AUDIO IM BROWSER

Implementierung eines digitalen Klangerzeugers mit der Web Audio API

Bachelorarbeit

Hochschule der Medien
Studiengang: Audiovisuelle Medien
Wintersemester 2020/21
Erstprüfer: **Prof. Oliver Curdt**
Zweitprüfer: **Prof. Michael Felten**

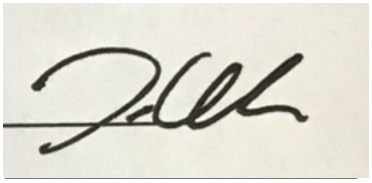
vorgelegt von
Jakob Getz
Matrikelnummer: 35681
Pfaffenwaldring 76-C | 70569 Stuttgart
jakob@getz.de

18. Februar 2021

ERKLÄRUNG

“Hiermit versichere ich, Jakob Getz, ehrenwörtlich, dass ich die vorliegende Arbeit mit dem Titel: ***Audio im Browser - Implementieren eines digitalen Klangerzeugers mit der Web Audio API*** selbstständig und ohne fremde Hilfe verfasst und keine anderen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester, § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. 4 19 Abs. 2Master-SPO (4 Semester und berufsbegleitend) der HdM einer unrichtigen oder unvollständigen Versicherung zu Kenntnis genommen.”



Jakob Getz
Suttgart, 18. Februar 2021

ZUSAMMENFASSUNG

Inhalt der Arbeit sind die theoretischen Grundlagen eines digitalen Klangerzeugers und dessen Implementierung. Der Fokus liegt hierbei auf dem Oszillator, welcher das zentrale Modul eines klassischen Synthesizers darstellt. Die Programmierung wird anhand praktischer Beispiele erläutert, sodass der Leser in der Lage ist, die Schritte nachzuvollziehen und zu reproduzieren, um so ein tiefgreifendes Verständnis digitaler Klangsynthese zu erlangen.
In den letzten Jahren ist die Nachfrage nach Möglichkeiten Audio im Web zu erstellen und zu bearbeiten, stark gestiegen, was Produkte wie *Amped Studio* und *Soundtrap* belegen. Somit beschäftigt sich diese Arbeit explizit mit der Implementierung für den Browser.
Anhand des durchgeführten Projektes werden verschiedene Technologien der Audio Programmierung vorgestellt und untersucht, welche von modernen Browsern unterstützt werden. Somit soll die Frage beantwortet werden, inwieweit das Web in der Lage ist Equipment für die professionelle Audioproduktion zur Verfügung zu stellen.

Topic of this thesis are the theoretical basics of a digital sound generator and its implementation. In this case the focus lies on the oscillator, which is the central module of a classic synthesizer. The programming is explained with practical examples, to enable the reader to comprehend and reproduce every step in the process for gaining a profound understanding of digital sound synthesis.
In the last couple of years the demand to generate and edit audio in the Web has increased significantly, which is verified by products like *Amped Studio* and *Soundtrap*. Consequently this thesis deals with the implementation for the browser.
Different technologies of audio programming which are supported by modern browsers will be presented and examined based on the carried out project. This should answer the question to what extend the web is able to provide equipment for the professional audio production.

INHALT

1 EINLEITUNG

1.1 Geschichte des künstlich erzeugten Klanges

Es ist nicht genau zu sagen, wann und wo die Musik ihren Anfang nahm. Eine Flöte, bestehend aus einem Knochen eines jungen Bären, welche in Divje Baba, Slowenien gefunden wurde, schätzen Forscher auf ein Alter von 60 000 Jahren.[1] Weitere Funde belegen Musikinstrumente in Deutschland, Vietnam und China von vor 35 000 und 10 000 Jahren.[2] Dass Klangerzeuger jedoch schon deutlich früher eine wichtige Rolle in der Kultur des modernen Menschen spielten, ist nicht auszuschließen. Somit liegt die Vermutung nahe, dass Musik und damit die Herstellung von Instrumenten auf vielen verschiedenen Orten der Welt zu unterschiedlichen Zeiten simultan entstanden sein muss.

Im Laufe der Jahrtausende erweiterte der Mensch sein Repertoire an Instrumenten stetig. Nachdem zunächst nur einfache Klänge und Rhythmen mit Flöten und Trommeln erzeugt werden konnten, wuchs der Umfangreichtum bis hin zu großen Orchestern an, bestehend aus eine Vielzahl kleiner, fein aufeinander abgestimmter Instrumente, welche im Einheitlichen Klänge aller Frequenzspektren erzeugen konnten.

Alle diese Klangerzeuger haben jedoch eines gemeinsam: Sie basieren auf den physikalischen Eigenschaften eines Materials, welches, von einem Menschen mehr oder weniger direkt auf eine bestimmte Art und Weise zum Schwingen gebracht wird, wodurch sich als Folge dessen ein dem entsprechenden Instrument einzigartiger Klang ergibt. Der Klang ist jedoch nicht nur einzigartig, er ist auch statisch mit dem Instrument verknüpft. So lange keine Veränderungen an Form und Material des Objektes stattfinden, wird eine Violine immer den ihr typischen Klang von sich geben.

Mitte des 20. Jahrhunderts gab es erste Versuche diese fundamentalen Eigenschaften zu durchbrechen. Der erste spiel- und konfigurierbare Synthesizer wurde von Robert Moog 1964

[1] vgl. Narodni Muzej Slovenje

[2] vgl. Bläsig

vorgestellt.[3] Anders als bei herkömmlichen Klangerzeugern basierte dieses Instrument auf einer elektronischen Schaltung, deren Zustand mit Hilfe von Potentiometern und Schaltern verändert werden konnte und somit in der Lage war eine Vielzahl von unterschiedlichen Geräuschen zu erzeugen. Zwar gelang es mithilfe dieser neuen Technologie nicht alle vorstellbaren Klänge zur Genüge darzustellen und herkömmliche Instrumente zu ersetzen, jedoch erweiterte diese Erfindung den Pool der Klangerzeuger um ein weiteres Werkzeug.
Mit der Digitalisierung Ende des 20. Jahrhunderts verschob sich die analoge, elektronische Entwicklung des Synthesizers hin zur programmseitigen Implementierung am Computer. Die Möglichkeiten, welche sich hieraus ergaben, waren eine leistungsfähigere Berechnung von Audiosignalen und somit die Erzeugung von einer noch größeren Vielzahl von Klängen. Ein bekanntes Beispiel dieser Anfänge ist der 1983 erschienene Yamaha DX7, welcher sich großer Beliebtheit erfreute und somit seine Anwendung in zahlreichen Musikstücken dieser Zeit fand.[4] [5]

1.2 Audio im Web

Trotz der weiten Verbreitung von digitalen Klängen in populärer Musik finden Softwaresynthesizer meist Anwendung im professionellen Bereich. Häufig kommen sie im Plugin-Format vor und sind somit fest gekoppelt an eine DAW (Digital Audio Workstation), welche ihrerseits ein professionelles Werkzeug für ausgebildete Toningenieure darstellt. Für die Allgemeinheit zugängliche Anwendungen zur Klangerzeugung existieren bisher nur wenige. Eine Ausnahme hiervon ist das OpenSource Projekt *Audiokit Synth One* welcher die Möglichkeiten umfangreichen Sounddesigns einer breiten Masse zugänglich und dabei genug Funktionalität für professionelle Produktionen liefern soll.[6] *Synth One* ist als App für das Betriebssystem IOS entwickelt.
Ein weiterer Nachteil, welcher sich durch die Nutzung einer festinstallierten Software ergibt, ist der Mangel an Flexibilität bei wechselnder Arbeitsumgebung und Hardware. Auch heute können eingerichtete Setups nicht ohne erheblichen Aufwand auf andere Rechner übertragen werden, zudem sind Programme oft nur mit einem bestimmten Betriebssystem nutzbar. Eine Lösung hierfür könnte die Implementierung von Audiotools im Web sein. Anwendungen, welche im Browser ausgeführt werden, laufen bei entsprechender Leistungsstärke des

[3] vgl. Audio Engineering Society

[4] vgl. Rogozinsky & Goryachev S.1

[5] siehe: https://de.yamaha.com/de/products/contents/music_production/synth_chronology/modal/modal_dx7.html

[6] siehe: https://audiokitpro.com/synth/

Computers auf allen üblichen Betriebssystemen. Des Weiteren wäre es durch eine einfache Nutzerauthentifizierung möglich, den eigenen persönlichen Arbeitsbereich auf allen verwendeten Geräten jederzeit abzurufen und wieder herzustellen.
In der nachfolgenden Arbeit soll die Frage beantwortet werden, ob und inwieweit die im Web zur Verfügung stehenden Technologien in der Lage sind, Klangerzeuger zu implementieren und auszuführen. Um eine hinreichende Beurteilung zu ermöglichen, wird ein Klangerzeuger in Typescript, eine auf dem ECMAScript Standart beruhende Programmiersprache[7], entwickelt.

1.3 Ziel der Arbeit

Die Arbeit richtet sich an all jene, die Interesse an einer Workstation übergreifenden Audioproduktion zeigen. Für Toningenieure und Musikproduzenten können die vorgestellten Technologien zeigen, welche Möglichkeiten sich in der Audioproduktion der Zukunft ergeben können. Darüber hinaus soll die Implementierung der nachfolgend beschriebenen Anwendung dazu beitragen, Lesern ein tiefgreifendes Verständnis der Algorithmik und Mathematik hinter digitaler Klangsynthese zu vermitteln. Dieses Wissen kann nicht nur für die Verwirklichung eigener Applikationen verwendet werden, sondern auch zu einer gesteigerten Produktivität und realistischeren Einschätzung von Klängen während des Arbeitsprozesses in Sounddesign und Mischung führen.
Informatikern kann dieser Text einen Einstieg in die Welt der Audio Programmierung bieten und sie für die Möglichkeiten zur Gestaltung innovativer Software für das Web sensibilisieren.

1.4 Anmerkungen

Um das praktische Vorgehen im nachfolgenden Projekt verstehen zu können, sind einige Grundkenntnisse erforderlich. Alle Quelltext-Beispiele sind in *JavaScript* oder *TypeScript*[8] geschrieben, was Erfahrung nicht nur in diesen spezifischen Sprachen, sondern auch ein gutes Verständnis von *HTML5* voraussetzt. User Interfaces nutzen *React*, eine auf *HTML5* und *Javascript* basierende Library welche ihre Anwendung in der Frontend-Entwicklung für

[7] vgl. Bierman & Abadi & Torgersen S.1

[8] siehe: https://www.typescriptlang.org

Webapplikationen findet. Zustandsdaten werden an einem zentralen Ort gespeichert, welcher mithilfe der Library *Redux* organisiert wird.

Grundlagen des digitalen Audios sind nicht Bestandteil dieser Arbeit, wodurch die Wandlung von analogen zu digitalen Signalen und deren Speicherung bereits bekannt sein sollte. Dazu gehören Begriffe wie Sampling und das Nyquist-Shannon-Abtasttheorem wie auch Quantisierung. Audiodaten werden in der digitalen Welt im Gegensatz zur analogen in zeit- und wertdiskreten Samples dargestellt, welche das eigentlich kontinuierliche Signal immer nur mit einer gewissen Ungenauigkeit darstellen können.

Die im Folgenden beschriebene Anwendung kann unter *www.jakobgetz.com/digital-soundsynthesis/oscillator* aufgerufen und genutzt werden, für eine reibungslose Laufzeit empfiehlt sich hierfür der Browser Google Chrome. Der Quelltext befindet sich auf GitHub, abzurufen unter: *https://github.com/jakobgetz/digital-soundsynthesis*

Reduzierte Codebeispiele zur Verdeutlichung der Algorithmik sind als Abbildungen in den entsprechenden Kapiteln der Arbeit enthalten.

Zum weiteren Selbststudium wird empfohlen, die vom Hersteller der verwendeten Technologien bereitgestellten Dokumentationen auf der Website deren offiziellen Internetpräsenz zu lesen und nachzuvollziehen. Im Anhang auf Seite 62 befindet sich eine Sammlung von Links zu verschiedenen Bildungsresourcen aus dem Internet, die bei einem einfachen Einstieg in die Web-Entwicklung und Audio-Programmierung helfen sollen.

2 SYNTHESIZER

2.1 Definition

Um das Thema weiter eingrenzen zu können, muss die genaue Bedeutung eines Klangerzeugers zunächst definiert werden. Als Klangerzeuger kann im Prinzip alles gelten, was durch ein bestimmtes Ereignis Schwingungen des Luftdrucks erzeugt, welche im für den Menschen hörbaren Bereich liegen. Im allgemeinen sind dies Schwingungen mit einer Frequenz von 16 bis 20 000 Hz[9] und einer Druckänderung von 20 μPa 20 Pa.[10] Dies können zum Beispiel Instrumente sein, der Ruf eines Tieres oder ein Regentropfen der auf den Boden fällt. Da es in dieser Arbeit jedoch ausschließlich um den künstlich erzeugten Klang geht, ist diese Definition nicht ausreichend. Ein Gerät, welches eben dieses Kriterium erfüllt, ist der Synthesizer. Hier kann die Art der erzeugten Klänge frei bestimmt werden, indem Veränderungen an den Einstellungen des Klangerzeugers selbst vorgenommen werden.[11] Ergo: Hier berechnet eine analoge oder digitale Einheit ein bestimmtes Audiosignal anhand unterschiedlicher, verstellbarer Parameter. Diese beschriebenen Eigenschaften treffen auf einen sogenannten *echten Synthesizer* zu und wird die Definition eines Synthesizers sein, welche als Grundlage für den folgenden Text dient. Aus diesem Grund ist es hier wichtig, den Synthesizer nicht mit seinem nahem Verwandten, dem Sampler, zu verwechseln. Bei einem Sampler werden lediglich voraufgenommene und gespeicherte Signale auf bestimmte Befehle hin wieder abgespielt, was die Möglichkeit eröffnet, Klänge akustischer Instrumente nah am Original wieder zu geben.[12]

[9] vgl. Webers S. 96

[10] vgl. Webers S. 102

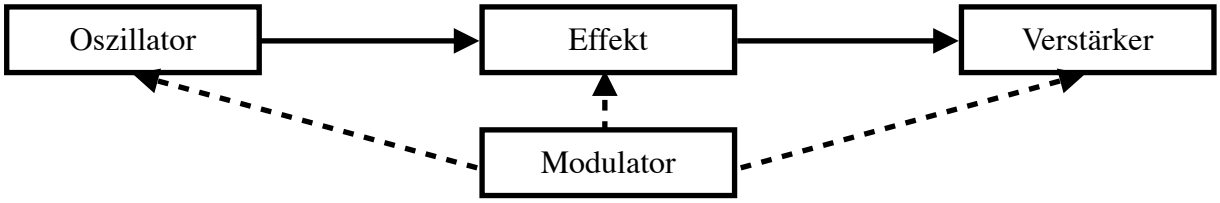
[11] vgl. Anwander S. 178

[12] vgl. Stange-Elbe S. 209

2.2 Aufbau

Die mittlerweile am häufigsten vertretene Form des Synthesizers ist der modulare Synthesizer. Wie schon der Name besagt, besteht ein solches Gerät aus mehreren zusammenspielenden Modulen, welche auf unterschiedliche Art und Weise miteinander verbunden werden können. Die Reihenfolge und Verschränkung, in welcher die einzelnen Einheiten miteinander verbunden sind, machen so die Unterschiede zwischen verschiedenen Modellen aus. Auf diesem Prinzip beruhen fast alle heutigen Synthesizer, analog wie digital. Ein modularer Synthesizer hat in der klassischen Definition drei Arten von Modulen, die in folgender Reihenfolge miteinander verschaltet sind.[13]

Selbst in der analogen Welt ist dieses System jedoch unzureichend, um den grundlegenden Aufbau eines Synthesizers zu veranschaulichen. Erst wenn man den Modultyp “Filter” durch “Effekt” ersetzt und das Modell um ein weiteres, der geschalteten Reihenfolge unabhängiges Modul, den Modulator erweitert, erhält man ein vollständiges Bild.



2.2.1 Oszillator

Der Oszillator ist die Signalquelle eines Synthesizers und sozusagen der eigentliche Klangerzeuger. Er generiert einen Ton, welcher als Ausgangsbasis für alle weitern Arbeitsschritte verwendet wird. Somit kann dieses Modul schon alleine als vollwertiger Synthesizer gelten. Je nach Ausführung bietet der Oszillator an sich bereits viele verschiedene Möglichkeiten der Klangbeeinflussung. Zunächst muss ein Ausgangssignal gewählt oder erstellt werden. Dies kann entweder eine der verfügbaren elementaren Wellenformen sein oder andere künstlich berechnete Funktionen. In der analogen Welt standen als periodische Schwingungen ursprünglich nur vier dieser elementaren Wellenformen zur Verfügung: Dreieck, Sägezahn, Rechteck und Pulswelle.[14] Erst später mit der Weiterentwicklung der Technologie konnte dieses Repertoire um den Sinus erweitert werden.

