Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Eugen Winter

Analyse und Reimplementierung einer plattformunabhängigen Software für das 3D-Audio-Rendering

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Eugen Winter

Analyse und Reimplementierung einer plattformunabhängigen Software für das 3D-Audio-Rendering

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Wolfgang Fohl
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 16. Dezember 2016

Eugen Winter

Thema der Arbeit
Analyse und Reimplementierung einer plattformunabhängigen Software für das 3D-Audio-Rendering

Stichworte
Wellenfeldsynthese, WONDER, CoRGII, Latenz, liblo, OSC, JACK

Kurzzusammenfassung
In dieser Abschlussarbeit wird die Software WONDER, die im Wellenfeldsynthese-Labor der HAW Hamburg zum Einsatz kommt, analysiert. Der Schwerpunkt für die Analyse liegt dabei auf der Ergründung der Ursachen einer übermäßigen Latenz und positionsabhängigen Amplitudenschwankungen. Anhand der gewonnenen Ergebnisse soll bewertet werden, ob eine Modifikation der bestehenden Software oder eine komplette Reimplementierung in Frage. Wenn nötig soll diese durchgeführt werden.
Eugen Winter

Title of the paper
Analysis and reimplementation of a cross-platform software for 3D audio rendering

Keywords
Wave field synthesis, WONDER, CoRGII, Latency, liblo, OSC, JACK

Abstract
In this thesis the software WONDER that is used in the wave field synthesis lab at the Hamburg University of Applied Sciences shall be analysed to find the reasons for a excessive delay and position dependant variations in the amplitude. The results of this analysis shall be used to decide whether it is usefull to modify the current software or to code it from scratch and if so perform the necessary steps.

Inhaltsverzeichnis

1 Einleitung

Die Wellenfeldsynthese stellt ein besonderes Verfahren zur Wiedergabe von Klangquellen dar. Statt wie bei herkömmlichen Lautsprecher-Aufstellungen kanalbasiert zu arbeiten, wird bei der Wellenfeldsynthese ein objektbasiertes Audio-Rendering eingesetzt, welches nicht nur in der Lage ist die Klangquellen bei der Wiedergabe aus einem bestimmten Lautsprecher auszugeben, sondern alle Lautsprecher dafür benutzt das physikalische Schallfeld der Klangquelle in Abhängigkeit ihrer Position korrekt zu synthetisieren.

Ein Grund, warum sich das Verfahren bislang nicht in der breiten Masse durchgesetzt hat, ist die Anforderung für eine korrekte Synthese besonders viele Lautsprecher einzusetzen. Damit verbunden ist die steigende Rechenlast, die benötigt wird, um alle Audiosignale in Echtzeit zu berechnen und korrekt auf den Lautsprechern auszugeben. Beide Faktoren sind eine Ursache dafür, dass sich Software, die sich mit dem Thema Wellenfeldsynthese beschäftigt, wenig verbreitet ist und meist nur in Forschungseinrichtungen oder an Hochschulen vorzufinden ist.

1.1 Ziel und Abgrenzung dieser Arbeit

Die Hochschule für Angewandte Wissenschaften Hamburg ist seit 2011 im Besitz einer Wellenfeldsynthese-Anlage. In einem Labor wird über ein verteiltes Computersystem die Wellenfeldsynthese für insgesamt 208 Kanäle berechnet. Dabei kommt es zu positionsabhängigen Amplitudenschwankungen sowie einer übermäßigen Latenz bei der Wiedergabe. Im Rahmen dieser Bachelorarbeit soll analysiert werden, welche Gründe für diese Anomalien verantwortlich sind und wie sie mit der Software zusammenhängen. Notwendige Korrekturen an der Software sollen gegen eine komplette Reimplementierung abgewägt werden. Analysiert wird dabei nur der Bereich innerhalb der Software, welcher für die Berechnungen der Wellenfeldsynthese verantwortlich ist.

1.2 Aufbau dieser Arbeit

Die Analyse der Software führt über die an der Wellenfeldsynthese beteiligten Softwarekomponenten und die verwendeten Bibliotheken. Es wird geprüft wie sich der Startvorgang der

Anlage auf das Verhalten auswirkt und ob die Bibliotheken korrekt implementiert worden sind. Die Latenz wird gemessen und mit vorangegangenen Ergebnissen verglichen. Die gesammelten Erkenntnisse und die damit verbundenen Korrekturen werden in Relation zu einer kompletten Reimplementierung der Software gesetzt. Mit der Entscheidung einer Reimplementierung wird, so weit im Rahmen dieser Bachelorarbeit möglich, eine neue verteilte Software für das Rendering der Wellenfeldsynthese im Labor der HAW implementiert.

2 Grundlagen

Dieses Kapitel gibt zunächst einen kompakten Einblick in die physikalischen Grundlagen des Verfahrens der Wellenfeldsynthese (im Folgenden auch kurz WFS genannt). Auf die formale Herleitung der Gleichungen zur Berechnung der Wellenfeldsynthese wird dabei bewusst verzichtet. Betrachtet werden sollen nur die für eine Software-Implementierung relevanten Ansteuerungsfunktionen in der Zeitdomäne. Eine detaillierte Herleitung ist unter anderem in [Ber88], [Sta97], [Ver97], [Ahr12], [Old13], [Wie14] oder [Sch16] zu finden. Am Ende des Kapitels werden kurz die Probleme und Grenzen der Wellenfeldsynthese beleuchtet und wie man diesen begegnen kann.

2.1 Grundprinzip der Wellenfeldsynthese

Die Physik hinter der Wellenfeldsynthese wird durch das Huygenssche Prinzip beschrieben, welches durch den gleichnamigen, aus den Niederlanden stammenden Astronomen, Mathematiker und Physiker Christiaan Huygens beobachtet und formuliert worden ist. Dieses besagt, dass jeder Punkt einer Wellenfront als Entstehungspunkt einer neuen Welle betrachtet werden kann. Die neue Welle wird als Elementarwelle bezeichnet. Überlagern sich all diese Elementarwellen, dann bilden sie erneut die anfängliche Wellenfront (siehe Abbildung 2.1). Dabei bleiben die ursprüngliche Frequenz und Phase erhalten.

Auf die Wellenfront einer Klangquelle angewendet, lässt sich auch diese durch die Summe einzelner Elementarwellen rekonstruieren, wobei für deren Erzeugung Lautsprecher zum Einsatz kommen. Das Verfahren für die Wiedergabe solcher „virtuellen“ Klangquellen unter Zuhilfenahme einer Reihe von Lautsprechern wurde 1988 durch Berkhout an der TU Delft entwickelt [Ber88]. Dabei werden die verwendeten Lautsprecher jeweils soweit verzögert, dass deren Wiedergabe exakt in dem Moment stattfindet, in dem auch die Wellenfront der virtuellen Klangquelle die Koordinate des Lautsprechers passieren würde. Gleichzeitig findet eine Gewichtung der jeweiligen Lautstärke statt. Diese ist abhängig vom Einfallswinkel der Wellenfront auf den einzelnen Lautsprecher. In ihrer Summe bilden sie so die virtuelle Klangquelle ab.

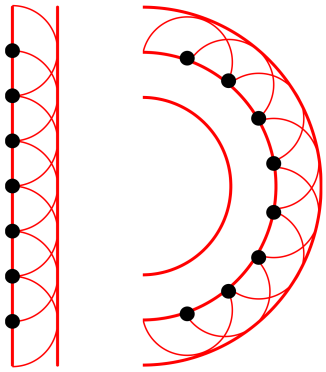

Abbildung 2.1: Huygenssches Prinzip am Beispiel einer ebenen und kreisförmigen Wellenfront

Als Basis für die Berechnungen der Wellenfeldsynthese dient zunächst das Kirchhoff-Helmholtz-Integral. Dieses besagt, dass innerhalb eines Volumens, welches frei von Quellen ist, für jeden Punkt der Schalldruck ermittelt werden kann, wenn der Schalldruck und die Schallschnelle aller Oberflächenpunkte des Volumens bekannt sind [Sta97, Kapitel 2.2].

Bei der tatsächlichen Realisierung einer Wellenfeldsynthese-Anlage muss diese Formulierung jedoch vereinfacht werden. Zum einen, weil nicht unendlich viele und unendlich kleine Lautsprecher zur Verfügung stehen, um jeden Punkt auf der Oberfläche eines Volumens abzubilden. Zum anderen wird meist aufgrund von räumlichen Gegebenheiten auf eine Lösung mit zweidimensionalen Lautsprecher-Arrays gesetzt.

Das Kirchhoff-Helmholtz-Integral wird daher mittels Green'scher Funktion und Rayleigh-Integral [Ahr12, Kapitel 2.4 und 2.5] soweit vereinfacht, dass eine Gleichung entsteht, bei der die Oberflächenpunkte eines Volumens durch eine diskrete Anzahl von vertikalen, monopolen Linienquellen ersetzt werden, die Druck in Richtung einer Ebene ausüben. Statt den Schalldruck eines Punktes innerhalb eines quellfreien Volumens zu bestimmen, wird dieser nun für einen Punkt in einer Ebene bestimmt, die dem Zuhörerraum entspricht. An die Stelle der monopolen Linienquellen treten Lautsprecher, deren Schallwellen durch Überlagerungen nach dem Superpositionsprinzip den Schalldruck im Bereich des Zuhörerraumes verändern können. Die Umformungen und Vereinfachungen führen zur Aufstellung einer Gleichung, die es unter Berücksichtigung der Lautsprecher- und Klangquellenpositionen ermöglicht, mittels Gewichtung und Verzögerung einzelner Lautsprecher, bestimmte Arten virtueller Klangquellen (siehe Unterabschnitt 2.2.3, Unterabschnitt 2.2.4 und Unterabschnitt 2.2.5) im Zuhörerraum zu synthetisieren. Dieses entspricht der 2D-Wellenfeldsynthese.

2.2 Wellenfeldsynthese in 2.5D

Ein geschlossener Lautsprecher weist die dreidimensionale Abstrahlcharakteristik einer monopolen Punktquelle mit kugelförmiger Ausbreitung auf. Damit entspricht er nicht den vertikalen, monopolen Linienquellen, die bei der Herleitung der 2D-Wellenfeldsynthese angenommen werden. Um dieser zusätzlichen Dimension Rechnung zu tragen, wurde die 2.5D-Wellenfeldsynthese formuliert, bei deren Herleitung von einer monopolen Punktquelle ausgegangen wird, um die Abstrahlcharakteristik eines Lautsprechers widerzuspiegeln [Sta97, Kapitel 3.1]. Ein daraus resultierender Nachteil ist jedoch, dass die Amplitude nur noch für einen bestimmten Punkt x_{ref} (siehe Abbildung 2.3) im Zuhörerraum korrekt berechnet werden kann. Die restlichen Punkte weisen lediglich eine korrekte Phase auf [Ver97, Kapitel 2.3].

2.2.1 Koordinatensystem

Das in dieser Bachelorarbeit verwendete Koordinatensystem ist in Abbildung 2.2 zu sehen. Punkte können durch die Länge eines Vektors $\vec{v}$, dessen Winkel ϕ in der Azimut-Ebene, sowie seiner Steigung θ auf der z-Achse und der daraus resultierenden Höhe bestimmt werden. Die Winkel in der Azimut-Ebene verlaufen gegen den Uhrzeigersinn und sind, wie auch die Höhe auf der z-Achse, positiv.

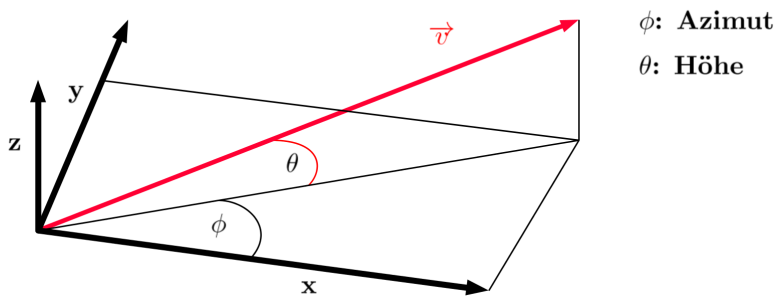

Abbildung 2.2: Verwendetes Koordinatensystem

2.2.2 Raumkoordinaten

Den Ursprung für die Raumkoordinaten bildet der Mittelpunkt des Zuhörerraumes und wird mit x_{ref} bezeichnet (siehe Abbildung 2.3). Dieser Punkt entspricht ebenfalls der optimalen Position eines Zuhörers bei der 2.5D-Wellenfeldsynthese und weist während der Wiedergabe von virtuellen Klangquellen die korrekte Amplitude auf.

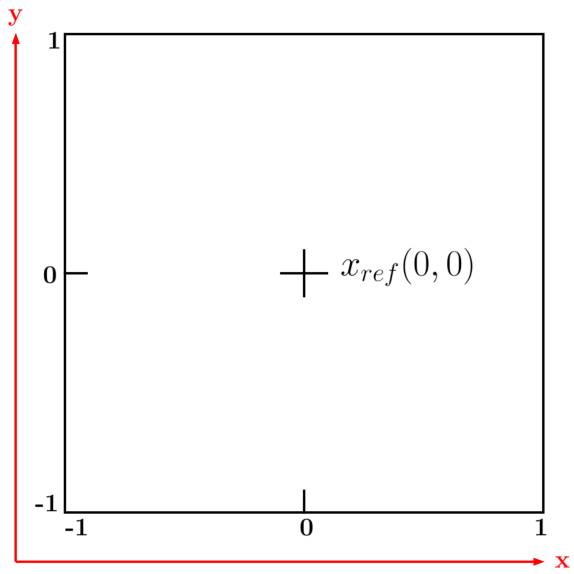

Abbildung 2.3: Referenzpunkt x_{ref} des Zuhörers im Zuhörerraum

2.2.3 Linearquelle als virtuelle Klangquelle

Eine Linearquelle (im Englischen *plane wave* genannt) zeichnet sich durch ein Klangbild aus, welches an jeder Position des Zuhörerraumes identisch klingt und den Bewegungen des Zuhörers folgt. Einzig der Winkel ihrer Richtung kann als Ursprung für die Ortung herangezogen werden. Damit eignet sich die Linearquelle besonders gut, um sehr weit entfernte, virtuelle Klangquellen abzubilden. Erreicht wird die Synthese des Schallfeldes durch die Vorgabe eines Winkels für den Ursprung der Linearquelle und einer korrekt aufeinander folgenden Verzögerung der einzelnen Lautsprecher (siehe Abbildung 2.4).

Ansteuerungsfunktion für die 2.5D-WFS einer Linearquelle

$$d_{2.5D_{pw}}(x_0,t) = \underbrace{2 \cdot g_0 \cdot a(t) \cdot h_{2.5D}(t) \cdot \langle n_k, n_{x_0} \rangle \cdot w(x_0)}_{\text{Amplituden-Gewichtung}} \cdot \underbrace{\delta\left(t - \frac{\langle n_k, n_{x_0} \rangle}{c}\right)}_{\text{Verzögerung}} \tag{2.1}$$

g_0 : 2.5D-WFS-Amplituden-Korrekturfaktor für den Punkt x_{ref} [Wie14, Gleichung 2.29]

$$g_0 = \sqrt{2\pi \cdot ||x_{ref} - x_0||}$$

x_{ref} : Koordinate für den Mittelpunkt des Zuhörerraumes / Position des Zuhörers (in Meter)

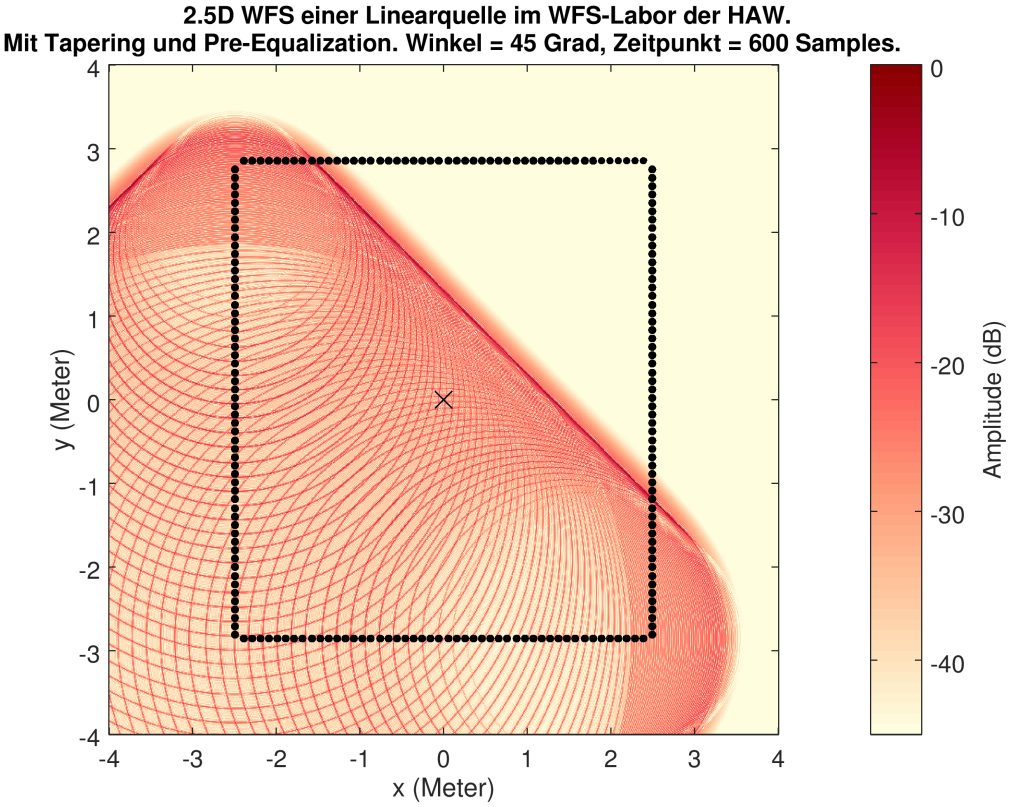

Abbildung 2.4: Simulation von 2.5D-WFS für eine Linearquelle in der Zeitdomäne

x_0 : Koordinate des aktuellen Lautsprechers (in Meter)

t : Aktueller Zeitpunkt (in Sekunden)

$a(t)$: Aktuelles Audiosignal

$h_{2.5D}(t)$: Vorfilterung des Audiosignals (im Englischen *pre-equalization filter* genannt)

$$h_{2.5D}(t) = \mathcal{F}^{-1}\left\{\sqrt{i\frac{\omega}{c}}\right\}$$

n_k : Normalenvektor (Richtung) der Linearquelle

n_{x_0} : Normalenvektor (Richtung) des aktuellen Lautsprechers

$\langle n_k, n_{x_0} \rangle$: Skalarprodukt von n_k und n_{x_0}

c : Schallgeschwindigkeit (in Meter pro Sekunde)

$w(x_0)$: Fensterfunktion für die Aktivierung des aktuellen Lautsprechers [SRA08]

$$w(x_0) = \begin{cases} 1 & \text{falls } \langle n_k, n_{x_0} \rangle > 0 \\ 0 & \text{sonst.} \end{cases}$$

Ein Lautsprecher ist nur dann aktiv, wenn das Skalarprodukt der Vektoren n_k und n_{x_0} größer ist als 0, also der von ihnen eingeschlossene Winkel $\cos(\alpha)$ kleiner ist als 90°.

Siehe dazu [Wie14, Gleichung 2.57].

2.2.4 Punktquelle als virtuelle Klangquelle

Bei der Punktquelle (im Englischen *point source* genannt) handelt es sich um eine außerhalb des Zuhörerraumes befindliche virtuelle Klangquelle mit kreisförmiger Ausbreitung. Diese behält, wie man es von einer realen Klangquelle erwarten würde, ihre Position bei, wenn sich der Zuhörer im Zuhörerraum bewegt. Damit ist sie im Gegensatz zur Linearquelle genau lokalisierbar und die Entfernung kann anhand ihrer Amplitude bestimmt werden (siehe Abbildung 2.5).

Ansteuerungsfunktion für die 2.5D-WFS einer Punktquelle

$$d_{2.5D_{ps}}(x_0, t) = \underbrace{\frac{1}{\sqrt{2\pi}} \cdot g_0 \cdot a(t) \cdot h_{2.5D}(t) \cdot \frac{\langle (x_0 - x_s), n_{x_0} \rangle}{||x_0 - x_s||^{\frac{3}{2}}} \cdot w(x_0)}_{\text{Amplituden-Gewichtung}} \cdot \underbrace{\delta\left(t - \frac{||x_0 - x_s||}{c}\right)}_{\text{Verzögerung}} \tag{2.2}$$

g_0 : 2.5D-WFS-Amplituden-Korrekturfaktor für den Punkt x_{ref} [Sta97, Gleichung 3.11]

$$g_0 = \sqrt{\frac{||x_{ref} - x_0||}{||x_{ref} - x_0|| + ||x_0 - x_s||}}$$

x_{ref} : Koordinate für den Mittelpunkt des Zuhörerraumes / Position des Zuhörers (in Meter)

x_0 : Koordinate des aktuellen Lautsprechers (in Meter)

x_s : Koordinate der Punktquelle (in Meter)

t : Aktueller Zeitpunkt (in Sekunden)

$a(t)$: Aktuelles Audiosignal

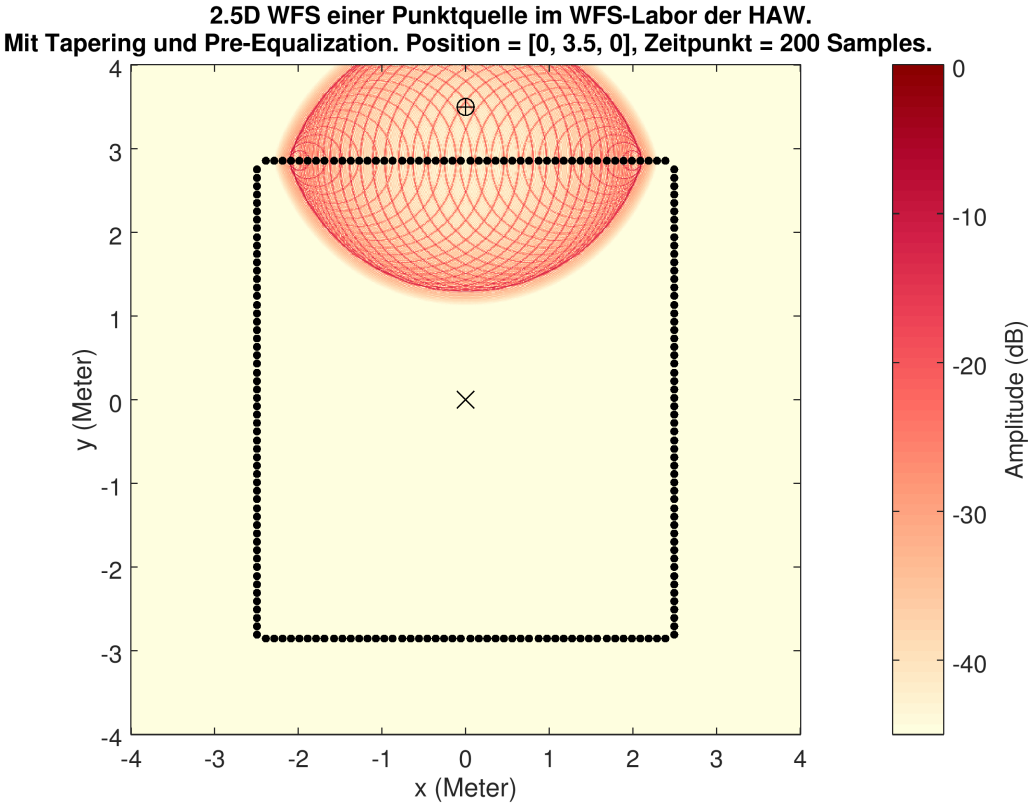

Abbildung 2.5: Simulation von 2.5D-WFS für eine Punktquelle in der Zeitdomäne

$h_{2.5D}(t)$: Vorfilterung des Audiosignals (im Englischen *pre-equalization filter* genannt)

$$h_{2.5D}(t) = \mathcal{F}^{-1}\left\{\sqrt{i\frac{\omega}{c}}\right\}$$

n_{x_0} : Normalenvektor (Richtung) des aktuellen Lautsprechers

$\langle (x_0 - x_s), n_{x_0} \rangle$: Skalarprodukt von $(x_0 - x_s)$ und n_{x_0}

c : Schallgeschwindigkeit (in Meter pro Sekunde)

$w(x_0)$: Fensterfunktion für die Aktivierung des aktuellen Lautsprechers [SRA08]

$$w(x_0) = \begin{cases} 1 & \text{falls } \langle (x_0 - x_s), n_{x_0} \rangle > 0 \\ 0 & \text{sonst.} \end{cases}$$

Ein Lautsprecher ist nur dann aktiv, wenn das Skalarprodukt des Vektors zwischen Lautsprecher und Punktquelle $(x_0 - x_s)$ und des Normalenvektors des Lautsprechers n_{x_0} größer ist als 0, also der von ihnen eingeschlossene Winkel $\cos(\alpha)$ kleiner ist als 90°.

