HAW HAMBURG

Master Thesis

Eugen Winter

Reimplementation of a Real-Time 3D Audio Rendering Software Using an Entity Component System Architecture

Faculty of Engineering and Computer Science
Department Computer Science

Eugen Winter

Reimplementation of a Real-Time 3D Audio Rendering Software Using an Entity Component System Architecture

Master thesis submitted for examination in Master's degree
in the study course *Master of Science Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at Hamburg University of Applied Sciences

Supervisor: Prof. Dr. rer. nat. Wolfgang Fohl
Supervisor: Prof. Dr.-Ing. Andreas Meisel

Submitted on: March 20, 2023

Eugen Winter

Thema der Arbeit

Reimplementierung einer Echtzeit 3D Audio Rendering Software mit Hilfe einer Entity Component System Architektur

Stichworte

Entity Component System, ECS, Sparse Set, Wellenfeldsynthese, WFS, WONDER, CoRGII, JACK, OSC, LibLO, C++

Kurzzusammenfassung

Das Ziel dieser Arbeit ist die Reimplementierung der *WONDER* Software Suite, einer Netzwerk-gesteuerten, Echtzeit 3D Audio Rendering Software für *Wellenfeldsynthese* (*WFS*). Diese umfangreiche Aufgabe wird in zwei separate Teile aufgeteilt. Zuerst werden die Softwarekomponenten der WONDER Suite analysiert. Die Unterprogramme werden diskutiert und alle notwendigen Elemente für die bevorstehende Reimplementierung ausfindig gemacht. Ein Benchmark wird durchgeführt, um die aktuelle Performance zu beurteilen und mögliche Einschränkungen aufzuzeigen.
CoRGII wird im zweiten Abschnitt behandelt und stellt den designierten Ersatz für WONDER dar. Auf Grundlage der vorangegangenen Testergebnisse werden die Designentscheidungen und Kernelemente der neuen Softwarearchitektur besprochen. Das Paradigma der datenorientierten Programmierung mithilfe des *Entity Component System* wird vorgestellt und weitere Implementierungsdetails, etwa das *Sparse Set*, werden aufgezeigt. Zum Schluss werden alle umgesetzten Softwareteile diskutiert und zukünftige Aufgaben skizziert.

Eugen Winter

Title of Thesis

Reimplementation of a Real-Time 3D Audio Rendering Software Using an Entity Component System Architecture

Keywords

Entity Component System, ECS, Sparse Set, Wave Field Synthesis, WFS, WONDER, CoRGII, JACK, OSC, LibLO, C++

Abstract

The goal of this work is the reimplementation of the *WONDER* software suite, a network controlled, real-time 3D audio rendering software for *Wave Field Synthesis* (*WFS*). This comprehensive task is divided into two separate parts. First, the software components of the WONDER suite are analyzed. The subprograms are discussed and all elements necessary for the upcoming reimplementation are identified. A benchmark is performed to assess their current performance and to reveal possible limitations.
The second part covers *CoRGII*, the designated replacement for WONDER. Based on the previous test results, the design decisions and core elements of the new software architecture are discussed. A data-oriented programming paradigm, using the *Entity Component System*, is introduced and further implementation details, such as the *Sparse Set*, are illustrated. Eventually, all implemented software parts are discussed and future tasks outlined.

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

AoIP Audio Over IP.

AoS Array of Structures.

API Application Programming Interface.

AR Augmented Reality.

AVX Advanced Vector Extension.

CoRGII Controlled Renderer, Graphical User Interface and Immersive Interaction.

CPU Central Processing Unit.

DAW Digital Audio Workstation.

DOD Data-Oriented Design.

DOP Data-Oriented Programming.

DSP Digital Signal Processing.

ECS Entity Component System.

FFT Fast Fourier Transform.

FIFO First In First Out.

FIR Finite Impulse Response.

FPGA Field Programmable Gate Array.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HPF High-Pass Filter.

IDE Integrated Development Environment.

ITD Interaural Time Difference.

JACK JACK Audio Connection Kit.

JND Just-Noticeable Difference.

LibLO Lightweight OSC Library.

LPF Low-Pass Filter.

LWFS Local Wave Field Synthesis.

MR Mixed Reality.

OOD Object-Oriented Design.

OOP Object-Oriented Programming.

OS Operating System.

OSC Open Sound Control.

POD Plain Old Data.

POV Point-Of-View.

RAM Random Access Memory.

SIMD Single Instruction Multiple Data.

SoA Structure of Arrays.

SSD Secondary Source Distribution.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

UEFI Unified Extensible Firmware Interface.

VR Virtual Reality.

WFS Wave Field Synthesis.

WONDER Wave field synthesis Of New Dimensions of Electronic music in Realtime.

XML Extensible Markup Language.

1 Introduction

Wave field synthesis Of New Dimensions of Electronic music in Realtime (WONDER) is a software suite designed to render *Wave Field Synthesis (WFS)*. Using a large number of loudspeakers, WFS is able to recreate the sound field of virtual sound sources by synthesizing their wavefronts. This particular audio rendering technique also features the unique property to be independent of the listeners position, which allows for highly interactive use cases, while maintaining a consistent soundscape. The downside of this method is its computationally intensive nature, where every correlation of a virtual sound source with each individual loudspeaker has to be calculated in real-time. This is why the WONDER software suite consists of subprograms that are specifically designed to be distributed and controlled across multiple computers on the same network to balance and handle the heavy audio rendering workload.

1.1 Motivation

As part of my bachelor thesis, I took a deep dive into the code base of the WONDER software suite to make it platform independent and compilable on modern build tools, in order to use it on Linux, MacOS and Windows. I also gained valuable insights of its inner workings and came across a lot of unmaintained, partly finished or aging source code. Some inadequate implementation details, quirks and limitations were also discovered and pinpointed. Since WONDER's intended purpose is to be used in a real-time environment and all of these factors have a negative impact on its performance, I decided that it would be a better choice to pursue an entire reimplementation process, rather than applying partial fixes throughout the code. Therefore, I started the *Controlled Renderer, Graphical User Interface and Immersive Interaction (CoRGII)* project as a future replacement for WONDER.

1.2 Goal

Following my groundwork, the first goal of this thesis is to conclude my work on the WONDER software suite by measuring the baseline performance of its core elements: the WFS audio renderer and the distributed networking functionality. This documents the current performance level and holds as a reference for future comparisons. Since CoRGII is planned as a replacement for WONDER, some features are adopted and allow for backwards compatibility. Before reusing them, their algorithms and internal data flows are analyzed to determine which access types are used (read, write or both) to operate with the underlying data structures. This helps to avoid slow synchronization mechanisms and to use lock-free programming wherever possible.
The focus will then be on CoRGII and its new software architecture and programming approach. Here, the main goal is to achieve a better performance by using a data-oriented and CPU cache friendly design to utilize the full potential of the underlying hardware. This involves a breakup of the previously used object-oriented paradigms and requires different ways to handle program data. All of this is accomplished by implementing the *Entity Component System (ECS)*, where simple data is stored in components, algorithms are encapsulated in systems and entities are just IDs to label objects and to identify from which components they are assembled. A contiguous and gapless memory layout for storing components is provided by a *Sparse Set* implementation.
Originating from video game development, an ECS can be extended to a fully fledged game engine, but this would be well beyond the scope of this work. Therefore, features like 3D graphics, event or input handling are not covered. Instead, my aim is to create a basic, but well documented network controlled audio renderer, by using a data-oriented approach that can be compared with traditional and object-oriented implementations.
The novelty of this project is to combine an ECS with a sparse set, both known in computer graphics for quickly and efficiently rendering a lot of meshes and 3D objects, and repurpose them to improve the computations used in WFS audio rendering.

1.3 Outline

This thesis starts by presenting some related work on the topic of data-oriented and hardware dependent programming in the upcoming chapter. Then the theory of WFS, its possibilities, but also its limitations, are briefly revisited in the third chapter. An overview of the structure of the WONDER software suite is given in the fourth chapter. Benchmarks are conducted on the audio rendering and networking parts to assess and document WONDER's performance in its current, object-oriented form. The test results are presented and discussed.
The fifth chapter marks the start of the second part of this thesis, that is dedicated to CoRGII and its implementation process. It introduces the delay line, a core element for the WFS audio rendering functionality. Chapter six covers the sparse set model, which enables a fast iteration through sparse sets of integer values, by providing a dense storage layout. This approach is utilized for a dense memory layout when storing future objects. The ECS is shown in chapter seven and makes use of the sparse set. It represents the new architecture base and has a data-oriented programming approach. Chapter eight documents CoRGII's components and program structure, including the used libraries, the build system and its classes. The ninth chapter concludes the whole working process during this thesis and summarizes the achieved results. Finally, the tenth chapter gives an overview of future work and additions that can further increase CoRGII's performance and extend its feature set.

2 Related Work

A couple of decades ago, back in the days when computing power was scarce, it was self-evident to ensure a careful resource management when programming [73]. The programs were much simpler than today, but because of the limited computer performance, a developer had to know both, the software and the underlying hardware to implement fast and responsive programs. Over time, object-oriented design was made possible with new programming languages and the performance of a Central Processing Unit (CPU) increased more and more.
In today's world, object-oriented programming is the default and every year a new CPU generation is released. But the numbers are deceptive, since mostly the CPU frequencies are increasing, while other important hardware like the Random Access Memory (RAM), had a much smaller performance increase over the years [13]. This dramatically affects the performance of a computer as soon as the tiny CPU cache is full and the system has to switch to RAM, which is hundred times slower. As a developer, especially for high performance software, it is important to know how the data of a program interacts with the caches of the system [40].
A field in software, which always demands the most hardware performance, is the gaming industry. Gaming consoles, to an extent, reflect the old days, when hardware lasted for several years, before it was upgraded. This gives programmers enough time to study the unchanged hardware for years and optimize their code specifically for the gaming console. A data-oriented programming approach is ubiquitous and necessary to achieve maximum performance [1]. The Entity Component System (ECS) is a well-known software design pattern in the gaming industry. It combines data-oriented design and object creation through composition and is also built into game engines [66].
A comparable field is audio software. Basically a game engine, which is reduced to its audio components. It is demanding and has real-time requirements. The WONDER software suite is such a program and the SoundScapeRenderer as well [33] [23]. However, they are all strictly programmed in an object-oriented way.

The component-based software design, which rather uses components than objects, is nothing new in software development [63] [14]. It can be utilized in different ways and support or replace classical data structures and software patterns when performance is crucial or the software has real-time restrictions [31] [68]. Simulations benefit from the data-oriented and CPU cache friendly approach [22]. Especially if a lot of objects (components) are in use, the ECS can play out its advantages [54].
It is somewhat curious, that ECS apparently seems to originate from the gaming industry or was discovered there. In fact, it has existed for much longer and its data-oriented core goes way back in time. Considering the benefits of data-oriented design and ECS, it looks obvious to choose this programming approach wherever complex computations or simulations exist, which demand high performance. All the more it is surprising that there seems to be no audio software utilizing this approach.

3 Wave Field Synthesis

This chapter introduces the concepts of *Wave Field Synthesis (WFS)*. Since this work focuses on a software implementation, the formulas shown in this chapter are already derived and transformed from the frequency into the time domain. A complete derivation of the formulas is therefore omitted. For a more in-depth look into the derivations, the works of Berkhout [6], Spors et al. [60] and Ahrens [2] are recommended.

3.1 The Principle of Wave Field Synthesis

The physics behind WFS is based on the Huygens' principle, which was discovered and named by physicist Christian Huygens and describes the propagation behavior of waves, in particular light waves [27], but also holds true for other kind of waves, such as sound waves. The principle states, that every point on a wavefront can become the origin of a new *elementary wave*. These elementary waves superimpose and interfere with each other and form the shape of a new leading wavefront (see Figure 3.1). The sum of all elementary waves forms an envelope, which is a curve that resembles the shape of the original wavefront. Different angles of incidence cause a diffraction of the wavefront, however the elementary waves are still able to successfully reassemble and propagate its curvature (see Figure 3.2). An infinite amount of infinitesimal elementary waves is needed to achieve a perfect reconstruction of the original wavefront. Based on this assumptions, a *two-dimensional (2D)* simplification was formulated by Berkhout in 1988 [6], to create and control virtual sound sources by using a discrete number of loudspeakers as a substitute for elementary waves. Each loudspeaker is activated individually and contributes to an artificially generated but physically accurate wavefront. The distance between the virtual sound source and a loudspeaker defines a delay, which is used to play back the sound of the virtual sound source at the correct point in time when its wavefront would pass the loudspeaker. Additionally, the distance and the angle of incidence between the virtual sound source and a loudspeaker define the attenuation of its amplitude.

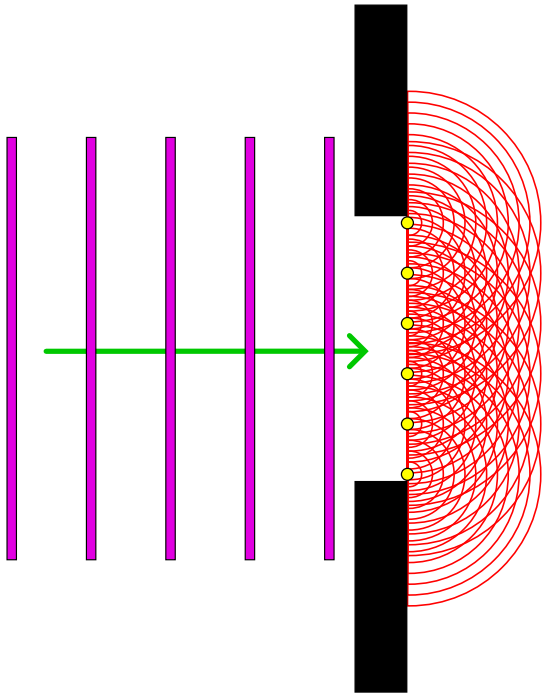

Figure 3.1: Elementary waves reconstructing a plane wavefront behind an aperture.

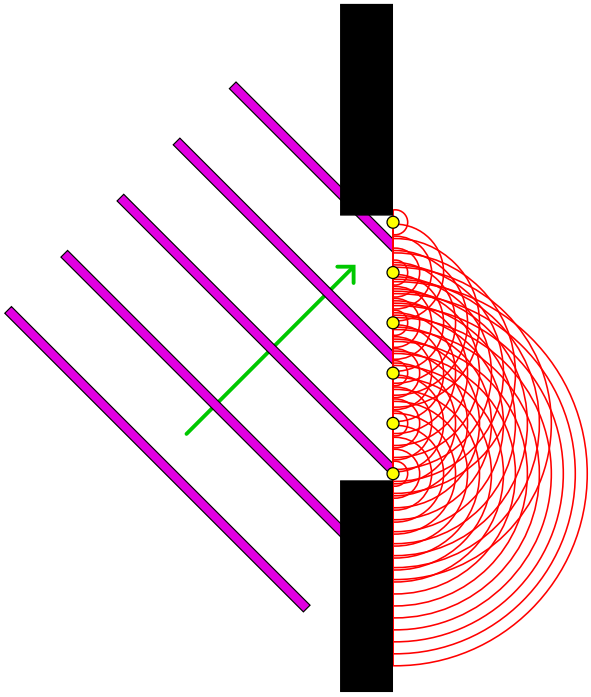

Figure 3.2: Elementary waves reconstructing an angled wavefront behind an aperture.

3.2 Wave Field Synthesis in 2.5D

The two-dimensional solution was an important step towards computationally feasible WFS implementations for real-time applications. However, since dense arrays with lots of loudspeakers are commonly used as *Secondary Source Distribution (SSD)* to generate the sound field, the simple 2D approach was not sufficient enough to precisely describe the radiation characteristics of a loudspeaker. Unlike a line source, which is assumed in the 2D solution, a loudspeaker has the radiation characteristics of a *three-dimensional (3D)* spherical point source [60]. Therefore, better equations were needed and this resulted in revised derivations.

Over time, the term *2.5-dimensional (2.5D)* WFS emerged, when new approaches were introduced to further improve the reproduction quality of WFS. One strategy was to go from 3D down to 2.5D by first using the 3D Neumann-Rayleigh integral and then applying a stationary phase approximation to synthesize the sound field of a virtual point source [62]. This produces a single reference point with the correct amplitude, while all other points only exhibit a correct phase. This is ideal for circular loudspeaker setups, but not suitable for a line array of loudspeakers. Hence, a second stationary phase approximation is used to achieve a correct amplitude on a line, which is parallel to the loudspeakers [61]. A similar approach was to go from 2D up to 2.5D by starting from the 2D Neumann-Rayleigh integral and also work with stationary phase approximations for a correct amplitude at a reference point or a reference line [60].

The approximated 2.5D WFS solutions still exhibit some flaws due to practical and physical limitations. Common loudspeakers need a pre-equalization of their signal and introduce amplitude artifacts when they are used in truncated or rectangular arrays instead of circular setups [59]. The number of loudspeakers and the spacing between them can also cause a coloration of the sound [69]. Additionally, the listening room has to be treated as well, since reflections can occur and distort the sound field [15]. Using *Local Wave Field Synthesis (LWFS)* with focused virtual sound sources can achieve a higher accuracy, but at the cost of a smaller listening area [72]. For a more comprehensive summary on sound field synthesis, the dissertation of Frank Schultz is advised [51].

In a newly and more generalized approach, Firtha et al. developed a unified WFS framework [18]. It uses a reference curve, on which the previously mentioned point and line approximations still holds true as special cases. Moving virtual sound sources, as an important part of interactive WFS setups, are also considered and realized within this framework [19]. This topic is extensively covered in Firtha's dissertation as well [17].

3.3 Virtual Source Types

This section covers the different types of virtual sound sources that can be synthesized with WFS. The focus is on the equations in the time domain and the resulting sound fields. All simulations are conducted with the *Sound Field Synthesis Toolbox*[1] for Python, using the WFS loudspeaker setup in the *I²AudioLab*[2] of the *HAW Hamburg*. It has a rectangular shape (5.0 m × 5.75 m) and consists of 208 audio channels with a loudspeaker spacing of 0.1 m. The referencing scheme used for the simulations relies on a single reference point in the center of the listening area and therefore satisfies an amplitude correct synthesis within the unified WFS framework [18] (see Figure 3.3).

