3.2 Engine

Das zentrale Element, das die *Application* benötigt um alle grafischen Elemente darzustellen und eine Kamerabewegung zu simulieren, ist die Render Loop. Diese ist häufig gleichzustellen mit der Game Loop, die aus Game Engines bekannt ist. Hauptaufgabe der Render Loop ist es, die Eingaben des Benutzers zu verarbeiten, das heißt die Attribute der betroffenen *SceneObjects* zu verändern und anschließend die *Scene* zu rendern. Abbildung 4.8 stellt vereinfacht den Ablauf einer Render Loop dar. Grundsätzlich lässt

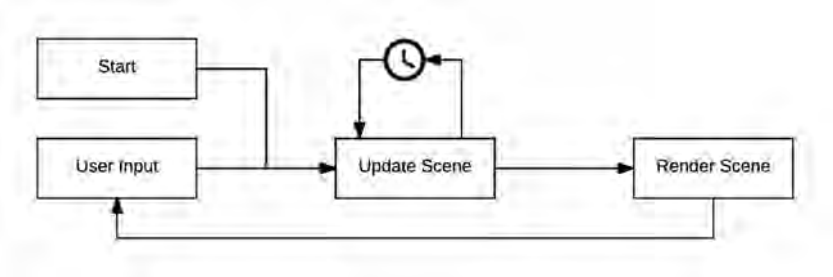


Abbildung 4.8: Funktionsweise der Render Loop

sich die Render Loop in unterschiedliche Phasen aufsplitten. Diese sind *start*, *update* und *render*. Im Weiteren Verlauf dieser Arbeit ist die *render*-Phase mit der *draw*-Phase gleichzustellen. Die *start*-Phase ist dazu da, um Objekte zu initialisieren, wie zum Beispiel die Initialisierung von *SceneObjects* oder auch die *Camera*. Diese Initialisierung findet allerdings nur beim Starten der Anwendung statt. Sind alle für die *draw*-Phase notwendigen Objekte initialisiert und noch keine Eingaben des Nutzers zu verarbeiten, so werden alle vorhandenen *SceneObjects* gerendert, was dann die Aufgabe der *draw*-Phase ist. Müssen hingegen Nutzereingaben verarbeitet werden, zum Beispiel das Verschieben eines *SceneObjects* oder eine Kamerabewegung, dann wird vor der *draw*-Phase die *update*-Phase eingeleitet, in der die Attribute der *Sceneobjects* anhand der Benutzereingaben verändert werden. Die *SceneObjects* werden also vorbereitet, um anschließend in der *draw*-Phase gerendert zu werden. Ziel der *Engine* ist es diese Phasen in Methoden zu kapseln, um beim Starten der *Application* die Render Loop zu initialisieren. Das heißt, die *Application* besitzt selbst die Phasen *start*, *update* und *draw*.

Diese werden allerdings von Funktionen der *Engine* aufgerufen, wie es in Abbildung 4.9 dargestellt wird.

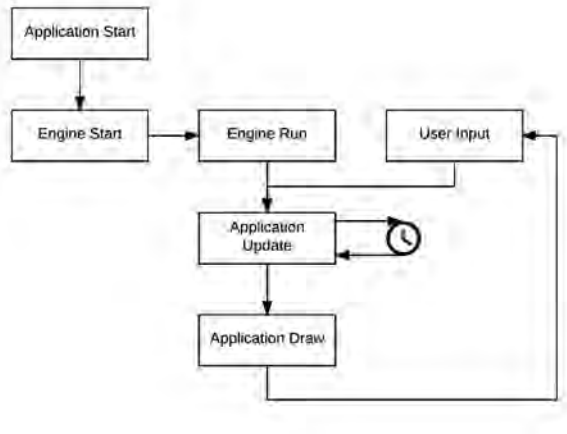


Abbildung 4.9: Zusammenspiel zwischen *Application* und *Engine*

4.3.3 SceneObject

SceneObjects sind Objekte, die mit Hilfe von WebGL gezeichnet werden. Dabei muss beachtet werden, dass das Zeichnen von mehreren Buffern zur gleichen Zeit in WebGL nicht möglich ist. Stattdessen müssen die einzelnen Buffer nacheinander gebunden und gezeichnet werden. Die Buffer-Verwaltung spielt daher eine wichtige Rolle in jeder Anwendung, die WebGL benutzt. Abbildung 4.10 soll den Initialisierungs- und Zeichenprozess der *Scene* verdeutlichen. Am Anfang wird die *Scene* initialisiert, das bedeutet, dass alle hinzugefügten *SceneObjects* initialisiert werden. Anschließend kann die *Scene* mit der in Quelltext 4.6 vorgestellten *draw*-Funktion gezeichnet werden. Grundsätzlich hilft die *Scene* bei der Verwaltung der *SceneObjects*, da alle Szenenobjekte direkt in der Szene abgespeichert werden. Dadurch kann in der *draw*-Phase nun über alle *SceneObjects* iteriert werden. Dabei wird der Buffer des *SceneObjects* am aktuellen Index gebunden und das Objekt kann gezeichnet werden. Verglichen mit den Phasen der Render Loop, hat der Renderprozess der *SceneObjects* nur zwei Phasen, genauer gesagt eine *init*- und *draw*-Phase. In der *init*-Phase werden sowohl Positionkoordinaten,

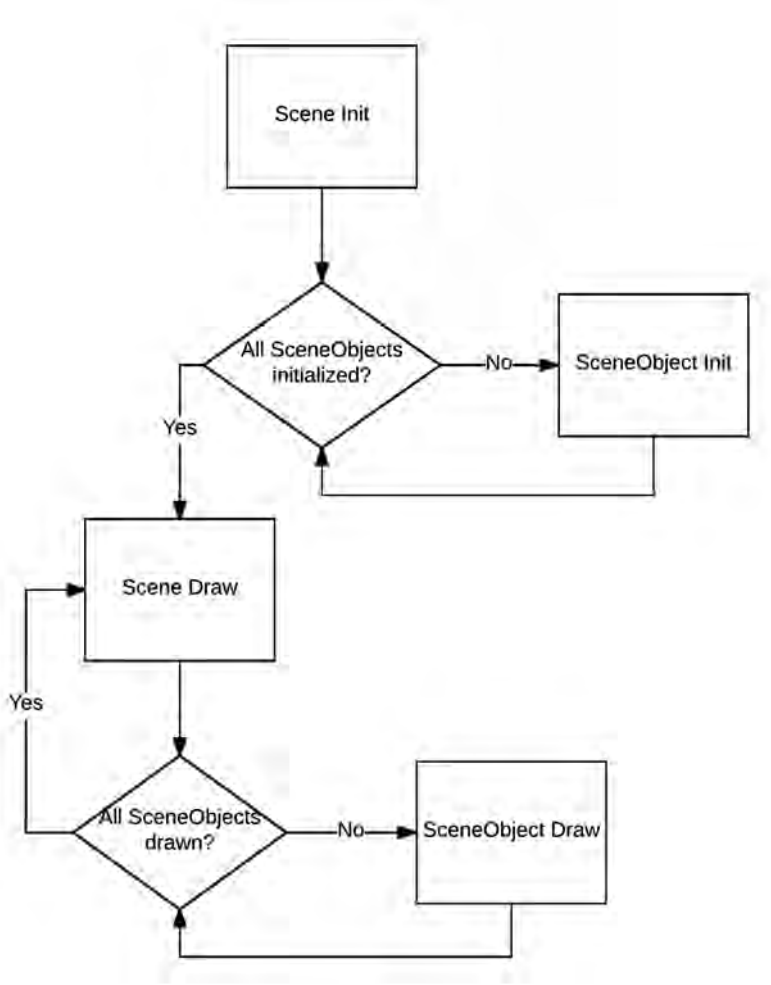


Abbildung 4.10: Zusammenspiel zwischen *Scene* und *SceneObject*

Texturkoordinaten und Texturen definiert, als auch die dazugehörigen Buffer initialisiert und befüllt. Quelltext 4.5 zeigt eine mögliche *init*-Funktion eines *SceneObjects*.

```
1 function init(){
2     object.position = [-1, -1, 0, 1, -1, 0, 0, 1, 0];
3     object.texture = [0, 0, 1, 0, 1, 1];
4
5     object.positionBuffer = gl.createBuffer();
6     var posArray = new Float32Array(object.position);
7     gl.bindBuffer(gl.ARRAY_BUFFER, object.positionBuffer);
```

```
8       gl.bufferdata(gl.ARRAY_BUFFER, posArray, gl.STATIC_DRAW);
9
10      object.textureBuffer = gl.createBuffer();
11      var texArray = new Float32Array(object.texture);
12      gl.bindBuffer(gl.ARRAY_BUFFER, object.textureBuffer);
13      gl.bufferData(gl.ARRAY_BUFFER, texArray, gl.STATIC_DRAW);
14  }
```

Listing 4.5: *init*-Funktion für *SceneObjects*

Ist das *SceneObject* erfolgreich initialisiert worden, kann in der *draw*-Phase der *Application* das *SceneObject* gerendert werden. Hierfür müssen alle Buffer, die für das *SceneObject* benutzt werden, wieder unter Verwendung eines *ShaderPrograms* gebunden werden. Eine mögliche *draw*-Funktion für das *SceneObject* wird in Quelltext 4.6 gezeigt.

```
1   function draw(){
2       var program = object.shaderProgram;
3       gl.useProgram(program);
4
5       gl.bindBuffer(gl.ARRAY_BUFFER, object.positionBuffer);
6       var posLoc = gl.getAttribLocation(program,"position");
7       gl.vertexAttribArray(posLoc, "position"),
8                           3, gl.FLOAT, false, 0, 0);
9
10      gl.bindBuffer(gl.ARRAY_BUFFER, object.textureBuffer);
11      var texLoc = gl.getAttribLocation(program, "texture");
12      gl.vertexAttribArray(texLoc, "texture"),
13                          2, gl.FLOAT, false, 0, 0);
14
15     gl.drawArrays(gl.TRIANGLES, 0, object.position.length/3);
16     gl.useProgram(null);
17  }
```