Siehe dazu [Sch16, Gleichung 2.138].

2.2.5 Fokussierte Punktquelle als virtuelle Klangquelle

Eine fokussierte Punktquelle (im Englischen *focused point source* genannt) ist eine virtuelle Klangquelle, die sich innerhalb eines Zuhörerraumes befindet und vom Zuhörer zu einem gewissen Grad umgangen werden kann, solange sich dieser nicht zwischen die Position der fokussierten Klangquelle und die aktiven Lautsprecher stellt. Für die Fensterfunktion $w(x_0)$ zur Wahl der aktiven Lautsprecher besitzt die fokussierte Punktquelle eine Richtung n_s, in welche sich die erzeugte Wellenfront ausbreitet. Um eine solche Klangquelle zu erzeugen, werden die Schallwellen in zeitlich invertierter Reihenfolge abgeschickt. Das führt dazu, dass die Schallwellen zuerst konvergent in Richtung der virtuellen Klangquelle verlaufen. Nach Erreichen der Position der virtuellen Klangquelle divergieren sie wieder in Richtung n_s und erzeugen damit eine kreisförmige Ausbreitung der Schallwellen (siehe Abbildung 2.6). Damit entspricht die fokussierte Punktquelle einer gewöhnlichen Punktquelle mit einem invertierten Zeitkoeffizienten. Das bedeutet, dass für eine Fokussierung innerhalb des Zuhörerraumes das eingehende Audiosignal eine negative Verzögerung besitzen müsste. Erreicht wird dieses durch eine Basisverzögerung, die auf alle eingehenden Audiosignale angewendet wird. Diese berechnet sich aus dem maximalen Abstand zwischen einer fokussierten Punktquelle und einem Lautsprecher.

Ansteuerungsfunktion für die 2.5D-WFS einer fokussierten Punktquelle

$$d_{2.5D_{fs}}(x_0, t) = \underbrace{\frac{1}{\sqrt{2\pi}} \cdot g_0 \cdot a(t) \cdot h_{2.5D}(t) \cdot \frac{\langle (x_0 - x_s), n_{x_0} \rangle}{||x_0 - x_s||^{\frac{3}{2}}} \cdot w(x_0)}_{\text{Amplituden-Gewichtung}} \cdot \underbrace{\delta\left(t + \frac{||x_0 - x_s||}{c}\right)}_{\text{Verzögerung}} \tag{2.3}$$

g_0 : 2.5D-WFS-Amplituden-Korrekturfaktor für den Punkt x_{ref} [Sta97, Gleichung 3.11]

$$g_0 = \sqrt{\frac{||x_{ref} - x_0||}{||x_{ref} - x_0|| + ||x_0 - x_s||}}$$

x_{ref} : Koordinate für den Mittelpunkt des Zuhörerraumes / Position des Zuhörers (in Meter)

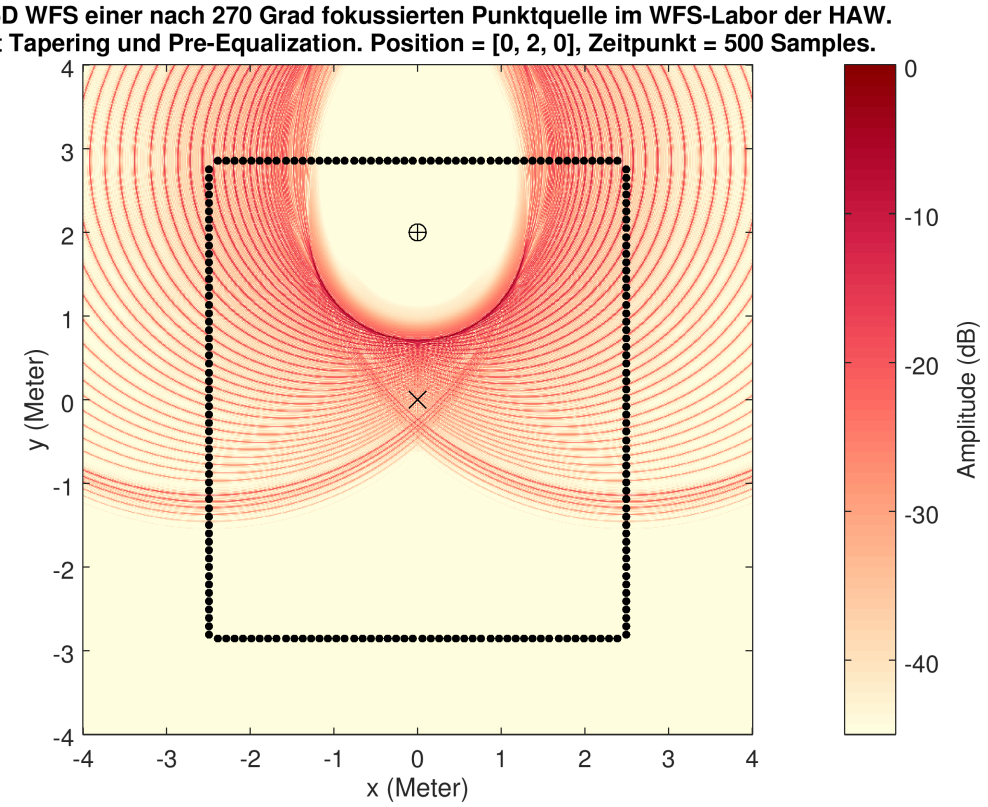

Abbildung 2.6: Simulation von 2.5D-WFS für eine fokussierte Punktquelle in der Zeitdomäne

x_0 : Koordinate des aktuellen Lautsprechers (in Meter)

x_s : Koordinate der Punktquelle (in Meter)

t : Aktueller „invertierter“ Zeitpunkt (in Sekunden)

$a(t)$: Aktuelles Audiosignal

$h_{2.5D}(t)$: Vorfilterung des Audiosignals (im Englischen *pre-equalization filter* genannt)

$$h_{2.5D}(t) = \mathcal{F}^{-1}\left\{\sqrt{i\frac{\omega}{c}}\right\}$$

n_{x_0} : Normalenvektor (Richtung) des aktuellen Lautsprechers

$\langle (x_0 - x_s), n_{x_0} \rangle$: Skalarprodukt von $(x_0 - x_s)$ und n_{x_0}

c : Schallgeschwindigkeit (in Meter pro Sekunde)

n_s : Normalenvektor (Richtung) der fokussierten Punktquelle

$w(x_0)$: Fensterfunktion für die Aktivierung des aktuellen Lautsprechers

$$w(x_0) = \begin{cases} 1 & \text{falls } \langle (x_s - x_0), n_s \rangle > 0 \\ 0 & \text{sonst.} \end{cases}$$

Ein Lautsprecher ist nur dann aktiv, wenn das Skalarprodukt des Vektors zwischen Punktquelle und Lautsprecher $(x_s - x_0)$ und der Richtung der fokussierten Punktquelle n_s größer ist als 0, also der von ihnen eingeschlossene Winkel $\cos(\alpha)$ kleiner ist als 90°.

Aufgrund der geometrischen Lage im Inneren des Zuhörerraumes muss bei der Fensterfunktion $w(x_0)$ der fokussierten Punktquelle darauf geachtet werden die Differenz zwischen Punktquelle und Lautsprecher $(x_s - x_0)$ zu bilden und nicht, wie bei der herkömmlichen Punktquelle außerhalb des Zuhörerraumes, zwischen Lautsprecher und Punktquelle $(x_0 - x_s)$.

2.3 Probleme bei der Synthese des Schallfeldes

Die Wellenfeldsynthese stellt ein Approximationsverfahren zur Rekonstruktion von echten Schallfeldern dar. Dabei kommt es, bedingt durch die Diskretisierung und Vereinfachung bei der Herleitung für die WFS-Ansteuerungsfunktionen zu Einschränkungen und Fehlern in der Darstellung des synthetisierten Schallfeldes. Diese Probleme sollen kurz benannt und mögliche Lösungsansätze aufgezeigt werden.

2.3.1 Amplitudenfehler

Das Verfahren der 2.5D-WFS berechnet die korrekte Amplitude des synthetisierten Schallfeldes nur für einen bestimmten Punkt x_{ref} innerhalb des Zuhörerraumes [Ver97, Kapitel 2.3]. Damit der Zuhörer sich frei im Zuhörerraum bewegen kann und dabei gleichzeitig die korrekte Amplitude wahrnimmt, kann dessen Position mit einem Tracking-System ermittelt werden. Der Referenzpunkt x_{ref} entspricht dann der aktuellen Position des Zuhörers. Das ermöglicht es der 2.5D-WFS stets die korrekte Amplitude an der Position des Zuhörers zu berechnen [Mel+08].

2.3.2 Truncation-Effekt

Der Truncation-Effekt ist ein Beugungseffekt, der an den Enden von Lautsprecher-Arrays auftritt. Der Grund dafür liegt darin, dass bei den Berechnungen der WFS nicht von einer endlichen Anzahl an Lautsprechern ausgegangen wird. Die fehlenden Lautsprecher an den Enden sorgen dafür, dass die Endlautsprecher Schattenwellen erzeugen, die mit den Schallwellen benachbarter Lautsprecher interferieren [Vog93, Kapitel 3.4] (siehe Abbildung 2.9).

Um diesem Verhalten entgegenzuwirken setzt man Tapering ein (siehe Abbildung 2.10). Darunter versteht man die Nutzung von Fensterfunktionen, mit denen der Amplitudenverlauf eines Lautsprecher-Arrays gedämpft wird. Die Fensterfunktion dämpfen dabei nur den Anfang und das Ende des Lautsprecher-Arrays (siehe Abbildung 2.8). Für die Simulationen in diesem Abschnitt wurde beim Tapering eine Tukey-Fensterfunktion verwendet.

Bei rechteckigen Lautsprecher-Installationen, in deren Ecken gleich zwei Enden eines Lautsprecher-Arrays aufeinandertreffen, fallen die störenden Interferenzen nicht ganz so stark aus, wie bei einem einzelnen Lautsprecher-Array [Ver97, Kapitel 2.4.2]. Dennoch kommt es in den Ecken zu ungewollten Interferenzen und verfälschten Amplituden (siehe Abbildung 2.11), die durch den Einsatz von Tapering reduziert werden können (siehe Abbildung 2.12).

2.3.3 Spatial-Aliasing

Das Spatial-Aliasing ist auf die Diskretisierung im Zuge der Vereinfachung und Herleitung der WFS-Ansteuerungsfunktionen zurückzuführen. Wegen der endlichen Anzahl von Lautsprechern einer WFS-Anlage und einem gewissen Mindestabstand, der Bauart bedingt zwischen zwei Lautsprechern herrscht, können bei der Rekonstruktion von Schallfeldern nicht alle Frequenzen fehlerfrei synthetisiert werden [Ver97, Kapitel 2.5]. Die Frequenz, ab welcher es bei einer WFS-Anlage zu Sampling-Fehlern kommt, lässt sich mittels [Ver97, Gleichung 2.35] bestimmen (siehe Gleichung 2.4).

$$f_{alias} = \frac{c}{2\Delta x_0} \tag{2.4}$$

Eine WFS-Anlage mit einem Lautsprecherabstand von $x_0 = 0.1m$ kann bei einer Schallgeschwindigkeit von $c = 343m/s$ das Schallfeld nur bis zu einer Frequenz von $f_{alias} = 1715Hz$ fehlerfrei darstellen (siehe Abbildung 2.13). Oberhalb davon kommt es zu Artefakten, die, abhängig von der Position im synthetisierten Schallfeld, als Verfärbungen des Klangbildes wahrgenommen werden [Bru04, Kapitel 5] (siehe Abbildung 2.14).

2.3.4 Hinweise zur Umsetzung einer Software-Implementierung

Bei der Umsetzung einer Software-Implementierung der 2.5D-Wellenfeldsynthese müssen bezüglich der Ansteuerungsfunktionen einige Dinge beachtet werden:

- Anzahl Lautsprecher × Anzahl Klangquellen = Maximale Anzahl von Berechnungen
- Eine Division durch Null ist möglich, weil Brüche involviert sind
- Beim Durchtritt einer normalen Punktquelle durch die Lautsprecher wird diese zu einer fokussierten Punktquelle und umgekehrt. In diesem Moment muss sich auch die jeweilige Ansteuerungsfunktion ändern
- Kommt eine virtuelle Punktquelle der Position eines Lautsprechers sehr nahe, dann verändert sich die Amplitude sehr stark. Eine Möglichkeit besteht darin einen Bereich vor und hinter dem Lautsprecher zu definieren, ab dem eine variable Fensterfunktion die Amplitude approximiert oder dämpft, um dadurch starke Amplitudenschwankungen an den Grenzen der Ansteuerungsfunktionen zu vermeiden [Baa08, Kapitel 3.4.1]
- Sollen fokussierte Klangquellen innerhalb des Zuhörerraumes berechnet werden können, müssen alle Audiosignale mit Hilfe einer Delay Line künstlich um die Dauer der negativen Latenz vorverzögert werden, da die Ansteuerungsfunktion für fokussierte Punktquellen einen negativen Zeitkoeffizienten aufweist (siehe Gleichung 2.3 und Abbildung 2.7)

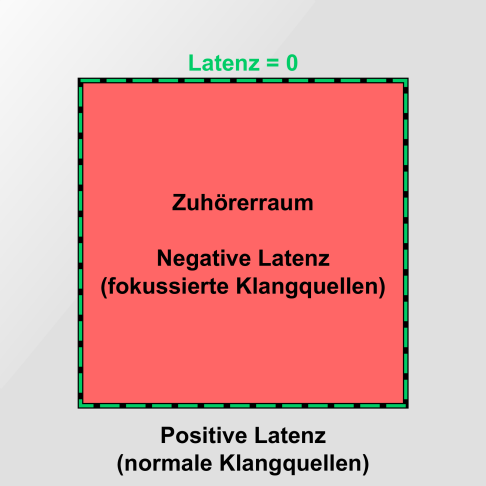

Abbildung 2.7: Latenz unterschiedlicher Klangquellen im Zuhörerraum

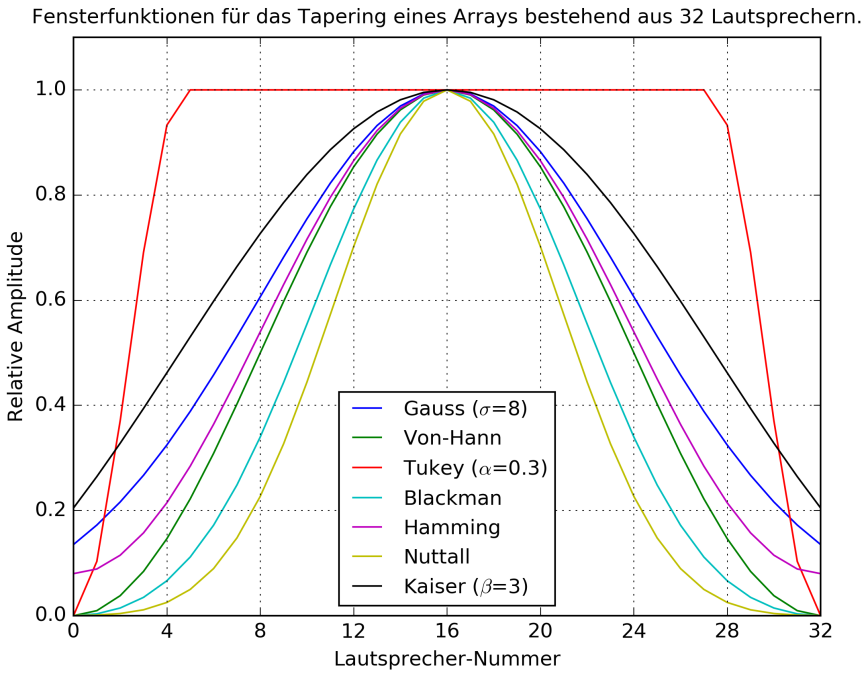

Abbildung 2.8: Mögliche Fensterfunktionen für den Einsatz von Tapering

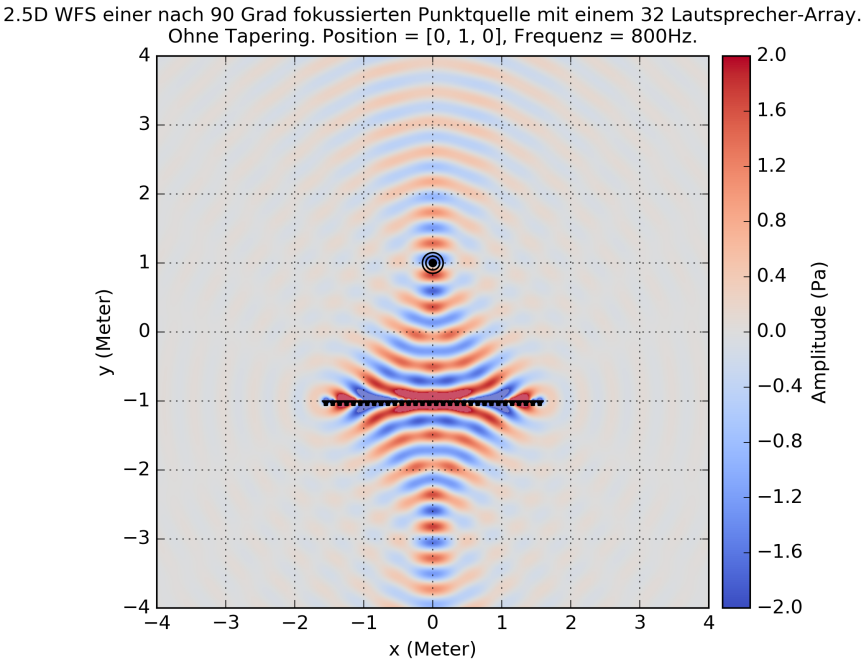

Abbildung 2.9: 2.5D-WFS eines Arrays und einer fokussierten Punktquelle ohne Tapering

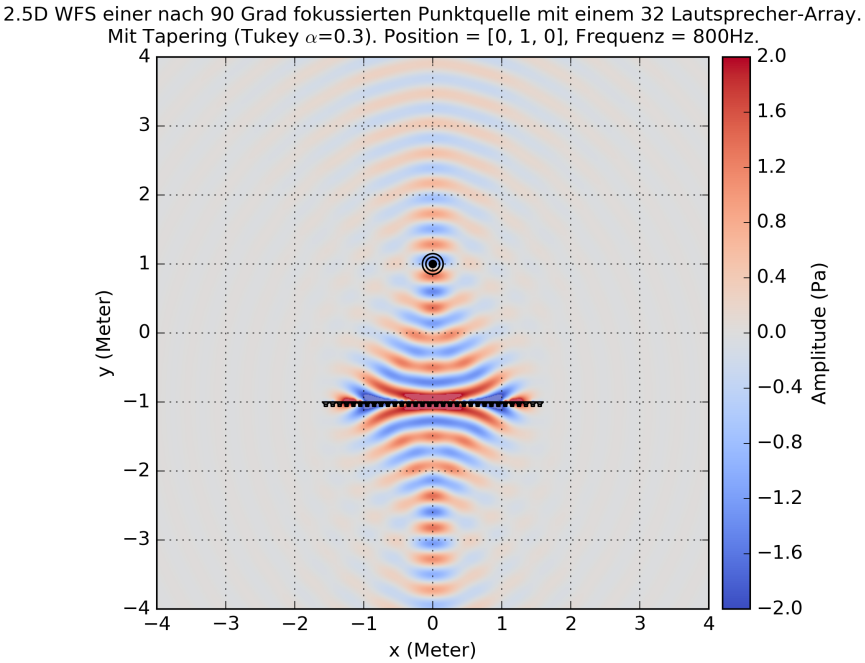

Abbildung 2.10: 2.5D-WFS eines Arrays und einer fokussierten Punktquelle mit Tapering