---

[13] vgl Stange-Elbe S. 197

[14] vgl. Ackermann S. 28

Das Kopieren und Neuverrechnen bereits vorhandener Signale ist ebenso vorstellbar, um ein bestimmtes Ergebnis zu erhalten.[15] Heute können die Funktionen von Oszillatoren sehr umfangreich sein und somit auch die Anzahl an verschiedenen Signalen, die erzeugt werden können. Auch Samples können als Ausgangsbasis für das vom Oszillator generierte Signal dienen. Neben periodischen Signalen kann der Sonderfall "Rauschgenerator" auch aperiodische Signale erzeugen. Zu diesen gehören beispielsweise White- und Pink-Noise.[16]
Je nach Art des Oszillators bieten sich zusätzliche Möglichkeiten der Ergebnisbeeinflussung. Nutzt ein solcher Klangerzeuger beispielsweise das Prinzip der Wavetable-Synthese, so kann dieser nicht nur eine statische Wellenform ausgeben, sondern eine ganze Liste an unterschiedlichen Signalen, deren exaktes Element vom Nutzer durch den Parameter "Wavetable-Position" gewählt wird. Hierdurch kann eine starke Klangverformung erreicht werden, die nicht in der Effektebene sondern schon innerhalb des Oszillators geschieht.[17]
Den Möglichkeiten dieses Moduls sind somit in der Klangerzeugung nur wenige Grenzen gesetzt.

2.2.2 Effekt

Nachdem ein Signal die Quelle verlässt, kann es durch einen Effekt geleitet werden. In der Welt der subtraktiven Synthese war dies ein Filter. Dieser hatte die Möglichkeit Frequenzen aus einem obertonreichen Signal herauszunehmen oder zu verstärken, was eine Veränderung der Klangfarbe mit sich führte. Somit konnte der Klang beeinflusst und neue Töne erzeugt werden.
Verschiedene Filter haben unterschiedliche Auswirkungen auf das Ausgangssignal. Unter anderem lassen sich folgende Arten voneinander unterscheiden:

Tiefpass
Dieser Filter senkt die Amplitude aller Frequenzen oberhalb einer bestimmten Cutoff Frequenz. Durch diesen Eingriff ergibt sich ein dumpfes Klangbild.

[15] vgl. Stange-Elbe S. 197
[16] vgl. Ackermann S. 28
[17] vgl. Stange-Elbe S. 201

Hochpass

Der Hochpass ist das Gegenteil eines Tiefpass-Filters. Hier werden alle Frequenzen unterhalb einer bestimmten Cutoff-Frequenz abgesenkt. Als Resultat ergibt sich ein hell und dünn klingendes Klangbild.

Bandpass

Dieser Filter vereinigt die Funktionen von Tiefpass und Hochpass wodurch nur noch Frequenzen unmittelbar um die Cutoff-Frequenz herum durchgelassen werden.

Bandsperre

Die Bandsperre ist wiederum das Gegenteil eines Bandpasses. Alle Frequenzen, bis auf die um die Cutoff-Frequenz herum, werden unverändert durchgelassen.[18]

Aus diesen Filtertypen lassen sich noch eine ganze Menge weiterer Formen zusammensetzen. Ein bekanntes Beispiel und häufig bei experimentellem Sounddesign verwendet, ist der Kammfilter. Auch gibt es eine Reihe von Effekten, welche auf der Basis von Filtern imitiert werden können, so zum Beispiel Phaser oder Flanger[19].

Da die subtraktive Synthese seit den 70er Jahren[20] lange nicht mehr die einzige Art der Klangerzeugung ist, traten an die Stelle des Filters zusätzlich auch andere Effekte. Zu diesen gehören Effekte wie Echo (Delay), Hall (Reverb) und Verzerrung (Distortion).

Während Delay und Reverb durch bereits vorhandene Funktionen des Filters umgesetzt werden können, bringt Distortion neue Arbeitsweisen mit sich, welche mit einfachen Filtern nicht darstellbar wären. So etablierte sich beispielsweise die Syntheseform der Phase-Distortion, welche das Ausgangssignal auf eine nur schwer vorhersehbare Weise verändert und in der CZ-Serie der Firma Casio Anwendung fand.[21]

2.2.3 Verstärker

Der Verstärker müsste eigentlich “Verschwächer” heißen, multipliziert er doch das Eingangssignal mit einem Wert im Wertebereich zwischen 0 und 1. Er entscheidet somit ob ein

[18] Dutilleux & Holgers & Disch & Zölzer S. 48

[19] vgl. Smith “Phaser”

[20] vgl. Ackermann S. 27

[21] vgl. Stange-Elbe S. 203

Signal durchgelassen werden soll oder nicht. Der Oszillator selbst erzeugt die gesamte Zeit einen Ton, welcher ohne einen Verstärker nicht an oder abgeschaltet werden könnte.

2.2.4 Modulator

Der Modulator ist ein Sonderfall, der sich nicht in die Einheit der drei oben beschriebenen Module eingliedert. Anders als ein Oszillator, welcher einen Klang ausgibt und Effekte wie auch Verstärker, welche Audiosignale erhalten und weitergeben, kommt ein Modulator im Regelfall nicht mit dem Audiosignal in Verbindung.[22] Auch er sendet jedoch Signale aus, die aber nicht als Audio gehandhabt werden sondern als Steuersignal. Dieses Steuersignal kann je nach Modell mit beliebigen Parametern der anderen Module verbunden werden und somit in die Klangerzeugung eingreifen. Es gibt verschiedene Arten von Modulatoren, weit verbreitet jedoch sind Folgende:

Envelope Generator

Ein Envelope Generator (zu Deutsch: Hüllkurvengenerator) erzeugt eine Funktion, welche einen Verlauf von 0% über 100% bis 0% in einem bestimmten Zeitraum darstellt und nur ein einziges mal pro Trigger durchlaufen wird. Meist wird diese Funktion definiert durch die vier Parameter Attack, Decay, Sustain und Release,[23] wodurch ein solcher Modulator auch häufig als ADSR Envelope bezeichnet wird. Es ist jedoch auch möglich weitere Parameter zur Beschreibung der Hüllkurve hinzuzufügen, um eine erhöhte Komplexität zu erreichen. In nahezu allen Fällen ist ein Envelope Modul mit dem Verstärker verbunden. Somit ermöglicht sich die dynamische Veränderung der Lautstärke des Ausgangssignals über die Zeit.[24] Da alle natürlichen Klänge Veränderungen unterworfen sind, ist eine solche Verwendung bei der synthetischen Imitation natürlicher Klänge auch zwingend erforderlich.[25]

Low Frequency Oscillator (LFO)

Der Low Frequency Oscillator ist eine periodische Modulationsfunktion, welche einmal gestartet kontinuierlich wiederholt wird. Von der Funktionsweise eines regulären Oszillators unterscheidet sich dieser nur in der Eigenschaft, dass der LFO mit einer geringeren Frequenz

---

[22] Es ist absolut denkbar, dass auch Modulatoren von Schall beeinflusst werden können. Beispielsweise könnte eine solche Einheit Audio über einen Side Chain Input erhalten und anhand dessen seine Modulationssignale berechnen. Diese Funktionalität ist zu diesem Zeitpunkt in der Synthesizer-Technologie nicht weit verbreitet.

[23] Deutsch & Deutsch S. 1

[24] vgl. Anwander S. 39

[25] vgl. Stange-Elbe S. 193

schwingt. Jedoch ist anzumerken, dass dies keinesfalls immer der Fall sein muss. Für Syntheseformen wie Frequenzmodulation (FM) oder Amplitudenmodulation (AM) kann diese Einheit auch durch einen normalen Oszillator ersetzt werden. Bei standardmäßig niedrigen Frequenzen bis 20 Hz jedoch kann dieses Modul in Verbindung mit einem Verstärker für einen Tremolo-Effekt sorgen, während ein Vibrato in Verbindung mit der Tonhöhe des klanggebenden Oszillators erzeugt werden kann.

Sequencer

Sequencer sind sehr komplexe Formen der Modulatoren. Sie können vom Nutzer frei programmierbare Funktionen ausgeben, welche genutzt werden können, um einen Synthesizer in einem bestimmten Rhythmus oder sogar nach einer bestimmten Melodie spielen zu lassen. Umsetzungen sind beispielsweise in der Form eines Stepsequencers in Synthesizern wie Apples Alchemy[26] oder Arturias Mini Lab[27] zu finden, oder in der Form eines Performers in der Massive Serie von Native Instruments.[28] Sequencer liegen nicht nur als Modul eines Synthesizers vor, sie sind auch als Midi-Recorder in nahezu allen gängigen DAWs vertreten. Auch hier werden Steuerbefehle für Klangerzeuger gespeichert,[29] was ihn somit abgesehen von seiner Position nicht von dem Modul Sequenzer unterscheidet.

2.3 Synthesearten

Synthesizer können Klänge auf verschiedene Arten erzeugen. Klassifiziert werden diese Vorgehensweisen in verschiedene Klangsyntheseverfahren. Zu diesen gehören unter anderem die subtraktive und additive Synthese, die Resynthese, Waveshaping und die Modulationssynthese. Wichtig ist hierbei festzustellen, dass dies nur eine Untergliederung zur besseren Verständigung zwischen Musikern und Technikern ist, wobei sich viele der Synthesearten in ihrer Funktionsweise überschneiden können. Daher ist eine exakte Kategorisierung unmöglich. Manche Verfahren finden auf der Ebene eines bestimmten Moduls, etwa dem Oszillator statt (additive Synthese), andere beschreiben das Zusammenspiel mehrerer Elemente (subtraktive Synthese). Ein Synthesizer kann mehrere Syntheseverfahren zur gleichen Zeit implementieren und berechnen. Besonders in einem digitalen Umfeld ist die

---

[26] siehe: https://www.apple.com/de/logic-pro/plugins-and-sounds/

[27] siehe: https://www.arturia.com/products/hybrid-synths/minilab-mkii/overview

[28] siehe: [1] https://www.native-instruments.com/de/products/komplete/synths/massive/
[2] https://www.native-instruments.com/de/products/komplete/synths/massive-x/

[29] vgl. Gorges & Meck 1989 S. 257

Vielfalt der verwendbaren Algorithmen sehr groß, was dem Nutzer die Möglichkeit bietet aus allen verfügbaren Formen der Klangerzeugung frei zu wählen.

Die sehr bekannte Form der subtraktiven Synthese die besonders zu Hochzeiten des analogen Synthesizers ihre Anwendung fand, jedoch bis heute in den meisten Technologien anzutreffen ist[30], basiert auf dem Prinzip, dass durch den Einsatz von Filtern von einem bereits bestehenden komplexen, obertonhaltigen Signal bestimmte Frequenzen entnommen werden, um so einen neuartigen Klang zu erzeugen. Die Popularität dieses Syntheseverfahrens stellt aber die heute verfügbaren Möglichkeiten längst nicht ausreichend dar. Da sie nur durch das Zusammenspiel mehrerer Module funktionieren kann, würde die Implementierung der subtraktiven Synthese den Umfang dieser Arbeit übersteigen, weshalb die exakte Funktionsweise hier nicht genauer beschrieben wird.
Eine weitere bekannte hier kurz kommentierte Form ist die Modulationssynthese, welche neben ihres Anwendungsbereiches im Audio auch häufig außerhalb der Klangerzeugung, wie zum Beispiel bei der Übertragung von Informationen im Radio, Telefon oder Internet, genutzt wird.[31] Hierbei wird entweder die Frequenz (FM), die Amplitude (AM) oder die Phase, des sogenannte Träger-(Carrier-)Signals mit einem anderen Signal (Modulator) moduliert,[32] wodurch völlig neuartige und unvorhersehbare klangliche Ergebnisse erzeugt werden können.

Bei der Programmierung eines Klangerzeugers kann man also aus einer Vielzahl verschiedener Vorgehensweisen wählen. Die Digitaltechnik erlaubt sogar - zumindest theoretisch - alle vorstellbaren Verfahren in nur einem einzigen Programm zu vereinen. Um dies zu bewerkstelligen muss mit den Grundlagen begonnen werden und zwar in der physikalischen und somit mathematischen Natur von Schwingungen selbst. Diese Basis, die sich der grundlegenden Eigenschaften von periodischen Funktionen bedient, heißt additive Synthese.

2.4 Einführung in die Implementierung

Bevor auf die theoretisch-mathematischen Grundlagen der verschiedenen Verfahren der Klangsynthese eingegangen wird, soll zunächst die grundlegende Vorgehensweise bei der

[30] vgl. Stange-Elbe S. 200

[31] vgl. Roads S. 215

[32] vgl. Stange-Elbe S. 206

Implementierung eines Klangerzeugers beschrieben werden. Ein Klangerzeuger kann im Kontext eines Synthesizers als Oszillator betrachtet werden.

Um einen Oszillator für den Browser mit Hilfe von *JavaScript* implementieren zu können, benötigt man einige Funktionen der von *Javascript* mitgelieferten *Web Audio API*.[33] Zunächst wird ein Objekt des Typs *AudioContext* erzeugt, was durch einen Aufruf des entsprechenden Konstruktors geschieht. Die Speicherung dieses Objekts erfolgt anschließend in einer entsprechenden Variable *context*.

Aus Sicherheitsgründen kann der Konstruktor nicht beim Laden der Seite, sondern erst nach einer Nutzerhandlung auf dieser aufgerufen werden.[34] Aus diesem Grund wird die Variable global definiert, während der eigentliche Audiokontext erst in einer entsprechenden Funktion *createAudioContext* erstellt wird.[35]

```
let context;
const createAudioContext = () => context = new AudioContext();
```

Auf eben jenes Objekt werden anschließend verschiedene Funktionen angewendet, was die Möglichkeit bietet, eigene Audioprogramme zu schreiben. Grundsätzlich funktioniert die Arbeit mit der *Web Audio API* ähnlich wie die Arbeit mit einem modularen Synthesizer. Es ist möglich, verschiedene Module - in der Welt von Web Audio Nodes - zu erzeugen und diese nach belieben zu kombinieren, um so am Ende komplexe Routings zu erhalten. Das hier benötigte Modul wird durch einen Aufruf der Funktion *AudioContext.createOscillator()* erzeugt. Im Anschluss ist das Starten des Oszillators ist erforderlich, um mit der Signalerzeugung zu beginnen. Obiges Programm wird also erweitert:

```
let context;
let oscillator;
const createAudioContext = () => {
       context = new AudioContext();
       oscillator = context.createOscillator();
       oscillator.start();
}
```

Durch die Verbindung des soeben erzeugten Klanggebers mit dem Ausgang des Audiokontexts wird dieser für den Nutzer hörbar gemacht. Der Ausgang ist erreichbar durch eine Referenz auf die Eigenschaft *AudioContext.destination*. Um der Unannehmlichkeit des durchgehenden Geräuschs auf den eigenen Lautsprechern oder Kopfhörern zu entgehen, wird

---

[33] mehr zur Web Audio API in Kapitel 6 auf Seite 51

[34] vgl. Adenot & Choi, “The AudioContext Interface”

[35] alle Code Beispiele nutzen die 2020 noch relativ neue ES6 (ECMAScript 6 / ECMAScript 2015) Syntax. Falls noch nicht vertraut, sind nähere Informationen zur Funktionalität und Anwendung unter
https://github.com/lukehoban/es6features
zu finden. (Abruf: 18.11.2020)

eine Möglichkeit den Ton an- und auszuschalten durch die Funktionen *connect* und *disconnect* implementiert.

```
const connect = () => oscillator.connect(context.destination);
const disconnect = () => oscillator.disconnect();
```

In nur zehn Zeilen gelingt es somit einen ersten einfachen Klangerzeuger zu Programmieren. Damit der Nutzer auch Kontrolle über die wiedergegebene Tonhöhe erhält, bedarf es einer Möglichkeit die Frequenz des Oszillators anzupassen. Auch hierfür bietet die *Web Audio API* ein geeignetes Interface. Über die Referenz *OscillatorNode.frequency.value* kann der Ausgangswert von 440Hz überschrieben werden.

```
const setFrequency = (value) =>
oscillator.frequency.value = value
```

Mithilfe einer solchen Methode eröffnen sich neue Möglichkeiten zum musikalischen Einsatz des Oszillators. Durch das Verknüpfen mit einer Midi-Schnittstelle kann dieser als Klanggeber eines Keyboards und somit für die Wiedergabe von Melodien genutzt werden. Um Oszillatoren polyphon, das heißt mehrstimmig erklingen zu lassen, ist es bei Benutzung der *Web Audio API* nötig, mehrere Instanzen des Objekts *OscillatorNode* zu erzeugen, welche jeweils unterschiedliche Tonhöhen wiedergeben. So ist es auch möglich, Harmonien und Akkordfolgen zu programmieren.