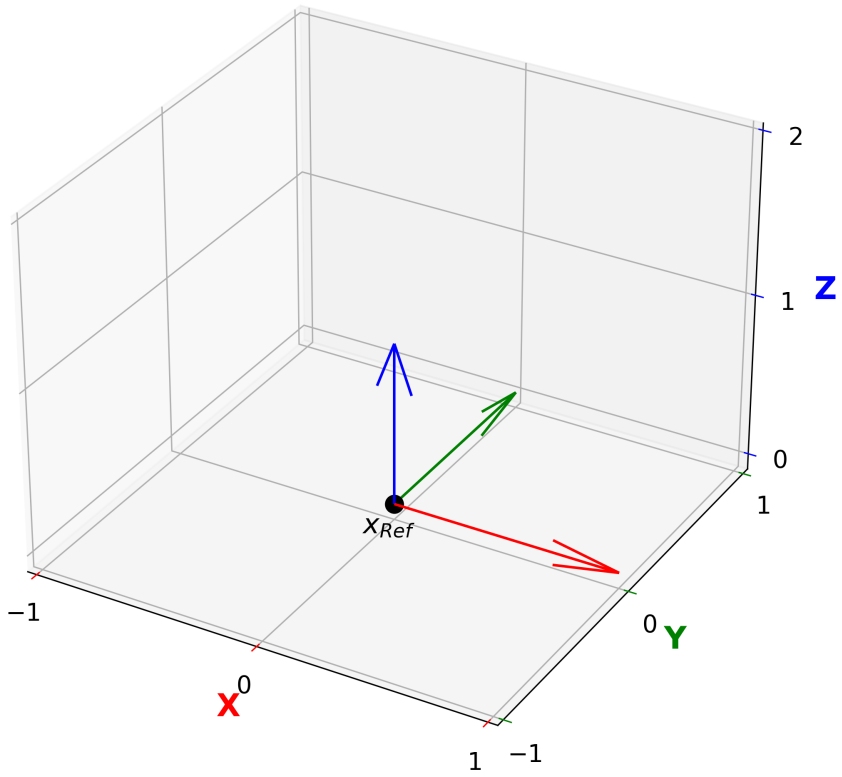

Figure 3.3: Coordinate system used for the simulations.

[1] https://sfs.readthedocs.io (Last accessed on: March 20, 2023)
[2] https://www.i2audiolab.de (Last accessed on: March 20, 2023)

3.3.1 2.5D WFS Driving Function for a Plane Wave Source

$$d_{2.5D_{pw}}(x_0,t) = \underbrace{2 \cdot g_0 \cdot \langle n_k, n_{x_0} \rangle \cdot w(x_0)}_{\text{Amplitude}} \underbrace{h(t) * s(t)}_{\text{Prefilter}} \underbrace{\delta\left(t - \frac{\langle n_k, n_{x_0} \rangle}{c}\right)}_{\text{Delay}} \tag{3.1}$$

g_0 : 2.5D WFS amplitude correction factor

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

x_{ref} : Vector with the reference point inside the listening area (in meters)

x_0 : Vector with the position of the loudspeaker (in meters)

t : Current point in time (in seconds)

$s(t)$: Virtual source audio signal

$h(t)$: 2.5D WFS prefilter applied to the audio signal $s(t)$ by using FFT [59]

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

n_k : Normal vector (propagation direction) of the plane wave

n_{x_0} : Normal vector (direction) of the loudspeaker

$\langle n_k, n_{x_0} \rangle$: Dot product of vector n_k and vector n_{x_0}

c : Speed of sound (343.1 m/s)

$w(x_0)$: Window function to activate a loudspeaker only within a specific angle [60]

$$w(x_0) = \begin{cases} 1 & \text{if } \langle n_k, n_{x_0} \rangle > 0 \\ 0 & \text{else (muted)} \end{cases}$$

A loudspeaker is only active, if the dot product of vector n_k and vector n_{x_0} is greater than zero, which means the angle of incidence $\cos(\alpha)$ is less than 90°.

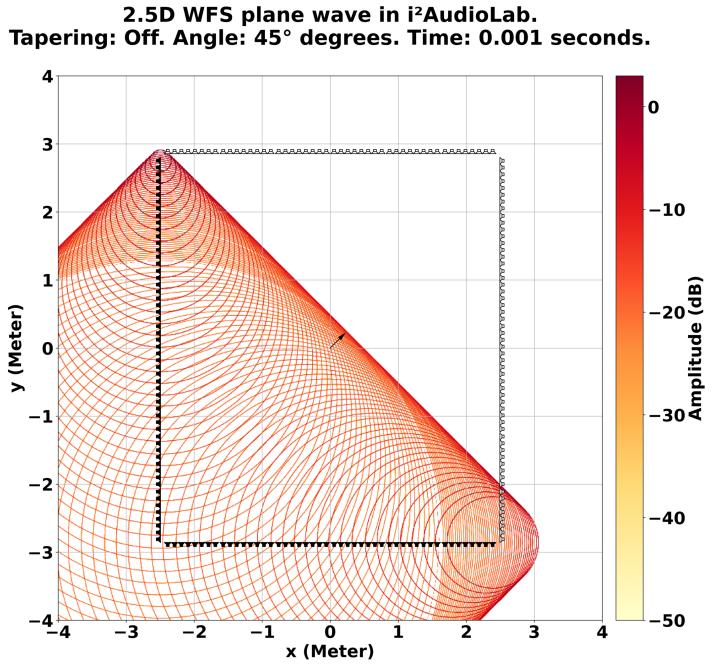

Figure 3.4: WFS simulation of a plane wave with 45° angle (time domain).

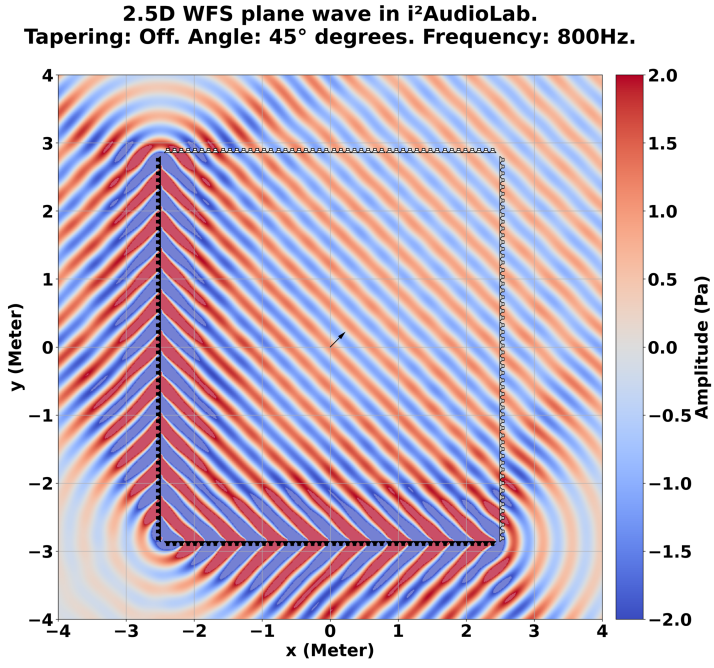

Figure 3.5: WFS simulation of a plane wave with 45° angle (frequency domain).

3.3.2 2.5D WFS Driving Function for a Point Source

$$d_{2.5D_{ps}}(x_0,t) = \underbrace{g_0 \cdot \frac{\langle (x_0 - x_s), n_{x_0} \rangle}{|x_0 - x_s|^{\frac{3}{2}}} \cdot \sqrt{\frac{1}{2\pi}} \cdot w(x_0)}_{\text{Amplitude}} \underbrace{h(t) * s(t)}_{\text{Prefilter}} \underbrace{\delta\left(t - \frac{|x_0 - x_s|}{c}\right)}_{\text{Delay}} \quad (3.2)$$

g_0 : 2.5D WFS amplitude correction factor

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

x_{ref} : Vector with the reference point inside the listening area (in meters)

x_0 : Vector with the position of the loudspeaker (in meters)

x_s : Vector with the position of the point source (in meters)

t : Current point in time (in seconds)

$s(t)$: Virtual source audio signal

$h(t)$: 2.5D WFS prefilter applied to the audio signal $s(t)$ by using FFT [59]

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

n_{x_0} : Normal vector (direction) of the loudspeaker

$\langle (x_0 - x_s), n_{x_0} \rangle$: Dot product of vector $(x_0 - x_s)$ and vector n_{x_0}

c : Speed of sound (343.1 m/s)

$w(x_0)$: Window function to activate a loudspeaker only within a specific angle [56]

$$w(x_0) = \begin{cases} 1 & \text{if } \langle (x_0 - x_s), n_{x_0} \rangle > 0 \\ 0 & \text{else (muted)} \end{cases}$$

A loudspeaker is only active, if the dot product of vector $(x_0 - x_s)$ and vector n_{x_0} is greater than zero, which means the angle of incidence $\cos(\alpha)$ is less than 90°.

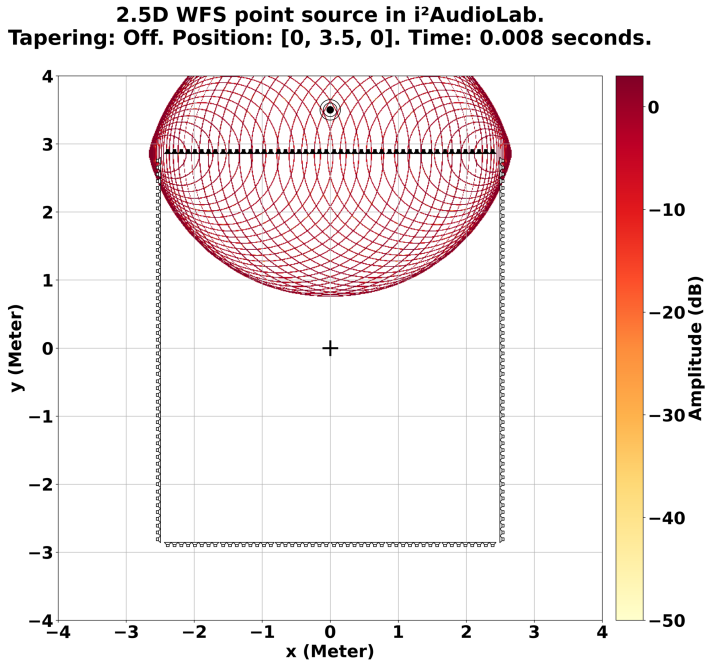

Figure 3.6: WFS simulation of a point source (time domain).

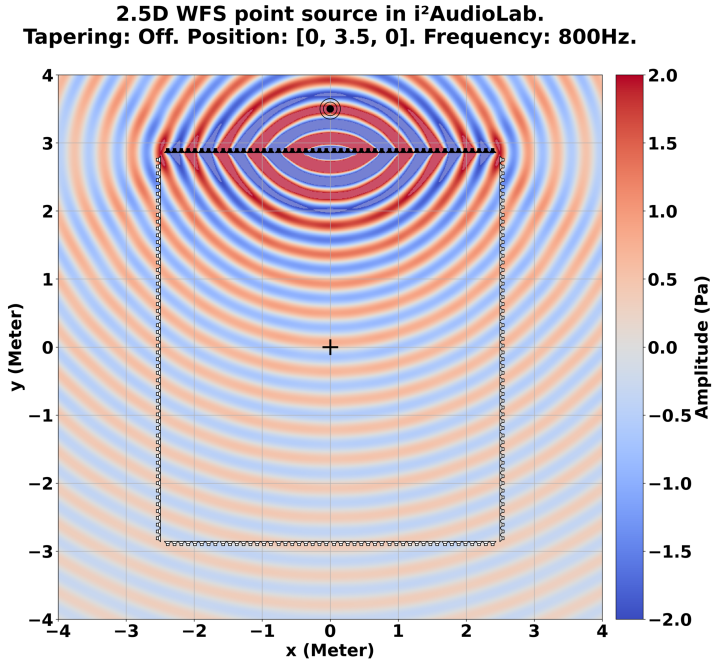

Figure 3.7: WFS simulation of a point source (frequency domain).

3.3.3 2.5D WFS Driving Function for a Focused Point Source

$$d_{2.5D_{fs}}(x_0,t) = \underbrace{g_0 \cdot \frac{\langle (x_s - x_0), n_{x_0} \rangle}{|x_0 - x_s|^{\frac{3}{2}}} \cdot \sqrt{\frac{1}{2\pi}} \cdot w(x_0)}_{\text{Amplitude}} \underbrace{h(t) * s(t)}_{\text{Prefilter}} \underbrace{\delta\left(t + \frac{|x_0 - x_s|}{c}\right)}_{\text{Delay}} \tag{3.3}$$

g_0 : 2.5D WFS amplitude correction factor

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

x_{ref} : Vector with the reference point inside the listening area (in meters)

x_0 : Vector with the position of the loudspeaker (in meters)

x_s : Vector with the position of the focused point source (in meters)

t : Current point in time (in seconds)

$s(t)$: Virtual source audio signal

$h(t)$: 2.5D WFS prefilter applied to the audio signal $s(t)$ by using FFT [59]

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

n_{x_0} : Normal vector (direction) of the loudspeaker

$\langle (x_s - x_0), n_{x_0} \rangle$: Dot product of vector $(x_s - x_0)$ and vector n_{x_0}

c : Speed of sound (343.1 m/s)

n_s : Normal vector (direction) of the focused point source

$w(x_0)$: Window function to activate a loudspeaker only within a specific angle [56]

$$w(x_0) = \begin{cases} 1 & \text{if } \langle (x_s - x_0), n_s \rangle > 0 \\ 0 & \text{else (muted)} \end{cases}$$

A loudspeaker is only active, if the dot product of vector $(x_s - x_0)$ and vector n_s is greater than zero, which means the angle of incidence $\cos(\alpha)$ is less than 90°. Differences between a normal and a focused point source are marked in red.

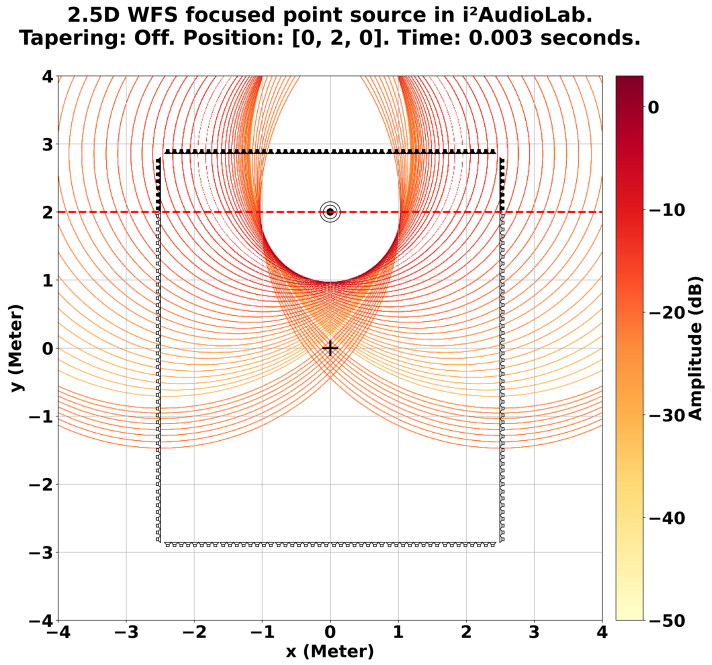

Figure 3.8: WFS simulation of a focused point source (time domain).

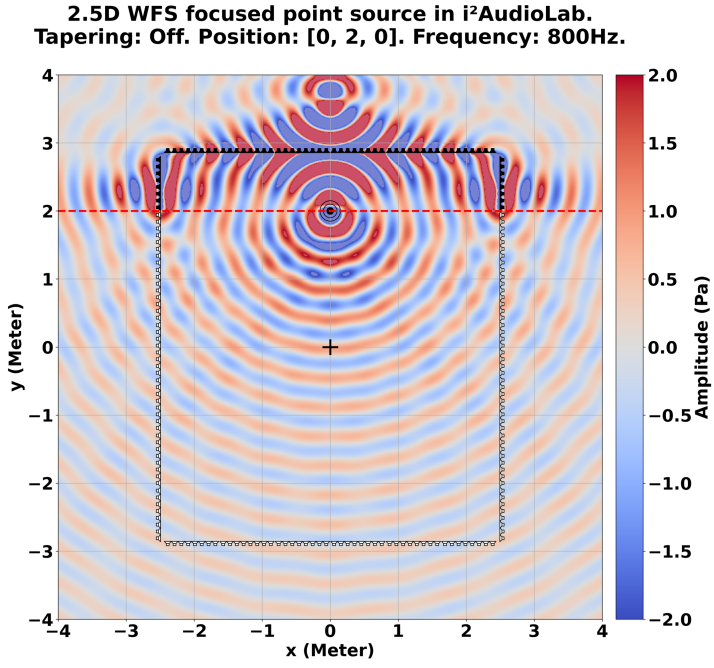

Figure 3.9: WFS simulation of a focused point source (frequency domain).

3.4 Limitations and Implementation Issues

The realization of a WFS audio system is a nontrivial task. It depends on specific hardware requirements and has physical limitations, that can be reduced to some degree, but not removed completely. The next subsections will cover these problems and show how to minimize them.