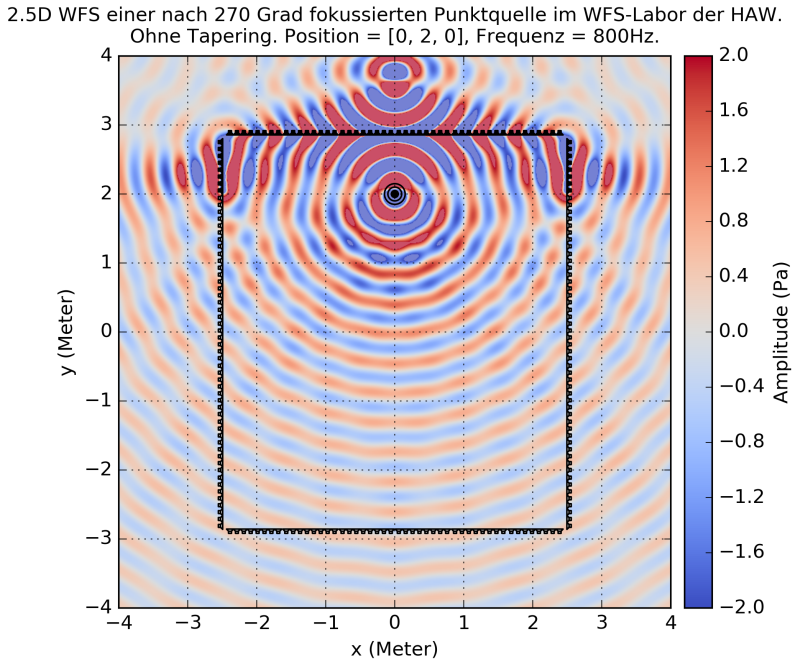

Abbildung 2.11: 2.5D-WFS im WFS-Labor und einer fokussierten Punktquelle ohne Tapering

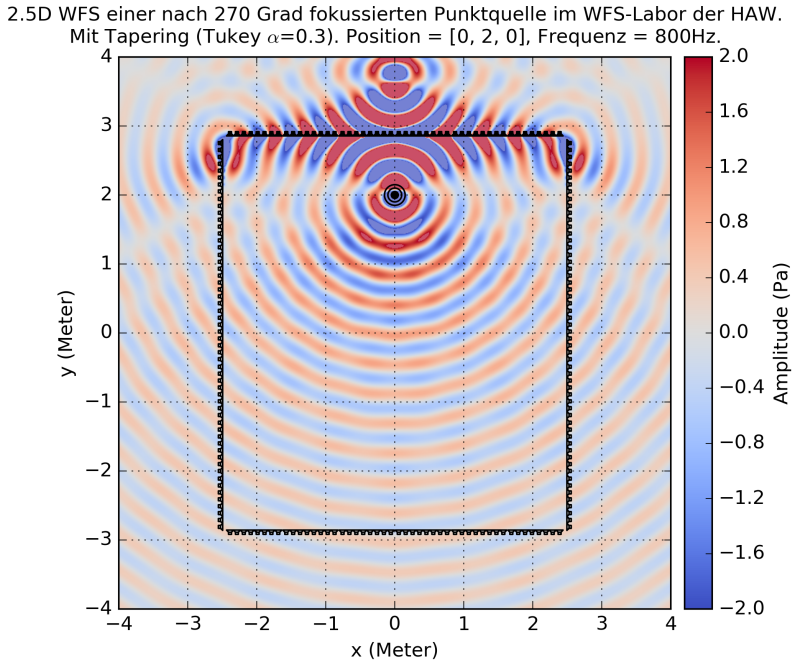

Abbildung 2.12: 2.5D-WFS im WFS-Labor und einer fokussierten Punktquelle mit Tapering

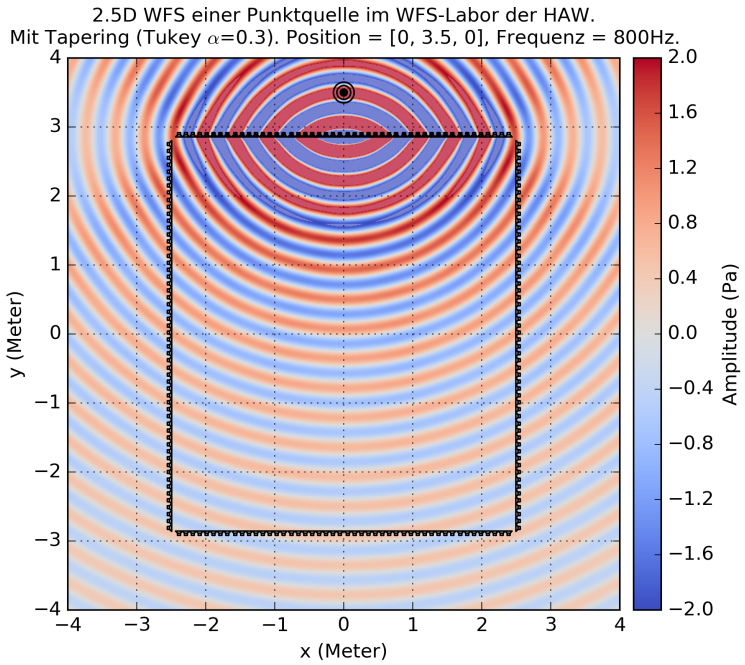

Abbildung 2.13: Kein Spatial-Aliasing unterhalb $f_{alias} = 1715Hz$ und $x_0 = 0.1m$

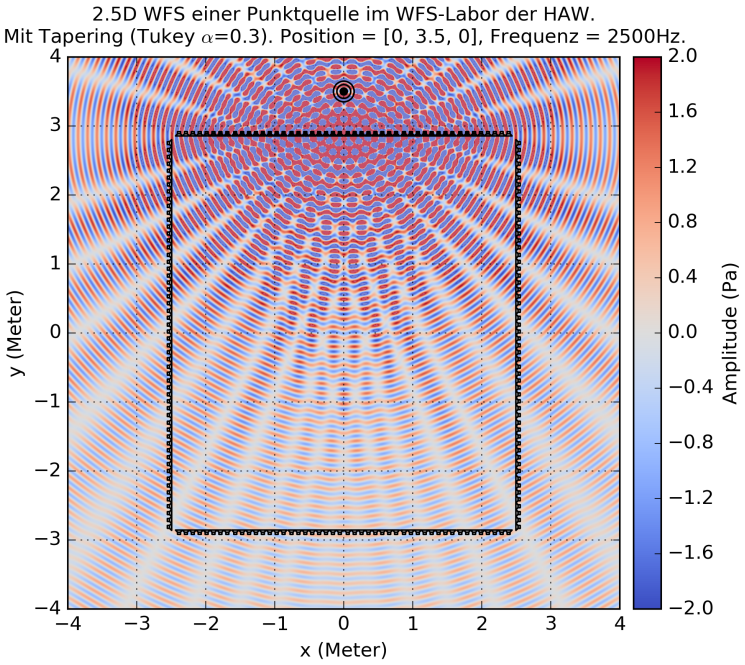

Abbildung 2.14: Deutliches Spatial-Aliasing oberhalb $f_{alias} = 1715Hz$ und $x_0 = 0.1m$

3 Analyse

Nachdem die Grundlagen der Wellenfeldsynthese erörtert worden sind, folgt nun der Blick auf eine konkrete Implementierung. Dabei geht es um die an der Hochschule für Angewandte Wissenschaften Hamburg (HAW Hamburg) installierte Wellenfeldsynthese-Anlage. Im Verlauf dieses Kapitels werden zunächst die in der WFS-Anlage verwendeten Hardware- und Softwarekomponenten betrachtet. Anschließend folgt eine detaillierte Quellcode-Analyse der für das WFS-Rendering zuständigen Software. Dabei soll herausgefunden werden welche Faktoren bei der übermäßigen Latenz der WFS-Anlage eine Rolle spielen und weshalb es bei virtuellen Klangquellen, abhängig von deren Position, zu Unregelmäßigkeiten in der Amplitude kommt.

3.1 Aufbau der Wellenfeldsynthese-Anlage

Die HAW Hamburg ist seit 2011 im Besitz einer Wellenfeldsynthese-Anlage [Foh13], welche durch die Firma Four Audio[1] installiert und eingemessen worden ist. Das System wurde auf einer Fläche von ca. 5 x 6 Meter und unter Zuhilfenahme eines Traversensystems realisiert (siehe Abbildung 3.1). Zusätzlich wurde ein Tracking-System der Firma Advanced Realtime Tracking GmbH[2] installiert, um das Tracking von Personen und Gegenständen innerhalb der WFS-Anlage zu ermöglichen.

3.1.1 Lautsprecher

Die insgesamt 26 Lautsprecher-Module, die in der WFS-Anlage der HAW Hamburg zum Einsatz kommen, sind eine Eigenkonstruktion der Firma Four Audio, die speziell für WFS-Anlagen entwickelt worden sind. Der Wiedergabe des Mittel- und Hochtonbereiches stehen drei Reihen mit je acht Breitbandchassis zur Verfügung, die im Abstand von 0.1 Meter angeordnet sind. Der Tieftonbereich wird durch jeweils zwei Tieftonchassis und Bassreflexöffnungen abgedeckt. Jedes Lautsprecher-Modul kann auf acht Kanälen mit Audiosignalen versorgt werden. Damit besitzt die WFS-Anlage der HAW Hamburg eine Gesamtzahl von 208 Audiokanälen.

[1] http://www.fouraudio.com - Abruf: 2016-12-16
[2] http://www.ar-tracking.com - Abruf: 2016-12-16

Neben einer aktiven Verstärkung hat jedes Lautsprecher-Modul zusätzlich noch einen digitalen Signalprozessor (DSP), mit dem Raumkorrekturen vorgenommen werden können.

3.1.2 Verkabelung und Vernetzung

Die hohe Zahl von 208 Audiokanälen erfordert ein effizientes Kabelmanagement. Dafür besitzen die Lautsprecher-Module ein Dante-Interface der Firma Audinate[3]. Dieses ermöglicht es mehrere Audiosignale mit einer sehr geringen Latenz von etwa einer Millisekunde in einem kabelgebundenen Netzwerk digital zu übertragen und zu verteilen. Mit Hilfe eines Netzwerk-Switches lässt sich ein Dante-Netzwerk aufspannen, in dem alle Geräte mit einem Dante-Interface untereinander verbunden werden können. Das Routing der Geräte sowie deren einzelne Audioeingänge und Audioausgänge lässt sich mit der ebenfalls von Audinate stammenden Software Dante Controller[4] konfigurieren.

Ein weiteres Netzwerk im WFS-Labor basiert auf Open Sound Control (OSC). Dieses dient in erster Linie der Kommunikation zwischen dem WFS-Control Computer und den beiden WFS-Renderern (siehe Abbildung 3.2). Dabei werden OSC-Nachrichten mit den aktuellen Positionen virtueller Klangquellen oder des Zuhörers, welcher zuvor vom Tracking-System erfasst worden ist, ausgetauscht. Zudem vereinfacht das OSC-Netzwerk ein Einbinden weiterer Software und Geräte, welche ebenfalls das OSC-Protokoll unterstützen. Diese können dazu genutzt werden, aktiv in die Steuerung des Rendering-Prozesses einzugreifen und mit der gesamten WFS-Anlage zu interagieren. Eine detaillierte Beschreibung des OSC-Protokolls folgt in Abschnitt 3.4.1.

3.1.3 Tracking-System

Das im WFS-Labor der HAW Hamburg verbaute ARTTRACK Tracking-System besteht aus insgesamt sechs Infrarot-Kameras. Diese ermöglichen das Tracking mit 60 Bildern in der Sekunde. Die Positionen von Personen oder Gegenständen können im dreidimensionalen Raum mittels passiver „Targets“ bestimmt werden. Diese sind mit mehreren, speziell beschichteten Kugeln, sogenannten „Markern“, besetzt, welche die Infrarot-Lichtblitze, die von an den Kameras angebrachten Infrarot-LEDs ausgesendet werden, reflektieren. Anhand der Reflexionen lässt sich die Lage der Marker und damit die Position des Targets bestimmen, die anschließend als ASCII codiertes UDP-Datagramm [Pos80] an eine Multicast-Gruppe [Dee89] geschickt werden kann.

[3] https://www.audinate.com - Abruf: 2016-12-16
[4] https://www.audinate.com/products/software/dante-controller - Abruf: 2016-12-16

3.1.4 Rendering-Hardware

Aufgrund der hohen Rechenlast, die bei den Berechnungen für die Wellenfeldsynthese entsteht, wird im WFS-Labor der HAW Hamburg auf ein verteiltes Rendering gesetzt. Die Hardwarearchitektur der WFS-Anlage stellt sich dabei wie folgt zusammen (siehe Abbildung 3.2):

- **Digital-Audio-Workstation (DAW):** Dieser Computer, ein Apple Mac Pro, ist sowohl in das Dante-, als auch das OSC-Netzwerk eingebunden und ist für die Abarbeitung mehrerer Aufgaben zuständig:
  - Steuerung virtueller Klangquellen auf den WFS-Renderern über die grafische Benutzeroberfläche (GUI) xWONDER [Mon07], welche dafür OSC-Nachrichten an den WFS-Control Computer verschickt
  - Steuerung virtueller Klangquellen über Gesten mit Hilfe des Tracking-Systems und der Software MoWeC [Nog12]
  - Anbindung in das Dante-Netzwerk über eine 256-Kanal Dante-PCIe-Soundkarte, die es Softwares wie Ardour[5], Cubase[6] oder SuperCollider[7] erlaubt die erzeugten Audiosignale zum Rendern an die beiden WFS-Renderer zu senden
- **WFS-Control:** Auf dem WFS-Control Computer läuft eine Instanz von cWONDER (siehe Unterabschnitt 3.3.1). Dabei handelt es sich um eine Software, welche die zentrale Kommunikationsschnittstelle zwischen den WFS-Renderern und anderer Ansteuerungs-Software wie xWONDER darstellt. Die Kommunikation wird über OSC-Nachrichten im OSC-Netzwerk abgewickelt.
- **WFS-Renderer:** Auf den WFS-Renderern laufen mehrere Instanzen von tWONDER (siehe Unterabschnitt 3.3.2). Die Software ist für das Rendern der eingehenden Audiosignale zuständig und verteilt diese anschließend auf die jeweiligen Audiokanäle der Lautsprecher-Module. Außerdem werden rendering-spezifische Daten wie die Art und Positionen von virtuellen Klangquellen in Form von OSC-Nachrichten empfangen und verarbeitet. Dafür sind die WFS-Renderer sowohl mit dem Dante-, als auch mit dem OSC-Netzwerk verbunden.

[5] https://ardour.org - Abruf: 2016-12-16
[6] https://www.steinberg.net/de/products/cubase/start.html - Abruf: 2016-12-16
[7] http://supercollider.github.io - Abruf: 2016-12-16

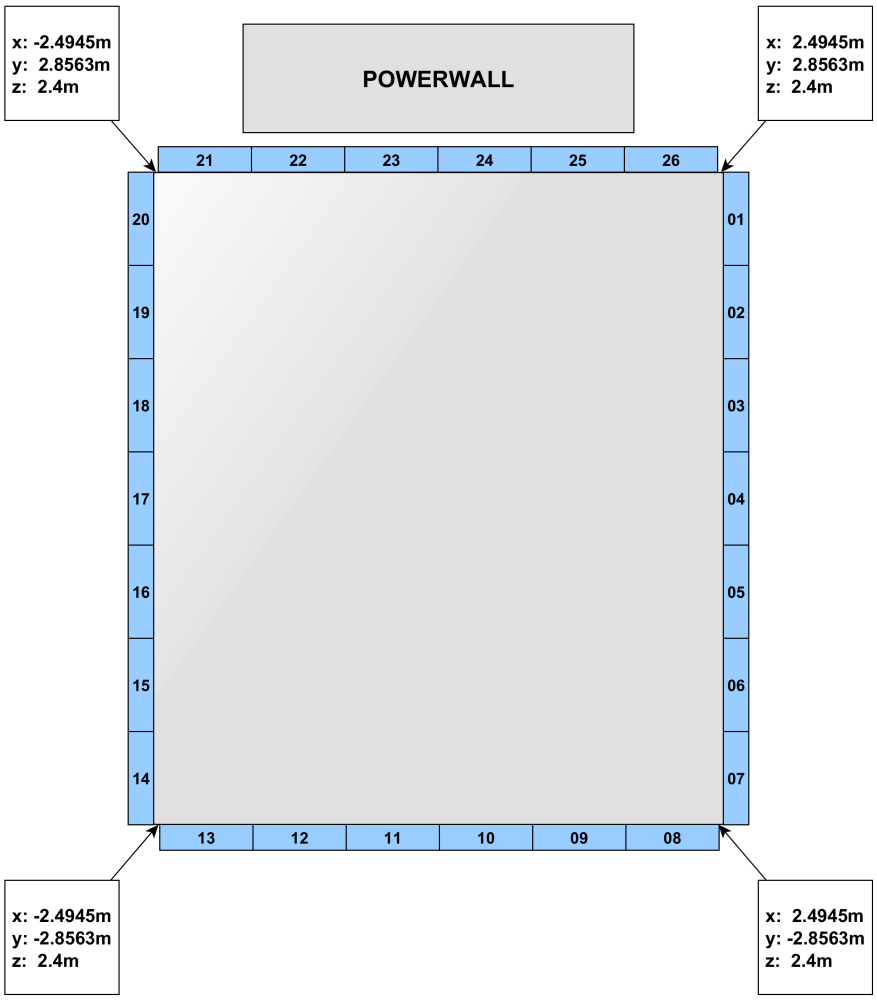

Abbildung 3.1: Abmessungen des WFS-Labors der HAW Hamburg

3.1.5 Rendering-Software

Für das verteilte Rendering kommt im WFS-Labor der HAW Hamburg die Software WONDER zum Einsatz (siehe Abschnitt 3.3). Im Rahmen von Abschlussarbeiten wurde diese bereits mehrfach modifiziert. So ist es nun möglich virtuelle Klangquellen auch in der Vertikalen und damit im dreidimensionalen Raum zu positionieren [Gei15]. Die in Unterabschnitt 2.2.5 genannte Einschränkung bei der Umgehung fokussierter Klangquellen wurde durch ein zuhörerbasiertes WFS-Rendering aufgehoben [Chr14]. Damit kann ein Zuhörer in der WFS-Anlage mit Hilfe des Tracking-Systems und einem aufgesetzten Brillen-Target eine fokussierte Klangquelle komplett in 360° umgehen.

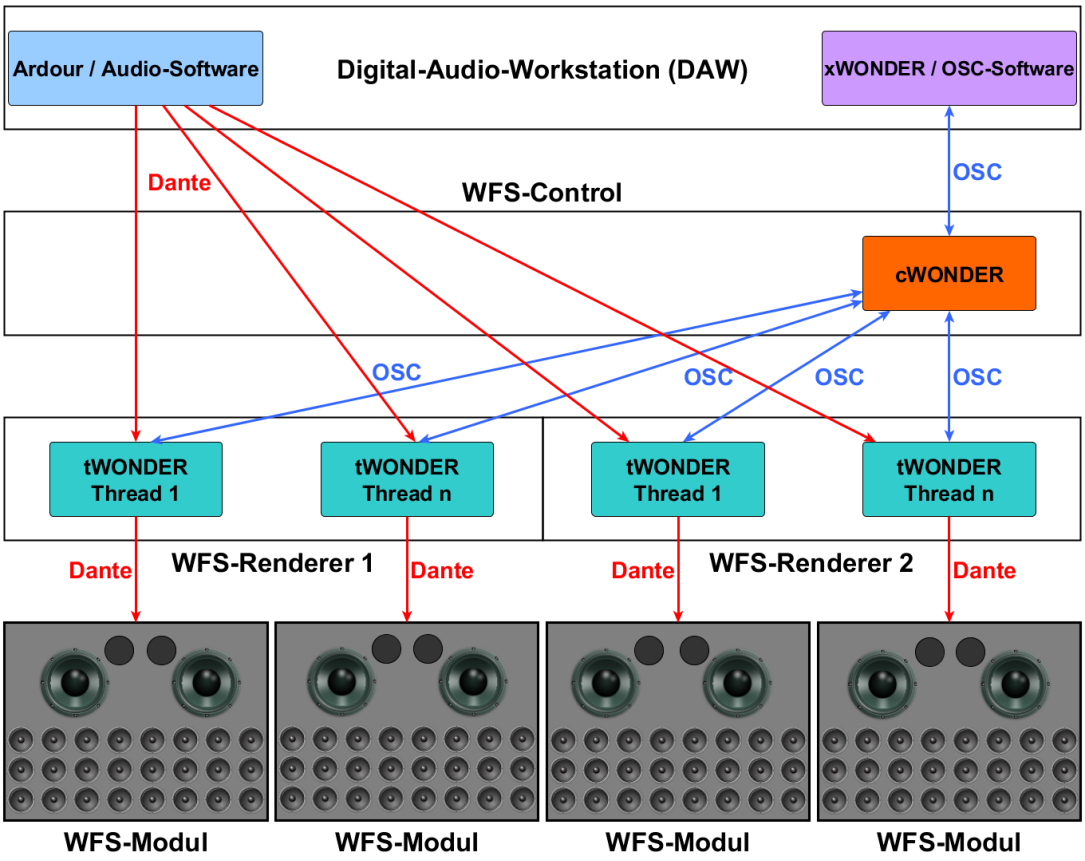

Abbildung 3.2: Rendering-Komponenten im WFS-Labor der HAW Hamburg

3.2 Problemstellungen bei der Wellenfeldsynthese-Anlage

Im Rendering-Betrieb zeigt die WFS-Anlage der HAW Hamburg zwei Auffälligkeiten. Zum einen die merklich erhöhte Latenz bei der direkten Wiedergabe von Audiosignalen und zum anderen die Schwankungen in der Amplitude von virtuellen Klangquellen in Abhängigkeit ihrer Position.

Im Rahmen der Analyse bilden diese beiden Problemstellungen den Schwerpunkt. Das weitere Vorgehen besteht darin in einem ersten Schritt die Unregelmäßigkeiten genauer zu untersuchen und im Falle der Latenz, diese auch zu messen. Im darauf folgenden zweiten Schritt bilden die Beobachtungen und gewonnen Ergebnisse die Grundlage für die Durchführung einer Analyse der eingesetzten Rendering-Software WONDER.

3.2.1 Erhöhte Latenz bei der Wiedergabe

Die erhöhte Latenz der WFS-Anlage macht sich vor allem bei einer direkten Interaktion mit dem erzeugten Audiosignal bemerkbar. Das betrifft zum Beispiel das Monitoring eines Mikrofons, bei dem man sich durch die Latenz selbst ins Wort fällt oder das Spielen eines Instrumentes in einer DAW, bei dem die verzögerte Rückmeldung den Musiker aus dem Takt bringt.

Es ist bereits bekannt, dass die WFS-Anlage der HAW Hamburg eine Latenz von etwa 120ms aufweist [Foh13]. Die Ergebnisse der 2013 durchgeführten Einmessung wurden mit freundlicher Genehmigung von Rainer Thaden von der Firma Four Audio zur Verfügung gestellt und ermöglichen eine differenzierte Analyse der Latenzen (siehe Tabelle 3.1 und Abbildung 3.3). Für die Messung wurde eine Punktquelle 2.5 Meter hinter den Lautsprecher-Modulen platziert. Der Abstand bis zum Messpunkt in der Mitte des Zuhörerraumes betrug damit 5 Meter.