Listing 4.6: *draw*-Funktion für *SceneObjects*

4.3.4 Camera

Heutzutage ist es unmöglich sich eine 3D Applikation ohne Kamera vorzustellen. Allerdings existiert keine Kameraklasse in der WebGL API. Jedoch ist es möglich mit der vorgegebenen *low-level* API von WebGL eine Kamera zu abstrahieren. Bei bisherigen Implementierungen ohne eine Kamera werden Transformationen, also Translationen, Rotationen und Skalierungen in unterschiedlichen Matrizen abgespeichert und beim Renderprozess an den Vertexshader übermittelt. Werden mit Hilfe der Funktion *drawArrays* Eckpunkte gezeichnet, werden auf diese die Transformationen angewendet. Das bedeutet, mit diesen Matrizen können Objekte positioniert und orientiert werden. Diese Transformationsmatrizen können zu einer Kompositionsmatrix multipliziert werden. Man spricht dann von einer *Model*-Matrix. Bei der Abstraktion einer Kamera in WebGL muss nun eine *View*-Matrix eingeführt werden. Diese Matrix ist dafür zuständig, dass die Eckpunkte eines Objekts im *world-space* in den *view-space* transformiert werden.

Diese *View*-Matrix besteht aus drei orthogonalen Vektoren, welche die Position, Richtung und Orientierung der Kamera repräsentieren. Der *look*-Vektor stellt die Richtung, in welche die Kamera zeigt, dar. Der *up*-Vektor definiert die vertikale Ausrichtung der Kamera. Der *position*-Vektor gibt den Ursprung der Kamera an. Eine Funktion, die genau diese Matrix erzeugt, wird in Kapitel 2.2.2 beschrieben. Um die Kamera bewegen zu können, müssen die Vektoren *up* und *look* rotiert werden. Zusätzlich existiert noch ein sogenannter *right*-Vektor, der für weitere Berechnung allerdings nicht notwendig ist. Die Rotation um den *look*-Vektor heißt *roll* und repräsentiert die Rotation um die z-Achse. Die Rotation um den *up*-Vektor wird *yaw* genannt und repräsentiert die Rotation um die y-Achse. Die Rotation um die x-Achse wird *pitch* genannt. Ursprünglich bezeichnet der *yaw-pitch-roll*-Winkel eine Möglichkeit zur Beschreibung der Orientierung eines Fahrzeugs im dreidimensionalen Raum. Abbildung 4.11 stellt diese Rotation grafisch dar. Abhängig von Benutzereingaben und den zuletzt gespeicherten Positionskoordinaten auf dem Bildschirm wird jeweils ein Offset für die x- und y-Achse berechnet und den Variablen *yaw* und *pitch* hinzugefügt. Eine Rotation um die z-Achse ist bei einer statischen Kamera dagegen nicht notwendig. Auf Basis der berechneten Werte des *yaw* und *pitch* werden die Vektoren *up* und *right* neu berechnet. Daraus ergibt sich dann

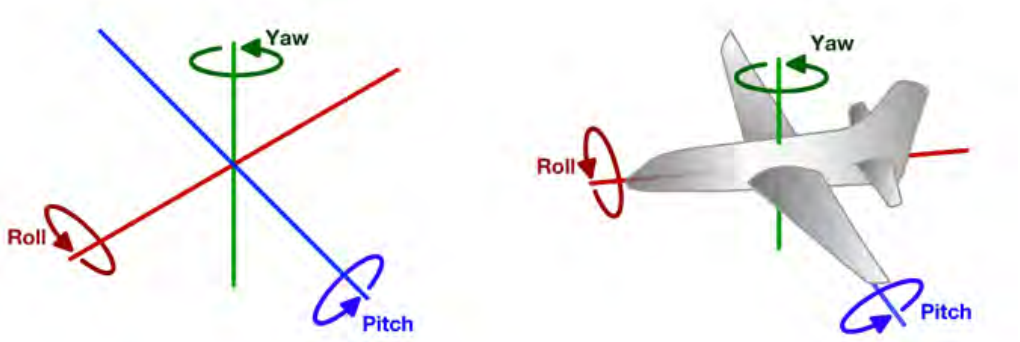


Abbildung 4.11: Visualisierung der einzelnen Rotationen im dreidimensionalen Raum [24].

eine neue *View*-Matrix, die in Kombination mit der perspektivischen Matrix, siehe Kapitel 2.2.2, eine Kamerabewegung simuliert. Abbildung 4.12 visualisiert den Renderprozess unter Berücksichtigung von Nutzereingaben. Eine ausführliche Implementierung wird in Kapitel 5 gegeben.

4.3.5 Log

Ein wichtiger Bestandteil der Anwendung ist das Aufzeichnen von Statistiken. Grundsätzlich sollen ab dem Zeitpunkt Informationen gesammelt werden, ab dem der Nutzer mit der Anwendung interagiert. Allerdings ergibt es in diesem Fall Sinn, die Aufzeichnung erst beim Starten einer Szene zu beginnen, da erst dort alle relevanten Informationen aufgezeichnet werden können. Dabei sollen die Daten während der Benutzung lokal gespeichert werden und beim Beenden der Anwendung an einen Server geschickt werden. Abbildung 4.13 verdeutlicht den Ablauf des Renderprozesses mit einem integriertem Log Vorgang. Der Log wird beim Initialisieren der Szene gestartet und fängt in der *draw*-Phase der *Scene* an Informationen zu sammeln. In jedem Frame werden nun Informationen aufgezeichnet und vor Beginn des nächsten Frames in einer lokalen Datenbank gespeichert. Falls das automatische Absenden der Log Informationen deaktiviert ist, werden diese erst nach einer Bestätigung durch den Benutzer an den Server weitergeleitet. Falls dies nicht zutrifft, werden die Daten lokal so lange gespeichert, bis

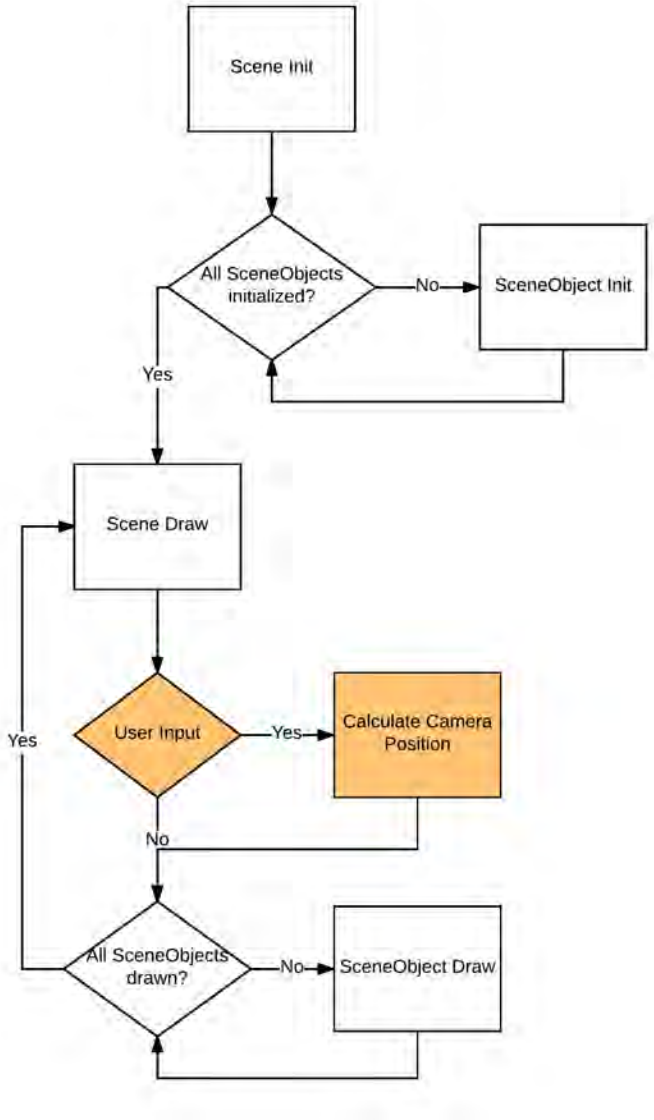


Abbildung 4.12: Zusammenspiel zwischen *Scene*, *SceneObject* und *Camera*

der Nutzer die Daten manuell sendet. Ist der automatische Upload aktiviert, werden beim Beenden der Applikation alle bisher gesammelten Informationen an den Server geschickt.