3 ADDITIVE SYNTHESE

Natürlich ist das bei weitem noch nicht alles, was ein Oszillator leisten sollte. Die wichtigste Eigenschaft, das Wiedergeben verschiedener Wellenformen, ist im obigen Programmbeispiel noch nicht implementiert. Abgesehen von Samplern und diesem verwandte Formen, arbeiten alle digitalen Oszillatoren nach dem Prinzip der additiven Synthese. Diese beruht auf einer mathematischen Methode, welche nicht nur bei der künstlichen Klangerzeugung sondern auch in vielen weiteren Bereichen der Ingenieurwissenschaften Anwendung findet.[36]

3.1 Fourier Analyse

1807 beschrieb Jean Baptiste Joseph Fourier in seinem Aufsatz “Mémoire Sur la propagation de la chaleur dans le corps solide” die mathematische Realisierung der Fourier Analyse,[37] welche auf dem Ansatz beruht, dass sich jedes beliebige periodische Signal aus einer beliebigen - auch unendlichen - Anzahl von Sinus- und Cosinus-Teilschwingungen zusammensetzen lässt.[38] Diese auch Fourier-Reihe genannte Reihe lässt sich durch die Fourier-Entwicklung berechnen und mithilfe der folgende Formel beschreiben:

$$f(t) = \sum_{k=1}^{N} a_k \cdot cos(k\omega t) + b_k \cdot sin(k\omega t) \qquad (3.1)$$

Üblicherweise wird die Reihe auch mit einem absoluten Summand a_0 beschrieben.[39] Da dieser aber für die künstliche Erzeugung und Generierung von Audioinformationen von nur begrenzt praktischer Bedeutung ist, wird die Beschreibung dessen hier außen vor gelassen.

[36] vgl. Osgood, S. 1

[37] vgl. Steppat, S. 103

[38] vgl. Randall & Tech, S. 8

[39] vgl. von Grünigen, S. 25

Abgebildet wird eine Funktion f in Abhängigkeit der Zeit t. Dabei ist f das Signal, welches sich durch eine Summe von Sinus- und Cosinusanteilen darstellen lässt. Die Winkelgeschwindigkeit ω lässt sich mit Hilfe der Grundfrequenz f_0 ausdrücken.

$$\omega = 2\pi f_0 \tag{3.2}$$

Das Produkt aus ω und k beschreibt die exakte Frequenz der einzelnen Teilschwingungen. Dabei repräsentiert k den Index des Elements in der Reihe aus Schwingungen, mit der sich die Funktion f zusammensetzten lässt.
Um die Fourier-Reihe darstellen zu können erfolgt zunächst die Berechnung der Fourier Koeffizienten a_k und b_k. Diese sind ein Maß dafür, wie stark die Frequenz $k\omega$ im Signal enthalten ist. Für jede Erhöhung von k um 1 wird eine Teilschwingung mit einer Frequenz, die einem ganzzahligen Vielfachen der Grundfrequenz entspricht, beschrieben.

Um das zu beschreibende Signal f exakt darzustellen, muss k gegen unendlich laufen,[40] jedoch wird N in der Realität frei gewählt und ist somit ein Indikator für die Auflösung der Fourier-Reihe. Je höher der Wert für N ist, desto genauer kann das Signal f approximiert werden. Im Bereich Audio gibt es allerdings keine Notwendigkeit, Werte jenseits von 1000 für N zu verwenden, da bei einer Grundfrequenz von 20Hz das tatsächliche Audiosignal bis zur menschlichen Wahrnehmbarkeitsgrenze von 20kHz[41] simuliert werden kann.
Die Fourier-Koeffizienten a_k und b_k können mit folgenden Formeln berechnet werden:

$$a_k := \frac{2}{T}\int_0^T f(t) \cdot cos(k\omega t)dt \tag{3.3}$$

$$b_k := \frac{2}{T}\int_0^T f(t) \cdot sin(k\omega t)dt \tag{3.4}$$

Die Funktion $f(t)$ steht hierbei für die Funktion, welche im Intervall von 0 bis T, was der Periodendauer einer einzelnen Schwingung entspricht, dargestellt ist. Sie muss für die Berechnung der Fourier-Reihe im Vorfeld bekannt sein.
Die Fourier Reihe-kann auch durch ihre komplexe Form dargestellt werden.

$$f(t) = \sum_{k=-N}^{N} c_k \cdot e^{ik\omega t} \tag{3.5}$$

---

[40] vgl. Von Grünigen, S. 25

[41] vgl. Webers 2007, S. 96

Diese Schreibweise ergibt sich durch das Anwenden der Euler-Form.

$$e^{i\Theta} = cos(\Theta) + isin(\Theta) \tag{3.6}$$

Auch hier finden sich die reellen Fourier-Koeffizienten a_k und b_k innerhalb des komplexen Koeffizienten c_k wieder.

$$c_k = a_k - ib_k \tag{3.7}$$

Für die Berechnung der Fourier-Koeffizienten c_k ergibt sich:

$$c_k := \frac{1}{2\pi} \int_0^T f(t) \cdot e^{-ik\omega t} dt \tag{3.8}$$

Die Fourier-Entwicklung lässt sich nur auf periodische Funktionen anwenden. Um diese auf nichtperiodische Funktionen zu übertragen, wird die Fouriertransformation mit der Fourier-Transformierten angewendet.

$$\hat{f}(\omega) := \frac{1}{2\pi} \int_{-\infty}^{\infty} f(t) \cdot e^{-i\omega t} dt \tag{3.9}$$

Auf diese Weise errechnet sich im Gegensatz zur Fourier-Reihe ein nicht mehr nur diskretes, sondern auch kontinuierliches Frequenzspektrum $\hat{f}$. Die Fourier-Rücktransformation erlaubt die Umwandlung dieses Spektrums zurück in die Ausgangsfunktion f.

$$f(t) = \int_{-\infty}^{\infty} \hat{f}(\omega) \cdot e^{i\omega t} d\omega \tag{3.10}$$

Die Fouriertransformation wird als Ausgangsbasis für die in den folgenden Kapiteln beschriebenen Algorithmen Discrete-Fourier-Transform (DFT) und Fast-Fourier-Transform (FFT) verwendet.

Zusammenfassend lässt sich feststellen, dass sich ein komplexes Signal in mehrere einfache Signale aufspalten lässt. Im Audio-Bereich spricht man hierbei von einer Grundschwingung und deren dazugehörigen Harmonischen.[42] Graphisch bedeutet dies, dass sich ein Signal nicht mehr nur noch nach dem Schema *Auslenkung über Zeit*, sondern auch durch *Amplitude über Frequenz* darstellen lässt. So erhält man ein sogenanntes Frequenzspektrum der Funktion.

[42] vgl. Grünigen, S. 26

2020 implementiert jede gängige DAW mindestens eines oder mehrere Programme zur Berechnung und Darstellung solcher Spektren, was dem Benutzer bei der Bewertung von Audioaufnahmen und Mischungen hilfreich entgegen kommen kann. Es ist jedoch nicht nur möglich diese Technik zur Analyse von Schwingungen zu nutzen, sondern auch bei dem Aufbau eigener Wellenformen von Beginn an.

3.2 Elementare Wellenformen

Die einfachsten und gängigsten solcher "selbst berechneter" Funktionen sind elementare Wellenformen, welche auch die häufigste Ausgangsbasis für die Klangerzeugung durch Oszillatoren darstellen. Die meisten unter ihnen ließen sich ebenfalls mit analogen Synthesizern einfach erzeugen und finden deshalb schon allein historisch bedingt häufig ihre Anwendung.[43] Auch sind sie in der digitalen Welt mathematisch nach einem simplen Muster berechenbar, was sie neben dem Besitzen charakteristischer Klangeigenschaften noch heute zu den beliebtesten Verwendungen machen.

3.2.1 Sinus

Die mathematisch, aber auch klanglich simpelste Wellenform ist der Sinus. Wie sich aus dem Namen bereits ableiten lässt, besteht diese Welle nur aus der Grundschwingung und besitzt keinerlei Harmonische. Folglich sind beide Fourier-Koeffizienten 0 und im Frequenzspektrum ist nur ein einziger Peak an der Stelle der Grundfrequenz zu erkennen, was diese Wellenform somit für die subtraktive Synthese unbrauchbar macht.
Trotz seiner kompakten mathematischen Beschreibung gehört der Sinus aus analoger Sicht nicht zu den elementaren Wellenformen, denn er lässt sich nicht durch eine einfache elektronische Schaltung erzeugen, sondern muss durch eine gezielte Modifikation der Dreieckskurvenform geschaffen werden.[44]
Der klare, weiche Ton findet in der Produktion von Musik häufig dann Anwendung, wenn es darum geht, den Bassbereich durch einen sogenannten Subbass weiter anzudicken.

[43] vgl. Anwander, S.50 für mehr Details zur analogen Erzeugung Elementarer Wellenformen

[44] vgl. Anwander, S.53

3.2.2 Dreieck

Um eine Dreiecksschwingung zu erhalten werden auf die Grundschwingung alle ungeraden Harmonischen aufaddiert. Die Amplitude fällt bei jeder neuen Addition um das Verhältnis von $\frac{1}{k^2}$ ab. Cosinus-Komponenten sind für die Erzeugung dieser Wellenform wie auch für die Erzeugung aller Wellenformen nicht zwingend erforderlich und für die Programmierung eines Oszillators sogar hinderlich. Dies hat den Grund, dass der Oszillator nur exakt eine Periode des Signals benötigt um zu funktionieren. Es bietet sich daher an eine ungerade Funktion zu nutzen, da nur ungerade Funktionen zum Zeitpunkt $t = 0$ auch eine Auslenkung von 0 besitzen. Somit lassen sich Knackser beim Starten des Klanges vermeiden. Damit eine Funktion diese Eigenschaft besitzt, darf die Fourier Reihe nur Sinus-Komponenten enthalten, da die Cosinusfunktion ihrerseits gerade ist und somit die Phase der Ausgangsfunktion verschiebt. Für die Fourier-Koeffizienten ergeben sich also:

$$a_k = 0 \tag{3.11}$$

$$b_k = \frac{1}{k^2} \cdot (k mod 2) \tag{3.12}$$

Die Dreieckswelle ist nach dem Sinus die weichste Form der Elementarwellen und wird in der Musik für die klangliche Reproduktion einer Flöte ein gesetzt.[45]

3.2.3 Rechteck

Die gleiche Anzahl, jedoch lautere Obertöne oder Harmonische enthält die Rechteck-Schwingung, auch Pulswelle genannt. Hier fällt die Amplitude jedoch deutlich schwächer ab.

$$a_k = 0 \tag{3.13}$$

$$b_k = \frac{1}{k} \cdot (k mod 2) \tag{3.14}$$

Optisch ähnelt die graphische Darstellung dieses Signals einem Rechteck, wodurch diese Welle auch ihren Namen hat. Sie existiert in nur zwei Zuständen: Hoch oder Niedrig.[46]
Die Pulswelle kann in ihrer Pulsweite moduliert werden.[47] Dazu wird bei gleichbleibender Amplitude das Integral der Funktion einer Periode von 0 entweder erhöht oder verringert, was

[45] vgl. Durmus “Die Dreieckwelle”

[46] vgl. Crombie, S. 13

[47] vgl Anwander, S. 54

mit einer Veränderung der Obertonstruktur einher geht. Optisch bedeutet dies, dass beispielsweise der Wert der Funktion nur die ersten 40% einer Periode oben ist, während er die anderen 60% unten liegt, was ein asymmetrisches Bild der Welle erzeugt. Die Rechteckwelle eignet sich ebenfalls hervorragend für die Imitation von Holzbläsern.[48]

3.2.4 Sägezahn

Die Letze, aber auch eine der wichtigsten elementaren Wellenformen ist der Sägezahn. Im Kontrast zu Drei- und Rechteck finden sich hier nicht nur alle ungeraden, sondern alle ganzzahligen Vielfachen der Grundschwingung. Für die Fourier Koeffizienten gilt hier:

$$a_k = 0 \tag{3.15}$$

$$b_k = \frac{1}{k} \tag{3.16}$$

Wie die Anzahl der Oberschwingungen bereits impliziert, wird beim Abspielen dieser Wellenform ein deutlich obertonreicheres Geräusch ausgegeben, welches sich durch seinen vollen Klang[49] unter anderem zur Imitation von Streichinstrumenten eignet.

3.3 Implementierung

Aufgrund des häufigen Vorkommens dieser vier Elementarwellenformen bietet die Oszillator Node der *Web Audio API* bereits vorgefertigte Versionen dieser Klänge an. Über die Eigenschaft *OscillatorNode.type* kann auf den aktuellen Typ der Wellenform zugegriffen oder dieser neu vergeben werden. Die Werte dieser Eigenschaft müssen dem Datentyp String entsprechen und lassen bei der Zuweisung nur vier Wörter zu: “sine”, “triangle”, “square”, “sawtooth”.[50]

Um nun von dieser Funktionalität Gebrauch zu machen, soll obiges, bereits aufgesetztes Programm um ein weiteres Feature erweitert werden. *HTML* liefert beispielsweise mit dem Input *type=“radio”* eine gute Möglichkeit zwischen den verschiedenen Formen hin und her zu schalten. Die Nutzereingabe wird anschließend in folgender Funktion *setWaveForm* verarbeitet:

```
const setWaveForm = (waveForm) => oscillator.type = waveForm
```

[48] vgl. Crombie, S. 12

[49] vgl. Crombie S. 14

[50] vgl. Adenot & Choi, “The OscillatorNode Interface”

Im Falle der Werte “sine”, “triangle”, “square” und “sawtooth” ist diese Herangehensweise selbsterklärend. Komplizierter, dafür aber umfangreicher ist jedoch der Typ “custom”.

OscillatorNode.type kann noch einen weiteren Wert “custom” annehmen. Versucht man diese jedoch direkt auf besagten String zu setzen, so exekutiert diese Zuweisung zunächst nicht.
Hierfür gibt es die Funktion *OscillatorNode.setPeriodicWave(wave)*. Diese Funktion nimmt als ihr erstes Argument ein *PeriodicWave* Objekt und setzt das vom Oscillator verwendete Signal mit diesem gleich.[51] Der Clou dabei ist, dass der Nutzer die Wellenform durch das Prinzip der additiven Synthese selbst bestimmen kann. Sobald diese Funktion mit einem validen Parameter aufgerufen wurde, wird der Typ des Oszillators auf “custom” gesetzt.
Um die daraus resultierenden Möglichkeiten zu verdeutlichen wird ein neues Programm aufgesetzt, welches im grundlegenden Aufbau dem ersten ähnelt. Im Script Tag des *HTML* Dokuments also:

```
let context;
let oscillator;
const createAudioContext = () => {
       context = new AudioContext();
       oscillator = context.createOscillator();
       oscillator.start();
}
const connect = () => oscillator.connect(context.destination);
const disconnect = () => oscillator.disconnect();
```

Um nun von der eben eingeführten Funktion Gebrauch zu machen muss zunächst ein *PeriodicWave* Objekt erzeugt werden. Auch hierfür liefert die *Web Audio API* die passende Funktion: *AudioContext.createPeriodicWave(real, imag[, constraints])*,[52] bei der das benötigte Ergebnis als Rückgabewert ausgegeben wird. Die ersten beiden Parameter der Funktion *createPeriodicWave* müssen für eine erfolgreiche Ausführung jeweils eine Sequenz von Float-Werten sein. Für einen einwandfreien Ablauf bietet sich der von *Javascript* implementierte Datentyp *Float32Array* an.[53]
Nachdem zuvor die Funktionsweise der Fourier Transformation genau beschrieben wurde, ist nun bereits ersichtlich, welche Informationen die beiden Arrays enthalten müssen. Hierbei handelt es sich um die Amplituden der Sinus- und Cosinus-Komponenten des zu berechnenden Signals. Die Länge der Liste bestimmt hierbei die Auflösung, also *N*. Der erste Parameter enthält alle Amplituden der Cosinus-Komponenten, der zweite alle Amplituden der

[51] vgl. Adenot & Choi, “The OscillatorNode Interface”

[52] Parameter welche in mit eckigen Klammern umgeben sind, sind optional

[53] vgl. o.V. MDN web docs “BaseAudioContext.createPeriodicWave”

Sinus-Komponenten. Aus eben diesem Grund werden diese auch in den Dokumentationen der *Web Audio API* als real und imag bezeichnet.[54] Wie bereits unter 4.1 erklärt können nämlich die Fourier Koeffizienten und somit die Amplituden auch in einer komplexen Zahl, bestehend aus Real- und Imaginärteil dargestellt werden.[55]
Nach der Beschreibung des Sinus unter 4.2.1 ist ebenso klar ersichtlich welche Werte der Funktion übergeben werden müssen, um ein, den Sinus repräsentierendes *PeriodicWave* Objekt zu erhalten. Dem Beispielprogramm hinzugefügt wird eine Funktion *createSine* welche als Rückgabewert einen Sinus in einem *PeriodicWave* Objekt liefert.

```
const createSine = () => {
       const real = new Float32Array(2);
       const imag = new Float32Array(2);
       imag[1] = 1;
       return context.createPeriodicWave(real, imag);
};
```

Da der Sinus nur die eine Frequenz der Grundschwingung enthält, kann das Signal auch bei einer Auflösung von 1 exakt beschrieben werden. Der erste Wert des Amplitudenarrays (Index 0) existiert nicht in der Fourier Synthese und muss daher 0 sein,[56] wodurch eine Länge von 2 völlig ausreichend ist. Da bei der Erzeugung von elementaren Wellenformen für den Oszillator keine Cosinus-Komponenten benötigt werden (vgl. 4.2.2) sind alle $a_k = 0$. Infolge dessen sind auch alle im ersten Parameter enthaltenen Werte 0. Der zweite Wert des zweiten Parameters ist dem entgegengesetzt gleich 1, was die Grundschwingung des zu berechnenden Signals markiert.