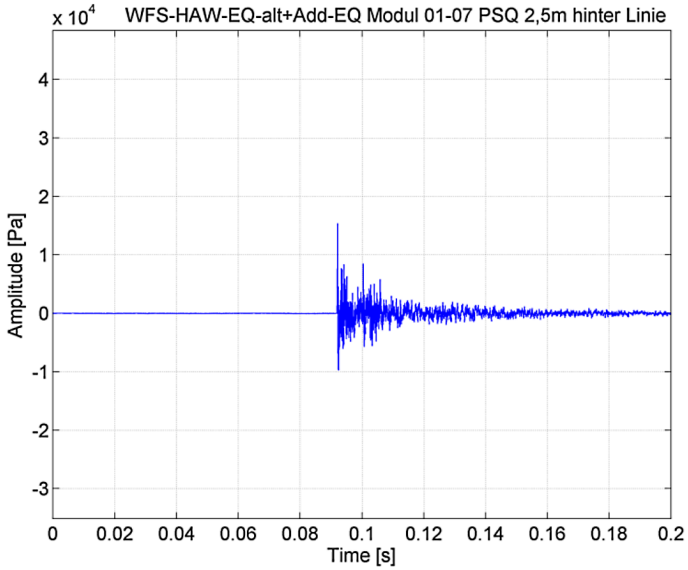

Abbildung 3.3: Gemessene Latenz im WFS-Labor der HAW Hamburg durch Four Audio (2013)

Latenz-Verursacher	Latenz
Punktquelle (5 Meter vom Messpunkt entfernt)	ca. 15ms
FIR-Filter des Lautsprecher-Moduls	ca. 4ms
WONDER, Soundkarte, Audiopuffer etc.	ca. 70ms
Systemlatenz	**ca. 90ms**

Tabelle 3.1: Aufteilung der einzelnen Latenzen bei der Einmessung durch Four Audio (2013)

Vernachlässigt man die Latenz des Dante-Netzwerkes, welches ca. eine Millisekunde benötigt, um das Audiosignal zu übertragen, dann ergibt sich für alle am WFS-Rendering beteiligten Hardware- und Softwarekomponenten eine Latenz von ca. 70ms.

Latenzmessung von WONDER

Um gezielt die von WONDER verursachte Latenz auszuschließen, wurde eine erneute Messung durchgeführt. Dafür wurden auf beiden WFS-Renderern die Audioeingänge mit den Audioausgängen verbunden, um das Audiosignal direkt zu den Lautsprecher-Modulen durchzuschleifen. Für eine Referenzmessung mit aktiviertem WONDER wurde eine Punktquelle genau auf der Position eines Lautsprecher-Moduls platziert. Vor diesem wurde dann mit einem Abstand von ca. 10 cm das Messmikrofon aufgestellt (siehe Abbildung 3.4). Die Maßnahmen dienten der Vermeidung störender Latenzen.

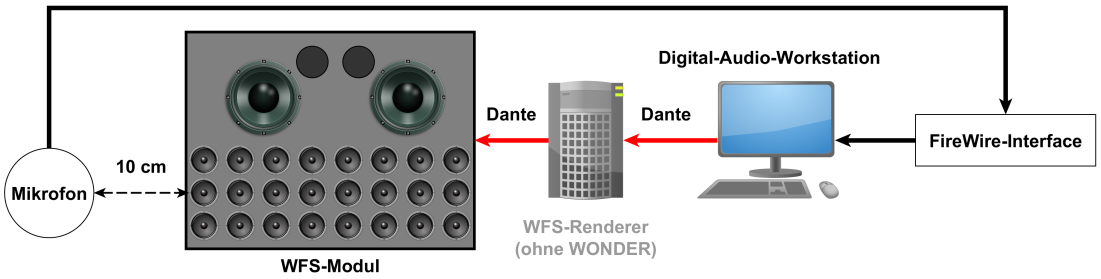

Abbildung 3.4: Versuchsaufbau der Latenz-Messung im WFS-Labor der HAW Hamburg

Für die Latenzmessung wurde folgendes Audio-Equipment aus dem WFS-Labor verwendet:

- AKG C3000B Großmembran Kondensatormikrofon[8]
- Focusrite Liquid Saffire 56 FireWire-Audio-Interface mit Mikrofon-Vorverstärker[9]
- Digital-Audio-Workstation (Apple Mac Pro) mit der Aufnahmesoftware Audacity[10]

Der Messvorgang begann mit der Erzeugung eines Audioimpulses. Auf der DAW wurde dieser mit Hilfe des Messmikrofons aufgenommen sowie auf einen Audioausgang der DAW durchgeschliffen. Das durchgeschliffene Audiosignal durchlief die gesamte WFS-Anlage und wurde am Ende auf einem Lautsprecher-Modul ausgegeben, wo es erneut vom Messmikrofon aufgenommen wurde.

[8] http://www.akg.com/de/p/c3000 - Abruf: 2016-12-16
[9] https://focusrite.de/firewire-audio-interfaces/liquid-saffire-56 - Abruf: 2016-12-16
[10] http://www.audacityteam.org - Abruf: 2016-12-16

Die Zeit zwischen den beiden Aufnahmen entspricht der Systemlatenz. Auf der DAW erfolgte die Aufnahme mit einer Samplerate von 48 KHz und mit Audiopuffergrößen von 128 und 1024 Samples. Die Puffergrößen müssen verdoppelt werden, da sie jeweils für den Audioeingang und Audioausgang gelten und beide benutzt werden. Daraus ergeben sich folgende Latenzwerte:

- DAW Latenz bei einem 128 Samples großen Audiopuffer: $2 \cdot \frac{128}{48000} \cdot 1000 = 5.33ms$
- DAW Latenz bei einem 1024 Samples großen Audiopuffer: $2 \cdot \frac{1024}{48000} \cdot 1000 = 42.66ms$

Latenz-Verursacher	**Latenz**
Mikrofon und Audiopuffer des FireWire-Interfaces	unbekannt
DAW Audiopuffer mit 128 Samples	ca. 5.33ms
DAW Audiopuffer mit 1024 Samples	ca. 42.66ms
Dante-Netzwerk	<= 1ms
WFS-Renderer Audiopuffer mit 128 Samples	ca. 5.33ms
Dante-Netzwerk	<= 1ms
FIR-Filter des Lautsprecher-Moduls	ca. 4ms
Hardwarelatenz ohne WONDER (128 Samples)	**ca. 24ms**
Hardwarelatenz ohne WONDER (1024 Samples)	**ca. 61ms**
Systemlatenz mit WONDER (128 Samples)	**ca. 86ms**
Systemlatenz mit WONDER (1024 Samples)	**ca. 123ms**

Tabelle 3.2: Aufteilung der einzelnen Latenzen bei der Einmessung der Rendering-Hardware

Analyse der gemessenen Latenzwerte

Die erneute Einmessung der WFS-Anlage bestätigt die vorangegangenen Messungen und deren Ergebnisse. Die ca. 123ms Systemlatenz mit WONDER entsprechen den Werten aus [Foh13] und zeigen, dass die Messung mit einer Audiopuffergröße von 1024 Samples durchgeführt worden ist. Eine Reduzierung auf 128 Samples senkt die Latenz um ca. 37ms und beträgt dann nur noch ca. 86ms. Dabei entsprechen die ca. 37ms genau der Differenz zwischen den Latenzen der Audiopuffergrößen.

$$42.66ms - 5.33ms = 37.33ms$$

Des Weiteren sind die ca. 86ms Latenz bei einer Audiopuffergröße von 128 Samples in etwa identisch mit den von Four Audio gemessenen ca. 90ms Latenz (siehe Abbildung 3.3 und Tabelle 3.1).

Latenz-Verursacher	Latenz
Dante-Netzwerk	<= 1ms
WFS-Renderer (128 Samples und 48 KHz Samplerate)	ca. 5.33ms
FIR-Filter des Lautsprecher-Moduls	ca. 4ms
Konstante Systemlatenz	**ca. 10ms**

Tabelle 3.3: Konstante Systemlatenz der WFS-Anlage

Latenz-Verursacher	Latenz
Systemlatenz mit WONDER (128 Samples)	ca. 86ms
-Konstante Systemlatenz	ca. 10ms
- DAW Latenz (128 Samples)	ca. 5.33ms
- Mikrofon und Audiopuffer des FireWire-Interfaces	unbekannt
Latenz von WONDER (128 Samples)	**ca. < 70ms**

Tabelle 3.4: Latenz von WONDER bei einer Audiopuffergröße von 128 Samples

Latenz-Verursacher	Latenz
Systemlatenz mit WONDER (1024 Samples)	ca. 123ms
- Konstante Systemlatenz	ca. 10ms
- DAW Latenz (1024 Samples)	ca. 42.66ms
- Mikrofon und Audiopuffer des FireWire-Interfaces	unbekannt
Latenz von WONDER (1024 Samples)	**ca. < 70ms**

Tabelle 3.5: Latenz von WONDER bei einer Audiopuffergröße von 1024 Samples

Aus den Messergebnissen resultiert eine konstante Latenz der WFS-Anlage von ca. 10ms (siehe Tabelle 3.3). Für die Latenz, welche durch die Messtechnik verursacht worden ist, gibt es keine genauen Werte. Da das FireWire-Interface aber eine analog-digital Wandlung für die Signale des Mikrofons durchführen muss, intern Audiopuffer einsetzt und die FireWire-Schnittstelle zur DAW ebenfalls nicht frei von Latenzen ist, kann davon ausgegangen werden, dass die Messtechnik mehrere Millisekunden zu den gemessenen Latenzen beisteuert.

Unabhängig von der gewählten Audiopuffergröße der DAW (siehe Tabelle 3.4 und Tabelle 3.5) und abzüglich der Latenz des Messaufbaus, lässt sich für WONDER eine Latenz von weniger als 70ms festmachen. Dieser Wert deckt sich somit ebenfalls mit der ermittelten Latenz von ca. 70ms im Zuge der Einmessung der WFS-Anlage durch die Firma Four Audio (siehe Tabelle 3.1).

Fazit der Latenzmessung von WONDER

Für eine abschließende Bewertung der Latenz von WONDER muss berücksichtigt werden, dass die Software in der Lage ist fokussierte Klangquellen innerhalb des Zuhörerraumes zu rendern. Diese Funktionalität kann ausgeschaltet werden, ist aber standardmäßig aktiviert und kommt auch im WFS-Labor zum Einsatz. Das bedeutet, dass WONDER dafür alle Audiosignale vorverzögern muss, um Berechnungen mit einem negativen Zeitkoeffizienten durchführen zu können (siehe Unterabschnitt 2.2.5). Der korrekte Wert für die Vorverzögerung lässt sich aus dem größten Abstand, den eine fokussierte Klangquelle zu einem Lautsprecher haben kann, berechnen. Im Falle der rechteckigen Anordnung der Lautsprecher-Module im WFS-Labor, entspricht die Vorverzögerung der Raumdiagonale (siehe Gleichung 3.1).

$$\text{Pre-Delay}_{HAW} = \frac{\sqrt{(4.989m)^2 + (5.713m)^2}}{343m/s} \cdot 1000 = \frac{7.585m}{343m/s} \cdot 1000 = 21.866ms \quad (3.1)$$

Ausgehend von dem in Gleichung 3.1 ermittelten Wert, müsste die gemessene Latenz von WONDER bei der WFS-Anlage der HAW Hamburg eigentlich im Bereich von ca. 22ms liegen. Die große Abweichung lässt darauf schließen, dass die Software bei der Bestimmung eines korrekten Wertes für die Vorverzögerung fehlerhaft arbeitet oder als Basis einen falschen Wert nutzt. Im Rahmen der Quellcode-Analyse von WONDER, die im nächsten Abschnitt folgt, sollte dieser Umstand bedacht und der Quellcode gezielt auf fehlerhafte Stellen geprüft werden.

3.2.2 Amplitudenschwankungen bei der Wiedergabe

Die Schwankungen in der Amplitude von virtuellen Klangquellen treten in der WFS-Anlage der HAW Hamburg nur in bestimmten Situationen auf. Um diese wahrzunehmen muss sich eine virtuelle Klangquelle zum einen in Bewegung befinden. Zum anderen muss die virtuelle Klangquelle in einem bestimmten Bereich der WFS-Anlage positioniert werden. Die beiden betroffenen Bereiche sind die Raumecken und das Areal direkt vor und hinter einem Lautsprecher-Modul.

Analyse der Amplitudenschwankungen

Wird eine virtuelle Klangquelle im Bereich einer Raumecke der WFS-Anlage bewegt, fällt ihre Amplitude stellenweise merklich ab. Das trifft vor allem auf den Bereich der Raumdiagonale zu. Handelt es sich bei der virtuellen Klangquelle um eine fokussierte Punktquelle, die in den Bereich außerhalb des Zuhörerraumes bewegt wird, dann verstärkt sich dieser Effekt.

Neben den Raumecken sind auch die Bereiche direkt vor und hinter den Lautsprecher-Modulen von Amplitudenschwankungen betroffen. Wird eine normale Punktquelle in das Innere des Zuhörerraumes bewegt, so steigt die Amplitude merklich an. Umgekehrt sinkt die Amplitude zu schnell ab, wenn eine fokussierte Punktquelle außerhalb des Zuhörerraumes bewegt wird.

Fazit der beobachteten Auffälligkeiten

Die Amplitudenschwankungen in den Raumecken sind typische Anzeichen für den Truncation-Effekt. Dieser kann mittels Tapering und einer Auswahl der richtigen Fensterfunktion reduziert werden (siehe Unterabschnitt 2.3.2).

Die Möglichkeit fokussierte Punktquellen zu rendern bringt mehrere Schwierigkeiten mit sich. Beim Durchtritt einer Punktquelle durch einen Lautsprecher muss je nach deren Position darauf geachtet werden die korrekte Ansteuerungsfunktion auszuwählen und eine Division durch Null zu verhindern (siehe Unterabschnitt 2.3.4).

In der Quellcode-Analyse von WONDER sollte bezüglich der Amplitudenschwankungen in den Raumecken geprüft werden, ob Tapering zum Einsatz kommt und welche Fensterfunktion dafür benutzt wird. Auch der Durchtritt einer Punktquelle durch einen Lautsprecher kann mit Fensterfunktionen approximiert werden, um eine Division durch Null zu unterbinden. Sind also Fensterfunktion im Quellcode zu finden, dann sollten deren Parameter überprüft werden. Diese sind abhängig vom Wiedergaberaum und den eingesetzten Lautsprechern. Möglicherweise ist dafür auch eine Einmessung und Evaluierung der Parameter nötig.

3.3 WONDER - Aufbau der Rendering-Komponenten

Für das verteilte Rendering wird im WFS-Labor der HAW Hamburg die Software WONDER[11] (**W**ave field synthesis **O**f **N**ew **D**imensions of **E**lectronic music in **R**ealtime) eingesetzt [Baa08, Kapitel 3.2]. Diese ist quelloffen und unterliegt der GNU General Public License Version 2.0 (GPLv2) [GPLv2]. Die Software existiert bereits seit 2004 [BP04] und bekam mit der Zeit neue Funktionen, wie das Rendering-Verfahren für WFS sowie die grafische Benutzeroberfläche xWONDER. Im Jahr 2007 fand eine umfassende Änderung der Softwarearchitektur statt, als diese für den Einsatz in verteilten Systemen reimplementiert worden ist.

Neben dem WFS-Labor der HAW Hamburg zeigt auch die WFS-Installation im Hörsaal WellenFeld H 104[12] an der TU Berlin die Fähigkeiten von WONDER in einem verteilten System. Dort wird die Rechenlast für das WFS-Rendering von insgesamt 832 Audiokanälen auf 15 Linux-Cluster verteilt.

Bei der Implementierung von WONDER wurde die Programmiersprache C++ verwendet. Als Programmbibliotheken kamen unter anderem die Lightweight OSC Library und das JACK Audio Connection Kit zum Einsatz (siehe Abschnitt 3.4). Das Zielbetriebssystem der Software ist Linux, sie kann jedoch auch unter OSX kompiliert und eingesetzt werden.

Die für das WFS-Rendering zuständigen WONDER-Komponenten, deren Aufbau im Weiteren betrachtet wird, sind cWONDER und tWONDER. Andere Komponenten, die ebenfalls Teil der WONDER-Software sind, sollen bei dieser Betrachtung nicht berücksichtigt werden.

3.3.1 Aufbau der Komponente cWONDER

Die Komponente cWONDER (control unit) wurde im Zuge einer Neuausrichtung der Softwarearchitektur für den Einsatz in verteilten Systemen entwickelt [Baa+07]. Sie dient als zentrale Vermittlungseinheit für alle übrigen WONDER-Komponenten. Dabei wickelt sie die Kommunikation zwischen ihnen ab und speichert alle wichtigen Daten des Systems, wie die Eigenschaften des Zuhörerraumes oder die Positionen von virtuellen Klangquellen. Somit lagern alle Informationen über das System an einer zentralen Stelle, von wo aus sie abgerufen werden können. Auf diese Weise sorgt cWONDER für einen konsistenten Zustand aller Informationen unter den Komponenten. Der Informationsaustausch findet über OSC-Nachrichten mit Hilfe der Lightweight OSC Library statt (siehe Unterabschnitt 3.4.1).

[11] https://github.com/dvzrv/wonder - Abruf: 2016-12-16
[12] https://www.ak.tu-berlin.de/menue/forschung/wellenfeldsynthese - Abruf: 2016-12-16

Stream-Modell

Für die Kommunikation zwischen den einzelnen WONDER-Komponenten nutzt cWONDER sogenannte „Streams“. Dabei handelt es sich um spezifische Kommunikationskanäle zur Nachrichtenübertragung.

Der für das Rendering relevante Stream ist der „Render Stream“ (siehe Abbildung 3.5). Erhält cWONDER zum Beispiel eine OSC-Nachricht mit den Positionsdaten für eine virtuelle Klangquelle, dann speichert es diese für sich selbst ab und sendet sie dann an alle Teilnehmer, die sich am Render Stream angemeldet haben. Im Fall von Abbildung 3.5 also an alle tWONDER-Instanzen von 1 bis n.

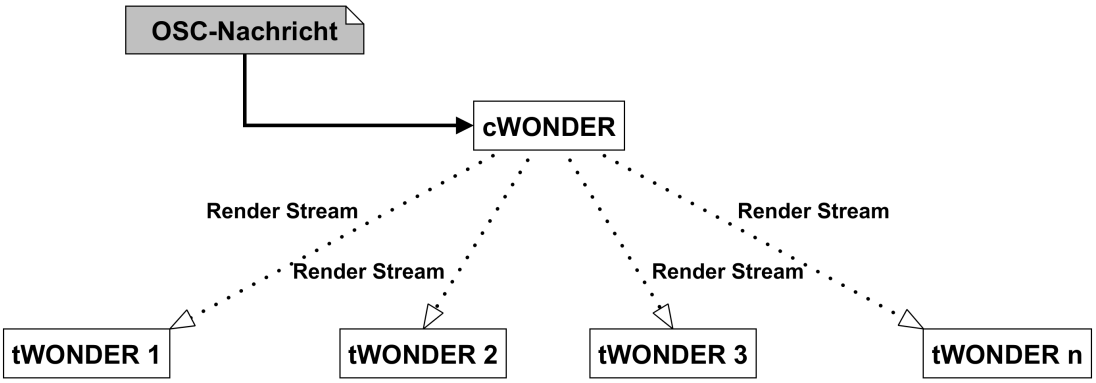

Abbildung 3.5: Kontrollfluss zwischen den einzelnen WONDER-Komponenten

3.3.2 Aufbau der Komponente tWONDER

Die für das WFS-Rendering zuständige Komponente ist tWONDER (time domain render unit) [Baa05]. Diese liest in Realtime alle Audiosignale ein, die am Audioeingang des Computers ankommen, auf dem die tWONDER-Instanz läuft. Für jeden einzelnen Audiokanal (Klangquelle) berechnet die Software dann anhand der WFS-Ansteuerungsfunktionen eine Gewichtung der Amplitude und die nötige Vorverzögerung. Mit Hilfe der berechneten Vorverzögerung wird das gewichtete Audiosignal in einem Puffer (Delay Line) abgelegt, aus dem es kontinuierlich sowie in zeitlich korrekter Reihenfolge gelesen und über den Audioausgang des Computers an die Lautsprecher des WFS-Systems gesendet wird. Für die Audioverarbeitung kommt dabei das JACK Audio Connection Kit zum Einsatz (siehe Unterabschnitt 3.4.2). Die Kommunikation wird, wie auch bei cWONDER, mit Hilfe der Lightweight OSC Library umgesetzt (siehe Unterabschnitt 3.4.1).

3.4 WONDER - Verwendete Bibliotheken

Bei der Implementierung von WONDER wurden eine Reihe von externen Programmbibliotheken verwendet. Zwei dieser Bibliotheken sind auch für das WFS-Rendering relevant. Deren Funktionalität soll im nächsten Abschnitt näher betrachtet werden.

3.4.1 Lightweight OSC Library

Die Lightweight OSC Library (im Folgenden auch kurz liblo genannt)[13] ist eine leichtgewichtige C-Implementierung des Open Sound Control (OSC) Protokolls und besitzt in ihrer aktuellen Version[14] auch einen C++-Wrapper. Die liblo-Bibliothek ist quelloffen und unterliegt der GNU Lesser General Public License Version 2.1 (LGPLv2.1) [LGPLv2.1]. Es gibt sie für die Betriebssysteme Linux, OSX und Windows.

Die Funktionen von liblo umfassen das Senden und Empfangen von OSC-Nachrichten. Für den Empfang stellt liblo die Implementierung eines Server-Threads zur Verfügung. Dieser arbeitet mit den Protokollen TCP sowie UDP und ist auch in der Lage eine Multicast-Gruppe zu „joinen“. Intern verarbeitet der Server-Thread die Nachrichten in einer Warteschlange. Das ermöglicht ihm OSC-Nachrichten, die mit einem in der Zukunft liegenden OSC-Timetag versehenen sind, auch erst später und nicht sofort beim Empfang zu verarbeiten. Beim Einsatz unter Windows gilt es jedoch zu beachten, dass der Server-Thread von liblo nach dem POSIX-Standard implementiert worden ist [IEE16]. Deshalb muss hier ein POSIX-Wrapper wie pthreads[15] benutzt werden, der sämtliche POSIX-Aufrufe auf äquivalente Win32-Thread-API-Aufrufe übersetzt.