3.4.1 Spatial Aliasing

The frequency dependent reproduction quality of a WFS system is defined by the size and distribution of its loudspeakers. While, in theory, a small size and high density is desirable for the reconstruction of a sound field, it is practically impossible. A loudspeaker needs a certain size to generate enough pressure and fill the listening area with sound. Therefore, the density is limited by the diameter of the loudspeaker chassis and its housing. This discretization leads to a limitation of the frequency range during the reproduction of a sound field. As a result, audio content above a certain frequency will experience *spatial aliasing* (compare Figure 3.10 and Figure 3.11). To calculate the spatial aliasing frequency of a WFS system, the spacing between adjacent loudspeakers and the speed of sound are utilized [59] (see Equation 3.4). The WFS audio system in the i2AudioLab can achieve a spatial aliasing frequency of 1715.5 Hz (see Equation 3.5), due to its compact loudspeaker spacing of just 0.1 m. Despite the relatively low frequency limit of 1715.5 Hz, the perceived artifacts and coloration may be noticeable but not disruptive for the human auditory system, since the higher frequency content of played back audio signals is usually very wide and diversified [57].

$$f_{alias} = \frac{c}{2 \cdot \Delta x_0} \tag{3.4}$$

$$f_{alias_{i2AudioLab}} = \frac{343.1\,\mathrm{m/s}}{2 \cdot 0.1\,\mathrm{m}} = 1715.5\,\mathrm{Hz} \tag{3.5}$$

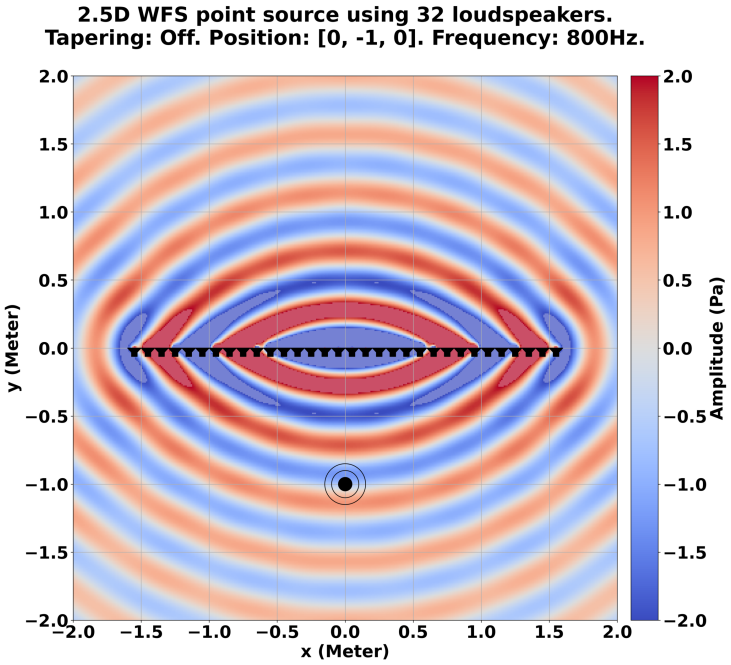

Figure 3.10: WFS simulation of a point source at 800 MHz without spatial aliasing.

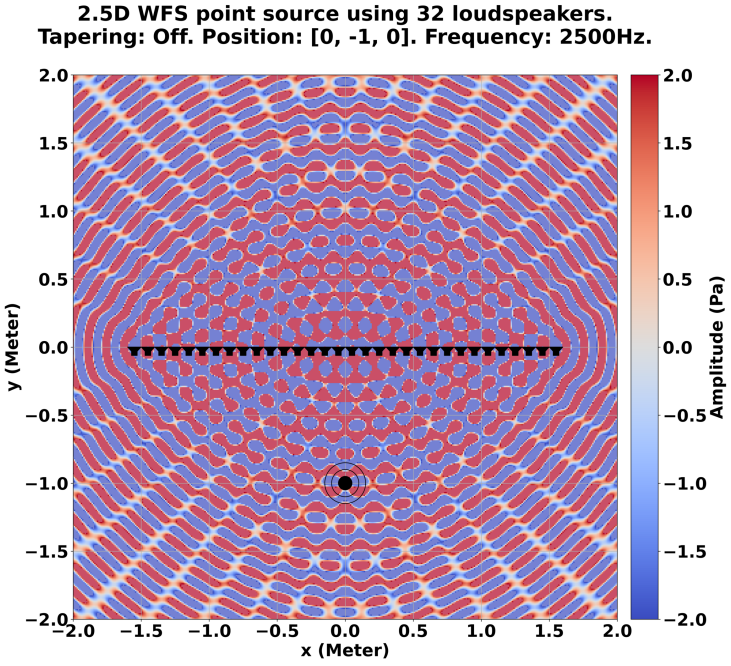

Figure 3.11: WFS simulation of a point source at 2500 MHz with spatial aliasing.

3.4.2 Truncation Effect

The principles behind WFS are based on the assumption that an infinite SSD array is used for the reproduction of the sound field. While circular setups fulfill this requirement by being closed, a linear loudspeaker array has discontinuities at its edges. Here, interference and diffraction effects cause artifacts and coloration [57] (see Figure 3.13). Focused point sources are particularly affected, since the artifacts are already present, even before the sound waves have reached the focusing point. These problems can be mitigated by tapering window functions [60] (see Figure 3.12). The gain of the utmost elements of an array is lowered and hereby their contribution to the artifacts (see Figure 3.14). Active loudspeaker elements are colored black, muted elements are white and attenuated elements have a gray color tone.

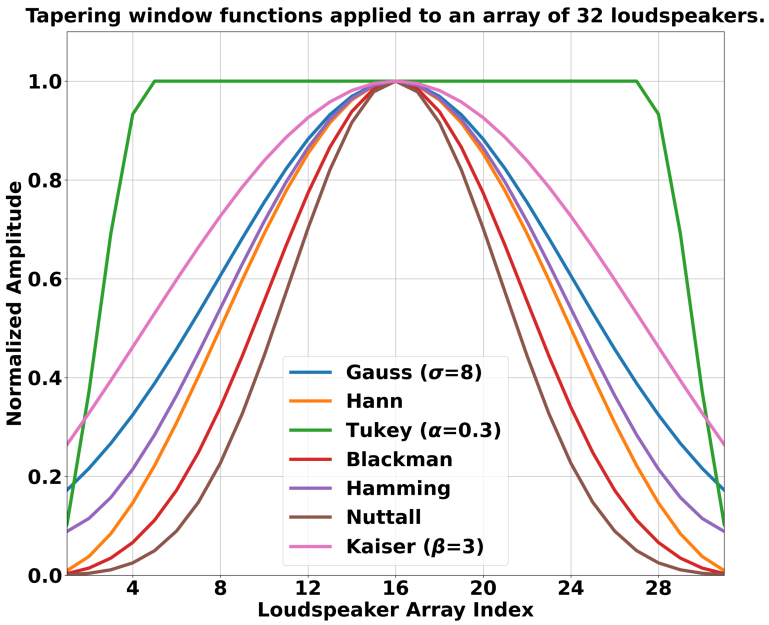

Figure 3.12: Different types of tapering window functions.

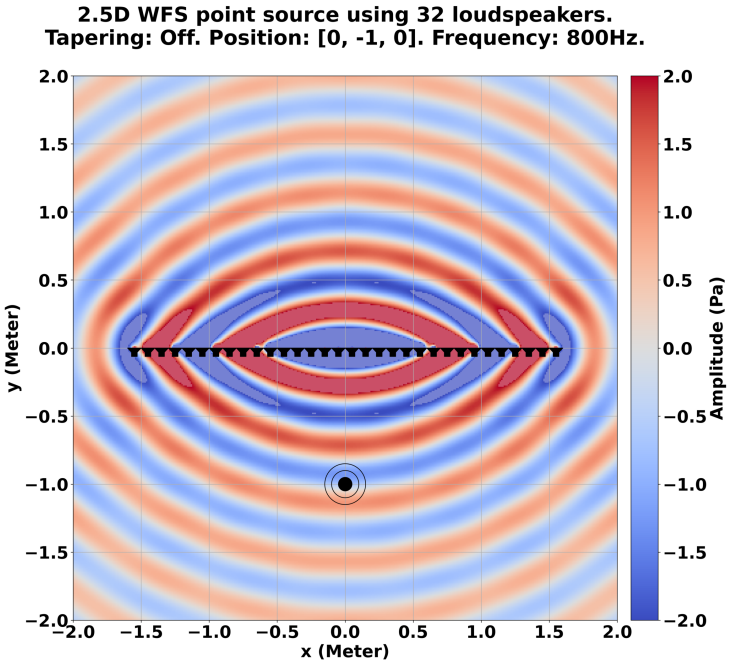

Figure 3.13: WFS simulation of a point source at 800 MHz without tapering.

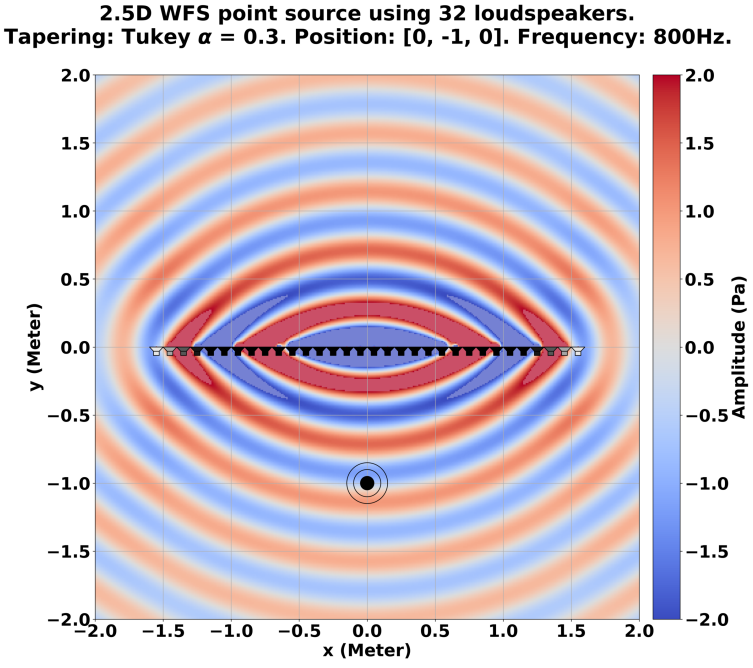

Figure 3.14: WFS simulation of a point source at 800 MHz with tapering.

3.4.3 Pre-Equalization Filter

The various approximations used to derive the 2.5D WFS driving functions, as well as the physical characteristics of a loudspeaker, exhibit different effects on the resulting sound field. While a loudspeaker array shows properties of a *Low-Pass Filter (LPF)*, the truncation leads to an interference, more energy in the higher frequencies and therefore acts like a *High-Pass Filter (HPF)*. Hence, a spectral correction is needed and both, the truncation and the aliasing frequency of the WFS audio system, have to be considered in the filter design. A typical WFS filter consists of a 3 dB per octave high-pass filter, which is only applied between the low end frequency of a loudspeaker and up to the spatial aliasing frequency (see Figure 3.15). The violet area shows the lower band where the spectral energy is filtered by the high-pass filter. A filter can be realized with a single convolution for each virtual sound source, using a *Fast Fourier Transform (FFT)* and the same *Finite Impulse Response (FIR)*. It is also independent of the weighting and delaying operations and can be implemented in a separate step. Omitting the pre-equalization filter is possible, but leads to a dull sound quality, especially in certain scenarios, where the virtual sound source is either placed too close to a loudspeaker (below 0.5 m) or the listener is standing too close to the loudspeaker array [4] [59].

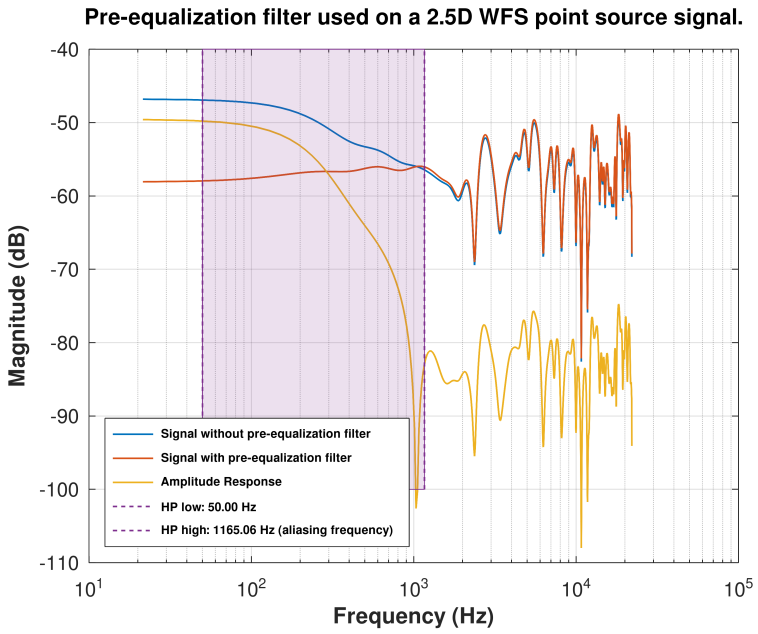

Figure 3.15: Exemplary 2.5D WFS high-pass (HP) pre-equalization filter.

3.4.4 Focused Point Sources

A focused point source can be used to project a virtual sound source in front of the loudspeaker array and into the listening area. It is a time inverted version of a regular point source, which means that the emerging wavefront initially has a concave shape and converges into the anticipated convex shape, after it passes the given focused point. This poses a problem, since the sound field is only perceived correctly after the wavefront has reached its convex shape (see red dashed line in Figure 3.8 and Figure 3.9). Standing in between the loudspeaker array and the focused point source, a listener would perceive the sound coming from the loudspeaker array first, due to the *Interaural Time Difference (ITD)*. The additional direction parameter of a focused point source n_s can be utilized to solve this problem. Usually, it points towards the reference point of the listening area x_{ref}. By integrating a tracking system, the listener's position can be tracked and used as the target direction for the focused point source. This enables the loudspeaker activation function $w(x_0)$ to select the right loudspeakers and synthesize a correct sound field at the position of a listener, even if the listener is moving [21].
The time inversion introduces another problem regarding the delaying of the audio signal. A focused point source is placed in front of the loudspeaker array and the time it takes for the wavefront to arrive at the focused point source is situated in the future. Since its position cannot be known beforehand, a system-wide pre-delay has to be ensured. The diagonal of a rectangular room is a possible estimation for the length of the pre-delay. Depending on the room size, interactive scenarios may experience lag and accuracy deficits, but gain the possibility to seamlessly move virtual sound sources from behind the loudspeaker array into the listening area.

3.4.5 Room Acoustics

The careful treatment of the listening room is an important and crucial task to guarantee an optimal listening experience and accurate sound field synthesis. Otherwise, reflections can distort or colorize the perceived audio signals [15]. There are passive solutions with sound-absorbing and damping materials, such as mats or curtains, that can be applied to walls, the ceiling, the floor and window sides. Active solutions can be implemented on different levels of the reproduction chain. Creating an impulse response of the listening room will help to understand its acoustical properties and to develop a suitable filter.

The filter can then be implemented at the hardware level in the built-in *Digital Signal Processing (DSP)* unit of an active loudspeaker, an intermediary equalizer or a standalone amplifier with a DSP unit. Software-wise, the filter can be implemented using a plugin inside a *Digital Audio Workstation (DAW)*, a separate filter stage inside the WFS software or adjustments performed at the driving function level [58] [44].

3.4.6 Hardware and Software

A WFS system consists of a combination of multiple hardware and software elements. Starting with the loudspeakers, the usage of passive models are unsuitable because of form, size, amplification and connectivity. Active loudspeakers are preferable, but mostly designed for a stereo setup. The necessity for a large amount of audio channels requires special, but also cost-intensive solutions. Unique modules specifically designed for WFS are distributed by *Four Audio*[3] and *Holoplot*[4]. These have dense loudspeaker layouts, active amplification and equalization and most important, the ability to handle lots of audio channels by using an *Audio Over IP (AoIP)* solution, such as the proprietary *Dante* protocol by *Audinate*[5].
The connection between the loudspeaker modules and a computer running the WFS software further relies on special sound cards, which support the AoIP protocol. These provide up to 128 input and output channels via a single ethernet cable, but, yet again, at a higher cost than regular sound hardware. Moreover, the computer needs enough computation power to handle the large amount of audio channels, low latency audio drivers or a real-time capable operating system.
Additional useful hardware, although optional, would be a tracking system to track a listener inside the listening area. This is important for the usage of focused point sources and interactive installations.
On the software side, audio content is mostly produced with a DAW application. In general, the content is mastered for individual channels. However, in the context of WFS, each audio channel represents a virtual sound source, an object, which includes further properties like a position. Contrary to the traditional mindset, audio content has to be handled object-based and not by individual channels. Controlling and modifying an audio object, such as its position, requires an additional layer of control. A well-established solution for this task is the message-based *Open Sound Control (OSC)*.

[3] https://www.four-audio.com (Last accessed on: March 20, 2023)
[4] https://www.holoplot.com (Last accessed on: March 20, 2023)
[5] https://www.audinate.com (Last accessed on: March 20, 2023)

Some DAW applications already have an integrated OSC support, others can be extended with plugins. The WFS software itself generally runs on a separate computer system, solely dedicated to render the sound field without wasting resources on other things. An exemplary setup is illustrated in Figure 3.16 and shows how a DAW computer generates an audio signal and an OSC message to control the position of a virtual sound source. The Dante protocol is used to transmit the virtual source signal to the WFS computer, which weights and delays the signal for every output channel, and to pass the processed signal to all loudspeakers. OSC messages are sent via a regular network connection using the *Transmission Control Protocol (TCP)* or the *User Datagram Protocol (UDP)*.

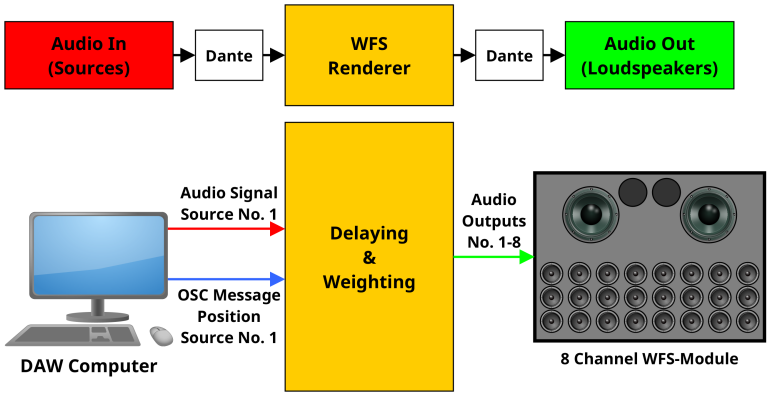

Figure 3.16: Exemplary hardware setup of a WFS audio system.

Looking at the implementation of the WFS software, the core requirement is first and foremost a high performance. For a real-time capability, all available hardware resources, including multi-threading and vector extensions, should be utilized to parallelize the costly computations of the software on the *CPU*. A standalone solution running on a separate computer is highly recommended and best accompanied with an interface to the popular OSC protocol. This allows the computationally heavy task of the WFS calculations to be distributed among multiple computers on the same network as well. Further solutions to improve the performance may use different hardware architectures, for example a *Graphics Processing Unit (GPU)* or a *Field Programmable Gate Array (FPGA)*, which also draws much less power than a regular computer [64]. Additionally, support for a non-proprietary AoIP integration like *AES67*[6] would be a benefit, but can be substituted by proprietary solutions if necessary.