Nach dem selben Prinzip wird nun vorgegangen, um auch die anderen elementaren Wellenformen mathematisch zu erzeugen. Die Berechnungen der Amplituden kann mit Hilfe der entsprechenden Fourier-Koeffizienten in einer for-Schleife erfolgen. Je nach gewünschter Auflösung kann die Anzahl der Iterationen frei gewählt werden. Nach den *Web Audio API* Spezifikationen ist diese Zahl jedoch auf 4096 beschränkt.[57] Für die Verrechnung eines Sägezahns wird eine Auflösung von 1024 gewählt.

[54] vgl. o.V. MDN web docs “BaseAudioContext.createPeriodicWave”

[55] vgl. Steppat, S. 104

[56] vgl. Adenot & Choi, “The PeriodicWave Interface”

[57] vgl. Adenot & Choi, “The PeriodicWave Interface”

```
const createSawtooth = () => {
	const resolution = 1024;
	const real = new Float32Array(resolution);
	const imag = new Float32Array(resolution);

	for (let k = 1; k< resolution; k++)
		imag[k] = 1 / k;

	return context.createPeriodicWave(real, imag);
};
```

Solange das Verhältnis der in den Sequenzen mitgegebenen Amplituden zueinander stimmt, ist die tatsächliche Größe dieser nicht von Belang, da *createPeriodicWave* den Nutzer auch mit einem Normalisierungsfeature versorgt. Um dieses Verhalten abzuschalten, kann der Funktion als dritter Parameter noch der boolsche Wert *true* übergeben werden.[58]

Nun erweitern sich die Möglichkeiten der Klanggestaltung erheblich. Um sich die additive Synthese insoweit zunutze zu machen, dass auch zur Laufzeit des Programms völlig neue eigene Geräusche synthetisiert werden können, kann das Programm nun nach eigenem Ermessen erweitert werden.
In dem in der Einleitung angegebenen GitHub Repository zu dieser Arbeit befindet sich unter dem Branch “custom-wave-shapes” eine umgesetzte Implementierung der additiven Synthese für die elementaren Wellenformen Sinus und Sägezahn. Die Anwendung kann unter *https://www.jakobgetz.com/digital-soundsynthesis/custom-wave-shapes* getestet werden.

[58] vgl. Adenot & Choi, “The BaseAudioContext Interface”

4 RESYNTHESE

In der Theorie ist nun klar, wie jedes mögliche periodische Signal erzeugt werden kann. Die additive Synthese in der Welt der Synthesizer kommt vor allem dem Modul Oszillator zu gute, welcher den grundlegenden Klangerzeuger eines solchen Gerätes darstellt. Es gibt nicht viele Syntheseverfahren die allein auf dieser Ebene stattfinden können. Ein weiteres Beispiel wäre die Wavetable-Synthese, die die Klangausgabe eines Oszillators um weitere Wellenformen erweitert, welche der Nutzer zur Zeit der Wiedergabe sukzessive erklingen lassen kann.[59] Die Wavetable-Synthese enthält also keine neuen audiotechnischen Konzepte, viel mehr arbeitet auch sie nach dem Prinzip der additiven Synthese.[60]
In der Praxis begegnet man bei der additiven Synthese dem Problem, dass Klänge nur mit einem hohem Aufwand zu erzeugen sind. Es dauert schlichtweg einige Zeit um, ein Geräusch aus all seinen Harmonien nach und nach aufzubauen und die damit erzielten Ergebnisse sind häufig nicht so einzigartig und interessant, wie sie von einem Sounddesigner gewünscht wären. Um diesem Problem zu begegnen, kann sich eines weiteren Syntheseverfahrens bedient werden, der Resynthese.

Streng genommen ist die Resynthese kein neues Verfahren, sondern lediglich eine Erweiterung der eben durchdringend beschriebenen additiven Ausführung. Tatsächlich wird die Zusammensetzung eines Klanges aus seinen einzelnen Sinus- und Cosinus-Komponenten hier umgekehrt ausgeführt. Man spricht hier von einer Analyse. Somit benötigt man für dieses Verfahren bereits ein Ausgangssignal, was als Basis für die anschließend entstehende Wellenform stehen soll. Sobald das Signal in seine Harmonien zerlegt ist, können diese wiederum von der additiven Synthese verwendet werden um die Eingangswellenform zu rekonstruieren. Zudem kann das Klangspektrum, sobald nach der Analyse die Amplituden der

[59] vgl. Stange-Elbe, S. 201

[60] ein im Jahr 2020 häufig genutzter und beliebter Syntesizer der diese beiden Syntheseformen miteinander vereint und dabei sämtliche mathematischen Berechnungen grafisch visualisiert, ist Xfer Records Serum. Die Arbeit mit diesem kann ein gutes Gespür für das Zusammenspielen unterschiedlicher Syntheseformen vermitteln. Erhältlich unter: https://xferrecords.com/products/serum

einzelnen Teilschwingungen vorliegen, gezielt modifiziert werden.[61] Die Resynthese bietet also die Möglichkeit bereits existierende und reale aufgenommene Geräusche zur Ausgangsbasis für einen Oszillator zu machen. Des Weiteren kann dieses Verfahren dazu genutzt werden, die mit der additiven Synthese erzeugten Wellenformen zu speichern und diese Datei anschließend als Voreinstellung für den Klangerzeuger zu nutzen. Der Vorteil hierbei ist, dass kein neues Dateiformat entworfen werden muss, sondern ein klassisches, wie das Wave-Format, völlig ausreichend ist.

Um die Resynthese umsetzen zu können, muss geklärt werden, wie die Fourier Synthese zur Analyse von Audio angewendet werden kann. Da in der digitalen Welt Signale anstatt in kontinuierlicher Form nur in diskreten Werten vorliegen, benötigt man einen Algorithmus der vielleicht zu den bedeutendsten Entwicklungen der Informatik gehört.

4.1 Discrete-Fourier-Transform (DFT)

Die Discrete-Fourier-Transformation setzt sich zum Ziel, Frequenzanteile anstatt aus einer mathematischen Funktion aus einer Ansammlung von Daten zu gewinnen. Dabei wird jeder eingehende Datenpunkt f als Teil eines Vektors verstanden, dessen Dimension N der Anzahl der Samples entspricht. Als Ausgabe liefert diese Berechnung einen Vektor mit der gleichen Anzahl an Elementen $\hat{f}$ bei dem nun jeder Wert die Amplitude einer bestimmten Frequenz beschreibt. Somit entspricht die maximal mögliche Auflösung der Länge an Samples mit denen der Algorithmus gefüttert wird.

$$\{f_0, f_1, \ldots, f_{N-1}\} \rightarrow DFT \rightarrow \{\hat{f}_0, \hat{f}_1, \ldots, \hat{f}_{N-1}\} \tag{4.1}$$

Zur Berechnung der einzelnen Frequenzanteile $\hat{f}_k$ wird die Formel der Discrete-Fourier-Transform verwendet.

$$\hat{f}_k = \sum_{n=0}^{N-1} f_n e^{-i2\pi k \frac{n}{N}} \qquad \text{für } k = 0,1,...,N \tag{4.2}$$

Im Hinblick auf die Eulerform kann festgehalten werden, dass der Multiplikator $2\pi k \frac{n}{N}$ eine Frequenz beschreibt. Beim Durchführen dieses Algorithmus bricht man die Ausgangsdaten in eine Summe aus Sinuswellen verschiedener Frequenzen auf und erhält als Ausgabe durch $\hat{f}_k$

[61] vgl. Stange-Elbe, S. 201

die Amplituden eben dieser. Natürlich kann die Formel auch rückwärts angewandt werden, um so von den Frequenzanteilen zu der durch Daten repräsentierten ausgehenden Wellenform zu gelangen. Diesen Vorgang nennt man analog dazu inverse Discrete-Fourier-Transform (iDFT).

$$f_n = \frac{1}{N} \sum_{n=0}^{N-1} \hat{f}_k \cdot e^{-i2\pi k \frac{n}{N}} \qquad \text{für } k = 0,1,...,N \tag{4.3}$$

In der Praxis wird die Transformation durch eine Matrizenmultiplikation ausgeführt.
Im Hinblick auf die diskrete Fouriertransformation wird für ω folgende Formel definiert:

$$\omega := e^{-i2\pi \frac{1}{N}} \tag{4.4}$$

Hierbei ist der Vektor, welcher zur linken steht, der Frequenz-Output, während der Vektor auf der rechten Seite den Daten-Input darstellt. Die Analysefunktion ist durch die in der Mitte stehende *Vandermonde* Matrix repräsentiert.

$$\begin{bmatrix} \hat{f}_0 \\ \hat{f}_1 \\ \hat{f}_2 \\ \vdots \\ \hat{f}_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & . & . & . & 1 \\ 1 & \omega_N & \omega_N^2 & . & . & . & \omega_N^{N-1} \\ 1 & \omega_N^2 & \omega_N^4 & . & . & . & \omega_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_N^{N-1} & \omega_N^{2(N-1)} & . & . & . & \omega_N^{(N-1)^2} \end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{bmatrix} \tag{4.5}$$

4.2 Fast-Fourier-Transform (FFT)

Selbstverständlich wird nicht nur für die Resynthese die Analyse von periodischen Signalen benötigt. In allen Bereichen der Ingenieurwissenschaften finden Frequenzanalysen ihre Anwendung. Zum einen können solche Berechnungen für die Kompression von Audio und Bild eingesetzt werden, zum anderen erlaubt dieser Vorgang auch das Streamen von Video wie auch Satellitenkommunikation.[62] Der Algorithmus der Wahl für diese Aufgaben ist hier die Fast-Fourier-Transform (FFT).
Tatsächlich kommt die Fast-Fourier-Transform zum exakt gleichen Ergebnis wie die Discrete-Fourier-Transform, der Unterschied liegt jedoch in der für die Berechnung benötigten Operationen. Bei der DFT ist es nötig für jede der N Reihen der Multiplikationsmatrix N Multiplikationen auszuführen, was in einer totalen Laufzeit von $O(n^2)$ resultiert. Nun ist es

[62] vgl. Brunton & Kutz, S. 68

leicht vorstellbar, dass sich der Rechenaufwand für jedes dazukommende Datenstückchen erheblich verlängert beziehungsweise exponentiell zunimmt, was die Ausführungsdauer schnell in astronomische Höhen treibt und die Umsetzbarkeit in einigen Fällen sogar unmöglich machen kann. Dieses Problem löst die Fast-Fourier-Transform, da sie die totale Laufzeit auf $O(nlog(n))$ absenkt. Gerade bei größeren Eingangslängen ist dieser Unterschied so stark spürbar, dass Steven Brunton, Professor an der University of Washington des Studiengangs Mechanical Engineering, die FFT als einen der wichtigsten entwickelten Algorithmen aller Zeiten bezeichnet.[63] Um ein Gefühl für diesen Unterschied zu entwickeln, kann die Anzahl der Operationen an einem konkreten Beispiel verdeutlicht werden. Um beispielsweise 10 Sekunden Audiomaterial, dass mit einer Samplingfrequenz von 44100 Hz abgetastet wurde, mithilfe des DFT Algorithmus auf sein Spektrum zu untersuchen, benötigt man bereits $(10 \cdot 44100)^2$ Rechenoperationen, was einer gesamten Größe von über 10^{11} entspricht. Für die selbe Länge eines Signals analysiert mithilfe der Fast-Fourier-Transform ergibt sich ein Rechenaufwand von unter 6 Millionen, also nur ein Bruchteil von der ursprünglichen Zahl. Je größer die Menge der Abtastwerte des Audios wäre, desto deutlicher wäre dieser Unterschied zusätzlich spürbar.

Veröffentlicht wurde die Entwicklung der FFT 1965 von den Ingenieuren J.W. Cooley und J.W. Tukey.[64] [65] Es ist jedoch sicher, dass der bekannte Mathematiker Carl Friedrich Gauss bereits um das Jahr 1807 eine dem heutigen Algorithmus ähnliche Form angewandt haben muss. Da Computer zu dieser Zeit jedoch noch nicht erfunden waren und Laufzeiten somit eine eher untergeordnete Rolle spielten, fand diese Methodik damals nicht die nötigen Anwendungen, um umfassend veröffentlicht zu werden.[66]

4.3 Implementierung

Die Implementierung der Fast-Fourier-Transform gestaltet sich kompliziert, da sie nicht nur das Verstehen der eigentlichen Algorithmik voraussetzt, sondern bereits Funktionalität zum Umgang mit komplexen Zahlen benötigt. Ein Paket, welches die FFT für *JavaScript* implementiert, ist *fft-js*, eine Library, erstellt zu Lehrzwecken, sowohl für die normale als auch die invertierte Form durch die Mitwirkenden Ben Bryan, Joshua Jung und Frederick

[63] vgl. Brunton

[64] vgl. Heideman & Johnson, S. 265

[65] siehe: https://www.jstor.org/stable/2003354?seq=1

[66] vgl. Heideman & Johnson, S. 266

Gnodtke.[67] Das Paket kann in einem initialisierten *Node* Projekt installiert und importiert werden.[68] Über den Befehl

```
node <dateiname>
```

in der Kommandozeile unter dem entsprechenden Verzeichnis wird die *JavaScript* Datei direkt im Terminal des Computers ausgeführt.

Zunächst wird ein Signal benötigt, welches es zu untersuchen gilt. Für dieses Beispiel wird sich der einfachen Sinus-Schwingung bedient, welche ein ziemlich eindeutiges Ergebnis liefern sollte bezüglich seiner Analyse auf Obertöne. Ein solches wird in der Funktion *createSine* berechnet. Als Parameter kann der Nutzer die gewünschte Länge und eine Frequenz in Umdrehungen pro Durchlauf der eingegebenen Länge angeben. Es ist hierbei definitiv darauf zu achten, dass diese genau einer Potenz von 2 entspricht, da fft-js die Eingabelänge nicht automatisch anpasst. Nach Abschluss der Berechnungen werden die Samples eines Sinustons in einem Wertebereich zwischen -1 und 1 herausgegeben und können in einer Konstante gespeichert werden.

```
const createSine = (length, frequency) => {
       let sine = new Array(length);
       for (let t = 0; t < length; t++)
              sine[t] = Math.sin(
                     (frequency * 2 * Math.PI * t) / length
              );
       return sine;
};
```

Im Anschluss kann das erzeugte Signal der Funktion *fft(signal)* übergeben werden, welche eine Reihe komplexer Zahlen als Antwort zurück liefert.

```
const FFT = require("fft-js);

const signal = createSine(1024, 50);
const phasors = FFT.fft(signal);
```

Die komplexen Zahlen werden in der Form eines Arrays aus Tupples gespeichert, bei denen jeweils das erste Element den Realteil und das zweite Element den Imaginärteil darstellt. Um also das tatsächliche Frequenzspektrum zu erhalten, muss die Größe eben jener komplexen

---

[67] siehe: [1] https://github.com/vail-systems/node-fft
[2] https://www.npmjs.com/package/fft-js

[68] siehe: https://nodejs.org/en/

Zahlen berechnet werden, welche man ähnlich wie bei der Längenbestimmung eines Vektors zum Beispiel durch

$$\sqrt{real^2 + imag^2} \tag{4.6}$$

erhält. Daraus folgen die nächsten Zeilen für das kleine Programm:

```
const spectrum = phasors.map(ph =>
       Math.squrt(Math.pow(ph[0], 2) * Math.pow(ph[1], 2))
);
console.log(spectrum)
```

Das Spektrum, welches nun auf der Konsole ausgegeben wird, liegt in der Form eines Arrays vor, welches die Koeffizienten der einzelnen Harmonien des analysierten Signals enthält. Bei einem Sinus gibt es nur eine einzelne Grundschwingung, welche für das Spektrum ins Gewicht fällt. Ihre Amplitude ist in der Liste unter dem Index gespeichert, welcher im selben Verhältnis zur Gesamtlänge steht wie die angegebene Frequenz zur Abtastrate. In diesem Fall wurde eine Länge von 1024 Samples gewählt sowie eine Frequenz von 50, was bedeutet, dass der Koeffizient der Schwingung auch ‘ungefähr’ an Index 50 auftauchen wird.
Das Adjektiv ‘ungefähr’ ist hier wörtlich zu nehmen, da es bei der Durchführung des FFT Algorithmus zu einem gewissen Fehler in Abhängigkeit der Problemgröße kommt (Leakage Effekt). Je kleiner die Liste der Eingabesamples, desto größer wird dieser Fehler ausfallen. Durch dieses Verhalten lassen sich auch die sehr niedrigen jedoch präsenten Multiplikatoren zu jeder Frequenz innerhalb des Spektrums erklären.[69]

Normalerweise ist für eine musikalisch brauchbare Durchführung der Resynthese noch ein Algorithmus zur Erkennung der Tonhöhe zu implementieren. Dieser Schritt gewährleistet das korrekte Abspielen der Grundfrequenz einer Welle durch den Oszillator. Um so einen Algorithmus umsetzen zu können, ist eine erfolgreiche Fourier-Transformation zum einen die Voraussetzung und zum anderen ihr größter Bestandteil. Es bleibt dem Leser daher selbst überlassen, sich mit den verschiedenen Möglichkeiten des Stimmens von Audiosignalen zu befassen. Für die nachfolgende Anwendung werden nur entsprechende Audiodateien in bereits festgelegter Tonhöhe verwendet. Die Nutzung von eigenen Dateien ist hier zwar möglich, führt aber nicht zu einer korrekten Wiedergabe eines Tons mit der gewählten Frequenz von 440 Hz.