Der C++-Wrapper von liblo bringt neben der Objektorientierung auch eine Vereinfachung beim Erstellen von Methoden für den Empfang von OSC-Nachrichten mit sich. Dafür werden Lamda-Funktionen eingesetzt, die seit dem C++11 Standard (ISO/IEC 14882:2011) ein Bestandteil der C++-Standardbibliothek sind. Anstatt mit einem Funktions-Aufruf beim Server-Thread einen OSC-Pfad für den Empfang von OSC-Nachrichten anzumelden und darin einen Funktions-Zeiger zu einer zweiten Funktion anzugeben, welche nach dem Empfang einer Nachricht ausgeführt werden soll, ist es nun möglich statt dem Funktions-Zeiger direkt eine Lambda-Funktion zu definieren. Somit sind nur noch die Hälfte an Funktionsaufrufen nötig, um neue OSC-Pfade und deren Funktionalität beim Server-Thread anzumelden.

[13] http://liblo.sourceforge.net - Abruf: 2016-12-16
[14] https://github.com/radarsat1/liblo - Abruf: 2016-12-16
[15] https://sourceware.org/pthreads-win32 - Abruf: 2016-12-16

OSC-Protokoll

Das Open Sound Control (OSC)[16] Protokoll ist ein Netzwerk-Protokoll, welches die Kommunikation über Nachrichten abwickelt. Entwickelt wurde es am Center for New Music and Audio Technology (CNMAT) der University of California in Berkeley [WFM03]. Anfangs war es für die Ansteuerung von Musik-Software und Peripheriegeräten vorgesehen, hat sich aber in den darauffolgenden Jahren immer mehr als universell einsetzbares Kommunikations-Protokoll behaupten können. Um den vielseitigen Charakter des Protokolls widerzuspiegeln, wurde von den Entwicklern bereits der Vorschlag geäußert den Namen des Protokolls nach *Open System Control* umzubenennen [FS09].

Die OSC-Spezifikation[17] definiert wie eine OSC-Nachricht auszusehen hat. Hierbei setzt sie sich aus den folgenden drei Teilen zusammen:

$$\underbrace{\text{/WONDER/source/position}}_{\text{Pfad}}\ \underbrace{\text{ifff}}_{\text{Typen}}\ \underbrace{\text{42 1.0 2.0 3.0}}_{\text{Argumente}}$$

1. **Pfad:** Der Pfad ist die Zieladresse einer OSC-Nachricht. Das empfangende Programm kann einen oder mehrere Pfade besitzen. Für jeden Pfad definiert das Programm eine Methode, die bei Erhalt einer Nachricht ausgeführt werden soll und in der auf den Inhalt der OSC-Nachricht zugegriffen werden kann.

2. **Typen:** Der zweite Teil einer OSC-Nachricht besteht aus der Spezifizierung jener Typen, welche im Argument, also dem Inhalt einer OSC-Nachricht enthalten sind. Die OSC-Spezifikation definiert alle möglichen Typen. Dazu zählen unter anderem:

   i Integer-Werte

   f Float-Werte

   c Zeichen (char)

   s Zeichenkette (string)

   t OSC-Timetag: Zeitstempel einer OSC-Nachricht

3. **Argumente:** Die Argumente bilden den Inhalt der OSC-Nachricht ab und müssen dabei den Typen entsprechen, welche zuvor definiert worden sind.

Die Zieladresse der obigen Beispielnachricht entspricht also */WONDER/source/position* und als Inhalt werden ein Integer-Wert (42) und drei Float-Werte (1.0, 2.0 und 3.0) erwartet.

[16] http://opensoundcontrol.org - Abruf: 2016-12-16
[17] http://opensoundcontrol.org/spec-1_0 - Abruf: 2016-12-16

Neben OSC-Nachrichten können auch OSC-Bundles verschickt werden. Dabei ist ein Bundle nichts weiter als ein Paket, bestehend aus einer oder mehreren OSC-Nachrichten. Es ist außerdem möglich jeder OSC-Nachricht und jedem OSC-Bundle ein OSC-Timetag zuzuweisen. Dieser Zeitstempel definiert den Zeitpunkt der Verarbeitung. Im Normalfall entspricht er „jetzt“, wodurch das empfangende Programm diese sofort verarbeitet. Es besteht aber auch die Möglichkeit einen Zeitpunkt in der Zukunft zu definieren und somit eine sequentielle Abarbeitung mehrerer OSC-Nachrichten zu erreichen.

3.4.2 JACK Audio Connection Kit

Das JACK Audio Connection Kit (im Folgenden auch kurz JACK genannt)[18] ist eine Bibliothek, die Programmen den Zugriff auf die Audio-Hardware eines Computers gibt und in der Lage ist Verbindungen zwischen diesen Programmen über sogenannte „Ports“ aufzubauen und unterschiedlich zu routen (siehe Abbildung 3.6).

Im Vordergrund steht dabei mit einer möglichst geringen Latenz zu arbeiten. Auch ein Realtime-Betrieb ist möglich, zum Beispiel beim Einsatz eines Realtime-Kernels in Linux. Die nachfolgenden Betriebssysteme werden mit ihren jeweiligen Audio-Backends und Treibern unterstützt: Linux (ALSA), OSX (CoreAudio) und Windows (ASIO, MME, DirectSound).

Um auch Programme in JACK einbinden zu können, die eigentlich kein JACK unterstützen, gibt es unter Windows die Möglichkeit den „JackRouter“ Treiber zu verwenden. Dieser gibt vor ein ASIO-Audio-Interface zu sein. Dadurch kann jede Audio-Software, die das ASIO-Protokoll beherrscht, den Treiber als Audioausgang sehen und diesen benutzen. Auf diese Weise kommen die Audiosignale über den JackRouter bei JACK an und können nach belieben geroutet werden.

Es existieren zwei Versionen von JACK: JACK1[19] und JACK2[20]. Beide Versionen sind untereinander kompatibel, wobei JACK2 neuer ist und auch Multiprozessor-Systeme unterstützt. Bei der Softwarearchitektur setzt JACK auf das Server-Client-Modell:

- **JACK-Server:** „jackd“ (jack daemon) ist der JACK-Server und läuft als Audio-Server mit Zugriff auf die Audio-Hardware im Hintergrund
- **JACK-Client:** Programm, welches die JACK-Bibliothek implementiert und sich für den Zugriff auf die Audio-Hardware beim JACK-Server registriert
- **JACK-Port:** Verbindung zwischen einzelnen JACK-Clients, um Daten auszutauschen

[18] http://www.jackaudio.org - Abruf: 2016-12-16
[19] https://github.com/jackaudio/jack1 - Abruf: 2016-12-16
[20] https://github.com/jackaudio/jack2 - Abruf: 2016-12-16

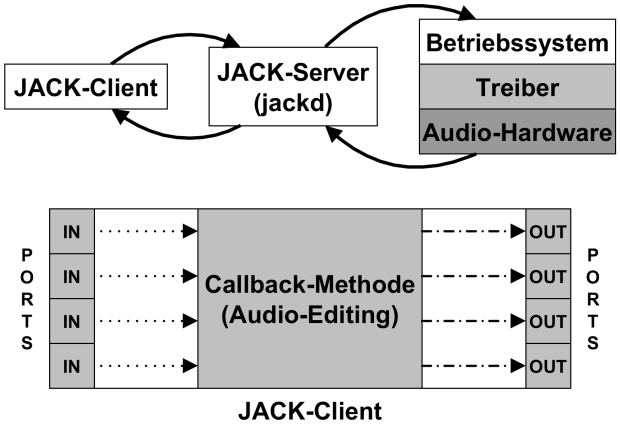

Abbildung 3.6: Aufbau der JACK Bibliothek

Bevor ein JACK-Client auf die Audio-Hardware zugreifen kann, muss er sich zuerst bei einem JACK-Server registrieren. Dafür muss jackd als Programm oder Dienst im Hintergrund laufen. Ein JACK-Client kann aber auch so programmiert sein, dass er automatisch den Start eines „Standard JACK-Servers“ initiiert, bevor er sich bei diesem registriert. Die Einstellungen des Standard JACK-Servers befinden sich im Benutzerverzeichnis in der Datei *.jackdrc*:

- **Linux:** /home/Benutzer/.jackdrc
- **Windows:** C:\Users\Benutzer\.jackdrc

Beim Start sucht der Standard JACK-Server nach der Datei und liest aus ihr die Konfiguration mit der er starten soll. Die Datei beinhaltet genau eine Zeile:

- **Linux:** /usr/bin/jackd -R -d alsa -d hw:0 -r 48000 -p 128
- **Windows:**
  C:\Program Files\Jack\jackd.exe -R -S -d portaudio -d ASIO::Realtek ASIO -r 48000 -p 128
  - **jackd:** Pfad zum JACK-Server jackd
  - **R:** Realtime-Betrieb
  - **S:** Synchroner Treiber-Betrieb (Unter Windows Pflicht!)
  - **d:** Treiber-Backend
  - **d:** Hardware-Backend
  - **r:** Samplerate
  - **p:** Puffergröße

Nachdem ein JACK-Server gestartet ist und sich ein JACK-Client bei diesem registriert hat, ist dieser in der Lage Audiosignale zu verarbeiten. Zuvor teilt ein JACK-Client im Rahmen der Registrierung dem JACK-Server eine Methode mit, in welcher die Verarbeitung der Audiodaten stattfinden soll. Diese „Callback-Methode“ wird, in Abhängigkeit der gewählten Samplerate und Puffergröße, periodisch in einem Realtime-Thread des JACK-Servers aufgerufen. Auf diese Weise kann der JACK-Server eine Synchronisierung aller JACK-Clients sicherstellen. Es gilt jedoch zu beachten, dass es von äußerster Wichtigkeit ist, dass in dieser Callback-Methode keinerlei Aufrufe geschehen dürfen, die nicht für eine Realtime-Umgebung geeignet sind. Aufrufe also, die zu viel Rechenzeit benötigen, lange blockieren könnten oder ein nicht-deterministisches Verhalten aufweisen. Dazu zählen:

- **I/O:** Datenträger, Terminal, Netzwerk
- **Speicheroperationen:** malloc, free
- **Ausgaben:** printf
- **Thread-Operationen:** Mutex-Lock, Condition-Wait, Thread-Join, Select, Sleep, Poll

Die Kommunikation zwischen JACK-Clients findet über die JACK-Ports statt. Diese stehen einerseits für die physikalischen Ports, also die Ein- und Ausgänge einer Audio-Hardware. Andererseits können sie aber auch für ein internes Routing benutzt werden und über die Anzahl der physikalischen Ein- und Ausgänge hinausgehen. Damit lassen sich mehrstufige Audio-Effekte umsetzen oder eine Vielzahl von Audiosignalen zu einem zusammenfassen.

Das Routing von JACK-Ports passiert über den JACK-Server und kann mit Hilfe einer GUI oder dem Konsolen-Programm „jack_connect“ gesteuert werden. Zusätzlich besteht die Möglichkeit einen JACK-Client so zu programmieren, dass er schon beim Starten ein konkretes, internes Routing aufweist. In Verbindung mit dem automatisierten Start des Standard JACK-Servers kann also ein JACK-Client entwickelt werden, der bereits nach seiner Ausführung funktioniert und keinerlei Konfiguration bedarf.

Neben den JACK-Ports stellt die JACK-Bibliothek noch eine weitere Schnittstelle für die Kommunikation zur Verfügung. Dabei handelt es sich um den JACK-Ringpuffer, mit dessen Hilfe genau zwei Threads „lock-free“ miteinander Daten austauschen können. Der Begriff lock-free bedeutet, dass für die Lese- und Schreibzugriffe auf den Ringpuffer keine Mutexe, Semaphore oder dergleichen notwendig sind. Beide Threads, einer lesend, der andere schreibend, haben jeweils einen Zeiger auf den Ringpuffer, mit dem sie abfragen können, ob es etwas zum Lesen gibt oder genug Platz zum Schreiben vorhanden ist. Damit eignet sich der JACK-Ringpuffer für den Einsatz als Kommunikationsschnittstelle zweier Threads in der Callback-Methode eines JACK-Clients.

3.5 WONDER - Startvorgang der Komponenten

Beim Startvorgang der WFS-Anlage der HAW Hamburg wird maßgeblich das Verhalten des im Anschluss daran laufenden Systems bestimmt. Das liegt an den zahlreichen Startskripten, die aufgrund der verteilten Hardware- und Softwarearchitektur zum Einsatz kommen. Diese haben über ihre Startparameter großen Einfluss auf einzelne WONDER-Komponenten. Aus diesem Grund können sie auch ein Einfallstor für mögliche Fehlerquellen sein.

3.5.1 Analyse des Startvorganges

Der Aufbau als verteiltes System hat zur Folge, dass sich einzelne WONDER-Komponenten auf unterschiedlichen Computern befinden (siehe Abbildung 3.2). Ausgehend von der DAW wird ein Startskript gestartet, welches über das Netzwerk auf dem WFS-Control Computer ein zentrales WONDER-Startskript ausführt. Dieses startet dort zuerst eine cWONDER-Instanz. Als nächstes wird über das Netzwerk auf jedem WFS-Renderer Computer zuerst ein JACK-Server gestartet, dann mehrere tWONDER-Instanzen und zum Schluss das JACK-Connect-Skript. Dieses führt ein Routing aller JACK-Ports der tWONDER-Instanzen durch, indem es sie mit den Ein- und Ausgängen des Audio-Interfaces verbindet (siehe Abbildung 3.7).

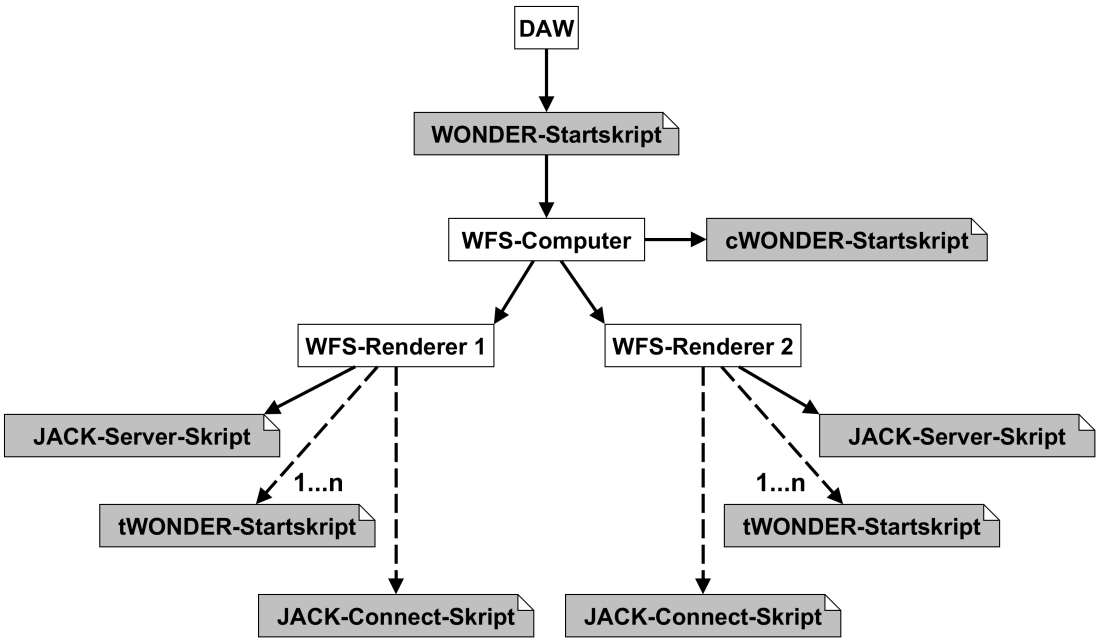

Abbildung 3.7: Verteilung der Startskripte auf den WONDER-Komponenten

Aufgrund der Menge an Skripten wird der Startvorgang der WFS-Anlage verzögert, weil zwischen den in den Skripten enthaltenen Befehlen kurze Pausen eingefügt worden sind, um die Beendigung eines vorangegangenen Befehls abzuwarten. Außerdem lassen sich durch die Menge an Befehlen schnell Fehler provozieren. So gibt es laut der WONDER-Dokumentation[21] für tWONDER den Startparameter *negdelay*, mit dem sich die Vorverzögerung für das Rendering fokussierter Klangquellen einstellen lässt. Trotz dessen Existenz fehlt er in den Startskripten von tWONDER. Das testweise Einfügen des Startparameters führte jedoch dazu, dass die WFS-Anlage nicht mehr gestartet werden konnte, weshalb er wieder aus den Startskripten von tWONDER entfernt wurde. Der WONDER-Dokumentation ist zu entnehmen, dass beim Fehlen der Angabe des Startparameters *negdelay* ein Standardwert von 20 Meter benutzt wird.

3.5.2 Fazit zur Analyse des Startvorganges

Im Hinblick auf die Möglichkeiten der JACK-Bibliothek, könnte ein Teil der Skripte ersetzt werden, indem der Quellcode von tWONDER so angepasst wird, dass der enthaltene JACK-Client automatisch den Standard JACK-Server startet und sein Routing intern selbst durchführt (siehe Unterabschnitt 3.4.2). Dadurch wären nicht nur alle drei Startskripte von tWONDER hinfällig, sondern es würde auch den Startvorgang der WFS-Anlage beschleunigen. Hinzu kommt die Reduzierung möglicher Fehlerquellen.

Eine Überraschung bei der Analyse des Startvorganges stellt der fehlende tWONDER Startparameter *negdelay* dar. Statt der in Gleichung 3.1 berechneten Vorverzögerung von 21.866ms bei einer Distanz von 7.585 Meter, entspricht die Vorverzögerung durch den *negdelay* Standardwert von 20 Meter stattdessen 58.309ms:

$$\text{Pre-Delay}_{default} = \frac{20.0m}{343m/s} \cdot 1000 = 58.309ms$$

Latenz-Verursacher	Latenz
Latenz von WONDER mit Messtechnik (128 Samples)	ca. 70ms
-Latenz von WONDER (Vorverzögerung für 20 Meter)	ca. 58ms
Mikrofon und Audiopuffer des FireWire-Interfaces	**ca. 12ms**

Tabelle 3.6: Latenz der Messtechnik mit WONDER bei einer Audiopuffergröße von 128 Samples

[21] https://github.com/dvzrv/wonder/raw/master/documentation/OSC_and_Commandline_Overview.pdf - Abruf: 2016-12-16

Über die Latenz von ca. 58ms, die für das Rendering fokussierter Klangquellen in WONDER stets präsent sind, kann nun auf die bis hierhin unbekannte Latenz der Messtechnik geschlossen werden (siehe Abschnitt 3.2.1). Diese Beträgt, abzüglich der ca. 58ms Latenz von WONDER, einem Wert von ca. 12ms (siehe Tabelle 3.6). Angesichts der analog-digital Wandlung für die Signale des Mikrofons, der internen Audiopuffer des FireWire-Interfaces sowie der FireWire-Verbindung zur DAW, entspricht die Latenz von ca. 12ms einem praxisnahen Wert.

Unerklärlich bleibt weiterhin, weshalb der Startparameter *negdelay* nicht in das Startskript von tWONDER eingefügt werden konnte, ohne dass dabei die WFS-Anlage nicht mehr startet. Innerhalb der im nächsten Abschnitt folgenden Code-Analyse wird geprüft, wie das Parsen dieses Startparameters umgesetzt worden ist.

Eine weitere Besonderheit, die sich aus den Startskripten herauslesen lässt, ist die Tatsache, dass die Zahl der gestarteten tWONDER-Instanzen genau der Anzahl an Prozessorkernen entspricht, die im WFS-Renderer Computer zur Verfügung stehen. Das lässt darauf schließen, dass es sich bei tWONDER um eine Single-Thread-Implementierung handelt.

3.6 WONDER - Quellcode-Analyse der Komponenten

Die Quellcode-Analyse der WONDER-Software beruht auf den Beobachtungen und Schlussfolgerungen aus der Analyse zu den übermäßigen Latenzen und Amplitudenschwankungen aus Abschnitt 3.2 und der Analyse des Startvorganges aus Abschnitt 3.5. Dabei wird sich ebenfalls nur auf die für das WFS-Rendering zuständigen Komponenten cWONDER und tWONDER beschränkt. Als Basis dient die WONDER-Version 3.1.9[22], die im WFS-Labor der HAW Hamburg zum Einsatz kommt, einschließlich aller in Unterabschnitt 3.1.5 beschriebenen Modifikationen.

3.6.1 Analyse von cWONDER auf Latenzen

Die zentrale Kommunikationseinheit cWONDER ist nach dem in Abschnitt 3.3.1 beschriebenen Stream-Modell aufgebaut, welches in der Klasse Cwonder in der Datei cwonder.cpp implementiert ist. Aufgrund des umfangreichen Quellcodes soll an dieser Stelle bewusst auf Auszüge verzichtet werden. Stattdessen soll vielmehr das dahinter stehende Konzept veranschaulicht und die damit verbundenen Latenzen aufgezeigt werden. Da das Konzept nicht nur für das WFS-Rendering in tWONDER, sondern auch für alle anderen WONDER-Komponenten zum Einsatz kommt, die von cWONDER abhängen, ist das Gesamtsystem als solches und alle Teilnehmer davon betroffen.

[22] Aktuellste Version zurzeit ist 3.2.0 (2016-12-16)

Für die Analyse der Latenz wird zunächst schrittweise ein beispielhafter Kommunikationsverlauf im Stream-Modell betrachtet:

1. Ein Teilnehmer im Render Stream, zum Beispiel ein Tracking-Programm, schickt eine OSC-Nachricht mit den Positionsdaten einer Klangquelle zum Render Stream
2. cWONDER erhält diese OSC-Nachricht, speichert deren Werte und verteilt sie an alle Teilnehmer des Render Streams
3. Alle Teilnehmer des Render Streams, darunter auch der Absender, erhalten eine OSC-Nachricht mit den Positionsdaten

Problematisch ist in diesem Szenario vor allem die Benachrichtigung aller Teilnehmer eines Streams im dritten Schritt. Das betrifft nicht nur den im Umfang dieser Analyse betrachteten Render Stream, sondern auch alle anderen verfügbaren Streams in cWONDER. Der Empfänger einer OSC-Nachricht kann nicht auseinanderhalten, ob die OSC-Nachricht von ihm selbst, einem anderen Teilnehmer im selben Stream oder von Außen an cWONDER mit dem Ziel eines Streams geschickt worden ist. Der offensichtliche Vorteil, die Teilnehmer einer Gruppe über eine zentrale Vermittlungsstelle auf demselben Wissensstand zu halten, kann sich dadurch schnell zum Nachteil entwickeln. Denn im obigen Beispiel fehlt die Latenz des Netzwerkes, in dem sich die Teilnehmer befinden. Da vom Moment des Absendens, bis zum erneuten Erhalt der eigenen OSC-Nachricht über einen Stream, eine gewisse Zeit vergeht, aktualisiert der Teilnehmer eines Streams, der gleichzeitig der Absender einer Nachricht ist, seinen eigenen Status mit Werten aus der Vergangenheit. Befinden sich diese Latenzen in einem kabelgebundenen Netzwerk im unteren, einstelligen Millisekunden-Bereich, können sie zum Beispiel in einem WLAN auch auf dutzende Millisekunden ansteigen.