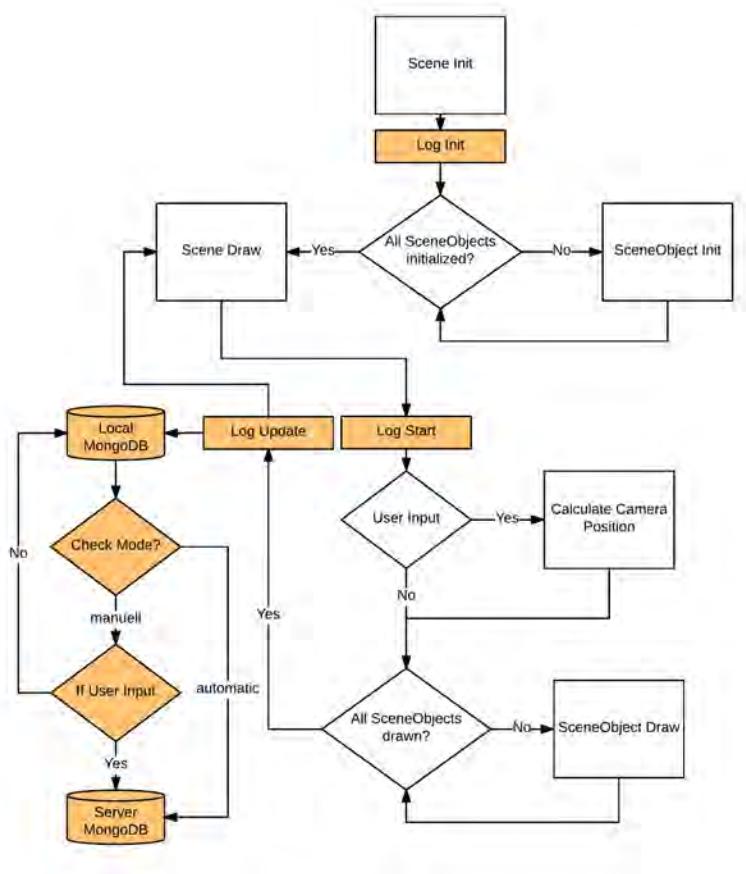


Abbildung 4.13: Zusammenspiel zwischen *Scene*, *SceneObject* und *Camera*

4.3.6 Räumlicher Sound

Ein weiterer wichtiger Punkt ist die richtige Aktualisierung der räumlichen Geräuschquellen in der Szene. Wie in Kapitel 2.4 beschrieben besitzt die Web Audio API die Schnittstelle PannerNode, mit der sich räumliche Audioquellen realisieren lassen. Jede Instanz der Schnittstelle PannerNode besitzt einen Ortsvektor und einen Richtungsvektor, der anzeigt, in welche Richtung der Ton gespielt wird. Die Berechnung, wie der Ton auf einem Ausgabegerät gespielt wird, hängt von der Position und Orientierung der Schnittstelle AudioListener ab. Da bei einer Rotation der Kamera nicht nur die Transformationen aller Szenenobjekte verändert werden, sondern auch die Orientierung des AudioListeners, müssen in der *draw*-Phase sowohl die Orientierung der PannerNodes,

als auch die Orientierung des AudioListeners in Abhängigkeit der Rotation aktualisiert werden. Dieser Prozess findet nur statt, wenn eine Benutzereingabe erfolgt ist und die *View*-Matrix neu berechnet worden ist. Abbildung 4.14 zeigt grafisch, an welcher Stelle der Richtungsvektor aktualisiert werden muss.

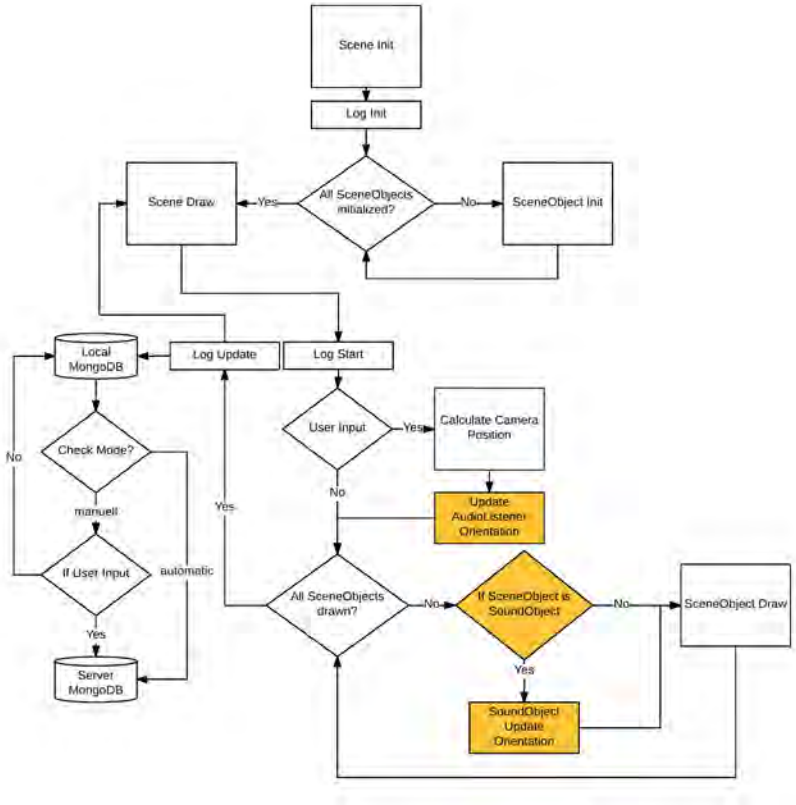


Abbildung 4.14: Zusammenspiel zwischen *Scene*, *SceneObject* und *Camera*

5

Implementierung

Im vorliegenden Kapitel wird anhand der in Kapitel 3 erhobenen Anforderungen und unter Berücksichtigung des in Kapitel 4 vorgestellten Entwurfs eine Implementierung der Applikation vorgestellt. Kapitel 5.1 werden zunächst die verwendeten Framework vorgestellt. Anschließend wird in Kapitel 5.2 beschrieben, wie eine Cubemap Textur erzeugt werden kann, die für die Anzeige eines Panoramabildes benötigt wird. Im Anschluss daran wird in Kapitel 5.3 die Implementierung von räumlichen Sound gezeigt. Des Weiteren werden in Kapitel 5.4 die Umsetzung der Kamerabewegung und in Kapitel 5.5 die Funktionsweise des visuellen Feedbacks vorgestellt. Abschließend wird in Kapitel 5.6 die Implementierung des Log Systems erklärt und in Kapitel 5.7 die finale Version der Applikation gezeigt.

5.1 Verwendete Frameworks

Im diesem Kapitel werden die in dieser Arbeit verwendeten Frameworks vorgestellt und erläutert. Kapitel 5.1.1 gibt eine Einleitung für das Vektor- und Matrix-Framework *glMatrix*, welches intensiv für Rechenoperationen genutzt wird. Anschließend wird in Kapitel 5.1.2 das *Hammer.js* Framework vorgestellt, welches Schnittstellen für Touch Eingaben auf mobilen Geräten ermöglicht. Das Bootstrap Framework wird danach in Kapitel 5.1.3 erläutert.

5.1.1 glMatrix

glMatrix ist eine Bibliothek, welche Rechenoperationen für Vektoren und Matrizen bereitstellt, die in der Computergrafik sehr häufig genutzt werden. Quelltext 5.1 zeigt die Erstellung von Vektoren und einer *lookAt*-Matrix. Meistens ist der erste Parameter von Funktionen, die Matrizen oder bestimmte Vektoren erzeugen, ein Pointer auf eine vorher angelegte Variable, in die das Ergebnis gespeichert wird. Alle in Kapitel 2.2 vorgestellten Funktionen, die in der Computergrafik benötigt werden, werden von dieser Bibliothek bereitgestellt.

```
1    let vEye = vec3.fromValues(0, 2, 0);
2    let vCenter = vec3.fromValues(2, 2, 2);
3    let vUp = vec3.fromValues(0, 1, 0);
4
5    let mView = mat4.create();
6    mat4.lookAt(mView, vEye, vCenter, vUp);
```

Listing 5.1: Erstellung einer *View*-Matrix

Hauptverwendung dieser Bibliothek sind aufwendige Matrixoperationen, wie zum Beispiel die Multiplikation von Matrizen oder das Erzeugen von *lookAt*- und *perspective*-Matrizen. Allerdings findet das Framework auch bei der Transformation von Szenenobjekten Verwendung, wenn deren Position oder Größe verändert wird. Aus diesen veränderten Werten wird dann eine *Model*-Matrix erstellt, welche die Änderungen abspeichert. Das ist bei der Verschiebung von Soundobjekten besonders wichtig, denn, wenn deren Position verändert wird, darf sich die Audioquelle nicht mehr an der vorherigen Stelle befinden, sondern muss um denselben Werten verschoben werden.

5.1.2 Hammer.js

Hammer.js ist ein Javascript Framework, das es ermöglicht, Touch-Gesten zu erkennen, um entsprechend darauf zu reagieren. Bisher werden folgende Touch-Gesten unterstützt.

- Pan
- Pinch

- Press
- Rotate
- Swipe
- Tap

Ein besonderer Vorteil von *Hammer.js* ist die einfache Benutzung und die *cross-platform* Unterstützung. Um *Hammer.js* nutzen zu können, muss eine neue Instanz des Hammer Objektes erzeugt werden, dessen Parameter das DOMElement ist, welches für Touch-Gesten empfänglich sein soll. Quelltext 5.2 zeigt wie diese Instanz erstellt wird.

```
1 let myElement = document.getElementById('hitarea');
2 let mc = new Hammer(myElement);
```

Listing 5.2: Initialisieren des Hammer Objekts

Ist das Objekt initialisiert können mit der Funktion *on* Eventlistener auf Touch-Gesten gelegt werden. Quelltext 5.3 verdeutlicht, wie auf *pan*- und *tap*-Events reagiert werden kann.

```
1  let myElement = document.getElementById('hitarea');
2  let mc = new Hammer(myElement);
3  mc.on('panstart', event => {
4      camera.setFirstMouse(true);
5  });
6  mc.on('panmove', event => {
7      camera.move(event);
8  });
9  mc.on('panend', event => {
10     camera.setFirstMouse(false);
11 });
12 mc.on('tap', event => {
13     // do something with event
14 });
```

Listing 5.3: Hinzufügen von Listenern, die auf die Touch-Gesten *pan* und *tap* reagieren

5.1.3 Bootstrap

Bootstrap ist ein für mobile Geräte entwickeltes Framework, mit dem effizient Frontends entwickelt werden können. Bootstrap wird über den Package Manager von Meteor hinzugefügt und lässt sich dann direkt verwenden. Grundsätzlich wurden in dieser Arbeit hauptsächlich CSS Komponenten verwendet. Eine detaillierte Auflistung der Komponenten, die in Bootstrap zur Verfügung stehen, findet sich unter [25].