[6] http://www.aes67.org (Last accessed on: March 20, 2023)

3.5 Conclusion

This chapter featured a brief introduction into WFS. It has the ability to synthesize a physically accurate sound field and therefore is a fitting choice for simulations and the creation of live or prerecorded soundscapes. Unlike other audio techniques, WFS is not restricted to a single *sweet spot* for the best listening experience, but instead covers a whole listening area. Combined with the possibility to render focused point sources inside the listening area, this allows for rich and interactive sound installations with a freely moving listener. It can even be further extended by using *Augmented Reality (AR)*, *Mixed Reality (MR)* or *Virtual Reality (VR)* headsets to create immersive audio-visual experiences [38]. The downside is the cost factor, since special and expensive hardware is needed for a viable realization. This is also the main reason why WFS technology is currently more likely to be found in research and rarely in commercial installations.

4 WONDER

WONDER is a software suite (short for ***W**ave field synthesis **O**f **N**ew **D**imensions of **E**lectronic music in **R**ealtime)*, developed at the Technical University of Berlin in 2008 [5], is a set of programs designed to render WFS (see chapter 3). Since the process uses tens of thousands of calculations per second and needs dozens or even hundreds of closely aligned loudspeakers to achieve a reasonable listening result, it represents a demanding and non-trivial task. To handle both problems, the components of WONDER are able to run on different computers inside a network. Using the OSC protocol, hundreds of audio channels can be rendered in real-time by distributing them across multiple machines, running the audio rendering component of WONDER.
In 2011, the Hamburg University of Applied Sciences (HAW Hamburg) installed a WFS audio system in the I²AudioLab [20]. Consisting of two rendering machines with AoIP interfaces and 26 loudspeaker modules, each using 8 channels, the system is capable of rendering 64 virtual sound sources and play them back on 208 loudspeaker channels with a loudspeaker spacing of 0.1 m. The listening area has a size of 5.0 m by 5.75 m.
Over the past years, multiple modifications were applied to tWONDER, WONDER's **t**ime delay based WFS renderer, by students at the HAW Hamburg [21]. A recent work, focusing on making WONDER's code base more platform independent, revealed issues in some implementation details that are not real-time safe. Additionally, due to its creation dating back to 2008, the hardly maintained software is aging and suffers from software erosion caused by changes in computer hardware, operating systems, libraries and the C++ programming language. It also lacks some important features like multi-threading and therefore is not capable to use a modern CPU to its full potential. Since the rendering of WFS is computationally intensive, the software structure has to be designed for maximum performance and proper utilization of the underlying hardware.

4.1 Benchmark Methods

In this section all parts involved in the test environment are described. It includes the used hardware and software, the structure of the code, as well as the design of the tests and the test conditions.

4.1.1 Hardware

The hardware used to perform the runtime measurements differs from the one actually installed in the I²AudioLab (see Table 4.1). On the one hand the tests should not be constrained by a specific hardware combination. On the other hand a more modern hardware choice shows a better view of the currently possible performance.
To achieve consistent test results, some settings for the CPU have to be adjusted. Since modern CPUs provide a different base and turbo clock speed, depending on the current load, voltage or temperature, there is no guarantee that it stays the same during all test runs. Therefore, the turbo is deactivated via the *Unified Extensible Firmware Interface (UEFI)* of the motherboard to ensure that the CPU, including all of its cores, is always running at its base clock, which is 3.5 GHz.
Another difference in the test setup is the audio hardware. Instead of a special Audio Over IP sound card, the built in stereo sound card is used. Rendering more than two input and output channels is still possible, due to the usage of the JACK library (see subsection 4.1.2).

	Intel Core i7 920	**AMD Ryzen 5 5600**	**Latency**
Release Date	2008	2022	
Cores/Threads	4/8	6/12	
Manufacturing	45 nm	7 nm	
Base Clock	2.66 GHz	3.5 GHz	
Turbo Clock	2.93 GHz	4.4 GHz	
L1 Cache	256 kB (4x 64 kB)	384 kB (6x 64 kB)	ca. 1 ns
L2 Cache	1 MB (4x 256 kB)	3 MB (6x 512 kB)	ca. 4 ns
L3 Cache	8 MB	32 MB	ca. 16 ns
Wattage (TDP)	130 W	65 W	
RAM	8 GB DDR3 800 MHz	16 GB DDR4 3600 MHz	ca. 100 ns

Table 4.1: Hardware comparison.

4.1.2 Operating System & Libraries

All tests are performed on an *Arch Linux* installation using kernel version 5.17 and the CPU governor of the *Operating System (OS)* set to *performance*. This leads to a fixed clock speed of 3.5 GHz for all CPU cores and a constant performance delivery without any fluctuations in the test results caused by power saving features of the hardware.

The compilation of the software is realized with the *GNU Compiler Collection (GCC)* in version 11.2.0 and the following compiler flags:

- **-O3:** Highest optimization level within C++ language standards.
- **-flto:** Run link-time optimizer.
- **-march=native:** Generate instructions for the currently used type of CPU.
- **-mtune=native:** Tune the generated code for the currently used type of CPU.

The *Lightweight OSC Library (LibLO)*[1] in version 0.31 is used to implement OSC support, so individual applications of the WONDER suite on the same network can communicate with each other.

Access to the audio hardware is handled by the *JACK Audio Connection Kit (JACK)*[2] library in version 1.9.21, which features virtual audio ports. This allows to address more audio ports than physically available on the sound card and to simulate the AoIP environment in the I²AudioLab. The computational effort stays the same, since virtual ports, just like real ports, have their own input and output buffers, that are also processed during runtime.

4.1.3 Software

This subsection describes the applications involved in the test runs and their purpose. A closer and more detailed look is taken at tWONDER, WONDER's WFS audio renderer, where all of the measurements take place.

[1] https://liblo.sourceforge.net (Last accessed on: March 20, 2023)
[2] https://jackaudio.org (Last accessed on: March 20, 2023)

Processing

Processing Java basiert und ist gleichzeitig auch eine Integrated Development Environment (IDE) und benötigt deshalb keine Kompilation der Programme *Processing*[3] in version 4.0b7 is used to add and move virtual sound sources (see Figure 4.1). It is a Java based tool to code animations and visualizations. A built-in extension system allows the installation of extra features, such as OSC support. This enables the communication with WONDER. The refresh rate, which also corresponds to the OSC sending rate, can be set to 60 Hz, 120 Hz or 240 Hz, in order to generate a higher workload. In addition, source movement, as well as WONDER's *duration* parameter, can be turned on or off. The duration parameter enables source movement over time and defaults to a value of one second.

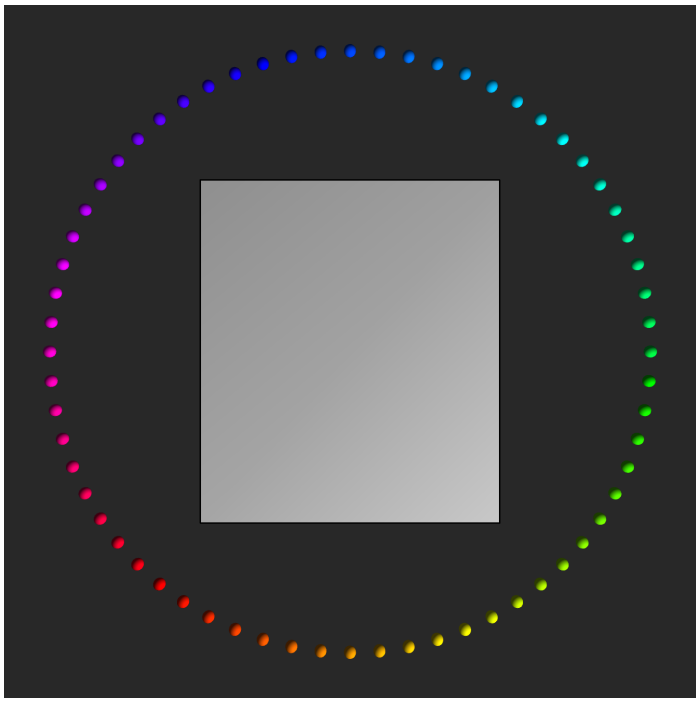

Figure 4.1: Processing project to move virtual sound sources in a circle.

[3] https://processing.org/ (Last accessed on: March 20, 2023)

xWONDER

The WONDER suite has a separate *Graphical User Interface (GUI)* called *xWONDER*. It is used to add, move and control all virtual sound sources in the current scene. During the test runs, it serves as a fallback to check for the right amount of active sources and their movement, which should match the circle pattern coded in Processing.

cWONDER

The central control unit and core of WONDER's distributed rendering approach is called *cWONDER*. It represents the state of the current rendering scene and all of its virtual sound sources. Similar to a *publish-subscribe* pattern, cWONDER acts as a server and offers different OSC *streams*, where incoming messages are propagated to all connected clients. Based on the needed information, renderers like tWONDER connect to the *render* stream, while GUI applications like xWONDER or Processing connect to the *visual* stream, that offers more source properties like colors or names.

JACK Server

The JACK server (*jackd*) is a professional audio server and provides real-time, low-latency connections between client audio applications that implement the JACK Application Programming Interface (API). It also handles the access to the sound card and the routing between the input and output ports of various connected clients. This enables tWONDER to use the audio output of other JACK enabled applications as input and apply WFS to it.

tWONDER

The WFS audio renderer of the WONDER suite consists of two parts: a *DSP* part, implemented with *JACK* and responsible for the WFS rendering, and a communication part, implemented with the LibLO and enabling distributed rendering on multiple machines in the same network to balance the heavy workload. Each part uses its own thread and both access and interact via the real-time command engine (see Figure 4.2).

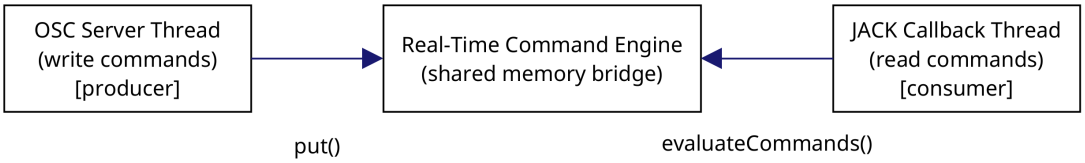

Figure 4.2: Thread access and shared memory approach in tWONDER.

The OSC thread receives new messages from the network and converts the containing simple data types, like integers or floats, into *Command* objects. These can trigger source movement and other events. To pass them safely into the JACK callback thread, where the commands are executed, they are stored in a *CommandQueue* object, which is based on the lock-free ring buffer implementation found in the JACK API (see Figure 4.3). It is worth mentioning that the CommandQueue object defaults to a size of only 256 elements.
Multiple CommandQueue objects and C++ *std::list* containers are embedded inside the real-time command engine (*RTCommandEngine*). They help to keep the chronological order of the incoming commands and execute commands with a special *duration* or *timestamp* parameter at a later point in time. The RTCommandEngine acts as a shared memory bridge between the OSC and JACK thread (see Figure 4.3).
The JACK thread first evaluates due commands inside the real-time command engine. Then it copies new audio input samples into the individual delay lines of virtual sound sources. Afterwards, a nested *for*-loop iterates over all source and speaker constellations and calculates their WFS parameters twice. Once for the current and once for the target source position. Both results, consisting of a delay and amplitude parameter, are used for interpolated reads of samples from the delay lines, that are then written into the audio output buffers. Finally, all current and target source positions are interpolated for smoother changes, in case a source is moving.

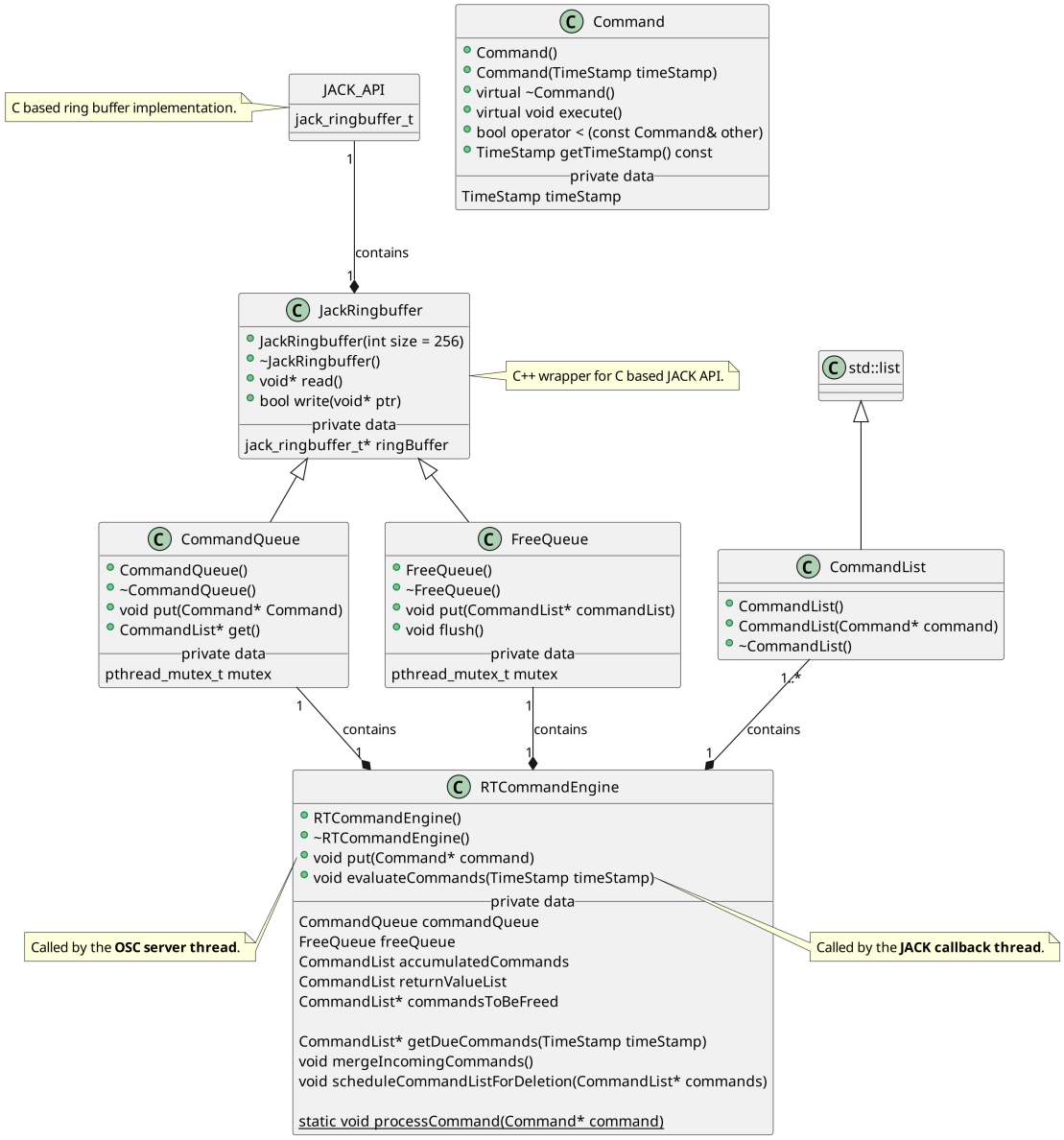

Figure 4.3: Class diagram for the real-time command engine (RTCommandEngine).

4.1.4 Test Design

The tests are performed on the DSP and networking components of tWONDER. Using a different test environment required some preceding adjustments. Without AoIP, the virtual port capability of JACK is necessary. However, a previously built-in check for the existence of enough physical ports had to be removed. This allows an arbitrary number of ports, where virtual ports complement the lack of physical ports.

The time measurements are implemented in a simple and unobtrusive way (see Listing 4.1). Each test is repeated three consecutive times to minimize errors and to form an average result. The iteration numbers vary for individual code sections and are measured in nanoseconds. Minimum, maximum, average and median results are calculated.

```
1 using namespace std::chrono;
2
3 auto startTimepoint = high_resolution_clock::now(); // Start measurement.
4 // ... Code to be measured.
5 auto endTimepoint = high_resolution_clock::now();   // Stop measurement.
6
7 duration<double, std::nano> duration = endTimepoint - startTimepoint;
8 double ns = duration.count();                       // Elapsed nanoseconds.
```

Listing 4.1: Time measurement code snippet.

OSC Test

Running JACK requires certain real-time conditions. Inside its callback process it is therefore not recommended to use blocking constructs like *mutexes*, *semaphores* or *sleeps*, as well as memory allocations with the keyword *new*. However, some of this constructs are used in the real-time command engine, which is evaluated inside the JACK callback process. This shared memory linkage between the OSC (producer) and JACK (consumer) thread, in addition to its small size of only 256 elements, has the potential to cause problems. Hence, this test focuses on the *put* function of the RTCommandEngine (see Listing 4.2). First, it writes into the CommandQueue (see Listing 4.3), then it flushes the already processed commands (see Listing 4.4). The *put* call of the RTCommandEngine object is measured inside the function called *oscSrcPositionHandler*, where the OSC server thread receives OSC messages with new source positions from the network.

Counters in both functions measure the received OSC messages and stored commands. This is important, since the buffer only holds 256 elements. Sending OSC messages at 60 Hz, would fill it up in 1 s with four moving sources. The *oscSrcPositionHandler* counter would further increase, while the counter in the *put* function of the real-time command engine would stop, indicating the loss of new source positions. Simultaneously, the JACK callback consumes commands from the buffer at 375 Hz (48000/128 = 375) and allows for six moving sources. The test cases therefore use one, six and eight moving sources.

```
1  void RTCommandEngine::put(Command* cmd) {
2    commandQueue.put(cmd); // Write process using new, mutex and sleep.
3    freeQueue.flush();     // Read process using mutex.
4  }
```

Listing 4.2: RTCommandEngine *put* function.

```
1  void CommandQueue::put(Command* command) {
2    CommandList* commandList = new CommandList(command);
3    pthread_mutex_lock(&mutex);
4
5    // Keep waiting for 100 microseconds, if writing to the CommandQueue
6    // and its underlaying JACK ring buffer implementation is not possible.
7    while(!write(reinterpret_cast<void*>(commandList))) {
8      std::this_thread::sleep_for(std::chrono::microseconds(100));
9    }
10
11   pthread_mutex_unlock(&mutex);
12 }
```

Listing 4.3: CommandQueue *put* function with *new*, *mutex* and *sleep* usage.

```
1  void FreeQueue::flush() {
2    CommandList* commandListToDelete;
3    pthread_mutex_lock(&mutex);
4
5    while(
6     (commandListToDelete = reinterpret_cast<CommandList*>(read())) != NULL) {
7      delete commandListToDelete;
8    }
9
10   pthread_mutex_unlock(&mutex);
11 }
```

Listing 4.4: CommandQueue *flush* function with *mutex* usage.