3.6.2 Fazit zur Analyse von cWONDER auf Latenzen

Auch wenn die beschriebenen Latenzen nicht direkt das WFS-Rendering betreffen, so wirken sie sich doch indirekt auf die Kommunikation und Interaktion mit der WFS-Anlage aus. Die in der Implementierung des Stream-Modells entstehenden Rückkopplungen für den Absender einer OSC-Nachricht, könnten durch eine Filterung der Absenderadresse unterbunden werden. Denkbar wäre auch der Einsatz eines Empfang-Time-outs, direkt nach dem Absenden einer OSC-Nachricht. Eine gewisser Rest an Latenz bliebe dabei jedoch weiterhin bestehen, weil Teilnehmer sich nicht gegenseitig aktualisieren, sondern den Zwischenschritt über cWONDER gehen müssen. Das ließe sich mit einer Implementierung auf Basis einer Multicast-Gruppe vermeiden.

3.6.3 Analyse des Startvorganges von tWONDER

Die gewonnenen Erkenntnisse aus Unterabschnitt 3.5.1 zum *negdelay* Startparameter von tWONDER lassen darauf schließen, dass es Fehler beim Parsen des Startparameters geben muss, da die WFS-Anlage bei dessen Nutzung nicht mehr gestartet werden kann.

Das Parsen der Startparameter erfolgt in der Klasse *TwonderConfig*, welche in der Datei twonder_config.cpp implementiert worden ist. Dort findet sich in Zeile 70 für *negdelay* der Standardwert von 20 Meter, welcher auch in der WONDER-Dokumentation beschrieben steht.

```
69 soundSpeed    = 340.0;
70 negDelayInit = 20.0;
```

In der Methode *parseArgs*, beginnend in Zeile 101, werden die Startparameter von tWONDER geparst. Alle zur Verfügung stehenden Startparameter werden im *struct long_options[]* ab Zeile 108 aufgelistet. Dort findet sich auch *negdelay*:

```
121 {"negdelay",       required_argument, 0, 1   },
122 {"speedofsound",   required_argument, 0, 2   },
```

Die Zahlen 1 und 2 am Ende der Zeilen sind jeweils die Definition eines Listeneintrags für „lange“ Startparameter. Bei deren Benutzung wird nicht nur ein Buchstabe angegeben, sondern ein ganzes Wort, welches optional auch ein Argument erfordern kann (*required_argument*), zum Beispiel:

Kurzer Startparameter: -v

Langer Startparameter mit Argument: --negdelay 7.585

Innerhalb einer *switch-case* Anweisung, beginnend in Zeile 134, werden die an tWONDER übergebenen Startparameter geparst. Die beiden langen Argumente, die mit den Zahlen 1 und 2 definiert worden sind, werden ab Zeile 186 verarbeitet. Dabei entspricht die Variable *optarg* den übergebenen Startparametern und wird mit der Methode *atof* zu einem Float-Wert umgewandelt:

```
186 case 1:
187   soundSpeed = atof( optarg );
188   break;
189
190 case 2:
191   negDelayInit = atof( optarg );
192   break;
```

Was nicht sofort ins Auge fällt, ist die Tatsache, dass die Nummerierung vertauscht worden ist. Für das Argument *negdelay* wurde in Zeile 121 der *case* mit der Nummer 1 definiert. Bei der Bearbeitung von *case* 1 wird das übergebene Argument aber nicht *negdelay* zugewiesen, sondern *soundSpeed*, also der für die Berechnungen verwendeten Schallgeschwindigkeit.

3.6.4 Fazit zur Analyse des Startvorganges von tWONDER

Der Zahlendreher in der *switch-case* Anweisung der Datei twonder_config.cpp erklärt, warum die WFS-Anlage bei der Benutzung des Startparameters *negdelay* abstürzt. Statt den Wert des Startparameters *negdelay* der gleichnamigen Variable zuzuweisen, wird dieser der Variable *soundSpeed* zugewiesen. Das verringert die Schallgeschwindigkeit bei den Berechnungen von 340m/s auf 7.585m/s, wodurch tWONDER zum Absturz gebracht wird. Dieser Bug ist nicht nur in WONDER Version 3.1.9, welche im WFS-Labor der HAW Hamburg zum Einsatz kommt, sondern auch in der aktuellen Version 3.2.0 zu finden.

3.6.5 Analyse von tWONDER auf Latenzen

Bei der Suche nach Stellen im Quellcode von tWONDER, die im laufenden Betrieb für Latenzen sorgen könnten, lässt sich aufgrund der verwendeten JACK-Bibliothek genau eine Methode ausfindig machen. Dieses ist die Callback-Methode des JACK-Clients, der in tWONDER dafür verwendet wird, um auf die Audiosignale für das WFS-Rendering zuzugreifen. Weil die Callback-Methode realtime-fähig sein muss, gilt es bei der Analyse auf die in Unterabschnitt 3.4.2 genannten Aufrufe zu achten, die es innerhalb der Callback-Methode zu vermeiden gilt. Diese würden zu Aussetzern bei der Wiedergabe führen oder einen Programmabsturz zur Folge haben. Da die Audiowiedergabe der WFS-Anlage jedoch fehlerfrei funktioniert, kann davon ausgegangen werden, dass innerhalb der Callback-Methode die Bereiche für WFS-Rendering und Verwaltung der Audiopuffer einwandfrei funktionieren.

Eine Analyse von tWONDERs Callback-Methode *process*, die in der Datei twonder.cpp ab Zeile 784 implementiert ist, läuft darauf hinaus, dass neben dem funktionierenden WFS-Rendering nur ein weiterer Methodenaufruf erfolgt:

```
789 realtimeCommandEngine->evaluateCommands( ( wonder_frames_t ) 10 );
```

An dieser Stelle werden mit dem Aufruf von *evaluateCommands* die Kommandos für die nächsten 10 „Sample-Sekunden“ (Sekunden × Samplerate) ausgewertet. Die Einheit Sample-Sekunden ist ein interner Zeitgeber von WONDER und besitzt eine Auflösung auf Sample-Ebene. Die Kommandos bestehen aus dem Inhalt von zuvor empfangenen OSC-Nachrichten und werden einheitlich mit der Methode put im *realtimeCommandEngine*-Objekt abgelegt.

Die Klasse *RTCommandEngine* ist in der Datei rtcommandengine.cpp implementiert. Intern setzt sie auf eine *CommandQueue*, die in commandqueue.cpp implementiert ist. Die Deklaration der *CommandQueue* in der Datei commandqueue.h zeigt, dass sich diese vom JACK-Ringpuffer ableitet (siehe Abbildung 3.8).

Abbildung 3.8: RTCommandEngine mit CommandQueue, die von JACK-Ringpuffer ableitet

Das ist, wie in Unterabschnitt 3.4.2 beschrieben, eine gute Möglichkeit, um genau zwei Threads einen lock-free Datenaustausch zu ermöglichen. Diese Bedingung trifft auch auf tWONDER zu, denn es besteht aus einem OSC-Thread für den Empfang von OSC-Nachrichten sowie einem JACK-Thread für den Zugriff auf die Audiosignale.

```
73 class CommandQueue : private JackRingbuffer
74 {
75 public:
76   CommandQueue();
77   ~CommandQueue();
78
79   void put( Command* command );
80
81   CommandList* get();
82
83 private:
84   pthread_mutex_t mutex;
85 };
```

```
109 void CommandQueue::put( Command* command )
110 {
111   CommandList* commandList = new CommandList( command );
112
113   pthread_mutex_lock( &mutex );
114
115   while( ! write( reinterpret_cast< void* >( commandList ) ) )
116     usleep( 100 );
117
118   pthread_mutex_unlock( &mutex );
119 }
```

Bei der Klassendeklaration von *CommandQueue* in commandqueue.h fällt in Zeile 84 ein Mutex auf. Der JACK-Ringpuffer ist jedoch bereits eine lock-free Implementierung. Zum Einsatz kommt der Mutex in der Methode *put*, die in der Datei commandqueue.cpp implementiert ist. Der Aufbau der Methode dient dem Zweck ein *Command*-Objekt in der *CommandQueue* abzulegen, aus der es in der Callback-Methode mit *evaluateCommands* gelesen wird. Das Ablegen selbst passiert mit dem Aufruf von *write* in Zeile 115. Das *write* gehört zum JACK-Ringpuffer und ist damit lock-free. Folgende Aufrufe sind es hingegen nicht:

- **Zeile 111:** Speicherallokation mit *new*
- **Zeile 113:** Einsatz eines Mutex
- **Zeile 116:** Einsatz von Sleep
- **Zeile 118:** Einsatz eines Mutex

Die Nutzung dieser Synchronisationsmechanismen sowie die Speicherallokation in Zeile 111 sind in diesem Fall unnötig und machen die Methode nicht realtime-safe. Da sie jedoch nicht in der Callback-Methode, sondern im OSC-Thread nach dem Eintreffen einer neuen OSC-Nachricht aufgerufen wird, bleibt dieser Umstand ohne Folgen für die Callback-Methode. Hier wird innerhalb von *evaluateCommands* das Äquivalent dazu, nämlich die Methode *get* aufgerufen:

```
122 CommandList* CommandQueue::get()
123 {
124   return reinterpret_cast< CommandList* >( read() );
125 }
```

Diese ist ebenfalls in commandqueue.cpp implementiert und nutzt in Zeile 124 einzig die *read* Methode des JACK-Ringpuffers, welche lock-free ist und somit der Aufruf von *get* im Rahmen der Callback-Methode realtime-safe bleibt.

3.6.6 Fazit zur Analyse von tWONDER auf Latenzen

Aufgrund der Eingrenzung zu Beginn der Analyse konnte der Bereich möglicher Fehlerquellen auf die aufgerufene Methode *evaluateCommands* innerhalb der Callback-Methode des JACK-Clients reduziert werden. Das Auslesen und Auswerten der *Command*-Objekte aus der *CommandQueue*, einer auf dem JACK-Ringpuffer basierenden Implementierung, erfolgt dabei lock-free und damit realtime-safe, was auch durch das Fernbleiben von Aussetzern oder Fehlern bei der Wiedergabe auf der WFS-Anlage bestätigt wird.

Der Schreibvorgang der *Command*-Objekte innerhalb der *CommandQueue* ist trotz der Implementierung des JACK-Ringpuffers nicht realtime-safe. Der Grund dafür liegt daran, dass neben den lock-free Methoden auch noch ein Mutex sowie ein Sleep verwendet werden. Für den Einsatz in einem Realtime-Programm sollte bei Puffern der maximal notwendige Speicher vorab bestimmt werden. Die Allokation von Speicher während des Realtime-Betriebes ist nicht immer realtime-safe. Die in der *CommandQueue* verwendete Methode *put* zum Abspeichern von *Command*-Objekten könnte dahingehend abgeändert werden, dass vorab genug Speicher für die *CommandQueue* alloziert wird und die *put* Methode nur noch die lock-free Methode *write* des JACK-Ringpuffers verwendet, um die *Command*-Objekte abzuspeichern.

Die Tatsache, dass der Schreibvorgang der *CommandQueue* nicht realtime-safe ist, wirkt sich zwar nicht auf die Latenz des WFS-Renderings und den JACK-Thread aus, ist aber zum Nachteil des OSC-Threads, der durch den Mutex und das Sleep unnötig ausgebremst wird.

3.6.7 Analyse von tWONDER auf Amplitudenschwankungen

Der Bereich für eine Analyse des Quellcodes von tWONDER auf Amplitudenschwankungen lässt sich auf ähnliche Art und Weise eingrenzen, wie bei der Analyse des Quellcodes auf Latenzen (siehe Unterabschnitt 3.6.5). Aus dem Fazit für die Beobachtungen zu den Amplitudenschwankungen in Abschnitt 3.2.2 ergeben sich zwei Schwerpunkte bei der Analyse. Zum einen, ob und wie Tapering eingesetzt wird, um damit die in den Raumecken befindlichen Lautsprecher zu dämpfen und den Truncation-Effekt zu verringern. Zum anderen, ob Fensterfunktionen vorhanden sind, die für den Fall, dass eine Punktquelle einen Lautsprecher durchtritt, eine Approximation der Amplitude durchführen. Das ist nötig, weil sich im Moment des Durchtritts die Ansteuerungsfunktionen ändern und die Möglichkeit einer Division durch Null besteht.

Da beide angegebenen Schwerpunkte die Audiosignale beeinflussen müssen und die JACK-Bibliothek zum Einsatz kommt, lässt sich die Suche nach Fehlerquellen im Quellcode von tWONDER auf die Callback-Methode des JACK-Clients eingrenzen. Diese Methode mit dem Namen *process* ist in der Datei twonder.cpp ab Zeile 784 implementiert. Darin finden die Berechnungen der Amplituden sowie Verzögerungen der Klangquellen in zwei verschachtelten *for*-Schleifen statt. Dabei iteriert die äußere *for*-Schleife über alle Lautsprecher und die innere über alle Klangquellen. Der Beginn für die Berechnung von Amplitude und Verzögerung ist in Zeile 827 der Datei twonder.cpp:

```
827 DelayCoeff c = (*sourcesIt)->source->getDelayCoeff(**speakersIt,
                                                       *listeners);
```

Die Klasse *DelayCoeff* speichert sowohl die Ergebnisse für die Amplitude, als auch für die Verzögerung einer Klangquelle. Der Methodenaufruf *getDelayCoeff* findet aufgrund der verschachtelten *for*-Schleifen für alle Permutationen zwischen Lautsprecher (*speakersIt*) und Klangquelle (*sourcesIt*) statt.

Auf Basis der Ableitungshierarchie in Abbildung 3.9, die in der Datei source.h deklariert ist, wird die Methode *getDelayCoeff* bei ihrer Ausführung auf eine Punktquelle (*PointSource*) oder eine Linearquelle (*PlaneWave*) angewendet. Die Implementierung der jeweiligen Methode ist in der Datei source.cpp zu finden und bildet somit den Ausgangspunkt für die Analyse der Amplitudenschwankungen.

Abbildung 3.9: Ableitungshierarchie für PointSource und PlaneWave

Bei der Betrachtung des Quellcodes in der Datei source.cpp fällt zunächst auf, dass weder für die Linearquelle, noch für die Punktquelle ein Amplituden-Korrekturfaktor existiert. Auch ein „Pre-Equalization“ Filter kommt bei beiden nicht zum Einsatz. Das lässt den Schluss zu, dass die Ansteuerungsfunktionen auf 2D-WFS und einer einfachen Gewichtung sowie Verzögerung der Audiosignale basieren und deshalb die in Abschnitt 2.2 genannten Korrekturmaßnahmen für die 2.5D-WFS keine Anwendung finden.

Ein Tapering und damit die Dämpfung der Enden des Lautsprecher-Arrays kommt ebenfalls nicht zum Einsatz. Das erklärt die Amplitudenschwankungen in den Raumecken des WFS-Labors, hervorgerufen als Folge des Truncation-Effekts.

Somit bleibt bei der Analyse des Quellcodes nur noch darauf zu achten, ob die Ansteuerungsfunktion für Punktquellen eine Fensterfunktion besitzt, die beim Durchtritt einer Punktquelle durch den Lautsprecher eine Approximation der Amplitude durchführt. Die Ansteuerungsfunktion für Linearquellen wird dabei außer Acht gelassen, da diese keine veränderliche Position besitzen, sondern nur einen Winkel aus dem sie stammen.

Die Implementierung der Methode *getDelayCoeff* der Klasse *PointSource* befindet sich in der Datei source.cpp und beginnt in Zeile 100. Nach den Berechnungen aller Vektoren und Normalen, die für die Bestimmung der Amplitude und der Verzögerung einer Punktquelle benötigt werden, wird in Zeile 118 eine Variable mit dem Namen *transitionRadius* definiert. Zudem ist dort ein hilfreicher Kommentar vermerkt:

```
116 // define a circular area around the speakers in which we adjust
        the amplitude factor to get a smooth
117 // transition when moving through the speakers ( e.g. from focussed
        to non-focussed sources )
118 float transitionRadius = twonderConf->speakerDistance * 1.5; //XXX:
        empirical value
```

Es wird also ein Radius definiert, der dem Abstand zwischen zwei Lautsprechern entspricht und mit dem empirischen Faktor von 1.5 multipliziert wird. Im weiteren Verlauf wird mit dem positiven „Transitions-Radius“ zum einen festgestellt, ob eine Punktquelle fokussiert ist oder nicht. Zum anderen dient der Wert bei den Amplitudenberechnungen als Approximation. Gelangt eine Punktquelle in den Transitions-Radius, wird deren Amplitude aus dem Mittelwert zweier Amplituden bestimmt, die sich aus der Position einer Punktquelle $\pm$ dem Transitions-Radius berechnen lassen.

3.6.8 Fazit zur Analyse von tWONDER auf Amplitudenschwankungen

Die Analyse des Quellcodes zeigt, dass bei der Implementierung von tWONDER nur zum Teil etwas gegen mögliche Amplitudenschwankungen unternommen worden ist. Für den Fall, dass sich eine Punktquelle durch die Lautsprecher hindurch bewegt, wurde eine Approximation umgesetzt. Somit wird zum einen eine Division durch Null verhindert, wenn sich eine Punktquelle genau auf der Position eines Lautsprechers befindet. Zum anderen wird beim damit verbundenen Wechsel, von der normalen zur fokussierten Ansteuerungsfunktion einer Punktquelle, ein weicherer Übergang in der Amplitude geschaffen.

Bei der Berechnung des Transitions-Radius hinterlässt die Nutzung eines empirischen Wertes, der zusätzlich auch noch „hard-coded“ im Quellcode hinterlegt worden ist, einen fragwürdigen Eindruck. Gerade weil der Wert empirisch ist, sollte er parametrisiert werden können, zum Beispiel in der Klasse *TwonderConfig*, in der auch Konfigurationsdateien und Startparameter geparst werden.

Eine andere, als die oben genannte Maßnahme gegen Amplitudenschwankungen wurde in tWONDER nicht umgesetzt. Das Fehlen von Amplituden-Korrekturfaktoren und Pre-Equalization Filtern ist der Tatsache geschuldet, dass WFS-Ansteuerungsfunktionen bei der Implementierung benutzt worden sind, die diese Aspekte nicht berücksichtigen.

3.7 WONDER - Fazit zur Analyse der Software

In diesem Abschnitt sollen die Erkenntnisse, die bei der Analyse von WONDER gewonnen worden sind im Hinblick auf die zukünftige Verwendung und Modifikation der Software diskutiert werden. Dabei gilt es den Aufwand zu bewerten, der nötig wäre, um die gefundenen Fehler zu beheben oder fehlende Funktionen zu implementieren. Dieser Aufwand soll anschließend in Relation zu einer kompletten Reimplementierung der Software gesetzt werden.

Der Startvorgang von WONDER beinhaltet eine Vielzahl von Startskripten. Sie bergen nicht nur die Gefahr möglicher Fehlerquellen, sondern sorgen auch für ein träges Hochfahren der gesamten WFS-Anlage. Bei der Analyse der JACK-Bibliothek wurde nach Möglichkeiten gesucht, um die Anzahl an Startskripten zu verringern. Es entstand der Prototyp eines JACK-Clients, der in der Lage ist den Standard JACK-Server zu starten und sich mit diesem zu verbinden. Auf diese Weise konnte die Funktionalität aller Startskripte von tWONDER direkt in den Prototypen implementiert werden. Die Integration dieses Verhaltens in tWONDER würde jedoch zu einem aufwendigen Eingriff in dessen Implementierung des JACK-Clients führen. Dafür wäre das Ergebnis ein enorm vereinfachter Startvorgang der WFS-Anlage, der schneller und sicherer wäre.

Für den Kommunikationsteil von WONDER lies sich in der Analyse feststellen, dass das verwendete Stream-Modell ein guter Ansatz ist, um eine Gruppe von Teilnehmern in einem verteilten System auf derselben Datenbasis arbeiten zu lassen. Allerdings sorgt der Zwischenschritt über die Komponente cWONDER für Latenzen, die vor allem bei Funkverbindungen zum Nachteil werden können. Um die Vorgabe eines verteilten Systems mit der Anforderung einer geringen Latenz zu erfüllen, müsste die Komponente cWONDER entfernt oder umgangen werden. Eine ideale Möglichkeit dafür stellt die Nutzung einer Multicast-Gruppe dar. Das Funktionsprinzip ist dem Stream-Modell von cWONDER sehr ähnlich, allerdings ist die Netzwerk-Hardware für die Verteilung der Nachrichten zuständig. Sobald also ein Teilnehmer seine OSC-Nachricht an die Multicast-Gruppe sendet, sorgt die Netzwerk-Hardware, im WFS-Labor wäre das ein Switch, automatisch für die Verteilung dieser OSC-Nachricht an alle übrigen Teilnehmer. Der Zwischenschritt sowie die dadurch verursachte Latenz von cWONDER entfällt, genauso wie cWONDER selbst. Jedoch ist der Aufwand für die Modifikationen an den übrigen WONDER-Komponenten sehr hoch, da der Kommunikationsteil jeder verbleibenden Komponente reimplementiert werden müsste.

Bei der Implementierung des WFS-Renderings konnten, bis auf die Approximation von Punktquellen beim Durchtritt durch einen Lautsprecher, keine weiteren Maßnahmen gegen den Truncation-Effekt oder Amplitudenschwankungen gefunden werden.

Der Grund dafür liegt bei den eingesetzten WFS-Ansteuerungsfunktionen. Für mehr Kontrolle über die Amplituden von Klangquellen sowie Lautsprechern und für eine Filterung des Audiosignals, müsste das gesamte WFS-Rendering reimplementiert werden. Nur konfigurierbare WFS-Ansteuerungsfunktionen und Filter machen es möglich das Klangbild der WFS-Anlage zu verbessern, da dieses von Faktoren wie dem Raum und den eingesetzten Lautsprechern abhängig ist.