5.2 Cubemap

Cubemaps sind Texturen, die nicht aus einer einzigen Fläche bestehen, sondern aus sechs 2D-Texturen, die jeweils auf eine Seite eines Würfels gelegt werden. In dieser Arbeit werden Cubemaps verwendet um einen dreidimensionalen Raum zu erzeugen. Dabei bestehen die Wände des Würfels aus den einzelnen Texturen. Abbildung 5.1 zeigt die sechs Texturen und in welcher Position sie jeweils verwendet werden. *Top*

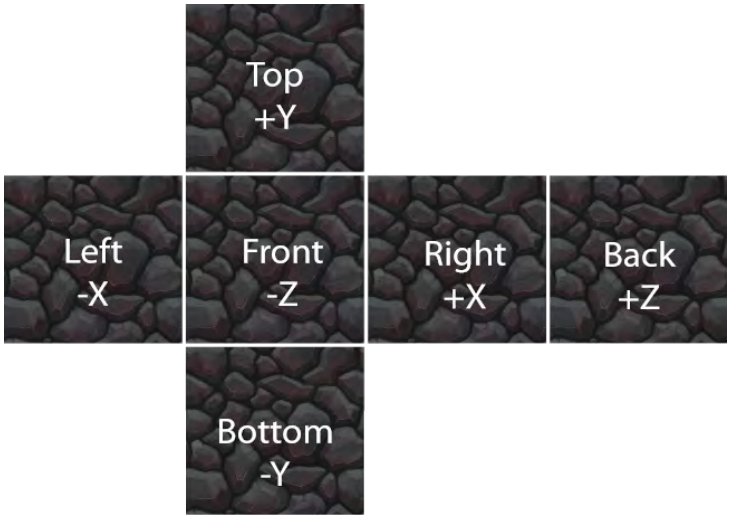


Abbildung 5.1: Aufbau einer Cubemap Textur, die aus sechs einzelnen Bildern besteht.

repräsentiert dabei die positive Y-Achse und *Bottom* die negative Y-Achse. *Front* und *Back* stehen für die Z-Achse und *Left* und *Right* für die X-Achse.

5.2.1 Laden von Cubemaps

Die Texturen für die Cubemap können statisch in HTML eingebunden werden, falls kein Server vorhanden ist. Da Meteor allerdings auch Server Code anbietet, können über Requests die einzelnen Texturen geladen werden. Dieser Prozess findet in der *start*-Phase statt, da das Laden und Erzeugen von Texturen nur einmal stattfinden muss und sehr rechenintensiv ist. Da diese Requests asynchron ablaufen, ist eine Struktur nötig, welche die Textur erst dann mit Hilfe von WebGL erzeugt, wenn alle Bilder geladen sind. Zur Kontrolle dafür wird ein Counter eingeführt, der zählt wie viele *onload*-Funktionen bereits gestartet sind. Startet die letzte *onload*-Funktion wird eine Textur erzeugt, gebunden und mit den vorher geladenen Bildern befüllt. Quelltext 5.4 soll diesen Ablauf verdeutlichen.

```
1
2   loadTexture(imagePaths) {
3     let counter = 0;
4     let gl = Core.mGetGL();
5     let images = new Array(6);
6     let type = gl.TEXTURE_CUBE_MAP;
7     for (let i = 0; i < imagePaths.length; i++) {
8       images[i] = new Image();
9       images[i].onload = function() {
10        counter++;
11        if (counter === 6) {
12            let textureBuffer = gl.createTexture();
13            gl.bindTexture(type, textureBuffer);
14            for (let j = 0; j < 6; j++) {
15                gl.texImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_X + j,
16                        0, gl.RGBA, gl.RGBA,
17                        gl.UNSIGNED_BYTE,
18                        images[j]);
19                gl.texParameteri(type,
20                            gl.TEXTURE_MAG_FILTER,
21                            gl.LINEAR);
22                gl.texParameteri(type,
```

```
23                                  gl.TEXTURE_MIN_FILTER,
24                                  gl.LINEAR);
25                      gl.texParameteri(type,
26                                  gl.TEXTURE_WRAP_S,
27                                  gl.CLAMP_TO_EDGE);
28                      gl.texParameteri(type,
29                                  gl.TEXTURE_WRAP_T,
30                                  gl.CLAMP_TO_EDGE);
31                  }
32                  gl.generateMipmap(type);
33                  gl.bindTexture(type, null);
34                  return textureBuffer;
35              }
36          };
37          images[i].src = imagePaths[i];
38      }
39  }
```

Listing 5.4: Laden der Cubemap Textur

Das Ergebnis von Quelltext 5.4 ist ein Texturbuffer, der gebunden werden kann, um so einen texturierten Würfel darzustellen. Abbildung 5.2 visualisiert den Würfel, nachdem

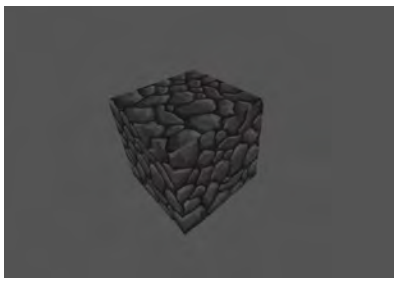


Abbildung 5.2: Ein Würfel, über dessen Seiten Texturen gelegt sind

die Textur an diesen gebunden worden ist. Wird der Würfel entsprechend skaliert und ist der Ursprung der Kamera innerhalb des Würfels, so umgeben die Innenseiten des Würfels den Ursprung, was ein Gefühl der Räumlichkeit erzeugt. Das kann genutzt werden um mit Hilfe von Texturen, die die reale Welt abbilden, einen virtuellen Raum zu

erzeugen. Abbildung 5.3 stellt die Cubemap, wie sie in der Applikation verwendet wird, dar.

Abbildung 5.3: Innenansicht der Cubemap in der mobilen Anwendung

5.3 Räumlicher Sound

Wie in Kapitel 4.3.6 beschrieben, muss bei einer Bewegung der Kamera durch den Anwender die Orientierung der Soundobjekte und auch die Orientierung des *AudioListeners* aktualisiert werden. Das geschieht durch die Funktion *update*, die von der Klasse *AudioRenderable* bereitgestellt wird. Diese Methode bekommt als einzigen Parameter ein *camera*-Objekt übergeben, welche die aktuelle *view-projection*-Matrix speichert. Weiterhin wird in der Kameraklasse die Orientierung des *AudioListeners* aktualisiert, da diese Orientierung vom Neigungsgrad der Kamerabewegung abhängt. Das bedeutet, dass abhängig davon, in welche Richtung die Kamera gedreht wird, die Orientierung angepasst werden muss. Quellcode 5.5 zeigt, dass die Orientierung direkt vom *lookAt*-Vektor abhängig ist, welcher jeden Frame neu berechnet werden muss. Die Berechnung des *lookAt*-Vektors wird in Quelltext 5.6 in Kapitel 5.4 genauer erläutert.

```
1   let ctx = Core.getContext();
2   let listener = ctx.listener;
3   listener.setOrientation(this._lookAt[0],
4                           this._lookAt[1],
5                           this._lookAt[2],
6                           0,
7                           1,
8                           0);
```

Listing 5.5: Innerhalb jedes Frames wird mit dem aktuellen *lookAt*-Vektor die Orientierung des *AudioListeners* aktualisiert

5.4 Kamerasteuerung

Die Kameraklasse stellt unterschiedliche Methoden bereit, die für die Bewegung der Kamera zuständig sind. Diese sind abhängig von Nutzereingaben und den daraus entstehenden Events. Navigiert der Nutzer mit der Touch-Geste *pan* durch die Szene, werden drei Methoden aufgerufen. Das erste Event, das vom Benutzer ausgelöst wird, ist das *panstart*-Event. Das hat zur Folge, dass die Kamera Variablen setzt, die angeben, welche die letzten X- und Y-Positionen sind. Die Information dafür stammt aus dem *panstart*-Event. Danach wird das *panmove*-Event ausgelöst, das zuständig für die Bewegung ist. Dabei wird ein Offset aus den letzten X- und Y-Positionen mit den Koordinaten des *panmove*-Events berechnet. Anschließend werden die letzten X- und Y-Positionen mit den Werten aus dem *panmove*-Event überschrieben. Der berechnete X-Offset wird der *yaw* Variable und der Y-Offset der *pitch* Variable hinzugefügt, siehe Kapitel 4.3.4. Mit Hilfe dieser zwei Variablen wird dann innerhalb jedes Frames eine neue *lookAt*-Matrix erzeugt. Für diese Matrix werden in der *glMatrix* Bibliothek drei Vektoren benötigt, wie es in Kapitel 4.3.4 beschrieben wird. Der *position*-Vektor bleibt gleich, da die Kamera immer an derselben Stelle bleibt. Der *lookAt*-Vektor muss allerdings neu berechnet werden, wie es in Quelltext 5.6 gezeigt wird. Dieser daraus entstandene

Vektor muss normalisiert und mit dem bisherigen *position*-Vektor addiert werden. Mit demselben *up*-Vektor wird jetzt eine neue *View*-Matrix erzeugt, die durch den Nutzer um die x- und y-Achse rotiert werden kann.

```
1     let lookAtVec = vec3.create();
2     lookAtVec[0] = Math.cos(yaw) * Math.cos(pitch);
3     lookAtVec[1] = Math.sin(pitch);
4     lookAtVec[2] = Math.sin(yaw) * Math.cos(pitch);
5
6     vec3.normalize(lookAtVec, lookAtVec);
7     vec3.add(lookAtVec, lookAtVec, positionVec);
8     mat4.lookAt(lookAtMat, positionVec, lookAtVec, upVec);
```

Listing 5.6: Die *lookAt*-Matrix wird mit Hilfe des *yaw* und *pitch* neu berechnet

5.5 Visuelles Feedback

Visuelles Feedback spielt nicht nur eine zentrale Rolle für die Benutzerfreundlichkeit der Anwendung, sondern liefert auch wichtige Informationen für den Log, wie zum Beispiel das erfolgreiche Lokalisieren einer Geräuschquelle.