JACK Test

There are multiple steps handled inside the JACK callback process, so the measurements of the execution times have to be performed in different code sections. The evaluation of the real-time command engine is measured first. Afterwards, both WFS calculations and the delay line interpolations are measured separately. Finally, the whole JACK callback process is measured, once with and once without the evaluation of the real-time command engine. Additionally, the DSP utilization of the JACK server is observed in the GUI application *QjackCtl*. It indicates how much of the JACK callback time (2.667 ms) was spent on computations. A 50 % utilization means a computation time of 1.333 ms. Workloads that exceed this time frame cause buffer under- or overruns (*xruns*).

JACK callback steps:

- Evaluate commands, whose executions are due in the next 10 audio samples.
- Put audio input in the corresponding delay lines of virtual sound sources.
- For every speaker (output buffer):
    - Mute (zero out) the audio output buffer first (safety measure).
    - For every source (input buffer):
        - 2x WFS: Calculate delay and amplitude for speaker source constellation.
        - If source is deactivated: Set both calculated WFS amplitudes to zero.
        - Interpolated read from the delay lines, using both WFS results.
        - Write the read samples into the corresponding audio output buffer.
- Interpolate all source positions.

Delay Line Test

The delay line test uses the *Hotspot*[4] profiler to measure the CPU cache references, the CPU cycles and the CPU intructions. The Processing project runs with 30 Hz, 60 Hz, 120 Hz and 240 Hz. tWONDER is using 16 loudspeakers and 64 sources. Additionally to the default implementation in Listing 4.5, the optimized version in section 5.4 is also tested at 60 Hz and 120 Hz. The code of the delay line *read* function is not listed due to its length of 267 lines of code.

[4] https://github.com/KDAB/hotspot (Last accessed on: March 20, 2023)

```
void DelayLine::put(float* samples, unsigned int nsamples) {
  // XXX: maybe use memcpy... or std::copy
  for (unsigned int i = 0; i < nsamples; ++i) {
    line[writePos] = samples[i];
    ++writePos;

    // wrap around if end of buffer is reached
    if (writePos >= lineLength) {
      writePos = 0;
    }
  }
}
```

Listing 4.5: DelayLine *put* function.

General Test Parameters

The tests are performed on a single computer that runs all necessary applications, instead of using a distributed system. Only a single instance of tWONDER is started. This serves as a baseline of tWONDER's single-core performance and allows comparisons between future implementations with multi-threading features and further optimizations.

The UEFI and operating system of the computer is set to a fixed CPU speed of 3.5 GHz.

Processing is used to simulate three test conditions:

1. 64 stationary sources (no OSC or networking).
2. 1, 6 and 8 moving sources (edge cases for the buffer size of the RTCommandEngine).
3. 1, 6 and 8 moving sources with a one second duration parameter for the movement.

The first test condition creates no network traffic, which means no work for the OSC thread and the real-time command engine. Test conditions two and three create work in both of this parts and also in the JACK callback during the interpolation of source positions. Additionally, the OSC sending speed in Processing can be switched between 60 Hz, 120 Hz and 240 Hz. This further increases the workload on the OSC part and the real-time command engine.

Program Parameters:

- **cWONDER:** ./cwonder -c ./cwonder_config.xml
- **xWONDER:** ./xwonder -i 127.0.0.1
- **tWONDER:** ./twonder -c ./twonder_config.xml -s ./twonder_speakers.xml -r 48000
- **JACK server:** 48 000 Hz and a buffer size of 128 samples (latency of 2.667 ms).
- **Inputs:** 64 audio inputs, representing 64 virtual sound sources.
- **Outputs:** 16 audio outputs, representing two WFS modules in the I²AudioLab.

Test Iterations:

- **OSC:** 10 000 × & 100 000 × RTCommandEngine *put* call in *oscSrcPositionHandler*.
- **JACK:** 100 000 × RTCommandEngine *evaluateCommands* call.
- **JACK:** 10 000 000 × both WFS calculations.
- **JACK:** 10 000 000 × interpolated delay line read and write into output buffer.
- **JACK:** 100 000 × JACK callback without *evaluateCommands* call (DSP only).
- **JACK:** 100 000 × JACK callback with *evaluateCommands* call.

4.2 Benchmark Results

The test results of the time measurements are divided into separate subsections. First, the OSC handler and its access to the real-time command engine is covered. Then the JACK callback process and all of its inner steps are examined in subdivided sections.

4.2.1 OSC: Real-Time Command Engine Access

The access of the OSC thread to the real-time command engine resulted in minimum execution times of 0.419 µs, which may also indicate the lowest possible resolution for measurements. Using the duration parameter had no influence on the results. The highest maximum execution times are listed separately, since they only occurred a single time on each run. Capturing the ten highest time values at the end of every test helped to identify this special cases that ranged between 554.2 ms and 1014.4 ms. The second highest and more realistic values for the maximum execution times varied between 2393.8 µs and 2540.6 µs. It is notable that these values nearly matched the JACK callback latency of 2666.7 µs. The average execution times resulted in 57.6 µs with one and 11.7 µs with eight moving sources. Interestingly, these values decreased with a higher OSC workload and more moving sources. One assumption would be a conflict about the acquirement of CPU resources that went in favor of the OSC thread after its workload increased with more incoming OSC messages.

Moving sources at 60Hz						
No.	**Minimum**	**Highest**	**Maximum**	**Average**	**Median**	**JACK DSP**
1	0.419 µs	554.168 ms	2396.596 µs	57.635 µs	1.746 µs	8.9 %
6	0.419 µs	985.170 ms	2540.614 µs	11.483 µs	1.397 µs	8.0 %
8	0.419 µs	1011.239 ms	2448.793 µs	11.687 µs	1.396 µs	8.0 %
Moving sources at 60Hz with 1s duration						
No.	**Minimum**	**Highest**	**Maximum**	**Average**	**Median**	**JACK DSP**
1	0.419 µs	564.954 ms	2393.808 µs	58.514 µs	1.769 µs	8.5 %
6	0.419 µs	980.437 ms	2454.662 µs	11.425 µs	1.397 µs	8.3 %
8	0.419 µs	1014.347 ms	2412.067 µs	11.738 µs	1.397 µs	8.5 %

Table 4.2: Access by OSC thread in *oscSrcPositionHandler* function.

In contrast to the average, the maximum execution times took tremendously longer and indicated the presence of 100 µs long *sleep* calls. The C++ reference states that the used *sleep_for* function blocks for at least the specified time, but may block much longer [46]. Resource conflicts or the scheduler of the operating system can cause longer delays and explain the huge difference between both results. The median times resulted in between 1.4 µs and 1.8 µs, showing that the majority of the results were much shorter than the maximum or average execution times.

The JACK DSP utilization peaked at 8.9 % and is noticeably higher than running tWONDER without the measuring code in place, where it fluctuated between 1 % and 2 %.

4.2.2 OSC: Message Throughput

The OSC and JACK threads are connected via the real-time command engine. It contains mutexes, a *sleep* call that is not real-time safe and has a relatively small default buffer size of just 256 elements. Furthermore, the frequency of the JACK callback thread (375 Hz) is different from the OSC thread (mostly 60 Hz). The test was executed for a duration of 60 seconds, used different refresh rates, numbers of sources and examined the handling of OSC messages. Here, the OSC thread worked as expected and reliably received the estimated number of sent OSC messages. A slightly larger amount of arrived messages is normal and due to the test not precisely ending after 60 seconds. However, the *put* function of the RTCommandEngine, which was called by the OSC thread inside the *oscSrcPositionHandler* function, only partially succeeded. The number of *put* calls always maxed out and stagnated at two moving sources. Hence, every additional source has caused data loss, that reached up to 97 % with 64 moving sources. Doubling the OSC sending rate doubled the *put* calls as well, but a successful storage of all incoming OSC messages was still only possible for up to two moving sources.

The JACK callback thread, reading and emptying the buffer of the real-time command engine at a much higher refresh rate of 375 Hz, showed no effect and did not help to cope with the amount of incoming OSC messages. Regarding the imbalance of CPU resources between the JACK and OSC thread in the previous test, this results showed a similar behavior, where an increased OSC workload led to an improved performance and a higher number of *put* calls. However, this also shows that there has to be something wrong with the real-time command engine, the thread synchronization mechanisms or both.

Moving sources at 60Hz for 60 seconds				
No.	**Estimated**	**Received via OSC**	**Stored in Buffer**	**Ratio**
1	3600	3795	3798	100.0 %
2	7200	7594	7600	100.0 %
3	10800	11397	7619	66.9 %
4	14400	15186	7622	50.2 %
6	21600	22693	7601	33.5 %
8	28800	30154	7580	25.1 %
64	230400	239017	7503	3.1 %
Moving sources at 120Hz for 60 seconds				
No.	**Estimated**	**Received via OSC**	**Stored in Buffer**	**Ratio**
1	7200	7560	7563	100.0 %
2	14400	15136	15142	100.0 %
3	21600	22615	15093	66.7 %
4	28800	30054	15050	50.1 %
6	43200	44926	15023	33.4 %
8	57600	59812	14984	25.1 %
64	460800	476381	14936	3.1 %
Moving sources at 240Hz for 60 seconds				
No.	**Estimated**	**Received via OSC**	**Stored in Buffer**	**Ratio**
1	14400	15025	15028	100.0 %
2	28800	29879	29885	100.0 %
3	43200	44675	29805	66.7 %
4	57600	59412	29729	50.0 %
6	86400	88963	29681	33.4 %
8	115200	118566	29672	25.0 %
64	921600	924816	28965	3.1 %

Table 4.3: Message handling between *oscSrcPositionHandler* and RTCommandEngine.

4.2.3 JACK: Real-Time Command Engine Evaluation

The JACK callback thread accesses the real-time command engine prior to any DSP related work. Available commands are merged, sorted and evaluated to execute all due commands in their chronological order. The minimum execution times for all this tasks resulted in 0.419 µs, again showing the possibly smallest measuring time. Without source movement, the maximum execution times resulted in 17.2 µs. Source movement resulted in mixed maximum times, ranging between 15.5 µs and 23.8 µs. Increasing the number of sources, and therefore the workload on the OSC thread, seemed to lower the maximum execution times. This is also supported by the fact, that the median values decreased from 1.257 µs to 0.987 µs, indicating more shorter execution times. The median results also stayed very close to the average results, showing equally distributed execution times. Only a slight increase was measured in the average execution times, resulting in 1.27 µs without source movement and values between 1.21 µs and 1.42 µs while using source movement. The duration parameter had no noticeable impact on the results.

The JACK DSP utilization stayed between 8.0 % and 8.9 % with moving sources, while stationary sources resulted in a peak of about 10.4 %. This shows that the JACK thread benefits from a higher workload on the OSC thread, since both threads are connected via the real-time command engine.

Stationary sources					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
64	0.419 µs	17.204 µs	1.272 µs	1.257 µs	10.4 %
Moving sources at 60Hz					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
1	0.419 µs	15.505 µs	1.257 µs	1.257 µs	8.5 %
6	0.419 µs	21.558 µs	1.291 µs	0.978 µs	8.1 %
8	0.419 µs	17.344 µs	1.349 µs	0.978 µs	8.4 %
Moving sources at 60Hz with 1s duration					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
1	0.419 µs	23.839 µs	1.212 µs	0.978 µs	8.1 %
6	0.419 µs	17.274 µs	1.304 µs	0.978 µs	8.0 %
8	0.419 µs	16.808 µs	1.420 µs	0.978 µs	8.9 %

Table 4.4: RTCommandEngine evaluation inside JACK's callback process.

4.2.4 JACK: WFS Calculations

The WFS calculations inside the JACK callback happens twice to interpolate between consecutive positions of moving sound sources and to ensure smoother value changes. All available sources (64) are always processed, while deactivated sources are just muted. This saves a case distinction during the calculations and improves the performance, which resulted in minimum execution times of 0.419 µs throughout all test runs, again indicating the probably lowest resolution for time measurements. The maximum execution times varied between 24.1 µs and 34.6 µs and decreased with higher numbers of moving sources, showing that a higher workload on the OSC thread led to a better overall performance. The average execution times resulted in about 1.1 µs throughout all test combinations and stayed very close to the constant median execution times of 0.978 µs, showing an equal distribution of execution times. The duration parameter had no influence on the test results.

The JACK DSP utilization increased to an abnormal level between 88.3 % and 93.0 %. Since the WFS calculations are numerous and very short, the high DSP usage was due to the integration of the measuring code, that introduced a lot of overhead and context switches.

Stationary sources					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
64	0.419 µs	24.119 µs	1.110 µs	0.978 µs	92.7 %
Moving sources at 60Hz					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
1	0.419 µs	32.174 µs	1.085 µs	0.978 µs	89.0 %
6	0.419 µs	32.872 µs	1.087 µs	0.978 µs	88.3 %
8	0.419 µs	25.027 µs	1.086 µs	0.978 µs	93.0 %
Moving sources at 60Hz with 1s duration					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
1	0.419 µs	34.619 µs	1.086 µs	0.978 µs	88.3 %
6	0.419 µs	23.979 µs	1.085 µs	0.978 µs	88.3 %
8	0.419 µs	24.724 µs	1.087 µs	0.978 µs	89.0 %

Table 4.5: WFS calculations inside JACK's callback process.

4.2.5 JACK: Delay Line Interpolation

The delay line interpolation inside the JACK callback uses the results of both preceding WFS calculations to read weighted and delayed samples from the delay line of each sound source. Then it copies the read samples into the individual audio output buffers. Again, all 64 sound sources are always processed to save a case distinction and improve the performance of the code. During the test runs the minimal execution times stayed at the lowest seen value of 0.419 µs. The maximum execution times ranged between 23.6 µs and 31.8 µs, while the average execution times resulted in about 1.2 µs. Both produced similar results to the WFS calculations. The same holds true for the median execution times, which resulted in values between 0.978 µs and 1.4 µs. This again shows that the execution times were equally distributed. Stationary sources, due to the lack of a workload on the OSC thread, returned worse results than moving sources, pointing out the entanglement between the OSC and JACK thread. The duration parameter had no effect on the execution times.

The JACK DSP utilization reached the same high values as during the WFS calculations. Results between 88.0 % and 94.3 % indicate a massive overhead that is caused by placing the measuring code inside the JACK callback process.

Stationary sources					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
64	0.419 µs	27.332 µs	1.221 µs	1.396 µs	94.3 %
Moving sources at 60Hz					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
1	0.419 µs	23.932 µs	1.180 µs	0.978 µs	88.0 %
6	0.419 µs	25.166 µs	1.184 µs	0.978 µs	89.0 %
8	0.419 µs	31.801 µs	1.189 µs	0.978 µs	90.0 %
Moving sources at 60Hz with 1s duration					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
1	0.419 µs	28.333 µs	1.184 µs	0.978 µs	88.7 %
6	0.419 µs	30.684 µs	1.188 µs	0.978 µs	91.0 %
8	0.419 µs	23.583 µs	1.188 µs	0.978 µs	89.3 %

Table 4.6: Delay line interpolation inside JACK's callback process.

4.2.6 JACK: DSP

The DSP part inside the JACK callback includes everything but the evaluation of the real-time command engine. This involves managing the audio buffers, compute the WFS parameters, use them to read interpolated samples from the delay lines and interpolate all source positions. Here, the minimum execution times resulted in values between 141.0 µs and 143.6 µs. The maximum execution times for stationary sources were much higher at 644.7 µs, while the execution times for moving sources varied between 320.0 µs and 358.6 µs. This is also true for the average execution times, where stationary sources resulted in 218.2 µs and moving sources took about 185.5 µs to 190.7 µs. The median execution times showed the same pattern and were nearly identical to the average times, indicating an equal distribution. A small impact of the duration parameter could be observed on the test results, ranging from 1 % to 4 %.

The JACK DSP utilization reached 7.9 % to 10.8 % and was higher than without the integration of the measuring code, where it only reached 1 % to 2 %. Moreover, a decline in the performance of the JACK thread could be seen while using stationary sources, resulting in 12.8 % of DSP usage. This again demonstrates how a missing workload on the OSC thread simultaneously reduces the performance of the JACK thread.

Stationary sources					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
64	143.618 µs	644.711 µs	218.212 µs	214.112 µs	12.8 %
Moving sources at 60Hz					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
1	141.174 µs	344.018 µs	185.473 µs	184.359 µs	9.5 %
6	142.221 µs	329.910 µs	187.252 µs	184.732 µs	8.2 %
8	142.384 µs	320.062 µs	187.530 µs	184.778 µs	7.9 %
Moving sources at 60Hz with 1s duration					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
1	140.987 µs	358.615 µs	186.225 µs	184.569 µs	10.8 %
6	142.501 µs	356.264 µs	188.419 µs	187.572 µs	8.1 %
8	143.642 µs	328.676 µs	190.696 µs	190.272 µs	8.5 %

Table 4.7: JACK callback process without RTCommandEngine evaluation (DSP only).

4.2.7 JACK: Callback Process

The JACK callback consists of the evaluation of the real-time command engine and the DSP computations. Since the evaluation process was very short (see Table 4.4), its impact should be negligible. However, the maximum execution times with stationary sources resulted in a 85.8 µs shorter time of 558.9 µs. The rest of the measuring results varied between 1 % and 9 %, but were basically the same.

Highlighted in red are the results for 64 moving sources at 60 Hz, which is the maximum number of active sources in the default configuration of the software. The purpose is to show the ability of the JACK thread to render 64 sources within the callback latency of 2667 µs, while the OSC thread is simultaneously receiving new positions for all sources. Despite the fact, that the RTCommandEngine can only store the data of up to two moving sources (see Table 4.3), this results confirm the feasible operation of the software.

The JACK DSP usage ranged from 7.8 % to 8.4 % and was about 2 % lower than in the previous test. Stationary sources caused a slightly lower utilization of 11.1 %. Even though this test included the evaluation of the real-time command engine and thus had more computations, the overall results improved a bit. The positioning of the measuring code at the start and end of the function may have led to better compiler optimizations.

Stationary sources					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
64	144.433 µs	558.922 µs	208.939 µs	202.938 µs	11.1 %
Moving sources at 60Hz					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
1	141.313 µs	332.819 µs	185.657 µs	184.033 µs	8.1 %
6	142.222 µs	333.542 µs	187.464 µs	185.174 µs	8.0 %
8	142.431 µs	341.899 µs	188.209 µs	184.872 µs	7.8 %
64	140.545 µs	393.327 µs	199.107 µs	185.524 µs	8.2 %
Moving sources at 60Hz with 1s duration					
No.	**Minimum**	**Maximum**	**Average**	**Median**	**JACK DSP**
1	142.012 µs	358.009 µs	187.142 µs	185.640 µs	8.4 %
6	143.479 µs	326.721 µs	188.732 µs	187.595 µs	8.0 %
8	144.014 µs	328.211 µs	190.353 µs	189.481 µs	8.2 %
64	154.025 µs	393.536 µs	214.053 µs	216.137 µs	8.3 %

Table 4.8: JACK callback process with RTCommandEngine evaluation.