Der Istzustand gleicht gewissermaßen einer Zwickmühle. Zwar läuft die WFS-Anlage mit den bestehenden Problematiken im Moment reibungslos, jedoch bedeutet jede Veränderung, die zu einer deutlichen Verbesserung der Netzwerkkommunikation oder der Audioqualität beitragen würde, dass ein Großteil der Software reimplementiert werden müsste. Da für den Betrieb der WFS-Anlage nur die beiden Komponenten cWONDER und tWONDER zum Einsatz kommen, gleicht die Modifikation einer einzelnen Komponente bereits einer Reimplementierung der Hälfte der eingesetzten Software. Hinzu kommen noch andere Auffälligkeiten, die bei der Nutzung der Software festgestellt worden sind:

- Die Single-Thread-Implementierung von tWONDER erfordert das lokale Starten von 8 Instanzen, um alle Kerne der CPU auszunutzen
- Aufgrund der 8 tWONDER-Instanzen laufen nicht nur 8 JACK-Client-Threads, sondern auch 8 OSC-Client-Threads, obwohl schon ein OSC-Client-Thread ausreichen würde
- Der JACK-Ringpuffer für die lock-free Kommunikation kann nur von 2 Threads verwendet werden. Eine Erweiterung auf Multithreading-Betrieb, um nicht immer 8 tWONDER-Instanzen starten zu müssen, ist nicht ohne weiteres möglich
- Die CPU-Auslastung jeder tWONDER-Instanz liegt immer bei 30-40%, selbst wenn keine Klangquellen gerendert werden
- Im WLAN des WFS-Labors machte sich bei der Nutzung einer Multitouch-GUI bereits die „Absender-Rückkopplung“ von cWONDER durch die Latenz des Stream-Modells störend bemerkbar. Daher könnte der Einsatz von cWONDER bei zukünftigen Augmented, Mixed und Virtual Reality Applikationen möglicherweise nicht responsiv genug sein und Probleme wie „Cyber Sickness“ verursachen [Nog15, Kapitel 2.10]

Mit Blick auf die Verbesserung der WFS-Anlage für zukünftige Anwendungsszenarien und den bei der Analyse gewonnenen Erkenntnissen im Rücken, macht die vollständige Reimplementierung der WFS-Software an dieser Stelle mehr Sinn, als das stellenweise Ausbessern einzelner Komponenten.

4 Reimplementierung

Die Analyse der im WFS-Labor der HAW Hamburg eingesetzten Software WONDER hat ergeben, dass sich die optimierbaren Stellen der Software in ihren beiden Kernkomponenten befinden und nur im Rahmen einer aufwendigen Reimplementierung verbessert werden können. Aus diesem Grund wurde die Entscheidung getroffen, dass es sinnvoller ist die gesamte Software neu zu implementieren, anstatt sie stellenweise auszubessern. Gestützt wurde die Entscheidungsfindung durch eine vorangegangene, ausführliche Analyse der Software WONDER sowie die Auseinandersetzung mit den verwendeten Bibliotheken. Für ein besseres Verständnis deren Funktionalität wurden kleine Prototypen erstellt. Anhand dieser konnte letztlich auch der bevorstehende Aufwand und der daraus resultierende Nutzen festgemacht werden, der die Entscheidung bekräftigte eine Reimplementierung der vorhandenen Software durchzuführen.

Im weiteren Verlauf dieses Kapitels wird der Prozess der Reimplementierung etappenweise durchlaufen. Zu Beginn geht es um das Konzept der neuen Software und welche Anforderungen an diese gestellt werden. Hiernach wird ein Blick auf die verwendeten Bibliotheken geworfen. Es folgt der Aufbau und die Umsetzung der Softwarearchitektur. Zum Schluss kommt eine Bewertung der Reimplementierung.

4.1 CoRGII - Motivation

CoRGII (oder auch CoRGI²) steht für „**Co**ntrolled **R**enderer, **G**raphical User Interface and **I**mmersive **I**nteraction“ und benennt die Software, die im Rahmen dieser Bachelorarbeit ihr Fundament erhalten und in Zukunft weiter ausgebaut und erweitert werden soll. Die Namensgebung versteht sich als ein Triumvirat aus Softwarekomponenten, welche die Aufgabenstellungen und Einsatzzwecke im WFS-Labor widerspiegeln und in einem verteilten System eingesetzt werden sollen. Dabei fließt das gewonnene Wissen aus der Arbeit mit der WFS-Anlage sowie die gesammelten Erkenntnisse aus der Analyse von WONDER in die Implementierung mit ein. Gleichzeitig sollen die zusammengetragenen Informationen anderen dazu dienen einen leichteren Einstieg in das Themengebiet und dem Umgang mit der Software zu erhalten, indem bei der Entwicklung auf Nachvollziehbarkeit und das Erstellen einer Dokumentation geachtet wird.

4.2 CoRGII - Abgrenzung der Implementierung

Im Rahmen dieser Bachelorarbeit erblickt CoRGII zum ersten Mal das Licht der Welt. Das Hauptaugenmerk liegt daher auf der Implementierung der Kernkomponente von CoRGII, welche durch den „Controlled Renderer“ dargestellt wird. Die beiden anderen Komponenten „Graphical User Interface “ und „Immersive Interaction“ werden nur in ihrem Zweck und der Funktionalität beschrieben, aber nicht implementiert.

4.3 CoRGII - Aufbau

Der Aufbau von CoRGII ist in drei Komponenten unterteilt. Jede für sich stellt unterschiedliche Funktionalitäten zur Verfügung. Die Kommunikation zwischen den einzelnen Komponenten von CoRGII geschieht über OSC-Nachrichten innerhalb einer Multicast-Gruppe.

4.3.1 Benutzersichten

Da der Controlled Renderer nicht nur das einfache Positionieren und Einstellen von Klangquellen, sondern auch die Parametrisierung des Audio-Renderers und der Lautsprecher zur Verfügung stellt, sollte bei der Umsetzung einer GUI mit Bedacht ausgewählt werden, welche Funktionen einem Nutzer zur Verfügung stehen sollen oder nicht. Selbiges gilt natürlich auch bei einer Umsetzung für Augmented Reality (AR), Mixed Reality (MR) oder Virtual Reality (VR). Im Groben lassen sich die Nutzer in drei Nutzergruppen unterteilen:

- **Endnutzer:** „Spielt“ mit der Audio-Szene, indem er Klangquellen verschiebt oder unterschiedliche Arten von Klangquellen (Punktquelle, Linearquelle) auswählt
- **Autor:** Erstellt Klanginstallationen oder Audio-Szenen. Mischt Musikstücke oder Filme ab, indem er Klangquellen in der Audio-Szene bewegt. Performt einen Live-Auftritt
- **Ingenieur:** Prüft die Funktionalität eines Audio-Renderers und dessen Parameter. Nimmt Korrekturen an den Parametern von Audio-Renderer und Lautsprechern vor. Misst eine WFS-Anlage ein oder justiert Fensterfunktionen (Tapering)

Alle drei Benutzergruppen können für die Interaktion sowohl eine GUI, als auch AR, MR und VR nutzen. Den Zugriff auf interne Renderer-Parameter sollte jedoch nur dem Ingenieur vorbehalten sein, um die WFS-Anlage nicht zu beschädigen.

4.3.2 Controlled Renderer

Der Controlled Renderer ist die Basiskomponente für das Audio-Rendering. Dieser ist für den Einsatz in einem verteilten System konzipiert und lässt sich über ein Netzwerk steuern. Die wesentlichen Funktionen, die der Controlled Renderer zur Verfügung stellt sind:

- Verteiltes Audio-Rendering, im speziellen 2.5D-WFS
- Echtzeit-Parametrisierung des Audio-Renderers
- Empfang von Tracking-Daten eines Zuhörers oder anderer Targets
- Kommunikation in einer Multicast-Gruppe
- Kommunikation über OSC-Nachrichten

4.3.3 Graphical User Interface

Das Graphical User Interface stellt eine grafische Benutzeroberfläche für CoRGII dar. Dabei ist nicht festgelegt mit welchem Framework, Betriebssystem oder auf welcher Hardware diese implementiert wird. Es kann sich um eine Desktop-Anwendung, eine Smartphone-App oder auch um ein VST-Plugin für eine DAW handeln. Der Funktionsumfang, den eine GUI-Umsetzung bieten könnte, wäre:

- Bewegen von Klangquellen
- Bewegungspfade für Klangquellen anlegen
- Aktivieren / Deaktivieren von Klangquellen
- Wechseln der Klangquellen-Art: Punktquelle oder Linearquelle
- Tracking eines Zuhörers steuern
- Auswahl des Audio-Renderers
- Echtzeit-Parametrisierung des Audio-Renderers
- Kommunikation in einer Multicast-Gruppe
- Kommunikation über OSC-Nachrichten

4.3.4 Immersive Interaction

Immersive Interaction zielt, wie der Name schon sagt, auf eine immersive Interaktion mit der WFS-Anlage ab. Es bietet mit AR, MR und VR die Möglichkeit wesentlich intuitiver mit der Audio-Szene umzugehen. Dabei bringt es nicht nur Vorteile für Endnutzer, die eine Audio-Szene erleben wollen. Für einen Autor ergibt sich die Chance sein eigenes Werk während des Schaffens zu betreten und innerhalb davon zu agieren. Der Ingenieur kann mit einem Blick sehen, welche Parameter der Audio-Renderer, ein Lautsprecher oder eine Klangquelle haben. Gleichzeitig kann er diese Parameter manipulieren und direkt einen Unterschied hören. Das gilt auch für die Auswahl von Fensterfunktionen für die Nutzung von Tapering oder dem Approximieren der Amplitude einer Klangquelle beim Durchtritt durch einen Lautsprecher. Hier kann der Ingenieur einer Klangquelle beim Durchtritt regelrecht zuschauen und direkt hören, ob Amplitudenkorrekturen vorgenommen werden müssen. Hinsichtlich seiner Funktionalität sollte der Bereich Immersive Interaction dasselbe bieten wie eine GUI. Durch AR, MR und VR kommen noch zusätzliche Aspekte hinzu:

- Bewegen von Klangquellen
- Bewegungspfade für Klangquellen anlegen
- Aktivieren / Deaktivieren von Klangquellen
- Wechseln der Klangquellen-Art: Punktquelle oder Linearquelle
- AR/MR/VR: Greifen von Klangquellen
- AR/MR/VR: Feedback durch Controller oder Datenhandschuhe
- AR/MR/VR: Wichtige Daten mit virtuellem „Heads-Up-Display“ (HUD) immer im „Blick“
- Tracking eines Zuhörers steuern
- Auswahl des Audio-Renderers
- Echtzeit-Parametrisierung des Audio-Renderers
- Kommunikation in einer Multicast-Gruppe
- Kommunikation über OSC-Nachrichten

4.4 CoRGII - Anforderungsanalyse

Aufgrund der Tatsache, dass CoRGII in einem verteilten System funktionieren soll und in Zukunft auf unterschiedliche Komponenten aufgeteilt sein wird, tangiert die Software sowie alle involvierten Komponenten gleich mehrere Anforderungsbereiche. Da sich im Rahmen dieser Bachelorarbeit auf den Controlled Renderer beschränkt werden soll, wird die Anforderungsanalyse für eine GUI oder ein immersives System außen vor gelassen.

4.4.1 Funktionale Anforderungen

- **Realtime-Betrieb:** Der Teil der Software, der für die Audioverarbeitung zuständig ist, wird von einem Realtime-Thread ausgeführt. Demnach muss sichergestellt sein, dass alle Berechnungen und Aufrufe innerhalb dieses Threads realtime-safe und lock-free sind.
- **Sequentielle Konsistenz:** Wenn mehrere Threads auf eine Variable zugreifen, muss sichergestellt sein, dass dieser Zugriff atomar passiert. Das heißt, dass das Lesen und Schreiben von Variablen durch multiple Threads so „geregelt“ sein muss, dass ein Lesevorgang erst abgeschlossen sein muss, bevor ein Schreibvorgang startet und umgekehrt.
- **Verteiltes System:** Die Rechenlast beim WFS-Rendering ist so hoch, dass diese auf unterschiedliche Computer verteilt werden muss. Die Software muss in der Lage sein im Verbund mit mehreren Computern zu kommunizieren und die Berechnungen aufzuteilen.
- **Multithreading:** Die aufwendigen Berechnungen des WFS-Renderings sollten zur effizienteren Lösung auf alle CPU-Kerne eines Computers aufgeteilt werden.
- **Geringe Latenz:** Da es sich bei der Software um ein verteiltes System handelt, entstehen auf den Laufwegen zwischen einzelnen Komponenten Latenzen. Diese sollten so gering wie möglich gehalten werden.

4.4.2 Nicht-funktionale Anforderungen

- **Quelloffenheit:** Die Software soll Interessierten und Entwicklern frei zur Verfügung stehen und von diesen modifiziert und benutzt werden können.
- **Dokumentation:** Die Software soll für ein besseres Verständnis und für zukünftige Modifikationen eine Dokumentation aufweisen.
- **Plattformunabhängigkeit:** Die Software soll plattformunabhängig sein und damit auf unterschiedlichen Betriebssystemen kompiliert und ausgeführt werden können.

4.5 CoRGII - Verwendete Bibliotheken

Bei der Auswahl des Compilers und der Bibliotheken, die im Controlled Renderer von CoRGII zum Einsatz kommen, wurde darauf geachtet, dass diese kostenlos und sowohl für Linux, als auch für Windows zur Verfügung stehen. Damit sollte von Anfang an eine Plattformunabhängigkeit geschaffen werden, die es dem Controlled Renderer ermöglicht auch bei einem Wechsel der Umgebung oder beim Betrieb mit gemischten Betriebssystemen seine Funktionalität zu wahren. Dieser Punkt ist vor allem dann wichtig, wenn plattformabhängige Software oder Hardware zum Einsatz kommt, die ein bestimmtes Betriebssystem oder einen bestimmten Treiber voraussetzt.

4.5.1 GNU Compiler Collection

Als Compiler kommt für den in C++11 (ISO/IEC 14882:2011) geschriebenen Controlled Renderer von CoRGII der GNU Compiler zum Einsatz[1]. Dieser ist bei vielen Linux Distributionen bereits standardmäßig installiert oder kann leicht nachinstalliert werden. Für die Windows Plattform gibt es die Portierung MinGW[2]. Diese muss in Form einer Distribution installiert werden. Es gibt mehrere Anbieter, wobei für die Entwicklung von CoRGII auf der Windows Plattform die Distribution MSYS2[3] benutzt worden ist. Auf diese Weise wurde eine plattformunabhängige Entwicklung und Kompilierung sichergestellt.

4.5.2 Lightweight OSC Library

Die Lightweight OSC Library (liblo) ist bereits in Unterabschnitt 3.4.1 ausführlich betrachtet worden. An dieser Stelle sei kurz erwähnt, dass es eine plattformunabhängige Bibliothek zum Senden und Verschicken von OSC-Nachrichten ist. Sie dient dem Controlled Renderer als Kommunikationsschnittelle zu anderer Software auf der Basis von OSC-Nachrichten.

4.5.3 JACK Audio Connection Kit

Das JACK Audio Connection Kit (JACK) ist bereits in Unterabschnitt 3.4.2 ausführlich betrachtet worden. An dieser Stelle sei kurz erwähnt, dass es eine plattformunabhängige Bibliothek zum Zugriff auf die Audio-Hardware eines Computers ist (siehe Abbildung 3.6). Mit dieser wird im Controlled Renderer von CoRGII innerhalb eines JACK-Realtime-Threads das WFS-Rendering berechnet.

[1] https://gcc.gnu.org - Abruf: 2016-12-16
[2] http://www.mingw.org - Abruf: 2016-12-16
[3] https://msys2.github.io - Abruf: 2016-12-16

4.5.4 OpenMP

OpenMP[4] bietet eine auf Compiler-Direktiven basierende Möglichkeit, um die Ausführung von Schleifen im Programmcode auf mehrere Threads zu verteilen [DM98]. Die Bibliothek wird mit dem GNU Compiler ausgeliefert und soll beim Controlled Renderer von CoRGII für das Multithreading eingesetzt werden. Damit soll das WFS-Rendering beschleunigt und die CPU-Kerne des ausführenden Computers effizienter ausgelastet werden. Das Starten mehrerer Instanzen, um alle CPU-Kerne auszulasten, soll im Vergleich tWONDER vermieden werden. Das untere Code-Beispiel zeigt das Einbinden einer OpenMP Compiler-Direktive:

```
1  #include <omp.h>
2
3  ...
4
5  int NUM_THREADS = 4;
6
7  #pragma omp parallel for
8  for(int = 0; i < NUM_THREADS; ++i) {
9    int id = omp_get_thread_num();
10   std::cout << "Thread " << id << ": Hello world!\n";
11 }
12
13 Output:
14 Thread 3: Hello world!
15 Thread 1: Hello world!
16 Thread 0: Hello world!
17 Thread 2: Hello world!
```

In Zeile 7 wird die Direktive für das parallele Abarbeiten einer *for*-Schleife angegeben. Diese reicht aus, um den nachfolgenden Programmcode auf die hier angenommenen 4 CPU-Kerne zu verteilen. Die Ausgabe ab Zeile 13 erfolgt von unterschiedlichen Threads.

4.5.5 TinyXML-2

Die Bibliothek TinyXML-2[5] dient dem Parsen von XML-Dateien. Sie ist sehr kompakt und portabel, weil sie aus nur zwei Dateien besteht. In Controlled Renderer von CoRGII wird sie benutzt, um die Einstellungen des Renderers aus einer XML-Datei zu lesen.

[4] http://www.openmp.org - Abruf: 2016-12-16
[5] http://www.grinninglizard.com/tinyxml2 - Abruf: 2016-12-16

4.5.6 Weitere Software

- **Versionsverwaltung:** Für die Versionsverwaltung kommt Git[6] zum Einsatz
- **Dokumentation:** Für die Dokumentation des Quellcodes wird Doxygen[7] verwendet
- **Kompilation:** Für die Kompilation wird ein *Makefile* benutzt
- **Quellcode Beautifier:** Für die Quellcode-Formatierung wird AStyle[8] mit folgender *.astylerc* Konfigurationsdatei benutzt:

```
--style=stroustrup
--indent=spaces
--indent=spaces=4
--indent-switches
--indent-col1-comments
--max-instatement-indent=120
--break-blocks
--pad-oper
--unpad-paren
--align-pointer=type
--add-brackets
--convert-tabs
--close-templates
--break-after-logical
--align-method-colon
--pad-method-colon=all
--preserve-date
--lineend=linux
```

Die Installation der Bibliotheken unter Linux ist sehr komfortabel und kann über die eingebaute Paketverwaltung durchgeführt werden. Unter Windows ist es komplizierter, weshalb Shell-Skripte für den Einsatz mit MSYS2 und dem GNU Compiler geschrieben worden sind. Diese werden sich in Zukunft im Repository von CoRGII befinden und erlauben es Windows Nutzern die Bibliotheken liblo und JACK mit nur einem Befehl aus deren Repositories zu klonen und automatisch zu kompilieren.

[6] https://git-scm.com - Abruf: 2016-12-16
[7] http://www.stack.nl/~dimitri/doxygen - Abruf: 2016-12-16
[8] http://astyle.sourceforge.net - Abruf: 2016-12-16

4.6 CoRGII - Softwarearchitektur

Der Aufbau der Controlled Renderer Architektur von CoRGII ist in Abbildung 4.1 zu sehen. Sie besteht dabei aus den drei Komponenten, die nun näher erläutert werden sollen.

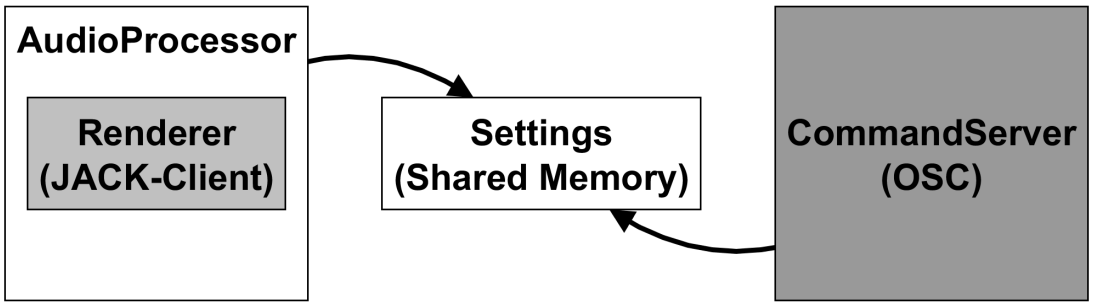

Abbildung 4.1: Aufbau der Controlled Renderer Architektur von CoRGII

4.6.1 Settings

Settings ist ein Shared Memory Objekt. Beim Programmstart liest dieses eine XML-Datei mit den Einstellungen für das gesamte System des Controlled Renderers ein. Diese Einstellungen beinhalten die Positionsdaten der Lautsprecher und die Parameter für AudioProcessor, Renderer und CommandServer. Diese Objekte haben auch jeweils einen Shared Pointer auf das Settings Objekt. Durch diesen Aufbau wird es möglich, dass beim Start bereits alles soweit initialisiert ist, dass kein Speicher mehr im Betrieb alloziert werden muss, was beim Realtime-Betrieb von Vorteil ist. Außerdem ermöglicht der Einsatz des Shared Memory Modells, dass mehrere Threads darauf zugreifen können und dadurch auch Multithreading möglich ist. Intern besteht Settings aus mehreren „passiven Klassen“, die weiter unten behandelt werden. Vorweg sei gesagt, dass diese lock-free implementiert sind und damit die Grundvoraussetzungen für den Multithreading-Betrieb sowie den Realtime-Betrieb erfüllen.

4.6.2 AudioProcessor

Der AudioProcessor ist für den Zugriff auf die Audio-Hardware zuständig. Er ist ein JACK-Client und so implementiert, dass er den Standard JACK-Server automatisch startet. Das Verbinden seiner JACK-Ports erledigt er ebenfalls selbständig. Am WFS-Rendering ist er jedoch nicht beteiligt, denn die Daten aus dem Audiopuffer werden von ihm an den Renderer weitergeleitet. Damit soll eine Trennung der Aufgaben erreicht werden, um Renderer austauschbar zu machen.

4.6.3 Renderer [work in progress]

Der Renderer ist für das 2.5D-WFS-Rendering zuständig und beinhaltet die nötigen Ansteuerungsfunktionen. Hier findet sich auch die Delay Line für die Verzögerung der Lautsprecher sowie der Pre-Equalization Filter zum Filtern der Audiosignale.

Der Renderer konnte leider nicht mehr im Zeitrahmen dieser Bachelorarbeit fertiggestellt werden und fehlt als letztes Glied in der Funktionskette. Bei der weiteren Implementierung wird stets darauf geachtet die Anforderungen an ein Audioprogramm einzuhalten. Das bedeutet realtime-safe und lock-free zu arbeiten. Einen guten Einblick in das Thema der Audioprogrammierung gibt es in einem Vontrag von Timur Doumler auf der CppCon2015 [Dou15].

4.6.4 CommandServer

Der CommandServer ist für die Netzwerkkommunikation in einem verteilten System zuständig und nutzt dafür OSC-Nachrichten. Er ist so implementiert, dass er eine Multicast-Gruppe joint, damit sich um die Verleitung der OSC-Nachrichten die Netzwerkhardware kümmern kann.