Im Folgenden wird eine Implementierung vorgestellt, mit der Objekte durch einen Klick im dreidimensionalen Raum lokalisiert werden können. Grundlage dieser Ray Picking Methode sind die unterschiedlichen Koordinatensysteme, die Vektoren ab ihrer Definition bis zur Darstellung auf dem Bildschirm durchlaufen. Abbildung 5.4 listet die unterschiedlichen Koordinatensysteme auf, die ein Eckpunkt bis zu seiner Darstellung als Pixel durchlaufen muss. Erster Input der Transformationen sind die lokalen Koordinaten, damit sind die Eckpunkte eines Objekts gemeint und haben die Form eines dreidimensionalen Vektors. Jedes Objekt hat somit sein eigenes Koordinatensystem. Werden dagegen mehrere Objekte platziert, so braucht man einen Bezugspunkt, wo diese Objekte in der virtuellen Welt lokalisiert sind. Daher werden die Eckpunkte mit einer *Model*-Matrix multipliziert, welche die Punkte in einem einheitlichen Weltkoordinatensystem darstellt. Bezugspunkt ist dabei der Ursprung der Szene. Anschließend werden die Punkte mit

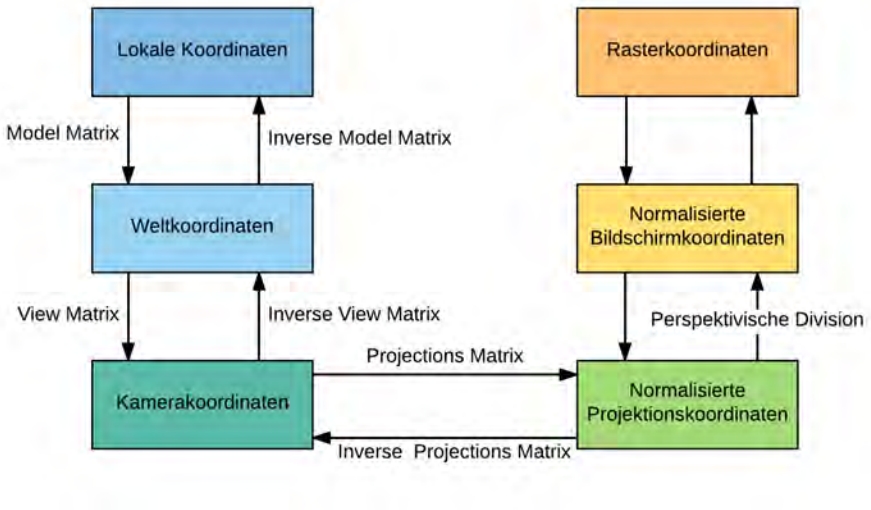


Abbildung 5.4: Koordinatentransformationen

einer *View*-Matrix in das Kamerakoordinatensystem transformiert, in welchem der Ursprung der Kamera der Bezugspunkt ist. Dadurch ist das Betrachten der Szene aus der Sicht der Kamera möglich. Die *projection*-Matrix transformiert danach die Eckpunkte in normalisierte Projektionskoordinaten. Die letzten beiden Transformationen werden ohne weitere Matrizen durchgeführt und transformieren den dreidimensionalen Vektor in einen zweidimensionalen Punkt, die Rasterkoordinaten. Bei der Methode Ray Picking wird nun ein Klick in Rasterkoordinaten aufgenommen und zurück in seine Weltkoordinaten transformiert. Diese Umwandlung wird anhand Quelltext 5.7 erklärt.

Zunächst werden die Rasterkoordinaten bestimmt, die danach in normalisierte Bildschirmkoordinaten umgewandelt werden. Diese befinden sich zwischen $[-1:1]$ und da die Kamera in Richtung negativer z-Achse zeigt, wird die z-Koordinate auf -1 gesetzt. Anschließend müssen die Koordinaten mit der inversen *projection*-Matrix multipliziert werden, um diese in das Kamerakoordinatensystem zu transformieren. Die letzte Transformation ist die Multiplikation der Koordinaten mit der inversen *View*-Matrix und man erhält die ursprünglichen Koordinaten des Punktes in Weltkoordinaten. Dieser so berechnete Vektor muss normalisiert werden, da es sich um einen Richtungsvektor handelt, dessen Ursprung das Zentrum der Kamera bildet.

```
document.onclick = function(e){
    // x,y in Rasterkoordinaten
    let x = e.clientX;
    let y = e.clientY;
    let z = -1;

    // x,y in Normalisierten Bildschirmkoordinaten
    x = (2.0 * x / 2) / canvas.width - 1.0;
    y = 1.0 - (2 * y / 2) / canvas.height;
    let ray_nbk = vec3.fromValues(x, y, z);

    // x,y,z,w in Normalisierten Projektionskoordinaten
    let ray_npk = vec4.fromValues(ray_nbk[0], ray_nbk[0], z, 1.0);

    // x,y,z,w in Kamerakoordinaten
    let inverse_projMatrix = mat4.create();
    let ray_kk = vec4.create();
    mat4.inverse(inverse_projMatrix, camera.projMatrix);
    vec4.transformMat4(ray_kk, ray_npk, inverse_projMatrix);

    // x,y,z,w in Weltkoordinaten
    let ray_wk = vec4.create();
    let inverse_viewMatrix = mat4.create();
    mat4.inverse(inverse_viewMatrix, camera.viewMatrix);
    vec4.transformMat4(ray_wk, ray_kk, inverse_viewMatrix);
    ray_wk = vec3.fromValues(ray_wk[0], ray_wk[1], ray_wk[2]);

    vec3.normalize(ray_wk, ray_wk);
}
```

Listing 5.7: Umwandlung von Rasterkoordinaten in Weltkoordinaten

Mit diesem Vektor kann nun berechnet werden, ob ein Schnittpunkt mit einem *SoundObject* existiert. Da das *SoundObject* als Würfel gezeichnet wird, existiert dafür ein Zentrum und ein Radius. Die Berechnung des Schnittpunktes mit einem Würfel erweist sich als deutlich aufwendiger als die Berechnung eines Schnittpunktes mit einer Kugel. Daher

wird überprüft, ob der Vektor stattdessen einen Schnittpunkt mit einer Kugel hat, dessen Zentrum und Radius mit den Werten des *SoundObjects* übereinstimmt. Quelltext 5.8 verdeutlicht, wie mit einem gegebenen Richtungsvektor und dessen Ursprung überprüft werden kann, ob Schnittpunkte mit dem *SoundObject* existieren. Um herauszufinden, ob ein gegebener Vektor eine Kugel schneidet, muss die Distanz zwischen dem Mittelpunkt der Kugel und einem Punkt auf der Geraden, die durch den Vektor gegeben ist, berechnet werden. Ist diese Distanz d größer als der Radius r der Kugel, dann existiert kein Schnittpunkt. Um diese Distanz d zu berechnen, muss zuerst der Abstand zwischen dem Mittelpunkt $center$ der Kugel und dem Urpsrung $origin$ des Vektors berechnet werden. Außerdem wird die Länge c des Vektors $\overrightarrow{OC}$ benötigt, um später mit dem Satz des Pythagoras die Distanz d zu berechnen.

$$\overrightarrow{OC} = center - origin \tag{5.1}$$

$$c = \mid \overrightarrow{OC} \mid \tag{5.2}$$

Danach wird mit Hilfe des Skalarproduktes zwischen $\overrightarrow{OC}$ und dem Richtungsvektor die orthogonale Projektion des Vektors $\overrightarrow{OC}$ auf die durch den Richtungsvektor v bestimmte Richtung berechnet.

$$a = \overrightarrow{OC} \bullet v \tag{5.3}$$

Durch die Anwendung des Satzes von Pythagoras kann jetzt die Distanz d berechnet werden.

$$\begin{aligned} a^2 + d^2 &= c^2 \\ \Rightarrow d &= \sqrt{c^2 - a^2} \end{aligned} \tag{5.4}$$

Falls diese Distanz d nun kleiner als der Radius r ist, so schneidet der Vektor die Kugel. Als Verbesserung kann die Distanz d direkt vom Radius r abgezogen werden. Dann existiert kein Schnittpunkt, falls das Ergebnis kleiner als 0 ist.

```
function intersect(object, camera, ray){
        let origin = camera.positionVector;
        let v = ray;
        let center = object.transformation.position;
        let r = object.transformation.size[0] * 0.75;
        let OC = vec3.create();
        vec3.subtract(Q, center, origin);

        let c = vec3.length(OC);
        let a = vec3.dot(OC, v);
        let d = r * r - (c* c- a * a);
        if (d < 0.0) {
            return false;
        }
return true;
}
```

Listing 5.8: Schnittpunkt eines Vektors mit einer Kugel

Das visuelle Feedback entsteht nun dadurch, dass eine nicht ausgefüllte Raute angezeigt wird. Allerdings ändert sich die Füllung, wenn der Anwender eine Geräuschquelle richtig lokalisiert hat. Abbildung 5.5 zeigt die Änderung der Raute.