[69] vgl. Werner S. 221

5 DIE APP

Nachdem nun die technischen Grundlagen für die digitale Klangsynthese beschrieben worden sind, soll sich diese Arbeit nun der aktiven Implementierung eines Klangerzeugers für den Browser widmen. Das nachfolgende Projekt wird aufzeigen, welche Fähigkeiten die heute nutzbaren Webtechnologien haben, um hochwertige Werkzeuge für die (professionelle) Arbeit mit Audio herstellen zu können. Dennoch sei gesagt, dass die hier vorgestellten Konzepte die Funktionsweise sowie die verfügbaren Möglichkeiten nur oberflächlich beleuchten und allerhöchstens ein grobes Ausgangsbild dessen darstellen können, zu was ein kompetenter Softwareentwickler fähig sein kann.
Die beschriebene Anwendung ist unter *www.jakobgetz.com/digital-soundsynthesis/oscillator* zur freien Nutzung verfügbar.

5.1 Funktionsweise

Beim Festlegen auf den genauen Inhalt der Anwendung stellt sich die Frage, was einen Klangerzeuger eigentlich genau aus macht. Zunächst ist dieser Begriff weit fassbar und kann von der Mechanik eines kompletten Synthesizers bis hin zur Aufgabe eines einfachen Media Players definiert werden. Wie bereits aus den vorangegangenen Kapiteln hervorgegangen, fasst diese Arbeit den Begriff Klangerzeugung an seinen Wurzeln und erklärt die Entstehung eines Geräusches von der Pike auf. Für die Erzeugung eines Klanges werden im Grundlegenden nur eine Reihe von Sinus- und Cosinus-Schwingungen benötigt, aus der sich jedes beliebige periodische Signal zusammensetzen lässt. Dieses Verfahren der additiven Synthese ist treibende Kraft hinter der Klangerzeugung eines in der Audiotechnik klassischen Oszillators, weshalb sich diese Arbeit ausschließlich mit der Implementierung eines solchen beschäftigt.
Das zu erstellende Programm soll in seiner finalen Form folgende Funktionen beinhalten.

1. Ein Oszillator, der auf Druck eines Knopfes einen Ton von sich gibt
2. Die Möglichkeit eine Wave Datei zu laden und deren Audioinhalt als Basis für die vom Oszillator wiedergegebene Wellenformen zu Nutzen
3. Eine optische Repräsentation der aktuell geladenen Wellenform in einer Auslenkung-über-Zeit-Grafik
4. Die optische Repräsentation der aktuell geladenen Wellenform in einem Frequenzspektrum und die freie Manipulierbarkeit dessen
5. Die Möglichkeit mehrere Wellenformen zu einem Wavetable zusammenzufassen und das aktuell wiedergegebene Signal zur Spielzeit zu verändern

Eine solche Anwendung beinhaltet alle bisherig behandelten Konzepte und stellt ihrerseits einen komplexen Oszillator dar, wie er als Ausgangsbasis eines Synthesizers verwendet werden kann. Es scheint, nach der Aufzählung der verschiedenen Verfahren der Klangsynthese von Joachim Stange-Elbe[70] nicht möglich, noch weitere Syntheseformen auf der Ebene eines einzelnen nicht Samplebasiertem Oszillator durchzuführen.

5.2 Architektur

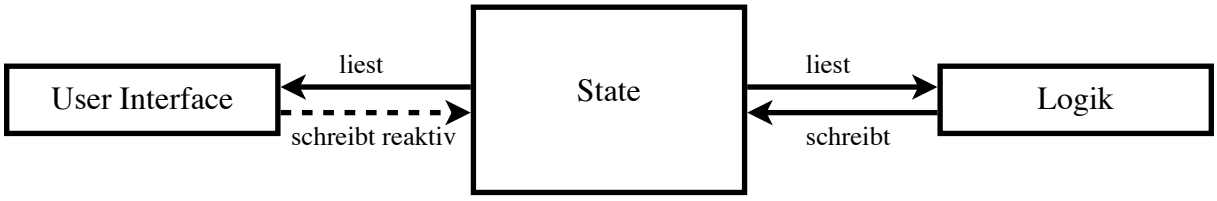
Um diese Funktionalität ermöglichen zu können, muss das Programm aus drei Bausteinen bestehen.

1. Das User Interface, welches dem Endnutzer die Möglichkeit bietet mit dem Klangerzeuger zu interagieren und eigene Einstellungen zu treffen. Hier werden sämtliche grafischen Informationen dargestellt.
2. Einen zentralen Ort der Datenverwaltung, den so genannten State (Zustand). Sämtliche Parameter auf deren Basis der Oszillator seinen Klang verrechnet, werden hier zwischengelagert.
3. Die Logikeinheit, in der alle für die Klangerzeugung relevanten Berechnungen durchgeführt werden. Diese Komponente wird das Herzstück der Anwendung darstellen. Hier ist der eigentliche Oszillator implementiert.

Eine Architektur nach diesem Schema hat den Vorteil, dass die Benutzeroberfläche und Logik komplett getrennt voneinander entwickelt werden können, um sich nicht gegenseitig zu

[70] vgl. Stange-Elbe, S. 198

beeinflussen, was die Gefahr von entstehenden Bugs und Fehlern durch eine Senkung der Komplexität bereits präventiv behandelt. Logikeinheit und Benutzeroberfläche müssen nur über ein Bindeglied miteinander kommunizieren, dem so genannten State, einem zentralen Ort der Datenverwaltung, welcher Schnittstellen zur Manipulation und Lesbarkeit bietet. Im State sind alle Informationen zur Klangerzeugung gespeichert, welche die Logikeinheit benötigt, um die Synthese korrekt zu berechnen. Somit kann die Benutzeroberfläche Daten aus dem State lesen und manipulieren, während die Logikeinheit die dort enthaltenen Daten nur reaktiv behandelt. Sollten sich also Veränderungen im Aufbau eines Komponenten ergeben, so bleiben die anderen beiden Einheiten des Programmes unangetastet und das Programm ist weiterhin lauffähig. Einzig und allein eine Architekturänderung oder Erweiterung des States führt zu Konsequenzen, die sowohl in der Logik- als auch in der Grafikeinheit aufgegriffen werden müssen.



5.3 Implementierung

Die Implementierung der App wird in ihren allgemeinen Prinzipien und Mechaniken beschrieben. Die Darstellung und Beschreibung des gesamten Quelltextes kann in dieser Arbeit nicht erfolgen, jedoch befindet er sich in einem Git Repository unter dem Branch “oscillator” welches unter der URL: *https://github.com/jakobgetz/digital-soundsynthesis* abzurufen ist. Nach dem Durcharbeiten der allgemeinen Prinzipien sollte der Leser, welchem grundlegende Kenntnisse zu *React*, *Redux* und *TypeScript* vorausgesetzt werden, in der Lage sein, den Programmablauf nachvollziehen und reproduzieren zu können. Außerdem soll durch die Vorstellung der verschiedenen Konzepte eine Basis geschaffen werden, auf welcher eigene Anwendungen im Bereich Audiobearbeitung und Klangerzeugung aufgebaut werden können.

5.3.1 Setup

Die Entwicklung der Anwendung erfolgt in der *React*-Umgebung. *React* ist eine von Facebook entwickelte und vertriebene Bibliothek für *JavaScript*, welche ab 2011 ursprünglich

unter dem Namen *FaxJS* erschien.[71] Diese Library stellt ein Grundgerüst für die Ausgabe von User Interface-Komponenten (Components) von Webseiten (*React*) oder Android beziehungsweise IPhone Apps (*React-Navive*) zur Verfügung. Als weitere Starthilfe wird nicht *JavaScript* als Sprache zur Implementierung der Software verwendet sondern das ebenfalls auf dem *ECMA06* Standard beruhende *TypeScript*. *TypeScript* bietet die Möglichkeit der Typisierung von Daten was eine erhöhte Elimination von Fehlern zur Laufzeit zur Folge hat. Um also ein Template für ein Programm eben jener Technologien zu bekommen liefern die Entwickler von *React* ein Script welches ein entsprechendes Projekt durch einen einfachen Befehl in der Kommandozeile des genutzten Computers in dem gewünschten Verzeichnis anlegt. Das Script kann nach der Installation von *nodejs* zusammen mit dem *Node Package Manager* (*npm*)[72] und *TypeScript* auf dem entsprechendem System gestartet werden.

```
npm create-react-app oscillator –template typescript
```
[73]

Der Source Ordner (*src*) ist anschließend für ein sauberes Setup aufzuräumen. Sämtliche Dateien bis auf die Dateien *App.tsx*, *index.tsx* und *react-app-env.d.ts* sind für die Implementierung nicht von nutzen und können gelöscht werden, was auch eine Säuberung des *tsx* im App-Componenten zur Folge haben muss.

Die beschriebene Architektur kann durch das Erstellen dreier Unterverzeichnisse im Ordner *src* aufgebaut werden. Ein Ordner *components* beinhaltet sämtlichen Code für die Darstellung der Nutzeroberfläche in den auch die Datei *App.tsx* verschoben wird, ein Ordner *logic* beinhaltet den Quelltext für die Funktionsweise des Oszillators und ein Ordner *state* begrenzt den Bereich aller zustandsorganisierenden Algorithmen. Die sich nun nicht in einem der Ordner befindliche Datei *index.tsx* wird später die Verbindung des States mit der Nutzeroberfläche regeln.

Nach dem Starten der Applikation im Browser wird *React* zunächst das User Interface laden. Um nun auch die Maschine des Klangerzeugers in Gang zu bringen, muss beim Rendern der Oberfläche auch der Quelltext des Oszillators einmalig ausgelesen werden. Aus diesem Grund muss dieser komplett in eine übergeordnete Funktion eingefasst werden. Im Ordner *logic* wird also ein File mit dem Namen *osc.ts* erstellt, welches eine Funktion *osc* implementiert, in der der gesamte Quelltext der Logikeinheit enthalten ist. Diese Funktion wird anschließend exportiert um im Component “App” aufgerufen werden zu können.

---

[71] vgl. Springer, S. 25

[72] siehe: https://nodejs.org/en/

[73] Bei diesem Kommando ist `oscillator` der gewählte Name für die Anwendung

```
// file osc.ts

const osc = () => {
       // Logik
       // …
};
export default osc;
```

Nach einem Import dieser Funktion in die Datei *App.tsx* und die Implementierung einer Schaltfläche im User Interface, welcher diese auf Klick aufruft kann die Anwendung nach Laden der Benutzeroberfläche einmalig gestartet werden.

5.3.2 State

Mit diesen Maßnamen ist die Implementierung des logischen Lebenszyklus der Anwendung beendet. Bevor jedoch mit der Entwicklung des User Interfaces begonnen werden kann, ist es zunächst notwendig, die allgemeinen Parameter zu definieren, auf der die Algorithmen des Klangerzeugers aufbauen. Sämtliche Daten für Berechnung des korrekten Geräusches werden an einem zentralen Ort aufbewahrt. Da sowohl die Nutzeroberfläche, als auch die Logik auf diesen Daten aufbauen, ist es daher empfehlenswert mit der Implementierung dieses Ortes, dem Zustand der Anwendung oder Application State zu beginnen.
Dieser State muss eine wichtige Eigenschaft besitzen. Da bei der Synthese von Klängen Echtzeitanforderungen herrschen, muss die Berechnung des Klanges bei jeder Veränderung der Parameter zumindest teilweise neu erfolgen. Dies bedeutet, dass die Logikeinheit den State konstant beobachten muss, um bei jeder Veränderung eines bestimmten Parameters eine bestimmte Funktion feuern zu können. Selbiges gilt für die grafische Ausgabe. Da sich zum Beispiel mit Veränderungen der Harmoniekonstellation die wiedergegebene Wellenform ändert, muss die grafische Ausgabe derer in Echtzeit aktualisiert werden.
Eine Bibliothek, welche den Entwickler in einer *JavaScript* Anwendung mit eben jener Funktionalität versorgt, ist *Redux*.[74] Diese kann in der *React* Umgebung durch folgenden Befehl in der Kommandozeile unter dem Verzeichnis *oscillator* dem Projekt hinzu gefügt werden:

```
npm install redux react-redux redux-thunk redux-watch
```

[74] siehe: https://redux.js.org

Genau genommen werden hier vier Pakete zugleich installiert, einmal *Redux* selbst und einmal ein Interface zur Kommunikation der Library mit *React*.[75] *Redux-thunk* ist eine Erweiterung der Bibliothek zur Ausführung asynchroner Aufrufe im Store.[76] Das letzte Paket *redux-watch*[77] liefert Funktionalität zur Beobachtung des States auf bestimmte Parameter und ist für die Umsetzung der Klangerzeuger Logik von Vorteil.

Der State ist wie ein normales *JavaScript* Objekt aufgebaut. Informationen werden in einer bestimmten Eigenschaft dieses Objekts gespeichert. Dabei ist die Größe dessen dynamisch und kann vom Entwickler nach Belieben angepasst werden. Eine genaue Architektur des States kann nach einer genauen Analyse der Funktionsanforderungen an die Anwendung erfolgen.
Der Nutzer der App soll die Möglichkeit haben eigene Wellenformen als Ausgangsmaterial für die Arbeit der Klangerzeugung zu wählen. Wie bereits erläutert, benötigt die *OscillatorNode* dafür ein Objekt des Typs *PeriodicWave*, allerdings ist dieses nicht in der Lage Informationen für die beiden Anforderungen nach einer grafischen Darstellung der Wellenform zu liefern. Um das Signal in einem Auslenkung-über-Zeit-Diagramm darstellen zu können, wird ein Array aus Zahlen benötigt, was Informationen für eben jene Grafik liefert. Durch die digitale Speicherung von Audio in zeit- und wertkontinuierlicher Form besitzen die tatsächlichen Samples eines Signals bereits die nötigen Informationen. Somit ist es notwendig neben des Objektes *PeriodicWave* auch ein Array mit den tatsächlichen Samples der Wellenform zu speichern. Um anschließend ein Frequenzspektrum zeichnen zu können, muss das Signal zusätzlich noch durch die entsprechenden Frequenzanteile repräsentiert werden. Da es für das Kreieren von Wellenformen ausreichend ist, sich auf Darstellung ungerader Funktionen zu beschränken, reicht eine Speicherung aller Sinus-Komponenten völlig aus. Um all diese für die Applikation wichtigen Informationen in einem einzelnem Objekt speichern zu können, wird für diese Eigenschaft ein neuer Datentyp *Wave* geschaffen.
Im State wird eine Instanz dieses Typs unter der Eigenschaft *currentWave* gelagert.

---

[75] siehe: https://react-redux.js.org/introduction/quick-start

[76] siehe: [1] https://redux.js.org/tutorials/fundamentals/part-6-async-logic
[2] https://github.com/reduxjs/redux-thunk

[77] siehe: https://github.com/ExodusMovement/redux-watch#readme

```
type Wave = {
       periodicWave: PeriodicWave;
       samples: number[];
       bins: number[];
};
```

Des Weiteren soll die Anwendung die Möglichkeit zur Umsetzung der Wavetable Synthese erhalten. Wavetable sind im Grunde nur unterschiedliche zu Listen zusammengefasste Wellenformen, welche durch einen Regler ausgewählt werden können. Daher reicht es also nicht aus, nur ein einzelnes Objekt des Typs Wave im State zu speichern, sondern ein ganzes Array. Um zur Laufzeit feststellen zu können, welches dieser Elemente vom Nutzer für den Moment ausgewählt wurde, muss zudem auch eine Information darüber im State gespeichert sein.