Für den korrekten Betrieb in einem Netzwerk sollten alle beteiligten Teilnehmer sich bei einem NTP-Zeitserver synchronisieren. Das hat den Vorteil, dass der Einsatz von OSC-Nachrichten, die mit einem OSC-Timetag Zeitstempel versehen sind, sehr genau ist und somit auch eine sehr genaue Abfolge von Befehlen über die OSC-Nachrichten realisiert werden kann.

4.6.5 Passive Klassen

Die passiven Klassen werden als solche bezeichnet, weil sie keinerlei Funktionalität im eigentlichen Sinne mitbringen. Sie haben einzig *Getter* und *Setter*, um die Werte und Eigenschaften des jeweiligen Objektes zu lesen oder zu schreiben. Dabei sind *Getter* und *Setter* lock-free implementiert und basieren auf den C++ Atomics, die mit C++11 eingeführt wurden [ISO14]. Im C++14 Draft finden sich in Abschnitt 1.10 (Multi-threaded executions and data races) unter Punkt 10 und 22 sowie in Abschnitt 29.3 (Order and consistency) und 29.4 (Lock-free property) die genauen Definitionen. Zusammenfassend wird garantiert, dass beim Einsatz von Atomics ein lock-free Betrieb möglich ist, weil eine sequentielle Konsistenz in der Abfolge einer Befehlsabarbeitung eingehalten wird. Die Deklaration eines Atomics basiert auf Templates und kann mit einfachen Datentypen verwendet werden, zum Beispiel einem *float*:

Float Atomic: std::atomic<float> f = 1.5;

Der Zugriff passiert dabei über gesonderte *Getter* und *Setter*, die die sequentielle Konsistenz sicherstellen und damit eine zeitlich korrekte Abfolge von Lese- und Schreibzugriffen garantieren. Diese Methoden sind:

Lesen: value.load(std::memory_order_acquire)

Schreiben: store(value, std::memory_order_release)

In den nun folgenden passiven Klassen sind die *Getter* und *Setter* deshalb mit den oben genannten *Gettern* und *Settern* des Atomic-Datentyps umgesetzt worden.

Listener

Beinhaltet die Position und die Blickrichtung eines Zuhörers, um die korrekte Amplitude für den Zuhörer berechnen zu können und eine zuhörerbasierte Interaktion mit der WFS-Anlage zu ermöglichen, indem vom Tracking-System Gebrauch gemacht wird.

Loudspeaker

Beinhaltet die Position und Ausrichtung eines Lautsprechers. Eine separate *Attenuation* (Englisch für Dämpfung) Variable ist vorhanden, die während des WFS-Renderings ausgelesen werden kann. In dieser wird die Dämpfung eines Lautsprechers, zusätzlich zu seiner Lautstärke, abgespeichert, um damit ein selektives Tapering einzelner Lautsprecher zu ermöglichen.

Source

Beinhaltet die Position und den Winkel (für Linearquellen und fokussierten Punktquellen) von Klangquellen. Zusätzlich kann der Typ von Linear- zu Punktquelle gewechselt und die Klangquelle deaktiviert werden, um Rechenleistung einzusparen.

4.6.6 Hilfsklassen

Position3D

Beinhaltet die dreidimensionalen Koordinaten einer Position. Dabei ist hier, wie bei den passiven Klassen auch, eine lock-free Implementierung benutzt worden. Alle passiven Klassen nutzen für ihre Position ein *Position3D*-Objekt.

Vector3D

Die Hilfsklasse *Vector3D* wird für die Vektorrechnungen in den WFS-Ansteuerungsfunktionen verwendet.

XMLParser

Der *XMLParser* nutzt die TinyXML-2 Bibliothek und parst die Einstellungen für das Shared Memory *Settings*-Objekt des Controlled Renderers aus einer XML-Datei.

4.7 CoRGII - Fortschritt der Implementierung

Auch wenn der Renderer von CoRGII, als Kern des eigentlichen WFS-Renderings, aufgrund des Umfangs der Reimplementierung noch nicht vollständig fertiggestellt ist, lässt sich sagen, dass der bisherige Fortschritt sehr zuversichtlich stimmt. Viele der Anforderungen konnten bereits umgesetzt werden und auch das Starten der Software in einem kleinen Rahmen unter Stereo-Bedingungen ist erfolgreich geglückt. Es folgt eine Auflistung von Tests und Funktionen, die bereits lauffähig sind und den aktuellen Fortschritt widerspiegeln. Das Fazit und der Ausblick folgen im nächsten Kapitel.

- **Plattformunabhängig:** Der Controlled Renderer kann mittels Makefile unter Linux und Windows fehlerfrei kompiliert werden. Ein Test im Mischbetrieb beider Betriebssysteme in einer Multicast-Gruppe verlief ebenfalls einwandfrei.
- **OSC-Server:** Der CommandServer ist vollständig implementiert und lauffähig, das heißt die OSC-Kommunikation innerhalb einer Multicast-Gruppe funktioniert einwandfrei. Nachrichten mit OSC-Timetags wurden ebenfalls verschickt und im Ziel sehr genau abgearbeitet. Voraussetzung ist jedoch ein zentraler NTP-Server im Netzwerk. Die Latenz liegt dann im einstelligen Millisekundenbereich.
- **JACK-Client:** Der JACK-Client ist vollständig implementiert und lauffähig. Der Lese- und Schreibzugriff auf die Audiopuffer funktioniert einwandfrei. Das Starten des Standard JACK-Servers und das anschließende Routing klappt einwandfrei.
- **Audio-Rendering:** Ein rudimentäres Stereo-Rendering hat funktioniert. Dabei wurde der Renderer bereits innerhalb einer Multicast-Gruppe von einem Zweit-Computer aus gesteuert, die Lautstärke und Dämpfung der Lautsprecher in Echtzeit verändert und die Klangquelle aktiviert und deaktiviert.

5 Zusammenfassung und Ausblick

An dieser Stelle soll abschließend eine Zusammenfassung über das Geleistete gegeben werden.

5.1 Was wurde erreicht?

Zu Beginn wurden die Grundlagen der Wellenfeldsynthese in kompakter Form wiedergegeben. Anschließend wurde sich mit den Problemen der WFS-Anlage beschäftigt. Dazu zählen die zu hohe Latenz, sowie Amplitudenschwankungen. Mit einer Messung der Latenz wurden vorangegangene Messungen verifiziert und damit nochmals belegt, dass etwas mit bestimmten Komponenten der eingesetzten Software WONDER nicht stimmen kann.

Es folgte die tiefgreifende Analyse der für das WFS-Rendering zuständigen Komponenten der Software WONDER. Dabei wurden neben dem Quellcode auch die verwendeten Bibliotheken und deren Umsetzung in WONDER analysiert. Ein Bug konnte ausfindig gemacht werden, der es nicht zulässt einen korrekten Wert für die Vorverzögerung einzustellen und somit immer ein zu hoher Wert für die WFS-Anlage benutzt wird. Damit kann jedoch nur das Problem mit der zu hohen Latenz behoben werden.

Weiterhin wurde festgestellt, dass gegen den Truncation-Effekt und andere Formen der Amplitudenschwankungen keine oder nur unzureichende Maßnahmen ergriffen worden sind. Der Grund dafür liegt in den implementierten WFS-Ansteuerungsfunktionen und kann nicht ohne weiteres behoben werden.

Bei der Umsetzung der Netzwerkkommunikation im Stream-Modell konnte eine Latenz festgestellt werden, die zwar das WFS-Rendering nicht beeinflusst, jedoch bei der Interaktion mit der WFS-Anlage als störend empfunden wird und in zukünftigen Anwendungen im Bereich Virtual Reality zu Problemen führen könnte (Cyber Sickness). Eine Behebung ist nicht ohne weiteres möglich, da auf dem Stream-Modell die Kommunikation der gesamten WONDER-Komponenten basiert.

Ferner wurde der Startvorgang der Software analysiert. Aufgrund zahlreicher Skripte ist dieser langwierig, umständlich zu konfigurieren und fehlerträchtig.

Am Ende der Analyse stand fest, dass Modifikationen an WONDER nötig sind, aber stets die Reimplementierung mindestens einer der beiden Kernkomponenten für das WFS-Rendering beinhalten. Aufbauend auf den während der Analyse gemachten Erfahrungen mit der Software und den verwendeten Bibliotheken, wurde sich für eine Reimplementierung der Software entschieden.

Bei der Reimplementierung entstand die Software CoRGII für das 3D-Audio-Rendering. Diese benutzt dieselben Bibliotheken wie WONDER, aber mit anderen Implementierungsansätzen. Sie ist ebenfalls für ein verteiltes WFS-Rendering ausgelegt, nutzt aber eine Multicast-Gruppe für die Kommunikation. Für den internen Informationsaustausch kommt eine für Multithreading geeignete Shared Memory Architektur zum Einsatz. Der JACK-Client startet und verbindet sich nun vollautomatisch und ersetzt somit die umfangreichen Startskripte.

Der aktuelle Zustand von CoRGII kann als fortgeschritten bezeichnet werden. Die Netzwerkkommunikation und der Zugriff auf die Audiopuffer sind vollständig implementiert und bilden gemeinsam den größten Funktionsumfang des Controlled Renderings. Die noch fehlenden Komponenten des Renderers werden im nächsten Abschnitt beleuchtet.

5.2 Was wurde nicht erreicht?

Auch wenn der Zustand von CoRGII und dem Conrolled Renderer relativ weit voran geschritten ist, fehlt im Bereich des Renderers noch einiges an Funktionalität. Das betrifft zum einen die Algorithmen der Ansteuerungsfunktionen sowie die Umsetzung der Puffer für eine Delay Line. Das Filtern der Audiosignale mit dem Pre-Equalization Filter steht ebenfalls aus. Dafür muss eine Funktion oder Bibliothek zum Einsatz kommen, die eine Fourier Transformation beherrscht, da die Filterung in der Frequenz-Domäne stattfinden muss.

Die Grundsteine für eine multithreading-fähige Architektur wurden bereits mit dem Shared Memory Modell gelegt, jedoch fehlt hier noch der Einsatz der OpenMP Bibliothek, um die noch fehlenden Ansteuerungsfunktionen auf Basis multipler Threads berechnen zu können.

5.3 Ausblick

In diesem Abschnitt soll neben der Auflistung noch fehlender Implementierungsdetails ein Ausblick auf kommende Funktionalitäten gegeben werden, die Teil weiterer Analysen und Arbeiten sein könnten.

- Fertigstellung des Controlled Renderers von CoRGII
- Evaluierung und Kalibrierung des WFS-Renderers für das WFS-Labor der HAW Hamburg
- Optimierung des WFS-Renderers, zum Beispiel *Improved Driving Functions for Rectangular Loudspeaker Arrays Driven by Sound Field Synthesis Equivalent Scattering Approach* [SSR15] oder *Wave Field Synthesis with Primary Source Correction: Theory, Simulation Results, and Comparison to Earlier Approaches* [VF12]
- Erstellen weiterer Renderer, zum Beispiel VBAP [Pul97], um CoRGII auch im *Creative Space for Technical Innovations* einzusetzen
- Erstellen einer Middleware für den Computer des Tracking-Systems im WFS-Labor, um die Tracking-Daten direkt an die neue CoRGII Multicast-Gruppe zu senden
- Erstellen einer grafischen Benutzeroberfläche (2D und als Zwischenschritt für 3D in Immersive Interaction)
- Analysieren, ob es sich lohnt xWONDER auf CoRGII umzubauen, weil es durch den Einsatz von Qt ebenfalls plattformunabhängig ist und es bei dieser Gelegenheit auch erweitert werden könnte:
  - Parametrisierung des Controlled Renderers von CoRGII
  - Anwenden von Tapering mit unterschiedlichen Fensterfunktionen
  - Parametrisieren der Amplituden-Approximation beim Durchtritt einer Klangquelle
  - Speicherung der Zustandsinformationen einer Audio-Szene im SpatDIF Format
- Erstellen einer grafischen Benutzeroberfläche (3D/AR/MR/VR) mit der ebenfalls das Bedienen, Demonstrieren und Konfigurieren der WFS-Anlage im dreidimensionalen Raum und im Echtzeitbetrieb möglich ist [Jan14]
- Bestehendes SPAOP VST-Plugin anpassen, zum Beispiel durch eine Middleware [Han14]

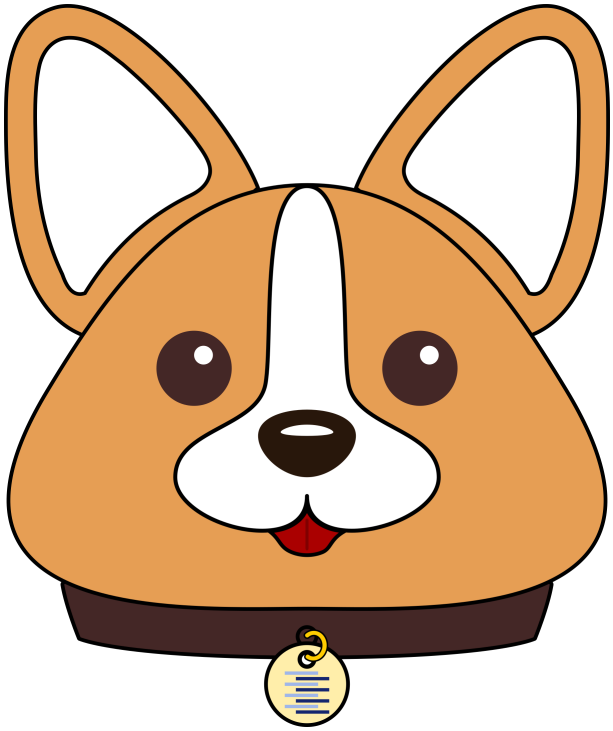

Abbildung 5.1: CoRGII Logo

Literatur

[Ahr12] Jens Ahrens. *Analytic Methods of Sound Field Synthesis*. 2012. Aufl. Berlin Heidelberg: Springer Science & Business Media, 2012. ISBN: 978-3-642-25742-1.

[Baa05] Marije A. J. Baalman. „Updates of the WONDER software interface for using Wave Field Synthesis“. In: *3rd International Linux Audio Conference*. Karlsruhe (ZKM), Germany, Apr. 2005.

[Baa08] Marije A. J. Baalman. *On Wave Field Synthesis and electro-acoustic music, with a particular focus on the reproduction of arbitrarily shaped sound sources*. Saarbrücken: VDM, 2008. ISBN: 978-3-639-07731-5.

[BP04] Marije A. J. Baalman und Daniel Plewe. „WONDER - a software interface for the application of Wave Field Synthesis in electronic music and interactive sound installations“. In: *30th International Computer Music Conference*. Miami, USA, Nov. 2004.

[Baa+07] Marije A.J. Baalman, Torben Hohn, Simon Schampijer und Thilo Koch. „Renewed architecture of the *sWONDER* software for Wave Field Synthesis on large scale systems“. In: *5th International Linux Audio Conference*. Berlin (TU Berlin), Germany, März 2007.

[Ber88] Augustinus J. Berkhout. „A holographic approach to acoustic control“. In: *Journal of the Audio Engineering Society* 36.12 (Dez. 1988), S. 977–995.

[Bru04] Werner de Bruijn. „Application of Wave Field Synthesis in Videoconferencing“. PhD-Thesis. Delft: Technische Universiteit Delft, 2004.

[Chr14] Carola Christoffel. „Modifikation der Software einer Wellenfeldsyntheseanlage zur Wiedergabe fokussierter Quellen in Abhängigkeit der Zuhörerposition“. Bachelor-Thesis. Hamburg: Hochschule für Angewandte Wissenschaften, 2014.

[DM98] Leonardo Dagum und Ramesh Menon. „OpenMP: an industry standard API for shared-memory programming“. In: *Computational Science & Engineering, IEEE* 5.1 (1998), S. 46–55.

[Dee89] Steve Deering. *Host Extensions for IP Multicasting*. RFC 1112. The Internet Engineering Task Force (IETF), Aug. 1989. URL: https://tools.ietf.org/pdf/rfc1112.pdf (besucht am 16. 12. 2016).

[Dou15] Timur Doumler. *C++ in the audio industry*. https://youtu.be/boPEO2auJj4. Bellevue (Washington), USA: CppCon 2015, 21. Sep. 2015. URL: https://github.com/cppcon/cppcon2015 (besucht am 16. 12. 2016).

[Foh13] Wolfgang Fohl. „The Wave Field Synthesis Lab at the HAW Hamburg". In: *Sound - Perception - Performance*. 2013. Aufl. Berlin Heidelberg: Springer Science & Business Media, 2013. Kap. 10, S. 243–255. ISBN: 978-3-319-00107-4.

[FS09] Adrian Freed und Andy Schmeder. „Features and Future of Open Sound Control version 1.1 for NIME". In: *New Interfaces for Musical Expression Conference 2009 (NIME09)*. Pittsburgh, USA, 4. Juni 2009, S. 116–120. URL: http://cnmat.berkeley.edu/system/files/attachments/Nime09OSCfinal.pdf (besucht am 16. 12. 2016).

[Gei15] Marvin Geitner. „Erweiterung der Wellenfeldsynthese-Software WONDER um 3D-Quellenpositionen und zusätzliche Lautsprecherkomponenten". Bachelor-Thesis. Hamburg: Hochschule für Angewandte Wissenschaften, 2015.

[GPLv2] *GNU General Public License*. Version 2. Free Software Foundation, Juni 1991. URL: https://www.gnu.org/licenses/gpl-2.0.txt (besucht am 16. 12. 2016).

[LGPLv2.1] *GNU Lesser General Public License*. Version 2.1. Free Software Foundation, Feb. 1999. URL: https://www.gnu.org/licenses/lgpl-2.1.txt (besucht am 16. 12. 2016).

[Han14] Martin Hansen. „Positionierung von Klangquellen einer Wellenfeldsynthese-Anlage mit Hilfe eines Audio-Plugins". Bachelor-Thesis. Hamburg: Hochschule für Angewandte Wissenschaften, 2014.

[IEE16] IEEE. „IEEE Standard for Information Technology – Portable Operating System Interface (POSIX(R)) - Base Specifications, Issue 7 – Technical Corrigendum 2". In: *IEEE Std 1003.1-2008/Cor 2-2016 (Corrigendum to IEEE Std 1003.1-2008) The Open Group Technical Standard Base Specifications, Issue 7* (Aug. 2016), S. 1–310. DOI: 10.1109/IEEESTD.2016.7542098. (Besucht am 16. 12. 2016).

[ISO14] ISO. *Programming Language C++ [Working draft]*. Standard. Geneva, Switzerland: International Organization for Standardization, 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf (besucht am 16. 12. 2016).

[Jan14] Sebastian Jandt. „Visualisierung und Steuerung von Wellenfeldsynthese-Quellen mit Android-Geräten“. Bachelor-Thesis. Hamburg: Hochschule für Angewandte Wissenschaften, 2014.

[Mel+08] Frank Melchior, Christoph Sladeczek, Diemer de Vries und Bernd Fröhlich. „User-Dependent Optimization of Wave Field Synthesis Reproduction for Directive Sound Fields“. In: *124th Convention of the Audio Engineering Society*. Amsterdam, Netherlands, Mai 2008.

[Mon07] Hans-Joachim Mond. „Implementierung einer graphischen Benutzeroberfläche zur Steuerung eines Wellenfeldsynthesesystems“. Master-Thesis. Berlin: Technische Universität, 2007.

[Nog12] Malte Nogalski. „Gestengesteuerte Positionierung von Klangquellen einer Wellenfeldsynthese-Anlage mit Hilfe eines kamerabasierten 3D-Tracking-Systems“. Bachelor-Thesis. Hamburg: Hochschule für Angewandte Wissenschaften, 2012.

[Nog15] Malte Nogalski. „Acoustic Redirected Walking with Auditory Cues by Means of Wave Field Synthesis“. Master-Thesis. Hamburg: Hochschule für Angewandte Wissenschaften, 2015.

[Old13] Robert Oldfield. „The analysis and improvement of focused source reproduction with wave field synthesis“. PhD-Thesis. Salford: University of Salford, 2013.

[Pos80] Jonathan B. Postel. *User Datagram Protocol*. RFC 768. The Internet Engineering Task Force (IETF), 28. Aug. 1980. URL: https://tools.ietf.org/pdf/rfc768.pdf (besucht am 16. 12. 2016).

[Pul97] Ville Pulkki. „Virtual sound source positioning using vector base amplitude panning“. In: *Journal of the Audio Engineering Society* 45.6 (Juni 1997), S. 456–466.

[Sch16] Frank Schultz. „Sound Field Synthesis for Line Source Array Applications in Large-Scale Sound Reinforcement“. PhD-Thesis. Rostock: Universität Rostock, 2016.

[SRA08] Sascha Spors, Rudolf Rabenstein und Jens Ahrens. „The Theory of Wave Field Synthesis Revisited“. In: *124th Convention of the Audio Engineering Society*. Amsterdam, Netherlands, Mai 2008.

[SSR15] Sascha Spors, Frank Schultz und Till Rettberg. „Improved Driving Functions for Rectangular Loudspeaker Arrays Driven by Sound Field Synthesis“. In: *41. Jahrestagung der Deutschen Gesellschaft für Akustik (DAGA)*. Aachen, Germany, März 2015.

[Sta97] E. W. Start. „Direct Sound Enhancement by Wave Field Synthesis“. PhD-Thesis. Delft: Technische Universiteit Delft, 1997.

[Ver97] Edwin Verheijen. „Sound Reproduction by Wave Field Synthesis“. PhD-Thesis. Delft: Technische Universiteit Delft, 1997.

[Vog93] Peter Vogel. „Application of Wave Field Synthesis in room acoustics“. PhD-Thesis. Delft: Technische Universiteit Delft, 1993.

[VF12] Florian Völk und Hugo Fastl. „Wave Field Synthesis with Primary Source Correction: Theory, Simulation Results, and Comparison to Earlier Approaches“. In: *133th Convention of the Audio Engineering Society*. San Francisco, USA, Okt. 2012.

[Wie14] Hagen Wierstorf. „Perceptual Assessment of Sound Field Synthesis“. PhD-Thesis. Berlin: Technische Universität Berlin, 2014.

[WFM03] Matthew Wright, Adrian Freed und Ali Momeni. „OpenSound Control: State of the Art 2003“. In: *New Interfaces for Musical Expression Conference 2003 (NIME03)*. Montreal, Canada, 22. Mai 2003, S. 153–159. URL: http://cnmat.berkeley.edu/system/files/attachments/Open+Sound+Control-state+of+the+art.pdf (besucht am 16. 12. 2016).

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 16. Dezember 2016 Eugen Winter