4.2.8 JACK: DSP Utilization

The DSP usage was captured with QjackCtl, a GUI application used to start the JACK server. Without connected clients, the DSP usage was below 1 %. Running tWONDER in its original state resulted in a DSP usage of about 2 %. Source movement and different refresh rates had no effect on the DSP usage. After implementing the measuring code, all computations showed an increased DSP usage between 7.8 % and 12.8 %. However, the WFS calculations and delay line interpolations resulted in a 94.3 % DSP usage and a lot of xruns. Since both computations are relatively short with plenty of iterations, this indicates that the measuring code, although simple, had a huge impact on the DSP usage, especially for small code fragments. It generated too much overhead and resulted in longer execution paths and poorer optimizations.

Interestingly, during the tests of the original code, while the DSP usage stayed the same, the CPU usage increased with higher refresh rates. The OSC and JACK threads were examined separately and both had the same CPU usage. Since the JACK thread has far more computations than the OSC thread, an equal CPU usage is highly unlikely. Also, higher refresh rates would only increase the workload on the OSC thread, while the computations of the JACK thread would stay the same. The equal CPU utilization of both threads indicates a mutual influence due to the real-time command engine and its synchronization mechanism that causes a negative impact on the performance of both threads.

JACK server DSP usage (without measuring code)	
64 stationary sources	1 % - 2 % (CPU usage: 2 % - 4 %)
64 moving sources (60Hz)	1 % - 2 % (CPU usage: 6 % - 8 %)
64 moving sources (120Hz)	1 % - 2 % (CPU usage: 12 % - 15 %)
64 moving sources (240Hz)	1 % - 2 % (CPU usage: 22 % - 25 %)
JACK server DSP usage at 60Hz (with measuring code)	
OSC RTCommandEngine access	8.0 % - 8.9 %
JACK RTCommandEngine evaluation	8.0 % - 10.4 %
JACK WFS calculations	88.3 % - 93.0 %
JACK Delay line interpolation	88.0 % - 94.3 %
JACK DSP	7.9 % - 12.8 %
JACK whole callback	7.8 % - 11.1 %

Table 4.9: JACK server DSP usage.

4.2.9 Delay Line: CPU Cache References

The results are discussed in subsubsection 4.3.3.

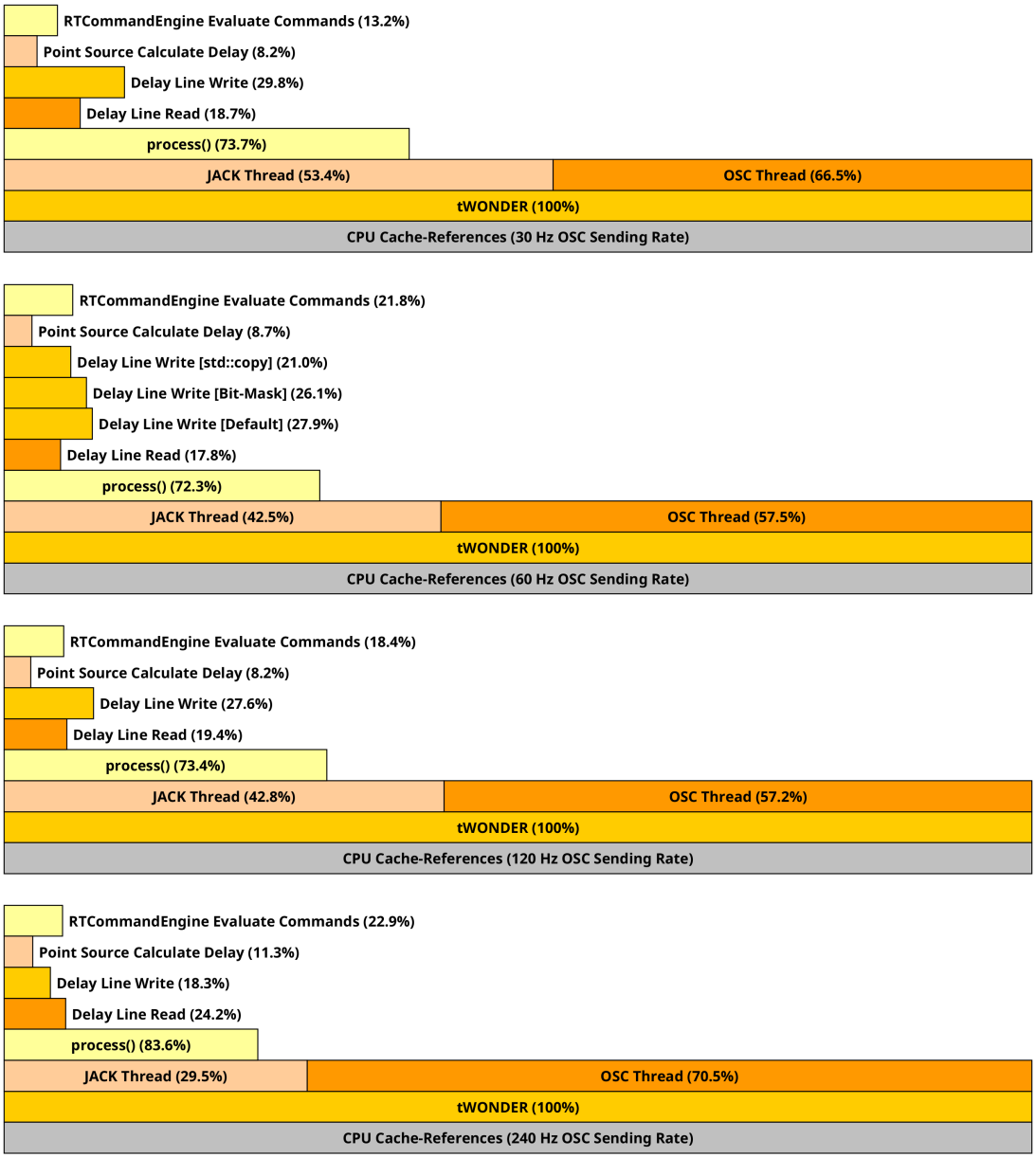

Figure 4.4: CPU cache references in tWONDER using different OSC sending rates.

4.2.10 Delay Line: CPU Cycles

The results are discussed in subsubsection 4.3.3.

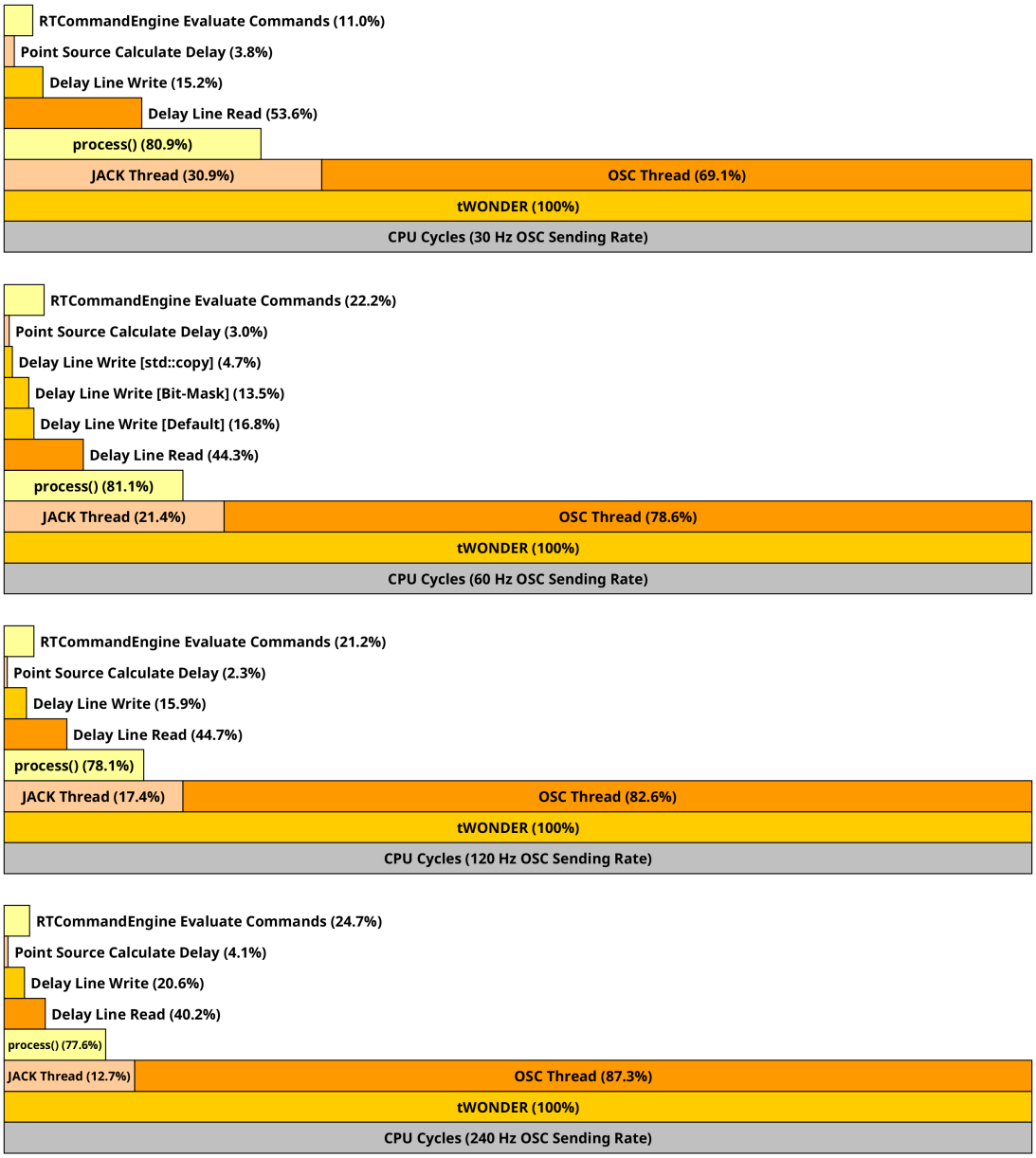

Figure 4.5: CPU cycles in tWONDER using different OSC sending rates.

4.2.11 Delay Line: CPU Instructions

The results are discussed in subsubsection 4.3.3.

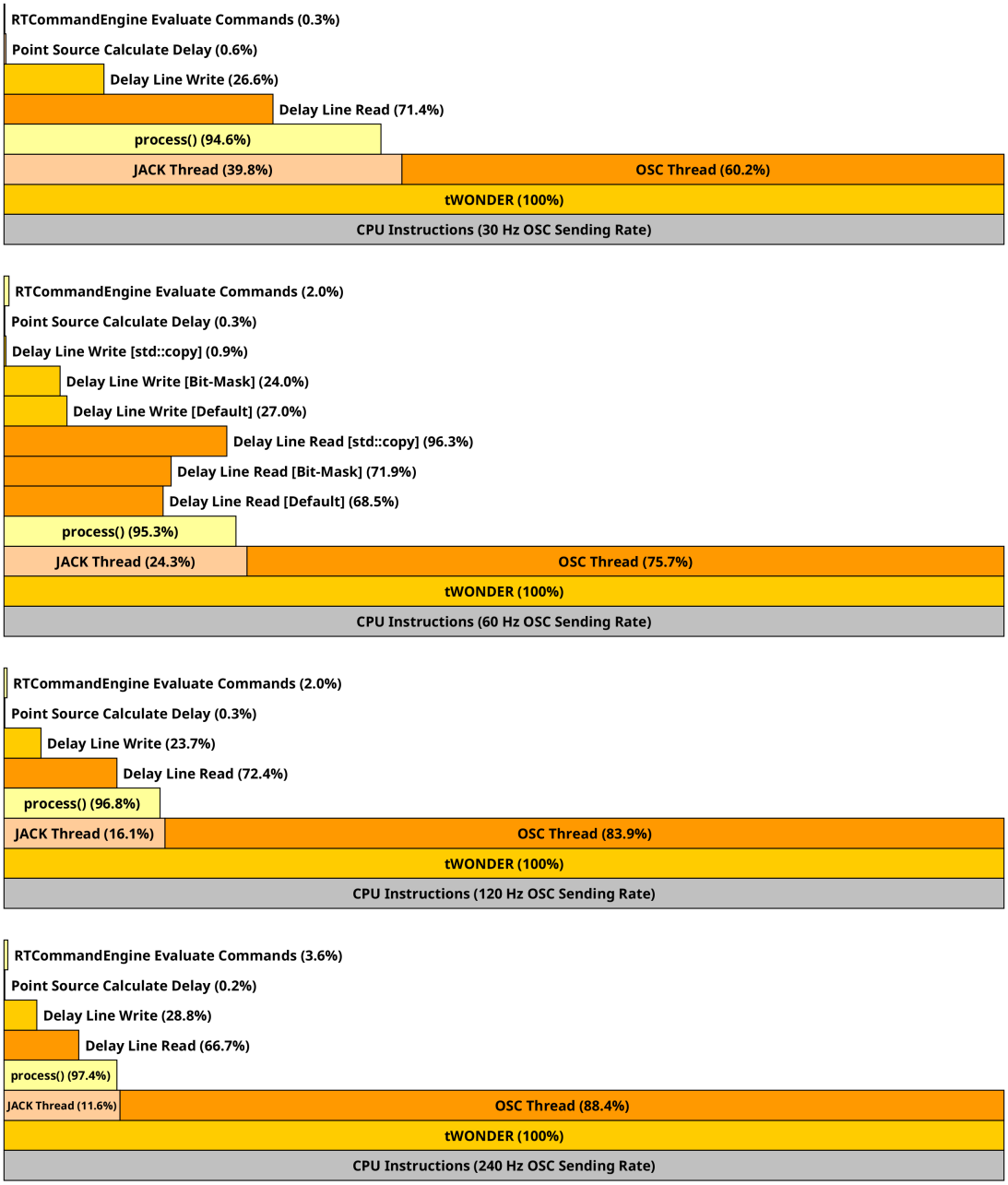

Figure 4.6: CPU instructions in tWONDER using different OSC sending rates.

4.3 Benchmark Discussion

In this section the results of the time measurements are interpreted. The OSC thread and its access to the real-time command engine is discussed first. Afterwards, the JACK thread is examined, including all the separated steps during the callback.

4.3.1 OSC Results

The OSC tests examined the performance and the impact of the OSC thread on the real-time command engine. Considering its performance, the OSC server worked as expected and handled every incoming message by reliably triggering the *oscSrcPositionHandler* function (see subsection 4.2.2). However, inside the *oscSrcPositionHandler* function, it had problems to continuously call the *put* function of the RTCommandEngine object and therefore was unable to store all new commands inside the buffer of the real-time command engine. The tests revealed, that no matter what refresh rate was being used, only the messages of two moving sources could be stored completely. Every additional source caused a data loss, since it increased the amount of incoming OSC messages, while the number of *put* calls stayed the same as for two moving sources. In the end, the inability to write new commands for more than two moving sources into the buffer, reached a total loss of 97 % when using 64 moving sources (see Table 4.3).

The results raise considerable doubts about the reliability of the real-time command engine. Tracing down the causes is complicated, since there are multiple factors that might have led to this unexpected behavior. It is likely, that a mix of different thread priorities, synchronization mechanisms, *sleep* calls and a small buffer size is responsible for the diminishing performance of the real-time command engine. This is supported by the fact, that the performance of the OSC thread increased with higher workloads and, in turn, also improved the performance of the JACK thread. Mutual influence can also be seen in the CPU utilization, where both threads, despite of different workloads, always share the same usage percentage (see subsection 4.2.8).

Recommending a general fix for these problems is difficult. A larger buffer might help, but thread priorities and the currently used synchronization mechanisms could still interfere. Other variables are the OSC sending rate, the number of sending devices, as well as the JACK sample rate and buffer size, which both define the JACK callback latency. All these parameters have to be considered, since each installation and setup is different.

4.3.2 JACK Results

The JACK callback thread consists of two separate computation steps. First, it evaluates due commands inside the real-time command engine and then it performs DSP. Compared to the DSP computations, the execution times of the evaluation were very short and only took about 1 % to 6 % of the JACK callback time. However, the evaluation step also interconnects the JACK thread with the OSC thread and causes several problems. One is the thread synchronization that introduces a harmful mutual influence. Since both threads have different priorities and refresh rates, the idleness of the OSC thread can slow down the performance of the JACK thread. Conversely, a higher workload on the OSC thread can also increase the workload on the JACK thread (see subsection 4.2.8). Another problem is the buffer of the real-time command engine. It is implemented with a *CommandQueue* object, which is derived from a *JackRingbuffer* object, which in turn is a C++ wrapper for a ring buffer implementation in the JACK C API (see Figure 4.3). Its size defaults to only 256 elements, but this number is hidden as a constructor parameter in the header file of the *JackRingbuffer* object. Despite running and evaluating commands at 375 Hz, the JACK callback thread is unable to free enough space in the real-time command engine, while the OSC thread simultaneously tries to write new commands into it. This results in an OSC data loss of up to 97 % (see subsection 4.2.2).
Short code sections with large iteration numbers, like the WFS calculations and delay line interpolations, showed problems with the measuring code, since its usage introduced more workload than the code to be measured itself. It is therefore inadvisable to measure such small fragments inside nested loops and instead, measure the whole loop or function. Regarding the overall performance of the JACK callback process, it had no problems to meet the latency deadline of 2667 µs (see subsection 4.2.7). By putting the DSP usage and the JACK callback time in relation, where 100 % utilization corresponds to 2667 µs, the average execution times with a roughly 8 % DSP usage would result in about 213.4 µs (2667 µs / 100 · 8 = 213.36 µs). This is nearly in line with the measured average execution times of about 190.4 µs. However, these values were achieved with twonder using the measuring code, while the original version only caused about 2 % DSP usage. Converting the DSP usage back to execution time results in about 53 µs (2667 µs / 100 · 2 = 53 µs). It should be noted that the DSP and CPU utilization increased differently in tWONDER. Without measuring code, an increased OSC workload showed a higher CPU than DSP growth (see subsection 4.2.8), indicating that thread synchronization and single-threaded programming significantly reduce the overall performance at a certain point.