(a) Richtige Lokalisierung

(b) Falsche Lokalisierung

Abbildung 5.5: Visuelle Umsetzung des Feedbacks

5.6 Log

Bei der Umsetzung des Logs wurde darauf verzichtet die Informationen über das Netzwerk zu senden, um mobile Geräte, die keine stabile Internetverbindung haben, nicht zu beeinträchtigen. Weiterhin ist die Anforderung bei der Aufzeichnung von Daten, dass dies in angemessener Zeit passieren soll [26]. Um das zu erreichen werden alle Informationen in einer lokalen Datenbank gespeichert. Dazu zählen die folgenden Informationen. Nachdem eine Szene geladen worden ist, fängt der *Log* an, die Zeit zu zählen. Interagiert der Benutzer über einen Klick mit der Anwendung, wird berechnet, ob der Nutzer damit eine Geräuschquelle lokalisiert hat. Falls ja, wird sowohl die Zeit, die er benötigt hat, um erfolgreich eine Geräuschquelle zu lokalisieren, als auch die Anzahl der bisherigen Erfolge gespeichert. Wird allerdings die Geräuschquelle nicht erfolgreich lokalisiert, so wird die Anzahl der Fehlversuche erhöht und die Zeit, die bis zu diesem Fehler vergangen ist, abgespeichert. Wird die Anzahl der Fehler und Erfolge verändert, so wird das Verhältnis

zwischen Erfolg und Fehler mit abgespeichert. Weiterhin wird auch die durchschnittliche Zeit, die bis zu einem Erfolg oder Fehler vergangen ist, aufgezeichnet. Eine weitere wichtige Information im Bezug auf das Erstellen von Statistiken ist der Abstand zwischen dem Punkt eines Fehlers und der tatsächlichen Position der Geräuschquelle. Für diese Information wurde ein k-d-Baum implementiert, der im Folgenden erklärt wird.

Ein k-d-Baum ist ein Binärbaum, mit dem Punkte in einem k-dimensionalen Raum gespeichert werden können [27]. Das heißt, dass jeder Knoten im Baum einen k-dimensionalen Punkt enthält. Im diesem Fall beinhaltet er den Mittelpunkt jedes *SoundObjects* im Szenengraphen. Um einen k-d-Baum zu erstellen, wird eine Liste von Punkten abwechselnd nach einer ihren Achsen sortiert. In diesem Fall sind das zuerst die x-Achse, dann die y-Achse und danach die z-Achse. Erreicht der Baum eine tiefere Ebene, dann wird wieder beginnend ab der x-Achse sortiert. In jedem rekursiven Durchlauf wird der Median der sortierten Liste berechnet. Der linke Teilbaum wird dann anhand der linken Teilhälfte der Liste erstellt, gleiches gilt für den rechten Teilbaum. Eine Implementierung findet sich in Quelltext 5.9. Der Vorteil von k-d-Bäumen besteht darin, dass bei Verwendung ebendieser das *nearest-neighbor*-Problem effizient gelöst werden kann.

```
1  KdNode.build = function (listPoints, dimension, currAxis) {
2      let median = Math.floor(listPoints.length / 2);
3      let currPosition = listPoints[median];
4      currAxis = currAxis % dimension;
5      if (listPoints.length == 0) {
6          return null;
7      }
8      if (listPoints.length == 1) {
9          return new KdNode(listPoints[0], null, null, dimension,
10         currAxis, true);
11     }
12     listPoints = KdNode.sortByAxis(listPoints, currAxis);
13     let left = KdNode.build(listPoints.slice(0, median),
14                             dimension, (currAxis + 1) % dimension);
15     let right = KdNode.build(listPoints.slice(median, listPoints.length),
16                             dimension, (currAxis + 1) % dimension);
```

```
17      return new KdNode(currPosition, left, right, currAxis, false);
18  };
```

Listing 5.9: Erstellung eines k-d-Baums

Ist der k-d-Baum initialisiert, so kann überprüft werden, welcher Punkt im Baum einem möglichen Punkt am nächsten ist. Der mögliche Punkt repräsentiert dabei den Punkt, der in Kapitel 5.5 unter Verwendung von Ray Picking erzeugt worden ist. Ziel ist es jetzt den Mittelpunkt des *Soundobjects* herauszufinden, der diesem Punkt am nächsten ist. Dabei wird beginnend ab der Wurzel des Baums so lange traversiert, bis ein Blatt gefunden worden ist. Nun wird die Distanz zwischen dem Punkt, der in diesem Blatt gespeichert ist, und dem Klick-Punkt berechnet. Danach wird diese Distanz und das Blatt als bisher bestes Ergebnis abgespeichert. Im anschließenden Rekursionsverlauf zurück zur Wurzel wird an jedem Knoten überprüft, ob die Distanz zum Klick-Punkt kleiner ist als die bisherige. Wenn ja, dann wird diese neue Distanz und der aktuelle Knoten als bestes Ergebnis gespeichert. Das Resultat ist dann der Mittelpunkt des *Soundobjects*, dass am nächsten am Klick-Punkt liegt. Mit dieser Information kann nun der Euklidische Abstand zwischen Klick-Punkt und *SoundObject* berechnet werden, so wie der Winkel zwischen diesen beiden Vektoren.

5.7 Realisierung

In diesem Kapitel wird die Realisierung der Applikation erläutert. Für eine geeignete Darstellung wurde das CSS-Framework *Bootstrap* aus Kapitel 5.1.3 verwendet. Wird die Anwendung gestartet, so gelangt der Nutzer in den Startbildschirm, wie es in Abbildung 5.6a zu sehen ist. Der Startbildschirm dient als Container, um eine *Application* zu starten.

Dazu wird über den *Load Scene*-Button ein Fenster geöffnet, in der bisher gespeicherte Szenen angezeigt werden. Aus dieser Liste kann nun eine Szene ausgewählt werden.

Abbildung 5.6b zeigt den Zustand der Applikation nachdem das Auswahlfenster für Szenen geöffnet worden ist. Ist bisher keine Szene gespeichert worden, so kann über den *Load*-Button eine Standardszene geladen werden. Anderenfalls wird die gewählte Szene

(a) Startbildschirm der Anwendung (b) Auswahl bisheriger Szenen

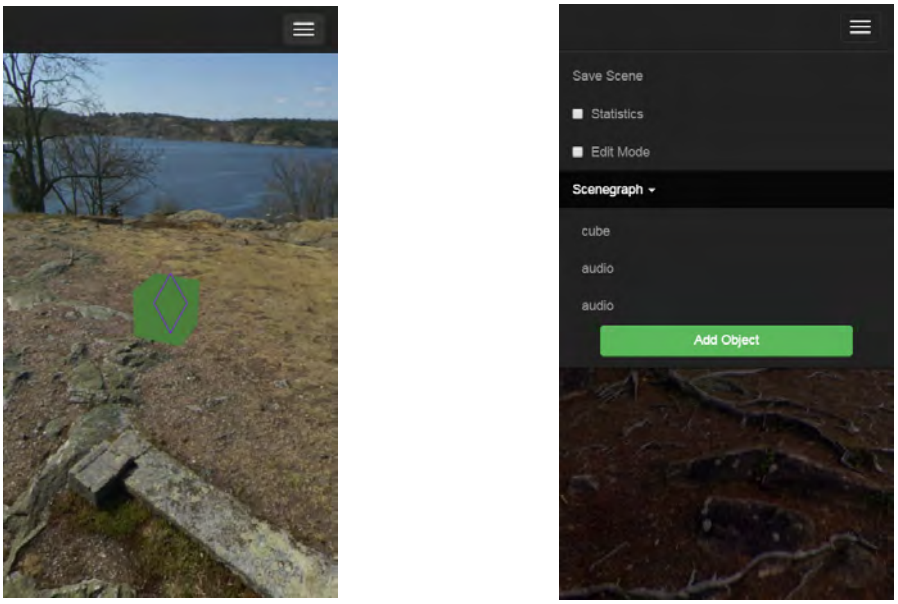
Abbildung 5.6: Abbildung 5.6a zeigt den Startbildschirm der mobilen Anwendung. Abbildung 5.6b zeigt das Auswahlfenster für Szenen

anhand ihres Namens aus der Datenbank gelesen. Alle in der Datenbank gespeicherten Szenen haben ein einheitliches Format, dass schon in Kapitel 4.2.2 vorgestellt worden ist. Mit dem zurückgelieferten JSON wird nun ein neues Szenenobjekt initialisiert und anschließend der *Application* übergeben. Die *Application* wird dann mit dieser Szene initialisiert und gestartet.

Im Hintergrund finden nun alle Initialisierungsprozesse statt, die in Kapitel 4.3 beschrieben sind. Dazu gehört das Initialisieren der Texturen und Sounds, sowie des Logs und der Render Loop. Ist der Initialisierungsprozess beendet, geht die *Application* in die *draw*-Phase über, das heißt, dass die Cubemap gerendert worden ist und der Benutzer nun mit der Anwendung interagieren kann, wie es in Abbildung 5.8a dargestellt ist.

Zunächst ist der Bearbeitungsmodus deaktiviert, um den Benutzer die Möglichkeit zu geben, sich akustisch zu orientieren. In diesem Modus sind die *SoundObjects* nicht sichtbar, werden allerdings schon vom Benutzer gehört. Wird der Bearbeitungsmodus dagegen im Menü aktiviert, werden alle Geräusche in Form eines Würfels visualisiert, wie es in Abbildung 5.7a zu sehen ist.

Im Menü hat der Benutzer die Möglichkeit sich den Szenengraphen anzeigen zu lassen. Der Szenengraph ist dabei eine Liste aus allen Szenenobjekten, die in der aktuellen Szene vorhanden sind. Dazu gehört genau ein *CubeObject* und beliebig viele *SoundObjects*. Grundsätzlich ist es immer möglich den Szenengraphen anzeigen zu lassen, dazu wird zuerst die Navigation geöffnet, wie in Abbildung 5.7b zu sehen ist.