Vollständig betrachtet sollte auffallen, dass für das Implementieren eines Oszillators, welcher auf Knopfdruck einen Ton erzeugt, wie bereits unter Punkt 2.4 beschrieben ein Objekt des Typs *AudioContext* und ein Objekt des Typs *OscillatorNode* benötigt werden. Diese beiden Informationen werden jedoch nur von der Logik Einheit des Programmes verarbeitet, wodurch eine zentrale Speicherung der *Web Audio API* Items nicht erforderlich ist. Dennoch muss fest gehalten werden, ob der Klangerzeuger gerade einen Ton von sich gibt oder nicht. Um den aktuellen Wiedergabestatus sichtbar zu machen, braucht man somit eine boolsche Messgröße, welche den Zustand angibt, in dem sich der Oszillator gerade befindet. Dieser binäre Wert kann unter der Eigenschaft *isPlaying* abgelegt werden.

Zu guter Letzt ist eine Speicherung der vom Nutzer hochgeladenen Datei im State notwendig, da ein Upload unweigerlich einige Konsequenzen für die Logikeinheit nach sich zieht. Standardmäßig werden Informationen zu diesen Aufrufen in einem Objekt untergebracht, was sowohl Aufschluss über den Ladezustand enthält, im Falle eines erfolgreichen Uploads die eigentlichen Daten und im Falle eines Fehlers die Error Message. Dieses Objekt kann in einem Typ *FileUpload* definiert werden.

```
type FileUpload = {
       loading: boolean;
       data: Float32Array;
       error: string
};
```

Lädt der Nutzer nun eine Datei hoch, wird zunächst der Zustand *loading* von *FileUpload* auf *true* gesetzt. Anschließend werden die Daten in ein Objekt des Typs *ArrayBuffer* umgewandelt, welcher nun als Ausgangsbasis für das Decodieren der Audiodaten dient. Der

AudioContext liefert eine Methode, um genau dies zu erreichen (*AudioContext.decodeAudio-Data(ArrayBuffer)*). Verlaufen all diese Schritte erfolgreich, ist das Hochladen der Datei abgeschlossen und die Eigenschaft *data* von *FileUpload* wird auf die Samples des ersten Kanals der Audiodatei gesetzt. Es ist nicht zu vergessen, sämtliche Fehler über einen entsprechenden catch Block aufzufangen und die Error Nachrichten in den State zu schreiben. Für das Uploaden eines Files wird die Funktion setAudioFile definiert:

```
export const setAudioFile = (data: File | Response | null) => (
	Dispatch: Dispatch
) => {
	if (data) {
		data
			.arrayBuffer()
			.then((buffer: ArrayBuffer) =>
				new AudioContext()
				.decodeAudioData(buffer)
			)
			.then((audioBuffer: AudioBuffer) =>
				dispatch(setAudioFileSuccess(
					audioBuffer.getChannelData(0))
				)
			)
			.catch((err: Error) =>
				dispatch(
					setAudioFileError(err.message)
				)
			);
	} else {
		const err = new Error("could not find data");
		dispatch(setAudioFileError(err.message));
	}
};
```

Beachtet man also alle für das Projekt notwendigen Daten, so ergibt sich daraus folgende Architektur für die Definition des States:

```
type State = {
	isPlaying: boolean;
	waveTablePosition: number;
	waveTable?: Wave[];
	currentWave?: Wave;
	audioFile?: FileUpload;
};
```

Drei der Eigenschaften sind optional was mit dem asynchronen Charakter einiger Informationen zusammen hängt. Zur ersten Instanzierung des States sind noch keine Wellenformen präsent, weswegen dieses Objekt noch nicht im zentralen Speicher enthalten

sein kann. Diese treten erst durch das Laden einer Wave Datei in Erscheinung. Aufgaben, bei welchen ein Programm auf die Antwort anderer Systeme angewiesen ist, wie beispielsweise das Laden einer Datei, sind asynchron und können nicht in Echtzeit durch den Quelltext ausgeführt werden.[78] [79] Dies hat zur Folge das diese Informationen nicht zum Zeitpunkt der ersten Instanzierung, sondern erst zu einem späteren Zeitpunkt zugriffsbereit sind.
Nachdem ein Datentyp für den State geschaffen wurde erfolgt anschließend die tatsächliche Implementierung dessen mit Hilfe von *Redux*. Der *Redux-Store* kann *React*, durch den Import des *Provider*-Component und das Umgeben der gesamten Anwendung mit eben diesen, bereit gestellt werden.

```
// file index.tsx

ReactDOM.render(
       <React.StrictMode>
              <Provider store={store}>
                     <App />
              </Provider>
       </React.StrictMode>,
       document.getElementById(“root”)
);
```

5.3.3 User Interface

Auch für den Aufbau der Benutzeroberfläche helfen die Spezifikationen des Funktionsumfangs der Anwendung weiter. Zu Beginn dieser Beschreibung sei angemerkt, dass sich dieser Punkt nur mit dem funktionalen Inhalt der grafischen Ausgabe beschäftigt und nicht mit dem künstlerischen Design. Das Nutzen von *Cascading Style Sheets* (*CSS*) tritt hier nur in Randerscheinungen auf und wird nicht näher erörtert.
Das User Interface besteht aus fünf Teilen:

1. Ein Knopf zum Starten und Stoppen der Wiedergabe
2. Ein Knopf zum Upload einer Wave Datei
3. Ein Regler zum Verstellen der Wavetable-Position
4. Eine Grafik zur Anzeige der aktuell ausgewählten Wellenform

---

[78] für mehr Informationen zu asynchronem JavaScript, siehe:
[1] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous
[2] https://eloquentjavascript.net/11_async.html

[79] für mehr Informationen zu JavaScripts Concurrency Model, siehe:
[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop
[2] https://html.spec.whatwg.org/multipage/webappapis.html#event-loops
[3] https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/#what-is-the-event-loop

5. Eine manipulierbare Grafik zur Anzeige der aktuellen Wellenform in ihrem Frequenzspektrum.

Alle diese Komponenten sollen erst gerendert werden, nachdem die Anwendung durch das einmalige Drücken des Startknopfes zum Laufen gebracht wurde. Während die ersten beiden Teile ohne große strukturelle Schwierigkeiten in den App-Component integriert werden können, bietet es sich für die Teile 3 und 4 an, diese in ihre eigenen Dateien auszulagern. Im Ordner components lohnt es sich daher, die funktionalen Components *WaveForm.tsx* und *Spectrum.tsx* anzulegen.
Auf einen Druck der Schaltfläche *Start/Stop* wird die Eigenschaft *isPlaying* des States invertiert. Beim Verändern des Reglers zur Wavetable Position wird die Eigenschaft *waveTablePosition* des States auf den Eingabewert des Reglers gesetzt.
Bei dem Upload einer Datei muss zunächst überprüft werden, ob auch wirklich ein File ausgewählt wurde. Falls keines selektiert ist, passiert auf einen Druck des Upload Buttons nichts, sind hingegen eine oder mehrere Dateien ausgewählt, wird diejenige, welche sich am Index 0 in der Eingabeliste befindet, weiter prozessiert, was bedeutet dass sie der unter 5.3.3 beschriebenen Funktion *setAudioFile* als Parameter übergeben wird.
Der Component *WaveForm* benötigt die Samples der aktuellen Wellenform aus dem State um die grafische Repräsentation rendern zu können. Hierbei bedient er sich allerdings nicht der Eigenschaft *currentWave* sondern der beiden Eigenschaften *waveTable* und *waveTable-Position*, wodurch er durch eine Abfrage des Arrays an die selben Informationen gelangt. Der Grund hierfür liegt in der Häufigkeit der Aktualisierung des *currentWave* Objekts. Da eine Aktualisierung des States immer mit einem gewissen Rechenaufwand verbunden ist, sollten große Objekte nur selten verändert werden. Somit liegt die Aktion bei einer Bewegung des Reglers zur Wavetable Position nur bei der Veränderung des entsprechenden Parameters und nicht bei der Aktualisierung des *currentWave* Objekts, welches seinerseits erst beim Loslassen des Reglers überschrieben wird. Um jedoch dauerhaft eine akkurate grafische Anzeige zu erhalten, wird die Sample-Information also mithilfe von *waveTable* und *waveTablePosition* extrahiert.
Für die Grafik wird in einer Box mit einer festgelegten Größe eine Linie durch alle Samples, welche die Auslenkung der Wellenform zu einem bestimmten Zeitpunkt repräsentieren, gezeichnet, wobei die Samples gleichmäßig über die gesamte Breite der Box verteilt sind. Eine Berechnung der Positionen der Punkte erfolgt in einer Funktion *calcCoordinates* welche eine weitere Funktion *calcX* für die Berechnung des x-Achsen Abschnitts zur Hilfe zieht. Eine

Library, welche sich gut für das Zeichnen komplexer Grafiken eignet, ist *Konva*.[80] Hier kann eine Container des types Stage mit einer bestimmten Höhe und Breite definiert werden, welche zu einem späteren Zeitpunkt über die Funktionen *Stage.width()* und *Stage.height()* ausgelesen werden können. Um mit dieser Bibliothek eine Linie zeichnen zu können, wird ein Array mit den Koordinaten der einzelnen Punkte nach der Form [x1, y1, x2, y2, … , xn, yn] benötigt.

```
// file WaveForm.tsx

const calcCoordinates = () => {
       let coordinates: number[] = [];
       if (waveTable)
              waveTable[waveTablePosition].samples.map(
                     (sample, i) => {
                            coordinates.push(
                                   calcX(i) * stage.width(),
                                   sample + (stage.height() / 2
                                   * stage.height() / 2
                                   );
                     });
       return coordinates;
};

const calcX  = (i: number) => {
       if (waveTable) {
              const len = waveTable[waveTablePosition]
                     .samples?.length;
              return len ? i / len : i;
       }
       return 0;
};
```

Der Component *Spectrum* benötigt ebenfalls Informationen der aktuell geladenen Wellenform. Da die Veränderung des Spektrums nur einmal nach Loslassen des Wavetable-Reglers geschehen soll, ist Objekt *currentWave* des States hier völlig ausreichend. Für die Darstellung des Spektrums wird nicht wie bei der Ausgabe der Wellenform eine eigene Grafik gezeichnet, sondern viel mehr eine Liste aus *HTML* Elementen des Typs *range* im *jsx* gerendert. Die Länge der Liste ist äquivalent zur Anzahl der berechneten Frequenzanteile des Signals. Um den Rechenaufwand zu verringern, kann sie aber auch auf eine kleinere Zahl wie beispielsweise 100 herab gesetzt werden. Höhere Frequenzanteile sind bei einer Grundschwingung von 440 Hz sowieso nicht zu hören. Die Auslenkung der Regler wird über die entsprechenden Amplituden der Harmonien bestimmt. Hierbei ist von Vorteil die Regler gegenüber des gesamten Spektrums ebenfalls in eine separate Datei *BinSlider.tsx* auszulagern,

[80] siehe: https://konvajs.org/docs/react/index.html

da so die Positionen exakt voneinander trennbar berechnet werden können. Zudem bietet es sich an, die Ausrichtung dieser hierbei zu verändern, um sie auf einer horizontalen Achse anzeigen zu können, was das Ablesen sowie die Bedienung intuitiver für den Nutzer gestaltet. Jeder Input erhält dabei eine eigene ID um Konflikte bei der Änderung von Obertönen zu vermeiden. Auch hier gilt: Erst beim Loslassen eines Reglers wird der neue Wert in den State übertragen um gigantische Rechenzeiten zu vermeiden. Durch das *MouseUp* event des *HTML* Elements wird also der Frequenzanteil der aktuellen Wellenform im State auf den neuen Wert gesetzt.

5.3.4 Logik

Nachdem der globale Datenspeicher sowie die Benutzeroberfläche implementiert wurden, dienen diese beiden Komponenten als Ausgangsbasis für die Programmierung der eigentlichen Algorithmen zur Erzeugung der Signale. Beinahe sämtlicher Quelltext wird hierfür in der Funktion *osc* der Datei *osc.ts* geschrieben. Dieser Teil des Programmes ist aus verschiedenen Abschnitten zusammengesetzt.
Zuoberst steht die Definition von Funktionen, welche das Verhalten des Oszillators bei Änderungen bestimmter Parameter des States beschreiben. Nach diesen folgt ein Abschnitt, welcher den Aufruf eben dieser definierter Funktionen steuert. In der *Redux*-Umgebung spricht man hier von einer Subskription. Es wird also der Store, welcher den State enthält, abonniert und auf dessen Veränderungen werden bestimmte Aktionen durch geführt. Da also Zugriff auf den State für die Datei *osc.ts* gewährleistet werden soll, ist es nötig, den Store in der Kopfzeile zu importieren. Hierfür bietet es sich an, diesen in der gleichnamigen Variable *store* zu speichern. Da *Redux* selbst keine mitgelieferte Möglichkeit bietet nur bestimmte Parameter im Kontrast zum gesamten State zu abonnieren, hilft hier das unter 5.3.2 installierter Paket *redux-watch* aus. Soll also beispielsweise eine Funktion *connect* auf die Veränderung von der State Eigenschaft *isPlaying* ausgeführt werden, so kann dies nun nach folgendem Schema definiert werden.

```
const watchIsPlaying = watch(store.getState, “isPlaying”);
Store.subscribe(watchIsPlaying(connect));
```

Der zweite Parameter der von *redux-watch* implementierten Funktion *watch* ist eine Repräsentation des zu beobachtenden Parameters im zentralen Datenspeicher im String-Format.

Nach der Regelung der Subscriptions folgen Codeblöcke, welche zum ersten Aufruf von *osc* ausgeführt werden. Hier binden sich die Definitionen globaler Variablen und das Starten der *OscillatorNode*, das "Entpacken" des States und der Aufruf der Funktion *setUpWaveForm()*, welche im späteren Verlauf des Kapitels noch genauer beschrieben wird. Da es sich auch bei dieser Anwendung genau wie bei vorherigen Beispielen im Grunde nur um die Programmierung eines einzelnen Oszillators handelt, muss auch hier ein neuer *AudioContext* kreiert und in einer Konstante gespeichert werden. Selbiges gilt für den Oszillator selbst, ein Objekt des Typs *OscillatorNode,* welcher nach seiner Instanzierung sofort gestartet wird. Im Anschluss an diese Grundinstallation sollen die Parameter des States ebenfalls in lokalen Variablen der Datei gespeichert werden, da häufige Abfragen des *Redux* Stores in einem erhöhten Rechenaufwand resultieren. Einige im State enthaltenen Informationen werden in mehreren Funktionen benötigt, welche aber nichts mit einer direkten Veränderung eben jenes Parameters zu tun haben. Die Informationen werden also nur dann vom Store abgerufen, wenn auch wirklich Änderungen an ihnen vorgenommen wurden, welche anschließend die bisherigen in den lokalen Variablen gespeicherten Daten überschreiben. Durch dieses Vorgehen kann ein möglichst effizientes Nutzen des Stores gewährleistet werden. Die Parameter des States, welche von übermäßig häufigen Abfragen betroffen sind sind, *waveTablePosition*, *waveTable* und *currentWave* weswegen eine entsprechende Destrukturierung oder ein Entpacken durchzuführen ist.

```
let {
       waveTablePosition,
       waveTable,
       currentWave
} = store.getState();
```

Da nun der statische Code der Logikeinheit hinzureichend beschrieben wurde, richtet sich das Augenmerk nun auf den eigentlich zentralen Teil: das Verhalten des Oszillators auf verschiedene Änderungen der Parameter, definiert durch die Algorithmik, eingegliedert in fünf Funktionen.

setUpWaveForm

Diese Funktion wird nicht nur nach der Veränderung der Wavetable Position aufgerufen, sondern auch gleich einmal zu Beginn der Anwendung, nach dem ersten Starten der App. Sie setzt das aktuell wiederzugebende Signal des Oszillators auf die aktuell ausgewählte Wellenform. Da dieser Aufruf bei Veränderung des Wavetable Reglers sehr häufig geschieht, nutzt diese Funktion anstatt des Objektes *currentWave* den gesamten Wavetable in Verbindung

mit der Position in diesem, um übermäßigen Datenverkehr zu vermeiden. Eine Neuspeicherung von *currentWave* geschieht, wie beschrieben, erst nach Loslassen des Reglers innerhalb des User Interfaces. Natürlich ist eine weitere Aufgabe dieser Zeilen auch die Neufestlegung der lokalen Variable *waveTablePosition* auf den aktualisierten Wert.

```
const setUpWaveForm = () => {
       waveTablePosition = store.getState().waveTablePosition;
       if (waveTable)
              osc.setPeriodicWave(
                     waveTable[waveTablePosition].periodicWave
              );
}
```

connect

Die Funktion *connect* wird nach der Negierung des State Parameters *isPlaying* aufgerufen. Dabei erledigt sie genau das, was die Funktionen *connect* und *disconnect* unter 2.4 zur Aufgabe hatten. Sie fragt den aktuellen Wiedergabestatus des Oszillators ab und verbindet oder trennt dessen Signal mit dem Audioausgang des Rechners.

createWaveTable

Etwas komplizierter wird es mit der Funktion *createWaveTable*, welche nach dem Laden einer neuen Wave Datei ausgeführt wird. Hier müssen die neu angekommenen Samples prozessiert und in ihre Frequenzanteile zerlegt werden. Aus dem durchgängigen Signal müssen hier also zunächst mehrere kleine Wellenformen extrahiert werden, welche anschließend in drei Darstellungen in ein Objekt des unter 5.3.2 selbsterstellten Typs *Wave* gespeichert werden. Benötigt wird jede Welle in der Form eines *PeriodicWave* Objekts, eines Number Arrays bestehend aus Sinus-Komponenten und eines Number Arrays, welches die tatsächlichen Samples beinhaltet, um später den Verlauf des Signals in der Benutzeroberfläche darstellen zu können. Für den Wavetable reicht aber nicht nur ein einzelnes Objekt dieses Typs. Es ist somit nötig, gleich ein ganzes Array mit *Wave*-Objekten zu füllen, um die entsprechende Eigenschaft des States zu bedienen.