4.3.3 Delay Line Results

The Bresenham algorithm was used for resampling to achieve the Doppler effect [5]. But it was implemented in a naive or brute force way. A lot of safety measures and *if*-cases were used to prevent possible sound distortions. Code-wise this leads to a construct that has more than 250 lines of code. Compared to a simple delay line implementation with a linear interpolation, this code is overly complicated and very slow.

CPU Cache References

The percentages show that a higher OSC sending rate led to an increased cache usage of the OSC thread, with going up to 70 % at 240 Hz. The delay line read function with the Bresenham algorithm uses the most references throughout the runs, while the RTCommandEngine stays the same. Using the optimized write implementations shows that the *std::copy* version performs the best.

CPU Cycles

The percentages show that a higher OSC sending rate led to an increased cycle count of the OSC thread, with going up to 87 % at 240 Hz. The delay line read function with the Bresenham algorithm uses the most cycles throughout the runs. Using the optimized write implementations decreases the cycle count significantly with the *std::copy* version performing the best.

CPU Instructions

The percentages show that a higher OSC sending rate led to an increased instruction count of the OSC thread, with going up to 88 % at 240 Hz. The delay line read function with the Bresenham algorithm uses the most instructions throughout the runs, however when using the improved write versions, it gets more instructions or more CPU time. Using the optimized write implementations reduces the instruction count immensely with the *std::copy* version performing the best.

4.4 Data Flow Analysis

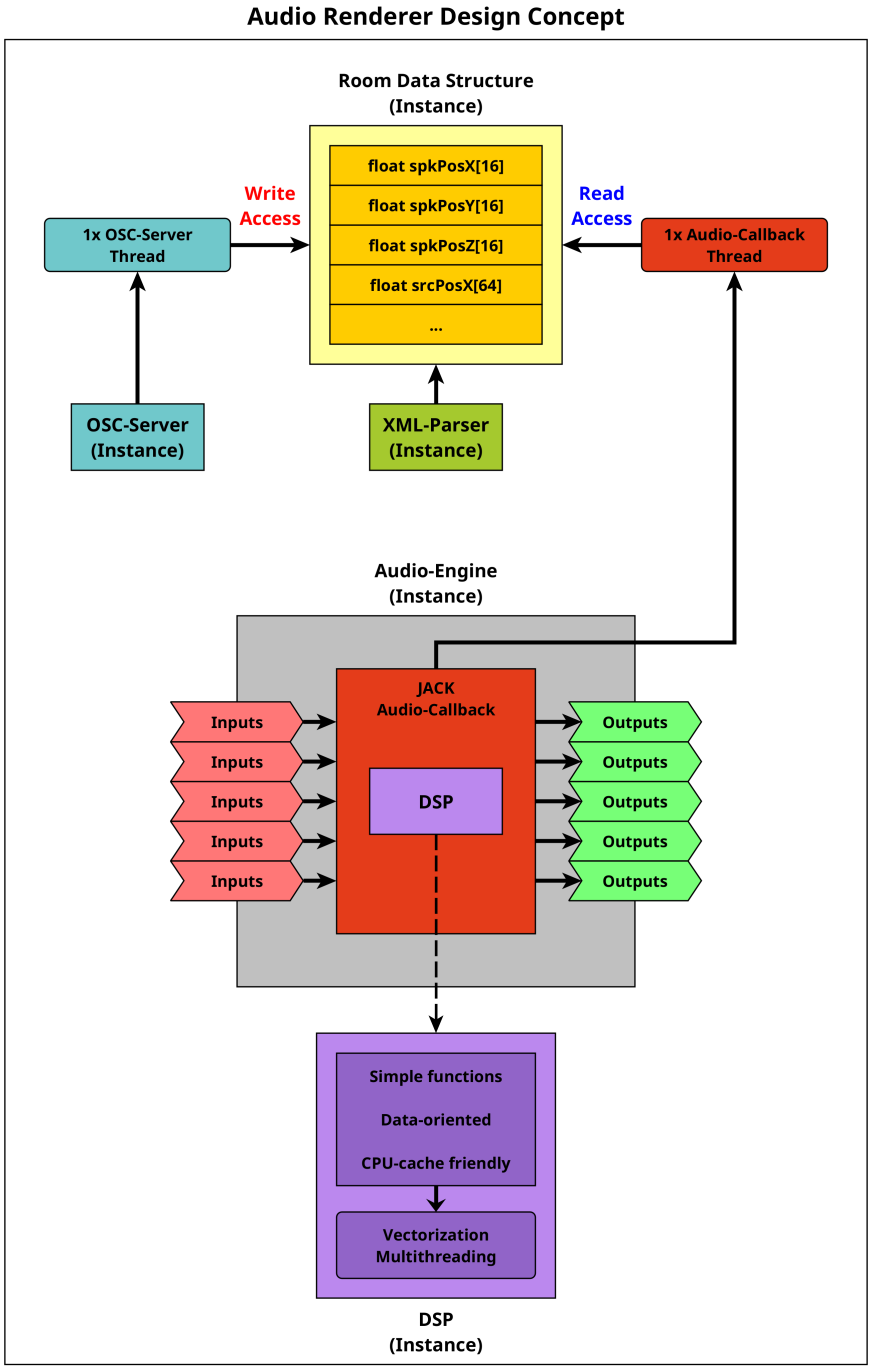

Figure 4.7: Design concept of a data-oriented audio renderer with data flow in mind.

4.5 Conclusion

This chapter introduced the WONDER software suite and its programs. A benchmark was planed and conducted to measure the performance of the audio and networking part in tWONDER and the delay line implementation. The benchmark results were analyzed and discussed. Additionally, a data flow analysis was created.

5 Delay Line

The delay line is a core element in the realization of a WFS application. It contains the delaying part of the WFS driving functions and ensures that all loudspeakers play back their audio signals precisely synchronized to synthesize a valid wavefront. This section starts with a basic step by step implementation of a delay line. It also contains the handling of moving virtual sound sources and the associated Doppler effect. Some adjustments and optimizations are discussed as well.

5.1 Circular Buffer

The delay line is implemented using a circular buffer approach with a single write and multiple read pointers. This is due to the fact that every audio input channel represents a virtual sound source and has its own delay line. Every loudspeaker is represented by an audio output channel and has its own delay and weight values calculated by the WFS driving functions. As a result, the same delay line buffer is read multiple times to obtain the individually delayed audio samples for each loudspeaker. An array or a *std::vector* can be used to store the incoming audio samples. After reaching the last index, i.e. the buffer is full, the pointers are wrapped around and restart from the beginning. At this point the linear storage becomes circular, hence the name (see Figure 5.1).

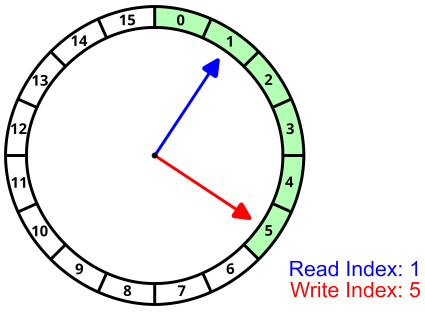

Figure 5.1: Circular buffer with read and write pointer.

5.2 Buffer Size

The size of the delay line buffer is affected by multiple factors. First, the chosen size has to be immutable. This prevents memory allocations and negative performance impacts during runtime. For the same reason all buffer memory is pre-allocated at the start of the application. The size itself depends on the longest distance a point source will be placed behind the loudspeaker array. In addition, using focused point sources introduces a necessary pre-delay (see subsection 3.4.4). This is based on the longest distance a focused point source will be projected into the listening area and placed in front of the loudspeaker array. The sum of both distances divided by the speed of sound results in the highest usable delay in seconds. A subsequent multiplication with the sample rate of the WFS system yields the final buffer size for the delay line in samples (see Equation 5.1).

$$\frac{d_{pre} + d_{post}}{c} \cdot SR = \frac{5\,\mathrm{m} + 250\,\mathrm{m}}{343.1\,\mathrm{m/s}} \cdot 48\,000\,\mathrm{Hz} \approx 35675\ samples \tag{5.1}$$

5.2.1 Two's Complement

Before introducing the read and write functions, an optimization will be applied to the delay line buffer. During their access, a constant boundary check is needed to see if the pointers have reached the end of the circular buffer and have to be wrapped around. The *modulo* operator ($\%$) or an *if*-statement are usually used for this, but especially the modulo operator is expensive and costs a lot of CPU cycles. To speed up the boundary checks, the buffer size is always set to a power of two. This enables the usage of bitwise operations, as long as both numbers have an unsigned integer type, which is the case here. The costly modulo operator (x % bufferSize) can now be replaced by a simple bitwise *AND* applied to a bit-mask (x & [bufferSize - 1]) (see Equation 5.2 and Equation 5.3).

$$x \,\%\, 2^n \equiv x \,\&\, (2^n - 1) \mid x, n \in \mathbb{N} \tag{5.2}$$

$$\begin{array}{cccc} & 0 & 0001 & 1010 & x \\ AND & 0 & 1111 & 1111 & Bitmask\ (2^8 - 1 = 255) \\ \hline = & 0 & 0001 & 1010 & Result \end{array} \tag{5.3}$$

However, this approach might use more memory in some edge cases, where picking the next fitting power of two results in a memory consumption twice as high. The above buffer size of 35675 ($> 2^{15}$) samples would rise to a buffer size of 65536 (2^{16}) samples.

5.3 DelayLine Class

The step-wise delay line implementation starts with a rudimentary body (see Listing 5.1). It acts as a foundation for functions that will be added in the upcoming sections. Since the audio samples are provided as an array of *float* samples, the buffer uses it as well. The memory of the array is pre-allocated in the constructor and freed in the destructor. When initializing the size of the buffer, an additional pre-delay for focused point sources should be kept in mind (see subsection 3.4.4). A separate function ensures that the buffer size is always a power of two. This is required to calculate a valid bit-mask and utilize its performance gains (see subsection 5.2.1).

```
1  class DelayLine {
2   public:
3    DelayLine(unsigned int bufferSize) {
4      m_bufferSize = powerOfTwo(bufferSize);
5      m_bufferMask = (m_bufferSize - 1);
6      m_writePointer = 0;
7      m_buffer = new float[m_bufferSize]; // Allocate memory.
8    }
9
10   ~DelayLine() {
11     delete[] m_buffer; // Free memory.
12   }
13
14  private:
15   inline unsigned int powerOfTwo(unsigned int value) {
16     unsigned int targetValue = value;
17
18     if((targetValue & (targetValue - 1)) != 0) {
19       targetValue = std::pow(2, std::ceil(std::log(targetValue) /
20                     std::log(2)));
21     }
22
23     return targetValue;
24   }
25
26   float* m_buffer;
27   unsigned int m_bufferSize;
28   unsigned int m_bufferMask;
29   unsigned int m_writePointer;
30 };
```

Listing 5.1: Initial DelayLine class.

5.4 Write Function

In this implementation the write process is the first step in the order of execution and happens before the read process. This naive version copies sample by sample in a *for*-loop and uses an *if*-statement to wrap around the write pointer (see Listing 5.2).

```
void write(float* audioInput, unsigned int numSamples) {
  for (unsigned int i = 0; i < numSamples; i++) {
    m_buffer[m_writePointer] = audioInput[i];
    m_writePointer++;

    if (m_writePointer >= m_bufferSize) {
      m_writePointer = 0;
    }
  }
}
```

Listing 5.2: Initial version of the DelayLine *write* function.

Using the two's complement (see subsection 5.2.1) omits the *if*-clause (see Listing 5.3).

```
void write(float* audioInput, unsigned int numSamples) {
  for (unsigned int i = 0; i < numSamples; i++) {
    m_buffer[m_writePointer] = audioInput[i];
    m_writePointer++;
    m_writePointer = (m_writePointer & m_bufferMask);
  }
}
```

Listing 5.3: Bit-mask version of the DelayLine *write* function.

The slow *for*-loop can further be replaced by a faster *std::copy* (see Listing 5.4).

```
void write(float* audioInput, unsigned int numSamples) {
  // Calculate the number of copyable samples before the end of the buffer.
  size_t first = std::min(numSamples, (m_bufferSize - m_writePointer));

  // Copy the first part and a possible rest of up to numSamples.
  // Increase the write pointer and apply the bit-mask.
  std::copy(audioInput, (audioInput + first), (m_buffer + m_writePointer));
  std::copy((audioInput + first), (audioInput + numSamples), m_buffer);
  m_writePointer = (m_writePointer + numSamples) & m_bufferMask;
}
```

Listing 5.4: *std::copy* version of the DelayLine *write* function.

5.5 Integer Read Function

The read process, in its simplest form, takes place after the write process and uses an integer delay parameter to read delayed samples from the delay line buffer and write them into the audio output array. An *if*-clause is necessary to check if the delay exceeds the size of the buffer. The read pointer is then derived by subtracting the given delay from the write pointer. Additionally, the number of previously written samples `numSamples` has to be subtracted as well (see Listing 5.5). This is important because a delay value of zero requires reading these currently written samples without any delay. The bit-mask is again utilized to remove an *if*-statement that would check the boundaries of the buffer. While the restriction on integer delay values ensures a sample correct delay and an easier handling of the read pointer, it should be emphasized that this solution is less suitable to be used in WFS. Here, the virtual sound sources and loudspeakers can have arbitrary positions, which results in decimal delay values. A delay line implementation with the ability to read fractional delays is necessary (see section 5.6).

```
1  void read(float* audioOutput, unsigned int d, unsigned int numSamples) {
2    // Constrain the maximum delay length.
3    unsigned int delay = 0;
4
5    if (d > m_bufferSize) {
6      delay = m_bufferSize;
7    } else {
8      delay = d;
9    }
10
11   // Calculate the delayed read pointer based on the write pointer.
12   unsigned int readPointer = (m_writePointer - delay) - numSamples;
13
14   for (unsigned int i = 0; i < numSamples; i++) {
15     // Apply bit-mask when reading from the buffer.
16     audioOutput[i] = m_buffer[(readPointer & m_bufferMask)];
17     readPointer++;
18   }
19 }
```

Listing 5.5: Integer version of the DelayLine *read* function.

5.6 Fractional Read Function

The functionality to use arbitrary delay values requires a read in between samples and therefore a fractional read pointer. Since the delay line buffer array and the write process still operate on an integer base, the read pointer now has to use two samples instead of just one to calculate an intermediate value. The fraction of the delay value is used to weight both samples proportionally and is applied as linear interpolation between two adjacent samples (see Equation 5.4 and Figure 5.2). Some modifications are added to the prior integer design (see Listing 5.6). The given delay must be checked for negative values. Two read positions are now needed and both have to be integer values, since accessing the buffer array still requires integer indices. A type cast to integer is used to form these indices of the current and upcoming read position. The bit-mask ensures that they stay inside the boundaries of the buffer size. Subtracting the integer part from the fractional read position extracts the fraction, valued between 0.0 and 1.0, which is then used for a linear interpolation between the two adjacent sample values x_0 and x_1. The increment of the fractional read pointer is treated similarly separated. After adding 1, the fraction has still the same value. By casting the incremented read pointer to an integer value, the bit-mask can be applied to wrap it around the array boundaries. The fraction is then added back to the read pointer and thus preserves the fractional delay.

$$y = ((1.0 - frac) \cdot x_0) + (frac \cdot x_1) \tag{5.4}$$

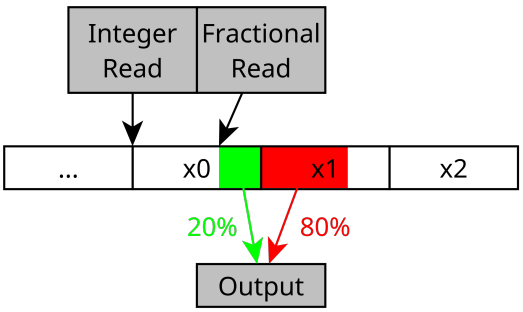

Figure 5.2: Fractional read with linear interpolation of two adjacent samples.

```
void read(float* audioOutput, float d, unsigned int numSamples) {
  // Constrain the minimum and maximum delay length.
  float delay = 0;

  if (d > m_bufferSize) {
    delay = m_bufferSize;
  } else if (d < 0) {
    delay = 0;
  } else {
    delay = d;
  }

  // Calculate the delayed read pointer based on the write pointer.
  float readPointer = (float)m_writePointer - delay - (float)numSamples;

  for (unsigned int i = 0; i < numSamples; i++) {
    // Form two adjacent integer read positions and a separate fraction.
    unsigned int x0 = (unsigned int)readPointer & m_bufferMask;
    unsigned int x1 = (x0 + 1) & m_bufferMask;
    float frac = readPointer - (float)x0;

    // Use linear interpolation to construct the output sample.
    audioOutput[i] = ((1.0 - frac) * m_buffer[x0]) + (frac * m_buffer[x1]);

    // Increase read pointer and wrap around with integer bit-mask.
    readPointer += 1;
    readPointer = (unsigned int)readPointer & m_bufferMask;

    // Transfer the separate fraction back to the read pointer.
    readPointer += frac;
  }
}
```

Listing 5.6: Fractional version of the DelayLine *read* function.

5.7 Doppler Effect

The Doppler effect, described by the Austrian physicist Christian Doppler in 1842, is a perceived change in the pitch of a moving sound source, depending on the relative velocity between the sound source and a listener. A moving sound source compresses its wavefront towards the moving direction and expands it behind it. The original wavefront occurs more frequently in the front and less frequently in the back (see Figure 5.3). As a result, a stationary listener perceives an approaching source at a higher and a leaving source at a lower pitch.
The calculation of the Doppler shift and the associated change in frequency and pitch depends on the relative velocity of both, the source and the listener. To reduce the complexity and possible outcomes, only four situations are considered, where either the source or the listener is moving, but not both at the same time (see subsection 5.7.1). It can also be expected that the listener is always stationary, as it seems highly unlikely that a listener inside the listening area can move fast enough to cause a noticeable Doppler shift. This covers the most common scenario when using WFS, namely the simulation of moving sound sources, and yields an implementation with two conditions, one with an arriving and one with a leaving sound source.
Implementation-wise, the rendering of moving sound sources needs a fractional delay line, which was already implemented in section 5.6. Additionally, the delay line processing has to be extended to integrate the velocity of a moving sound source as an extra argument. This so-called *time-varying* delay line read variant is presented in section 5.8.