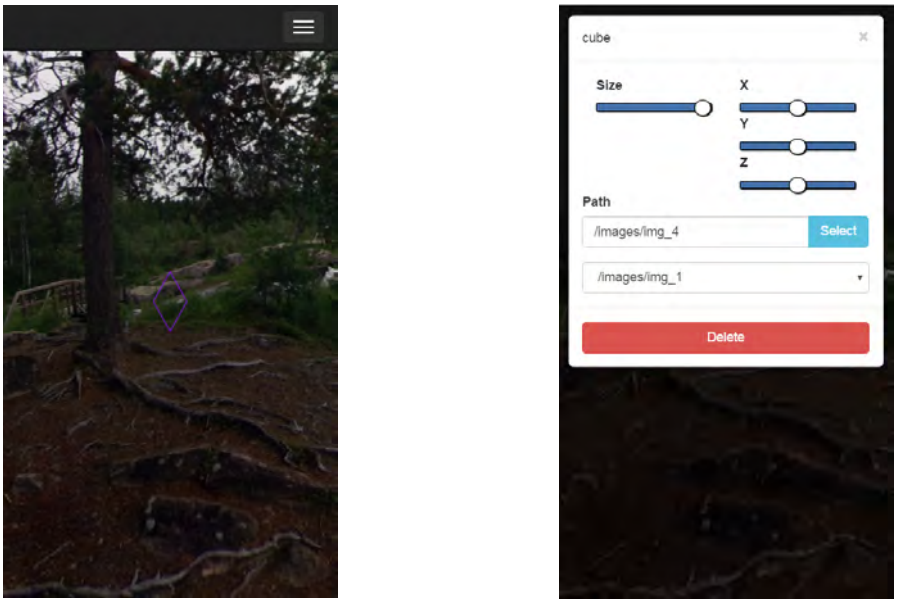


(a) Sichtbare Geräuschquellen (b) Anzeigefenster der Navigation

Abbildung 5.7: Abbildung 5.7a zeigt die visuelle Umsetzung der Geräuschquellen im Bearbeitungsmodus. Abbildung 5.7b zeigt die Navigation mit geöffnetem Szenengraphen

Des Weiteren ist es möglich, durch den *Add Object*-Button neue *SoundObjects* hinzuzufügen. Wenn das der Fall ist, wird die Liste mit den Szenenobjekten aktualisiert und der Szenengraph wird neu erstellt. Neu hinzugefügte Objekte werden im Ursprung positioniert und können dann verschoben werden. Wird dagegen ein einzelnes Objekt aus dem Szenengraphen ausgewählt, öffnet sich ein Bearbeitungsfenster.

Abbildung 5.8b zeigt den Aufbau dieses Bearbeitungsfensters. Hierin können die Größe und die Positionen auf den drei Achsen eingestellt werden. Werden Änderungen vorgenommen, so geschehen diese in Echtzeit, da die Anzeige für die Eigenschaften des Szenenobjektes direkt mit dem dahinterliegenden Objekt verknüpft ist.



(a) *Application* ist initialisiert (b) Bearbeitungsfenster für *CubeObjects*

Abbildung 5.8: Abbildung 5.8a zeigt die *Application* nachdem sie initialisiert worden ist. Abbildung 5.8b zeigt das Bearbeitungsfenster für die zwei unterschiedlichen Typen von *SceneObjects*

Weiterhin kann, je nachdem ob es sich um ein *SoundObject* oder ein *CubeObject* handelt, aus einer Liste mit möglichen Pfaden entweder eine Textur oder ein neuer Ton ausgewählt werden. Mit dem Bestätigen des *Select*-Buttons wird die neue Auswahl übernommen. Außerdem lassen sich alle Objekte mit einem Klick auf den *Delete*-Button aus der Szene löschen.

6

Anforderungsabgleich

Die in Kapitel 3 vorgestellten funktionalen und nichtfunktionalen Anforderungen an die Applikation sollen in diesem Kapitel anhand einer Tabelle nochmals aufgeführt und anschließend abgeglichen werden. Ziel dabei ist eine Gegenüberstellung zwischen Soll- und Ist-Zustand.

6.1 Funktionale Anforderungen

In diesem Kapitel werden die in Kapitel 3.1 vorgestellten funktionalen Anforderungen bezüglich ihrer Umsetzung in der Anwendung bewertet und anschließend erläutert. In der folgenden Tabelle 6.1 sind die einzelnen funktionalen Anforderungen aufgelistet und nach ihrer Umsetzung im System bewertet.

FA01 Dokumentdatenmodell für eine Szene

Alle Informationen, die für die Rekonstruktion der Szene notwendig sind, wurden in das Dokumentdatenmodell integriert, dazu gehören die Dokumentdatenmodelle für *CubeObjects*, *SoundObjects* und die Kamera.

FA02 Laden einer Szene aus der Datenbank

Durch das Erfüllen der funktionalen Anforderung 6.1 wird gewährleistet, dass Szenen korrekt aus der Datenbank gelesen werden können.

Nr.	Anforderung	Erfüllungsgrad
FA01	Dokumentdatenmodell für eine Szene	erfüllt
FA02	Laden einer Szene aus der Datenbank	erfüllt
FA03	Speichern einer Szene	erfüllt
FA04	Textur der Szene ändern	erfüllt
FA05	Neue Sounds der Szene hinzufügen	erfüllt
FA06	Vorhandene Sounds bearbeiten	erfüllt
FA07	Unterscheidung zwischen Bearbeitung und Benutzung	erfüllt
FA08	Kamerasteuerung	erfüllt
FA09	Szenengraph	erfüllt
FA010	Visuelles Feedback	erfüllt
FA011	Informationen über Nutzungsverhalten	erfüllt

Tabelle 6.1: Abgleich der funktionalen Anforderungen

FA03 Speichern einer Szene

Beim Speichern der Szene werden alle für die Rekonstruktion notwendigen Komponenten in das in Anforderung 6.1 entwickelte Dokumentdatenmodell überführt. Zusätzlich wird der Szene vor dem Speichervorgang einen Namen durch den Benutzer zugewiesen.

FA04 Textur der Szene ändern

Für die Änderung der Textur der Szene wurden Texturen in einem geeigneten Format und die dazugehörigen Pfade in die Anwendung integriert.

FA05 Neue Sounds der Szene hinzufügen

Der Benutzer hat die Möglichkeit über den Szenengraphen neue *SoundObjects* hinzuzufügen. Dargestellt wird diese Funktionalität über einen Button.

FA06 Vorhandene Sounds bearbeiten

Durch die Integration des Szenengraphen lassen sich alle Szenenobjekte und damit auch *SoundObjects* direkt adressieren. Dazu wurde eine GUI entworfen, die es dem Benutzer ermöglicht, die wichtigsten Parameter in Echtzeit zu ändern.

FA07 Unterscheidung zwischen Bearbeitung und Benutzung

Im Navigationsmenü wurde dem Benutzer die Möglichkeit gegeben den Bearbeitungsmodus entweder zu aktivieren oder zu deaktivieren. Bei Deaktivierung sind *SoundObjects* nicht mehr sichtbar.

FA08 Kamerasteuerung

Durch die Verwendung des Frameworks Hammer.js wurden *touch-listener* implementiert, die auf *pan* und *tap* reagieren. Dadurch ist die Kamerasteuerung auf mobilen Geräten intuitiver für den Benutzer geworden.

FA09 Szenengraph

Der Szenengraph listet alle vorhandenen Szeneobjekte auf und kann über eine GUI eingesehen werden. Diese wurde dynamisch gestaltet, dadurch lassen sich zur Laufzeit Objekte hinzufügen oder entfernen.

FA010 Visuelles Feedback

Visuelles Feedback wurde auf der Basis von WebGL und Ray Picking implementiert. Trotz aufwändiger Berechnungen erfolgt das Feedback in angemessener Zeit.

FA011 Informationen über Nutzungsverhalten

Informationen über das Nutzungsverhalten werden bei jedem Event, dass die Anwendung zum Pausieren bringt, in einer lokalen Datenbank gespeichert.

6.2 Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen beschreiben die technischen Merkmale des Systems. Tabelle 6.2 listet die in Kapitel 3.2 aufgezeigten nichtfunktionalen Anforderungen auf und bewertet diese bezüglich der Einhaltung. Im Anschluss werden die einzelnen Anforderungen und deren Aspekte genauer erläutert.

Nr.	Anforderung	Erfüllungsgrad
NFA01	Funktionalität	erfüllt
NFA02	Benutzerfreundlichkeit	teilweise erfüllt
NFA03	Zuverlässigkeit	erfüllt
NFA04	Effizienz	teilweise erfüllt
NFA05	Wartbarkeit	erfüllt
NFA06	Wiederverwendbarkeit	erfüllt

Tabelle 6.2: Abgleich der nichtfunktionalen Anforderungen

NFA01 Funktionalität

Die Anwendung erfüllt alle vorgegeben funktionalen Anforderungen.

NFA02 Benutzerfreundlichkeit

Die GUI der Anwendung wurde vollständig mit Hilfe des Front-End Frameworks *Bootstrap* entwickelt. Zusätzlich ist die Oberfläche minimalistisch gehalten worden, um den Benutzer nicht durch überflüssige Komponenten zu beeinträchtigen.

NFA03 Zuverlässigkeit

Die Anwendung stellt Methoden bereit, um Fehlereingaben durch den Benutzer zu minimieren. Dazu gehört das Bereitstellen von informativen Fehlermeldungen.

NFA04 Effizienz

Die Anwendung ist durch das Verwenden von WebGL prädestiniert für einen hohen Ressourcenverbrauch. Gerade Matrixoperationen oder das Laden von Texturen gestalten sich oft problematisch. Allerdings wurden durch das Framework *glMatrix.js* kostspielige Matrixoperationen optimiert und durch das Einschränken der Texturen auf eine maximale Höhe und Breite von jeweils 512 Pixeln reagiert die Anwendung in einer angemessenen Zeit.

NFA05 Wartbarkeit

Bei der Entwicklung der Anwendung wurde auf eine hohe Abstraktion geachtet. Dadurch lassen sich eventuelle Fehler direkt erkennen und bieten Spielraum für Verbesserungen.