Um das Sample Arrays also erstellen zu können, muss das hochgeladene Signal zunächst in definiert große Stücke zerlegt werden von denen jeweils eines an die korrespondierende Stelle im Wavetable gespeichert wird. Wie bereits erklärt, ist für eine optimale Durchführung der Fast-Fourier-Transform eine Samplelänge, welche einer Potenz von 2 entspricht, von Vorteil. Somit wird in der Funktion *createWaveTable* nach einer Destrukturierung der Audioinformationen aus dem State erst eine Konstante *waveTableSampleLength* mit einem Wert von 2048 instanziert. Da sich die Samples aus dem State zum jetzigen Zeitpunkt anstatt

innerhalb eines Arrays noch in einem Objekt des Typs *Float32Arrays* befinden und *JavaScript* das Anwenden von *Array.prototype* Funktionen auf dieses nicht unterstützt, müssen die Daten zunächst in ein normales Array überführt werden. Das nun zu definierende zweidimensionale Array *samples* wird nun Audioinformationen der tatsächlichen Wellenformen beinhalten. Dazu wird ein Teil des ursprünglichen Signals mit einer Länge von 2048 Samples an jeden Index dieser Liste kopiert.

Nach diesen Schritten liegen bereits die Samples für das final angestrebte Objekt des Typs *Wave* in finaler Form vor. Nun wendet man sich den Amplituden der einzelnen Sinus-Komponenten zu. Die Koeffizienten können durch ein praktisches Anwenden der Fast-Fourier-Transform erhalten werden. Um diesen Algorithmus durchführen zu können, wird ein weiteres Mal das Paket *fft-js* verwendet. Die in der Bibliothek enthaltene Funktion *fft()* nimmt ein Array aus Samples und liefert die Koeffizienten in einem Array aus komplexen Zahlen zurück. Um diese verwerten zu können, muss die Größe jeder dieser komplexen Zahlen bestimmt werden, was durch die ebenfalls mitgelieferte Funktion *util.fftMag()* erreicht werden kann. Da sich die so errechneten Werte teilweise außerhalb des durch die Spektrum-Regler definierten Bereichs zwischen 0 und 1 liegen, werden diese mithilfe der Funktion *normalize* in den festgelegten Wertebereich eingepflegt.

Dank der so gewonnenen Daten kann das gewünschte Objekt *waveTable* erstellt und anschließend zusammen mit der Eigenschaft *currentWave* in den State überführt werden.[81]

```
const normalize = (numArray: number[]): number[] => {
       const max = Math.max(…numArray);
       return numArray.map((a) => a / max);
};
```

[81] Aus Gründen der Einheitlichkeit wurde davon abgesehen, die verschiedenen Logikabschnitte der Funktion *createWaveTable* in mehrere kleinere Funktionen zu unterteilen. Dies kann bei größeren Projekten aber durchaus von Vorteil sein.

```
const createWaveTable = () => {

	// get audio data form the store
	const { audio } = store.getState().audioFile;

	// define the sample length for each wave
	const waveTableSampleLength = 2048;

	// copy audio data from store into sample Array
	let audioArray = new Array(audio.length)
	for (let i = 0; i < audio.length; I++)
		audioArray[i] = audio[i];

	// copy the actual waveform samples into 2D Array
	let samples = new Array<number[]>(
		Math.floor(audio.length / waveTableSampleLength)
	);
	let position = 0;
	samples.fill([]);
	samples = samples.map(() => audioArray.slice(
		position, (position += waveTableSampleLength)
	));

	// get phasors
	const phasors = samples.map((wave) => FFT.fft(wave));

	// fourier coefficients
	let magnitudes = phasors.map((ph) =>
		FFT.util.fftMag(ph)
	);
	magnitudes = magnitudes.map((m) =>
		m.map((a: number) => Math.floor(a))
	);
	magnitudes = magnitudes.map((m) => normalize(m));

	// create wavetable
	const newWaveTable = magnitudes.map(m, i) => ({
		periodicWave: ctx.createPeriodicWave(
			m.map(() => 0),
			m
		),
		samples: samples[I],
		coefficients: m,
	}));

	// set waveTable and currentWave in the state
	store.dispatch(setWaveTable(newWaveTable));
	store.dispatch(
		setCurrentWave(newWaveTable[waveTablePosition])
	);
};
```

constructWaveForm

Diese Funktion wird jedes Mal aufgerufen, wenn das aktuelle Frequenzspektrum geändert wird. Mit einer Veränderung der Fourier-Koeffizienten geht auch eine Veränderung der gesamten Welle in jeder ihrer Ausführungen, das heißt eine Veränderung ihres Klanges sowie der grafischen Darstellung einher.

Auch hier lässt sich die Logik der Funktion in verschiedene Abschnitte unterteilen. Zunächst muss wieder mittels von *AudioConext.createPeriodicWave(real, imag)* ein Objekt des Typs *PeriodicWave* erzeugt werden, welches der Oszillator später für die Wiedergabe des entsprechenden Klanges verwenden kann. Anschließend muss eine umgekehrt ausgeführte Fourier-Transformation zur Berechnung der neuen Samples erfolgen. Durch die Fourier-Koeffizienten muss also auf die entsprechende Wellenform geschlossen werden. Ein Algorithmus hierfür kann durch zwei ineinander verschachtelter for-Schleifen realisiert werden, bei denen die äußere den Index der Samplekette von 0 bis 2048 durchläuft, während die innere die Koeffizienten bis zu einer bestimmten Auflösung (in diesem Fall 1024) hin nachvollzieht. Die einzelnen Werte können so mithilfe von *Math.sin()* berechnet werden. Auch hier müssen die Samples nach einem erfolgreichen Abschluss der Berechnungen normalisiert werden.[82]

Zu guter Letzt soll selbstverständlich auch hier die Eigenschaft *waveTable* im State auf den neuesten Stand gebracht werden und da am Wavetable Veränderungen vorgenommen wurden, ist die Funktion *setUpWaveForm()* ein weiteres Mal manuell aufzurufen.

[82] Es ist zu beachten, dass es immer verschiedene grafische Lösungen für die Erzeugung ein und desselben Klanges und somit korrespondierend zu dem selben Frequenzspektrum geben kann. Je nach verwendeten Algorithmus offenbaren sich die erzeugten Wellenformen in einer anderen Form.

```
const constructWaveForm = () => {
	if (currentWave) {

		// set periodicWave
		currentWave
			.periodicWave = ctx.createPeriodicWave(
				currentWave.coefficients.map(() => 0),
				currentWave.coefficients
			);

		// fourier transform to get the new samples
		let sample;
		for (let x = 0; x < 2048; x++) {
			sample = 0;
			for (let k = 0; k < 1024; k++) {
				sample += currentWave.coefficients[k]
				* Math.sin(
					(-2 * Math.PI * x * k) / 2048
				);
			}
			currentWave.samples[x] = sample;
		}
		currentWave.samples =
			normalize(currentWave.samples);
	}

	// set new waveTable
	if (waveTable && currentWave) {
		waveTable[waveTablwaveePosition] = currentWave;
		store.dispatch(setWaveTable(waveTable));
		setUpWaveForm();
	}
};
```

6 WEB AUDIO API

Das Projekt zeigt zu einem gewissen Grad, welche "Macht" in der Verwendung moderner Technologien steckt, um allein für den Browser entwickelte Anwendungen auch für einen professionellen Bereich nutzbar zu machen. Dank der *Web Audio API* entstand eine Option nun Werkzeuge und Programme für die kreative und effiziente Arbeit mit Ton zu implementieren.

Die Möglichkeit überhaupt erst Audio nativ im Web abspielen zu können, kam erst mit dem durch *HTML5* eingeführten Element *<audio>*, welches von allen modernen Browsern unterstützt wird.[83] Vor dieser Aktualisierung waren Nutzer stets auf Plugins von Drittanbietern wie *Flash* oder *Silverlight* angewiesen, welche expliziert installiert werden mussten und diverse Sicherheitslücken mit sich brachten.[84] Trotz dieser neuen Freiheiten war die Verwendung des *<audio>* Tags mit einigen Limitierungen verknüpft. Es war zum Beispiel unter anderem nicht möglich, das Timing präzise zu kontrollieren, Echtzeiteffekte anzuwenden und Klänge zu analysieren.[85] Um diesem Problem entgegenzuwirken, wurde eine Reihe verschiedener APIs entwickelt, wie etwa die *Audio Data API* von *Mozilla Firefox*, welche die Funktionalität des in *HTML5* neuen Elements erweitern sollte.[86] Die *Web Audio API* arbeitet im Kontrast zu dieser jedoch komplett separat vom *<audio>* Tag und wurde mit dem Ziel entwickelt, Funktionalität, welche sich auch in Spiele Engines oder professionellen digitalen Audioproduktionsprogrammen finden, zu bieten.[87] Heute findet das Produkt in einer ganzen Reihe von Browseranwendungen seinen Platz. *Soundtrap* beispielsweise ist eine umfangreiche Digital Audio Workstation für den Browser mit der Möglichkeit zur Midi und Audio Eingabe und Bearbeitung, Automationen und einem von der Firma *Antares*

[83] vgl. Smus, S. 1

[84] vgl. Benjamin

[85] vgl. Smus, S. 1-2

[86] siehe: https://wiki.mozilla.org/Audio_Data_API

[87] vgl. Smus, S. 2

entwickeltem Auto-Tune-Feature.[88] Eine Möglichkeit die *Web Audio API* auf andere Art kreativ zu nutzen ist der Beatmaker *The Audio Machine*, welcher ebenfalls durch ein Team von Studenten an der *Hochschule der Medien* im Rahmen einer Studioproduktion 2020 entwickelt wurde und auf den Servern der Hochschule[89], sowie unter einer eigenen Domain[90] verfügbar ist. Die Arbeit an diesem Projekt brachte jedoch auch eine der größten Schwächen der *Web Audio API* zu Tage, welche auf die nur unzureichend plattformübergreifende Kompatibilität zurückzuführen ist. So ist der von *Apple* betriebene und entwickelte Browser *Safari*, nur teilweise in der Lage die Funktionalität der neuen Technologie zu unterstützen, während *Microsofts Internetexplorer* die Verwendung des *AudioContexts* ganz ausschließt.[91] Bevor es daher weithin profitabel werden kann, professionelle Software für das Web zu entwickeln, erscheint es erforderlich, dass Firmen durch eine aktive Weiterentwicklung ihrer Browser schnell eine Kompatibilität mit den neuen Technologien bereitstellen. Dies gilt nicht nur für Systeme im Bereich Audio, sondern auch für Schnittstellen in anderen multimedialen Feldern.

6.1 Grundlagen APIs

Bevor auf die prinzipielle Funktions- und Anwendungsweise der *Web Audio API* eingegangen werden soll, ist zunächst zu erörtern, was genau eine API ausmacht und wie sie sich definiert. API steht für *application programming interface* und wird im allgemeinen als Bezeichnung für eine Programmierschnittstelle verwendet. Kurzgesagt: eine API liefert Daten oder Funktionen, um die Interaktion verschiedener Computerprogramme miteinander zu erleichtern.[92] Mit einer API können also beispielsweise Daten von einer Quelle abgefragt werden. Möchte man als Host einer Webseite etwa Informationen zu der aktuellen meteorologischen Wettervorhersage bereitstellen, so kann man über die APIs der jeweiligen Firmen diese Daten abrufen.[93] Anhand dieses Beispiels offenbart sich auch die

---

[88] siehe: https://www.soundtrap.com

[89] siehe: http://im-prod.hdm-stuttgart.de/~TheAudioMachine/

[90] siehe: https://www.theaudiomachine.com/

[91] vgl. o.V. MDN web docs “Web Audio API”

[92] vgl. Massé, S. 5

[93] siehe: [1] http://wetterapi.metgis.com/?gclid=CjwKCAiAt9z-BRBCEiwA_bWv-HjwXK0AQSs1QiVugtIPFG2NBrsZSQr6L1D9d1iJOOMl3NPx_FheZBoCfs4QAvD_BwE
[2] https://www.predicthq.com/events/severe-weather?utm_campaign=Experimental&utm_term=%2Bweather%20%2Bapi&utm_source=adwords&utm_medium=ppc&hsa_net=adwords&hsa_ver=3&hsa_grp=111260510804&hsa_tgt=kwd-368383759597&hsa_kw=%2Bweather%20%2Bapi&hsa_cam=1082143176&hsa_ad=466866928708&hsa_src=g&hsa_acc=4773278510&hsa_mt=b&gclid=CjwKCAiAt9z-BRBCEiwA_bWv-LLmfRdDATcOD39Su8w8QxNaOgXEeqV5Y_VlZT8W67ex_kk3ys1ahBoCg_0QAvD_BwE
[3] https://openweathermap.org/api

Daseinsberechtigung dieses Design Patterns. Man nutzt APIs, um Daten mit spezialisierten Zulieferern, welche spezielle Probleme lösen können, auszutauschen, sodass man als Entwickler nicht von Neuem Lösungen für diese Probleme finden muss.[94] Natürlich gehen die Fähigkeiten von APIs weit über die Vermittlung von meteorologischen Daten hinaus. Auch die Software zum Betreiben der Soundkarte eines Computers stellt eine API zur Verfügung, sodass Programmierer dieser Rechnerkomponente leicht Anweisungen erteilen können. Ähnlich verhält es sich mit der *Web Audio API*. Der Quelltext dieses Systems implementiert alle wichtigen Funktionen, welche benötigt werden um Audio durch den Browser laden, bearbeiten und abspielen zu können. Die dahinterliegende Komplexität bleibt dem Nutzer allerdings verborgen, stattdessen nutzt er nur einige wenige an der Oberfläche bereitgestellte Funktionen, er kommuniziert also nur über ein festdefiniertes Interface, um gezielt Verhaltensweisen festzulegen. APIs entscheiden sich nur geringfügig von herkömmlichen Libraries, jedoch vor allem in dem Punkt, dass eine API mit einem bereits laufenden Programm kommuniziert, während eine Library hingegen nur einen wiederverwertbaren Block Code zur Verfügung stellt.[95]

6.2 Funktionsweise

Die *Web Audio API* ist auf eine Art und Weise aufgebaut, die alle Audio-Operationen innerhalb eines eigenen Audio-Kontextes bearbeitet. Dabei erlaubt sie modulares Routing.[96] Der Audio Kontext ist in diesem Zusammenhang nichts weiter als ein Graph, bestehend aus mehreren Komponenten, sogenannter Audio Nodes, welcher definiert, wie Audio von einer Quelle zu einem Ausgang fließt.[97] Jedes mal, wenn dieser Fluss durch eine bestimmte Node fließt, können die Eigenschaften des Audios verändert oder eingesehen werden. Ebenso können Nodes für das Mixen verschiedener Audioströme zuständig sein. Um die *Web Audio API* zu nutzen, muss zunächst ein Audio-Kontext initialisiert werden, was durch die bereits beschriebene Erzeugung des entsprechenden Objekts *AudioContext* geschehen kann. Durch Aufrufe der von diesem Objekt bereitgestellten Funktionen können verschiedene Nodes erzeugt werden, welche in vier Kategorien unterteilt werden können.

---

[94] vgl. Jin & Sahni & Shevat, S. 2

[95] vgl. o.V. RapidAPI “API vs Library (What’s the Difference?)”