NFA06 Wiederverwendbarkeit

Weiterhin wurde bei der Entwicklung auf eine hohe Modularität geachtet, dadurch lassen sich einzelne Komponenten auch in anderen Arbeiten weiter verwenden.

7

Zusammenfassung & Ausblick

7.1 Zusammenfassung

Bei der Entwicklung von mobilen Anwendungen spielt die Wahl der Plattform eine entscheidende Rolle. Sie gibt vor, in welcher Programmiersprache Anwendungen entwickelt werden und welche Eigenheiten jeweils zu beachten sind. Eine mögliche Lösung der Frage nach der richtigen Plattform liegt in der Verwendung eines plattformunabhängigen Frameworks, wie es in dieser Arbeit genutzt wird. Durch eine präzise Konzeption wurden die Anforderungen an die mobile Anwendung erfasst, analysiert und definiert. Dadurch war es möglich, den Umfang und die Funktionalität der Applikation einzugrenzen und bei der Entwicklung der Architektur auf einen modularen Aufbau zu achten. Außerdem ist es durch eine hohe Wiederverwendbarkeit und Wartbarkeit möglich, die Anwendung ohne Einschränkungen zu erweitern oder einzelne Komponenten für andere Arbeiten wiederzuverwenden. Im Allgemeinen stellt die Applikation eine Lösung für die Plattformen Android, iOs und Web bereit, um die Lokalisationsfähigkeit von Geräuschquellen im dreidimensionalen Raum zu untersuchen.

7.2 Kritikpunkte

Im Verlauf der Implementierung sind Punkte zustande gekommen, die für zukünftige Erweiterungen Optimierungen darstellen.

7.2.1 Quadtrees

Quadtrees sind eine Baumstruktur, bei der jeder innere Knoten vier Kindknoten hat. Sie werden hauptsächlich zur Unterteilung von zweidimensionalen Regionen eingesetzt, indem diese Regionen rekursiv immer weiter in vier Abschnitte unterteilt werden. Bezogen auf die Darstellung der Panoramabilder, können Texturen durch Quadtrees zerlegt werden, dass nur noch die Teile der Textur angezeigt werden, die auch im Sichtfeld liegen und die, die außerhalb des Sichtvolumens liegen, nicht mehr gerendert werden. Eine ausführliche Erklärung findet sich unter [27] und [28].

7.2.2 Verbesserung der Kamersteuerung

Die Kamerasteuerung entsteht durch die Touch-Events *panstart*, *panmove*, *panend*. Allerdings wird der vertikale *pan* im *panmove*-Event nicht aktiviert, was sich auf das bisherige *scroll*-Event zurückführen lässt.

7.2.3 Speicherung der Log-Daten

Log-Daten werden nur Lokal gespeichert, da eine permanente Übertragung dieser Daten einen hohen Verbrauch zur Folge hätte. Zukünftigen Arbeiten ist daher zu überlassen, wie diese Informationen ausgewertet werden.

7.3 Ausblick

In diesem Kapitel werden zukünftige Erweiterung oder Verbesserung im Hinblick auf die Anwendung vorgestellt. Ziel dieser Arbeit war die Entwicklung einer mobilen Anwendung zur Untersuchung und Verbesserung der Lokalisierungsfähigkeiten von Geräuschquellen im dreidimensionalen Raum. Die Anwendung kann auf den Plattformen Android, iOs oder aber auch als Webanwendung angeboten werden. Bisher beruht die Lokalisierung von Geräuschquellen über einen manuellen Input durch den Benutzer, dem eine 3D Szene vorliegt. Eine vielversprechende Möglichkeit durch die 3D Szene zu navigieren

bietet sich durch das Einsetzen der WebVR API, welche zur Zeit nur in der Chrome Beta(M56+), Firefox Nightly und dem Samsung Internet Explorer für Gear VR verfügbar ist. Allerdings existieren auch Schnittstellen in Browsern, die kein WebVR unterstützen. Diese sind allerdings sehr kostenintensiv und daher noch nicht zu empfehlen. Eine weitere Alternative bietet auch Augmented Reality, bei der die Szene nicht in Form von Bildern erzeugt wird, sondern durch die wirkliche Umgebung [29].
Weiterhin soll eine zusätzliche Plattform entwickelt werden, auf der Texturen und Töne gespeichert werden können, welche der Anwender für diese Applikation nutzen kann. Außerdem bietet sich für diese Plattform an, dass sie Metainformationen, die bisher bei der Verwendung der Applikation gespeichert werden, empfangen und sinnvoll verarbeiten kann.

Literaturverzeichnis

[1] Schickler, M., Reichert, M., Pryss, R., Schobel, J., Schlee, W., Langguth, B.: Entwicklung mobiler Apps: Konzepte, Anwendungsbausteine und Werkzeuge im Business und E-Health. eXamen.press. Springer Vieweg (2015)

[2] Schickler, M., Pryss, R., Reichert, M., Schobel, J., Langguth, B., Schlee, W.: Using mobile serious games in the context of chronic disorders - a mobile game concept for the treatment of tinnitus. In: 29th IEEE Int'l Symposium on Computer-Based Medical Systems (CBMS 2016). (2016)

[3] McCallum, S.: Gamification and serious games for personalized health. Stud Health Technol Inform **177** (2012)

[4] Pryss, R., Reichert, M., Langguth, B., Schlee, W.: Mobile crowd sensing services for tinnitus assessment, therapy, and research. In: Mobile Services (MS), 2015 IEEE International Conference on, IEEE (2015)

[5] Statista: Statista, Anzahl der Apps in den Top App-Stores. `https://de.statista.com/statistik/daten/studie/208599/umfrage/anzahl-der-apps-in-den-top-app-stores/` (2017) Zugriff: 31.01.2017.

[6] Meteor Development Group: Meteor.js. `http://docs.meteor.com/` (2016) Zugriff: 12.01.2017.

[7] MongoDB, Inc: mongoDB. `https://www.mongodb.com/de` (2017) Zugriff: 12.01.2017.

[8] Douglas Crockford: JSON. (`http://www.json.org/json-de.html`) Zugriff: 12.01.2017.

[9] Meteor Development Group: Meteor.js. `https://guide.meteor.com/mobile.html` (2016) Zugriff: 12.01.2017.

[10] The Apache Software Foundation: Cordova Architecture. (`https://cordova.apache.org/docs/en/latest/guide/overview/index.html`) Zugriff: 12.01.2017.

[11] Mozilla Developer Network and individual contributors: Mozilla, Kamera API. https://developer.mozilla.org/en-US/docs/Mozilla/B2G_OS/API/Camera_API/Introduction (2017) Zugriff: 12.01.2017.

[12] Blaze: Blaze Template. (http://blazejs.org/index.html) Zugriff: 31.01.2017.

[13] Brandon Jones: glMatrix. http://glmatrix.net/docs/mat4.html (2016) Zugriff: 12.01.2017.

[14] Shirley, P., Ashikhmin, M., Marschner, S.: Fundamentals of computer graphics. CRC Press (2015)

[15] Brandon Jones: glMatrix. http://glmatrix.net/docs (2016) Zugriff: 12.01.2017.

[16] Mozilla Developer Network and individual contributors: Mozilla, WebGL. https://developer.mozilla.org/en/docs/Web/API/WebGL_API (2017) Zugriff: 12.01.2017.

[17] Matsuda, K., Lea, R.: WebGL programming guide: interactive 3D graphics programming with WebGL. Addison-Wesley (2013)

[18] Mozilla Developer Network and individual contributors: Web Audio API. https://developer.mozilla.org/en/docs/Web/API/Web_Audio_API (2016) Zugriff: 12.01.2017.

[19] Mozilla Developer Network and individual contributors: Web Audio API. https://mdn.mozillademos.org/files/7893/web-audio-api-flowchart.png (2016) Zugriff: 13.01.2017.

[20] Mozilla Developer Network and individual contributors: Web Audio API. https://developer.mozilla.org/en-US/docs/Web/API/AudioContext/decodeAudioData (2016) Zugriff: 13.01.2017.

[21] Mozilla Developer Network and individual contributors: Web Audio API. https://mdn.mozillademos.org/files/9717/WebAudioAudioBufferSourceNode.png (2016) Zugriff: 13.01.2017.

[22] Mozilla Developer Network and individual contributors: Web Audio API. `https://mdn.mozillademos.org/files/14311/WebAudioListenerReduced.png` (2016) Zugriff: 13.01.2017.

[23] Network, M.D., individual contributors: Web audio api. `https://mdn.mozillademos.org/files/13815/WebAudioPannerNode.png` (2016) Zugriff: 13.01.2017.

[24] Alex: The Bored Engineer: The quadcopter - control the orientation. `http://theboredengineers.com/2012/05/the-quadcopter-basics/` (2012) Zugriff: 31.01.2017.

[25] Moreto, S.: Bootstrap By Example. Packt Publishing (2016)

[26] Schobel, J., Pryss, R., Schickler, M., Reichert, M.: A lightweight process engine for enabling advanced mobile applications. In: 24th International Conference on Cooperative Information Systems (CoopIS 2016). Number 10033 in LNCS, Springer (2016)

[27] Samet, H.: The quadtree and related hierarchical data structures. ACM Computing Surveys (CSUR) **16** (1984)

[28] Samet, H.: The design and analysis of spatial data structures. Volume 199. Addison-Wesley Reading, MA (1990)

[29] Pryss, R., Geiger, P., Schickler, M., Schobel, J., Reichert, M.: Advanced algorithms for location-based smart mobile augmented reality applications. Procedia Computer Science **94** (2016)

Abbildungsverzeichnis

Tabellenverzeichnis

Name: Felix Rottler Matrikelnummer: 807838

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Felix Rottler