[96] vgl. o.V. MDN web docs “Web Audio API”

[97] vgl. Smus, S. 3

1. Source Nodes, bei denen es sich um Quellen für Audio wie Audio Buffern und Live Inputs handelt, ebenso wie die Oszillator Node
2. Modification Nodes, wie zum Beispiel Filter, Panner und andere Effekte
3. Analysis Nodes, zur Analyse von Signalen auch auf ihre Frequenzanteile
4. Destination Nodes, zu denen Audio hingeleitet werden kann, um es beispielsweise über die mit dem Computer verbundene Anlage abzuhören oder aufzeichnen zu können.[98]

Im einfachsten Fall wird eine einzelne Quelle direkt zum Ausgang geroutet.[99]

Eine weitere Besonderheit und Stärke in der Funktionsweise der *Web Audio API*, durch die sie sich auch von dem Klassischen *HTML5 <audio>* Element abhebt, ist ihr Umgang mit Timing, welches durch hohe Präzision und geringe Latenz kontrolliert wird.[100]

6.3 Alternativen

Natürlich ist die *Web Audio API* nicht die einzige Möglichkeit, um in der Umgebung eines Browsers mit Tönen umzugehen. OpenSource Libraries wie *p5*[101] und *Tone.js*[102] implementieren ähnliche Funktionalität für die einfache Erstellung interaktiver Audio Software mit *Javascript*. *Tone.js* hat im Gegensatz zu der *Web Audio API*[103] sogar denn Vorteil globalen Transport für Synchronisation und Planung von Events zu ermöglichen.[104] Zudem ist diese Library vor allem dafür gemacht, eigene Synthesizer, Effekte und komplexe Kontrollsignale zu erstellen, während die *Web Audio API* den Begriff Audio etwas allgemeiner fasst und in erster Linie versucht, bei dem interaktiven Abrufen von Audioinhalten auftretende Probleme in den Griff zu bekommen, um so Konzepte wie *Next Generation Audio* (*NGA*) zu ermöglichen.

p5 ist eine in erster Linie zur Erstellung von Grafiken ausgelegte Bibliothek und wird erst durch die Erweiterung *p5.sound* zu einem Werkzeugkasten für digitale Klangerzeugung.[105]

---

[98] vgl. Smus, S. 5

[99] vgl. Adenot & Choi, “Modular Routing”

[100] vgl. o.V. MDN web docs “Web Audio API”

[101] siehe: [1] https://p5js.org
[2] https://github.com/p5py/p5

[102] siehe: https://tonejs.github.io

[103] vgl. Benjamin

[104] vgl. o.V. Tone.js “Tone.js”

[105] siehe: https://p5js.org/reference/#/libraries/p5.sound

Dafür wird hiermit ebenfalls durch Objekte wie *Soundfile*, *SoundRecorder* und *Oscillator* Funktionalität für die Eingabe, Wiedergabe und Analyse von Tönen bereitgestellt.
Es sei also gesagt, dass bei der Erstellung hochwertiger Programme für die Arbeit mit Audio im Browser keineswegs die *Web Audio API* das einzige Mittel der Wahl darstellt, sondern auch auf Libraries und Programme diverser Drittanbieter zurückgegriffen werden kann.

7 RESÜMEE

Diese Arbeit soll mit einer abschließenden Bewertung der aktuell zur Verfügung stehenden Möglichkeiten für die Schaffung hilfreicher, unterhaltender oder sogar professioneller Programme zur Arbeit mit Klängen und Geräuschen enden. Die Wege, die sich Dank moderner Technologien für das Web offenbaren, haben längst einen Punkt überschritten, bei dem Entwickler nahezu keine Grenzen in Punkto Umsetzung der eigenen kreativen Ideen gesetzt sind. Gerade Programme wie *Soundtrap* belegen, wie weit rein auf dem Browser basierende Anwendungen gehen können. Auch im Synthesizerbereich kann das Web durch Sammlungen wie die, auf der Webseite der Firma *Playtronica* zur Verfügung gestellte, punkten.[106] Die in diesem Verzeichnis auftauchenden Applikationen stellen ein grobes Bild dar, welche Tools zur Klangerzeugung bereits im Browser genutzt werden können. Für die dem entgegengesetzte Menge an Menschen, welche sich bereits mit *JavaScript* befassen, aber Anwendungen für den Desktop schreiben wollen, bietet *Node.js* nun sogar eine Möglichkeit diese Engine aus seiner gewohnten Umgebung zu nehmen. Somit ist der Fall klar: Der kreativen Implementierung von Audiosoftware scheinen aktuell keine Grenzen gesetzt zu sein.

Dennoch ist die kritische Frage zu stellen, inwieweit derlei Anwendungen überhaupt im Netz benötigt werden, schließlich bieten heutige Digital Audio Workstations wie *Logic Pro X*, *Ableton Live* und *ProTools Ultimate* durch die beinahe unbegrenzten Erweiterungsmöglichkeiten bereits alle nötigen Werkzeuge zur Bearbeitung und Erzeugung von Klang für den professionellen und kreativen Bereich. Wie bereits gesagt wurde, kann die Einbindung solcher Workstations in den Browser die Konnektivität und Kooperation verschiedener Toningenieure fördern, wodurch eine Verschiebung der Frage vom allgemeinen Nutzen hin zur Frage nach der Vorgehensweise für die sinnvolle Einbindung webbasierter Technologien in den herkömmlichen Workflow sinnvoll erscheint, um die aktuell erschließbare Arbeitsumgebung zu erweitern.

---

[106] siehe: https://synth.playtronica.com

Die Hohe Nachfrage an einsteigerfreundlichen Musikprogrammen rechtfertigt schon jetzt die Entwicklung solcher einfach zu bedienenden und für die Allgemeinheit zugänglichen Plattformen rein wirtschaftlich. Doch selbst bei Vernachlässigung dieses Kriteriums lohnt sich die Zeit, welche mit der Entwicklung solcher Software aufgebracht wird, für die Erschließung neuer Möglichkeiten und Wege für eine fortschrittliche Zukunft der Klangerzeugung.

VERZEICHNIS MATHEMATISCHER FORMELN

QUELLENVERZEICHNIS

Ackermann, Phillip (1991): ***Computer und Musik, eine Einführung in die digitale Klang- und Musikverarbeitung;*** Springer-Verlag, Wien

Adenot, Paul & Choi, Hongchan u.A. (2020): ***Web Audio API, W3C Candidate Recommendation, 11 June 2020;*** W3C, (MIT, ERCIM, Keio, Beihang); https://www.w3.org/TR/webaudio/#dom-audiocontextstate-suspended (Abruf: 19.11.2020)

Anwander, Florian (2000): ***Synthesizer, so funktioniert elektronische Klangerzeug-ung,*** 9. Auflage PPVMEDIEN GmbH, Bergkirchen

o.V. Audio Engineering Society (2004): ***117th AES CONVENTION AN AFTERNOON WITH BOB MOOG - BIOGRAPHIES;*** https://www.aes.org/events/117/specialevents/bios.cfm (Abruf: 05.11.2020)

Benjamin, netzwerkbewegung.com (2015): ***Web Audio API;*** https://www.netzbewegung.com/de/lab/web-audio-api/ (Abruf: 12.12.2020)

Bierman, Gavin & Abadi, Martin & Torgersen, Mads (2014): ***Understanding Tybescript;*** Springer Verlag, Berlin Heidelberg; https://link.springer.com/chapter/10.1007/978-3-662-44202-9_11 (Abruf: 14.11.2020)

Bläsig, Bettina (16.09.2006): ***Die ältesten Musikinstrumente;*** Welt, Berlin; https://www.welt.de/print-welt/article152896/Die-aeltesten-Musikinstrumente.html (Abruf: 05.11.2020)

Brunton, Steven (2016): ***ME565 Lecture 17: Fast Fourier transforms (FFT) and Audio;*** https://www.youtube.com/watch?v=4d6EeRJZLbo&t=1971s (Abruf: 09.12.2020)

Brunton, Steven L. & Kutz, J.Nathan (2017): ***Data Driven Science & Engineering, Machine Learning, Dynamical Systems, and Control;*** Copyright © Brunton & Kutz; http://databookuw.com/databook.pdf (Abruf: 09.12.2020)

Crombie, David (1984): ***The Complete Synthesizer: A Comprehensive Guide;*** Omnibus Press, London; https://exellon.net/book/The_Complete_Synthesizer.pdf (Abruf: 20.11.2020)

Deutsch, Ralph & Deutsch, Leslie J. (1976): ***Appl. No.: 652,217;*** Los Angeles: United States Patent; https://patents.google.com/patent/US4079650A/en (Abruf: 13.11.2020)

Dutilleux, P. & Holgers, M. & Disch, S. & Zölzer, U. (2011): ***Filters and Delays;*** in: DAFX: Digital Audio Effects, Second Edition John Wiley & Sons Ltd, Großbritannien

Durmus, Nadir (o.J): ***Analaoge Klangsynthese - Eine Einführung in die Programmierung von Synthesizern;*** http://www.analogeklangsynthese.de/index.html (Abruf: 20.11.2020)

Gorges, Peter & Merck, Alex (1989): ***Keyboards Midi Homerecording - Alles über Equipment und Anwendungen;*** Gunther Carstensen Verlag, München

von Grünigen, Daniel (2008): ***Digitale Signalverarbeitung mit einer Einführung in die kontinuierlichen Signale und Systeme;*** 4. Auflage Carls Hanser Verlag, München

Heideman, Michael T. & Johnson, Don H. & Burrus, C. Sidney (1985): ***Gauss and the History of the Fast Fourier Transform;*** Rice Univesity, Houston; https://www.researchgate.net/publication/226049108_Gauss_and_the_history_of_the_fast_Fourier_transform (Abruf: 09.12.2020)

Jin, Brenda & Sahni, Saurabh & Shevat, Armir (2018): ***Designing Web APIs; O'Reilly Media, Sebastopol;*** https://www.pdfdrive.com/designing-web-apis-building-apis-that-developers-love-d176234255.html (Abruf: 14.12.2020)

Massé, Mark (2012): ***REST API Design Rulebook; O'Reilly Media, Sebastopol;*** https://pepa.holla.cz/wp-content/uploads/2016/01/REST-API-Design-Rulebook.pdf (Abruf: 14.12.2020)

o.V. MDN web docs (2020): https://developer.mozilla.org/en-US/ (Abruf: 23.11.2020)

o.V. Narodni Muzej Slovenje (o.J.): ***Neanderthal flute;*** https://www.nms.si/en/collections/highlights/343-Neanderthal-flute (Abruf: 05.11.2020)

Osgood, Brad (o.J.): ***Lecture Notes for EE 261, The Fourier transform and its Applications;*** Electrical Engineering Department, Stanford University; https://see.stanford.edu/materials/lsoftaee261/book-fall-07.pdf (Abruf: 18.11.2020)

Randall, R. B. & Tech, B. (1977): ***Application of B & K Equipment to Frequency Analysis;*** Second Edition Brüel & Kjær, Nærum

o.V. RapidAPI (2020): ***API vs Library (What's the Difference?);*** https://rapidapi.com/blog/api-vs-library/ (Abruf: 14.12.2020)

Roads, Curtis (1996): ***The Computer Music Tutorial;*** The MIT Press, Cambridge

Rogozinsky, Gleb & Goryachev, Nickolay (2019): ***Modeling of Yamaha TX81Z FM Synthesizer in Csound;*** The Bonch-Bruevich St.Petersburg State University of Telecommunications; https://csound.com/icsc2019/proceedings/3.pdf (Abruf: 05.11.2020)

Smith, Julius Oliver (2010): ***Physical Audio Signal Processing;*** Center for Computer Research in Music and Acoustics (CCRMA), Stanford University; https://ccrma.stanford.edu/~jos/pasp/ (Abruf: 13.11.2020)

Smus, Boris (2013): ***Web Audio API; O'Reilly Media, Sebastopol;*** https://webaudioapi.com/book/Web_Audio_API_Boris_Smus.pdf (Abruf: 12.12.2020)

Springer, Sebastian (2020): ***React - Das umfassende Handbuch;*** 1. Auflage Rheinwerk Verlag, Bonn

Stange-Elbe, Joachim (2015): ***Computer und Musik - Grundlagen, Technologien und Produktionsumgebung der digitalen Musik;*** Walter de Gruyter GmbH, Berlin/Boston

Steppat, Michael (2014): ***Audioprogrammierung - Klangsynthese, Bearbeitung, Sounddesign;*** Carl Hanser Verlag, München

o.V. Tone.js (o.J): ***Tone.js;*** https://tonejs.github.io (Abruf: 14.12.2020)

Webers, Johannes (2007): ***Handbuch der Tonstudiotechnik für Film, Funk und Fernsehen, Digitales und analoges Audio Recording;*** 9. Auflage Franzis Verlag GmbH, Poing

Werner, Martin (2000): ***Signale und Systeme;*** 3. Auflage 2008, Vieweg + Teubner | GWV Fachverlage GmbH, Wiesbaden; https://link.springer.com/book/10.1007/978-3-8348-9523-3 (Abruf: 10.12.2020)

ANHANG

Audio im Browser - Ressourcen und Quellen zum Einstieg

Diese Liste enthält einen Überblick der Technologien, welche für die Durchführung der mit der Arbeit verbundenen Projekte genutzt wurden. Dabei kann sie als eine Art Einstiegshilfe in die Welt der Audioentwicklung für das Web verstanden werden und soll ein Gefühl dafür vermitteln, welche verschiedenen Fähigkeiten bei Tätigkeiten in diesem Bereich von Bedeutung sind. Die Auflistung enthält außerdem eine Auswahl nützlicher Links, welche zu Dokumentationen, Blogs und YouTube Kanälen führen, die sich allgemeinen mit entsprechenden Themen befassen.

Grundlagen:

1. HTML
   (i) https://html.spec.whatwg.org/#toc-introduction
   (ii) https://developer.mozilla.org/de/docs/Web/HTML
   (iii) https://www.w3schools.com/html/
2. Javascript
   (i) https://javascript.info
   (ii) https://developer.mozilla.org/de/docs/Web/JavaScript
   (iii) https://www.youtube.com/channel/UCFbNIlppjAuEX4znoulh0Cw
   (iv) https://www.youtube.com/user/shiffman
   (v) https://www.pdfdrive.com/javascript-books.html
3. Typescript
   (i) https://www.typescriptlang.org
   (ii) https://www.youtube.com/channel/UCW5YeuERMmlnqo4oq8vwUpg
   (iii) https://www.pdfdrive.com/learning-typescript-e43087833.html

User Interfaces

1. React
   (i) https://reactjs.org
   (ii) https://www.pdfdrive.com/react-books.html
   (iii) https://reactjs.de/artikel/react-tutorial-deutsch/
   (iv) https://www.youtube.com/channel/UC80PWRj_ZU8Zu0HSMNVwKWw

(v) https://www.youtube.com/c/TheMorpheus407/playlists

2. Konva
   (i) https://konvajs.org/docs/react/index.html
   (ii) https://www.youtube.com/user/RockyDeRaze

State Management

1. Redux
   (i) https://redux.js.org
   (ii) https://github.com/reduxjs/redux
   (iii) https://www.youtube.com/channel/UC80PWRj_ZU8Zu0HSMNVwKWw
   (iv) https://www.youtube.com/channel/UCWv7vMbMWH4-V0ZXdmDpPBA
2. Redux Thunk
   (i) https://www.npmjs.com/package/redux-thunk
   (ii) https://github.com/reduxjs/redux-thunk
   (iii) https://www.digitalocean.com/community/tutorials/redux-redux-thunk-de
3. Redux Watch
   (i) https://www.npmjs.com/package/redux-watch
   (ii) https://github.com/ExodusMovement/redux-watch#readme

Web Audio

1. Web Audio API
   (i) https://www.w3.org/TR/webaudio/
   (ii) https://webaudioapi.com/book/Web_Audio_API_Boris_Smus.pdf
   (iii) https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API
   (iv) https://www.youtube.com/watch?v=nCVoHy2Zk4c
2. ToneJS
   (i) https://tonejs.github.io
   (ii) https://www.youtube.com/channel/UCpKb02FsH4WH4X_2xhIoJ1A
3. p5
   (i) https://p5js.org
   (ii) https://github.com/processing/p5.js?files=1
   (iii) https://www.youtube.com/channel/UCvjgXvBlbQiydffZU7m1_aw
4. fft-js
   (i) https://www.npmjs.com/package/fft-js

(ii) https://github.com/vail-systems/node-fft

(iii) https://www.youtube.com/channel/UCm5mt-A4w61lknZ9lCsZtBw

Nützlich

1. https://stackoverflow.com
2. https://www.pdfdrive.com
3. https://developer.mozilla.org/de/
4. https://www.youtube.com/channel/UC8butISFwT-Wl7EV0hUK0BQ