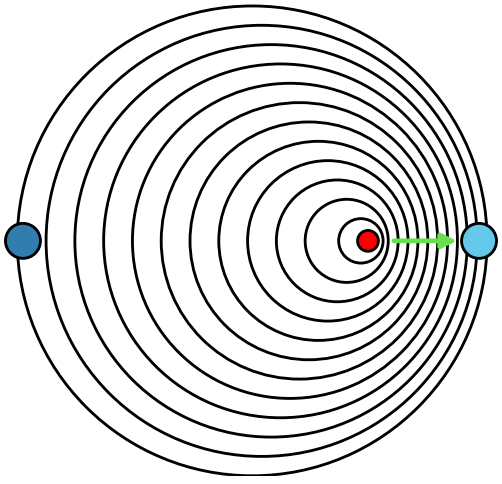

Figure 5.3: Propagating sound waves of a moving source.

5.7.1 Doppler Shift Equations

The Doppler frequency f_l perceived by a listener is calculated using the base frequency emitted by the stationary source f_s, the source velocity relative to the listener v_{rel} and the speed of sound c (343.1 m/s) [28]. As mentioned before, the implementation of the Doppler shift in the WFS application only utilizes the first two equations, since mostly moving sound sources are relevant for the rendering process, while a listener is considered to be stationary.

1. **Stationary listener and arriving source:**

$$f_l = \frac{f_s}{(1 - (\frac{v_{rel}}{c}))} \tag{5.5}$$

2. **Stationary listener and leaving source:**

$$f_l = \frac{f_s}{(1 + (\frac{v_{rel}}{c}))} \tag{5.6}$$

3. **Stationary source and arriving listener:**

$$f_l = f_s \cdot (1 + (\frac{v_{rel}}{c})) \tag{5.7}$$

4. **Stationary source and leaving listener:**

$$f_l = f_s \cdot (1 - (\frac{v_{rel}}{c})) \tag{5.8}$$

5.8 Time-Varying Read Process

The time-varying delay line read process is mainly based on the fractional variant shown in section 5.6. The main difference is the addition of a velocity parameter to simulate a moving sound source that exhibits a Doppler shift. This velocity parameter has to be incorporated into the read process to reflect the frequency changes. Considering the regular read process, the iteration of the read pointer is always one sample at a time. This represents the frequency of a stationary source. To simulate an arriving source, whose pitch appears higher, the read pointer has to advance faster through the delay line. Conversely, the read pointer has to advance slower to simulate a leaving source with a lower pitch. Adding a delay growth parameter g enables the modification of the reading speeds at a sample per sample ratio [55]. The relation of this ratio is the same as between the source velocity v_s and the speed of sound c (see Equation 5.9). A default increment of one sample represents a stationary source and thus the propagation of sound waves at the speed of sound (343.1 m/s). Here, the growth parameter has a value of zero, because the source velocity v_s is also zero. An arriving source at 10 % of the speed of sound (34.31 m/s) results in a growth parameter g of -0.1, since the velocity of arriving sources is treated with a negative sign. Looking at the code and the increment of the read pointer, the calculation `readpointer += (1 - g)` yields an increment of 1.1 and consists of the stationary propagation plus the velocity of the moving sound source (see Listing 5.7). A leaving source at 10 % of the speed of sound would result in a growth parameter of 0.1 and a read pointer increment of 0.9, therefore simulating a lower pitch. Code-wise, the increment of the read pointer is followed by a regular *if*-statement to wrap it around the size of the buffer. A bit-mask is omitted, since the growth parameter changes the fraction of the read pointer every time and would require a recalculation. One extra bit of code handles the correction of the delay growth parameter. Interestingly, a modification of the read pointer to simulate a moving sound source actually simulates a moving listener. This issue is examined separately in subsection 5.8.1.

$$g = \frac{v_s}{c} = \frac{v_s}{343.1\,\mathrm{m/s}} \tag{5.9}$$

```
void read(float* audioOutput, float d, float v, unsigned int numSamples) {
  // Constrain the minimum and maximum delay length.
  float delay = 0;

  if (d > m_bufferSize) {
    delay = m_bufferSize;
  } else if (d < 0) {
    delay = 0;
  } else {
    delay = d;
  }

  // Convert given velocity in meters per second to samples per sample.
  float g = v / 343.1; // Speed of sound = 343.1 m/s.

  // Correct the delay growth parameter "g" to represent a moving source.
  if (g < 0) {
    g = (-1.0 / (1.0 + g)) + 1.0; // Arriving source.
  } else if (g > 0) {
    g = 1.0 - (1.0 / (1.0 + g)); // Leaving source.
  }

  // Calculate the delayed read pointer based on the write pointer.
  float readPointer = (float)m_writePointer - delay - (float)numSamples;

  for (unsigned int i = 0; i < numSamples; i++) {
    // Form two adjacent integer read positions and a separate fraction.
    unsigned int x0 = (unsigned int)readPointer & m_bufferMask;
    unsigned int x1 = (x0 + 1) & m_bufferMask;
    float frac = readPointer - (float)x0;

    // Use linear interpolation to construct the output sample.
    audioOutput[i] = ((1.0 - frac) * m_buffer[x0]) + (frac * m_buffer[x1]);

    // Increase read pointer with delay growth parameter and wrap around.
    readPointer += (1 - g);

    if (readPointer >= m_bufferSize) {
      readPointer -= m_bufferSize;
    }
  }
}
```

Listing 5.7: Time-Varying version of the DelayLine *read* function.

5.8.1 Delay Growth Parameter Analysis

The implementation of a time-varying delay line and the introduction of the delay growth parameter g enables the simulation of a Doppler shift (see section 5.8). However, contrary to the intention to simulate a moving sound source, the result is the simulation of a moving listener. The deviation in the Doppler shift frequency is small but measurable and becomes larger at higher velocities (see Figure 5.5 and Figure 5.6). The reason for this can be described with an analogy, using a magnetic tape or a vinyl record [28]. Some sort of reading head represents the perceived Doppler shift by a listener. Reading sound information with varying rotation speeds from the tape or record, results in lower or higher pitched versions of the sound. This is also achieved with the time-varying delay line and the delay growth parameter. To simulate a moving source, the sound information has to be recorded with a higher or lower pitch on the tape or record. This means, only a writing head is able to exactly simulate the correct Doppler shift frequencies of a moving source. Revisiting the integer write process would involve resampling, since the audio input has to be shrinked (higher pitch) or stretched (lower pitch), before writing it into the buffer. This is why most delay line implementations use fractional reads and integer writes to achieve pitch shift effects [41].

1. **Delay growth parameter correction for an arriving source** ($g < 0$)**:**

$$g = \left(\frac{-1.0}{(1.0 + g)}\right) + 1.0 \tag{5.10}$$

2. **Delay growth parameter correction for a leaving source** ($g > 0$)**:**

$$g = 1.0 - \left(\frac{1.0}{1.0 + g}\right) \tag{5.11}$$

	Stationary (0.00 m/s)	**Arriving** (34.31 m/s)	**Leaving** (34.31 m/s)
Moving source (expected)	440 Hz	489 Hz	400 Hz
Moving listener (expected)	440 Hz	484 Hz	396 Hz
Default time-varying read	440 Hz	484 Hz	397 Hz
Corrected time-varying read	440 Hz	489 Hz	401 Hz

Table 5.1: Corrected read process yielding the frequencies of a moving source.

In search of a solution for simulating moving sound sources while still using the read pointer and omitting to resample the audio input, a delay growth parameter correction was developed (see Equation 5.10 and Equation 5.11). This alters the speed of the reading pointer in such a way, that it produces the correct pitch to simulate the Doppler shift of a moving sound source. A comparison is listed in Table 5.1 and shows the expected frequencies of a moving source and listener and the measured frequencies of the default and corrected time-varying read process. As can be seen, the uncorrected version created the same frequencies as a moving listener (red), while the corrected variant actually yielded the Doppler shift frequencies of a moving sound source (green). Sounds produced by both implementations were analyzed using a FFT (see Figure 5.4). It shows that the difference in frequency is relatively small, even at a velocity of 34.31 m/s (123.5 km/h) the frequency deviation is only about 1 %, which is just over 0.6 % and the *Just-Noticeable Difference (JND)* [30]. Comparing the results for a wider range of velocities, a significant frequency deviation is only present above a velocity of ±50 m/s and comes in at about 10 Hz (see Figure 5.5). While there are divergences between both implementations at high velocities (see Figure 5.6), it is unlikely that these are reached in a regular WFS setup or would be even noticeable. However, if simulations depend on an accurate reproduction of the Doppler effect at very high velocities, the correction might be an option.

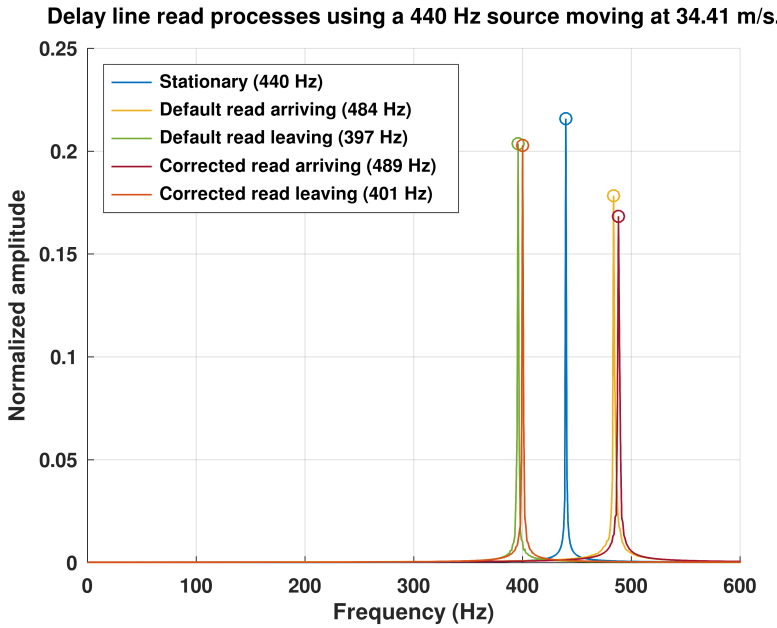

Figure 5.4: Analysis of the default and corrected read process using FFT.

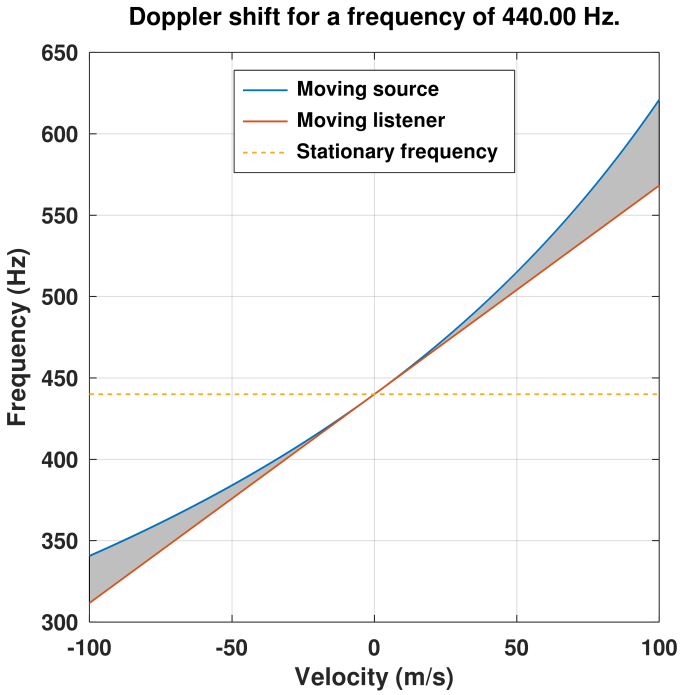

Figure 5.5: Frequency shifts for a moving or stationary source and listener.

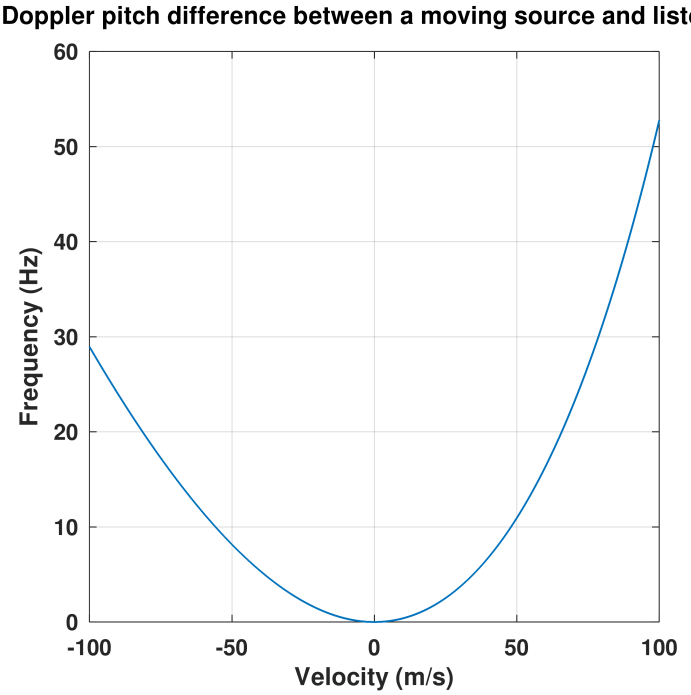

Figure 5.6: Difference in pitch between a moving source and listener.

5.9 Cross-Fading Read Process

There are situations, like spontaneous position changes of a sound source, where a Doppler effect might be undesirable due to pitch shift. An alternative way is to fade between the old and new delay. This approach is also implemented in the WONDER software suite and is called a *fade jump* [5]. Two different fade buffers can be selected, one with 3 dB and one with 6 dB. They contain attenuation factors that are applied to all samples of a buffer frame (`numSamples`) during an audio callback (see Listing 5.8). While the new delay position reads and applies the factors in normal order (fade-in), the old delay position reads the factors in reversed order (fade-out) (see Figure 5.7).

```
1  float* fadeIn3dB = nullptr;
2  float* fadeIn6dB = nullptr;
3
4  void initFadeBuffers(const unsigned int numSamples) {
5    fadeIn3dB = new float[numSamples];
6    fadeIn6dB = new float[numSamples];
7
8    float stepSize = 1.0 / (numSamples - 1);
9
10   for (unsigned int i = 0; i < numSamples; i++) {
11     float scaleFactor6dB = i * stepSize;
12
13     fadeIn6dB[i] = scaleFactor6dB;
14     fadeIn3dB[i] = std::sqrt(scaleFactor6dB);
15   }
16 }
```

Listing 5.8: Creating cross-fade arrays with different attenuation factors (WONDER).

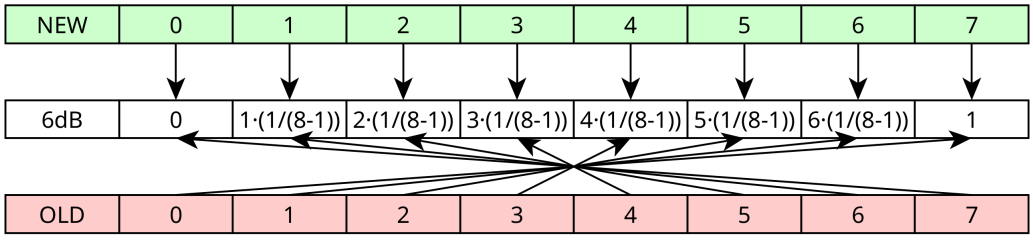

Figure 5.7: Cross-fade between old and new buffer frame using 6dB attenuation factors.

5.10 Artifacts and Noise

Despite a careful implementation, there is a possibility of unwanted noises and artifacts. Some may occur due to external failures, like wrong filters or incorrect amplitude values. Others can happen in special situations, for example with instantaneous position jumps or high velocities. It can help to restrict some parameters, like the maximum velocity, by clamping the delay growth factor to something between -0.25 and 0.25 (see section 5.8). This results in a maximum source movement of 25 % of the speed of sound or 85.8 m/s (308.8 km/h) and should be enough for most scenarios. The read pointer is particularly sensitive to big value changes, discontinuities and is not allowed to pass the write pointer, since then it would suddenly read the oldest samples. Some interpolation algorithms may sound undesirable and cause filtering or coloration of the sound [3]. In such situations, it can help to test various interpolation algorithms [71], or try something different like the fading approach in section 5.9.

5.11 Conclusion

This chapter went through an exemplary and performance oriented implementation of a delay line. Different approaches were shown for the write and read functions. By using the time-varying delay line and the delay growth parameter, the Doppler effect was incorporated to simulate a realistically sounding source movement. Furthermore, a solution was elaborated to correct the delay growth parameter and enabling the read process to actually render Doppler shifts of a moving sound source, instead of a moving listener. As an alternative, the cross-fade approach was introduced, which can replace the Doppler effect, if its pitch changes are not desired. Finally, a few tips and suggestions were mentioned, that might help to solve problems while implementing a delay line.
This was only a small insight into the world of audio processing and there are a lot more things to discover. Helpful recommendations with a more theoretical approach are the books by Udo Zölzer [74] [75]. Practical implementations and the development of audio plugins can be found in the books of Joshua D. Reiss [47] and Will C. Pirkle [41].

6 Sparse Set

The sparse set is a data structure that represents a tightly packed set of integer values [8]. All elements are organized with the help of two arrays. A *dense* array holds all integer values currently in the set in a contiguous way in the memory. The *sparse* array stores the index at which an added value can be found in the dense array. While the sparse array can exhibit gaps because of not yet added or removed integers, the dense array ensures a consecutive access to all added integers. This is especially helpful in situations with a big set size but scattered and unused elements. Besides a fast iteration, this level of indirection also makes it easy to check if an element is contained in the set. Most operations are very efficient and feasible in constant or linear time (see Table 6.1). This chapter features a basic and explanatory implementation of the sparse set, which will be used in the next chapter to realize an Entity Component System (see chapter 7).

Operation	**Cost**
insert	$\mathcal{O}(1)$
remove	$\mathcal{O}(1)$
contains	$\mathcal{O}(1)$
clear	$\mathcal{O}(1)$
iterate	$\mathcal{O}(n)$
copy	$\mathcal{O}(n)$
compare	$\mathcal{O}(n)$

Table 6.1: Complexity of different sparse set operations.

6.1 SparseSet Class

The SparseSet class uses *std::vector* to implement the dense and sparse arrays. Both are resized to their final size in the constructor. A fixed, non-growing array capacity is used to prevent memory allocations during runtime. For the sake of readability, a recurring `assert` call is omitted in the upcoming functions (see Listing 6.2). Normally embedded at the start of the functions, it helps to find invalid array accesses more easily using a debug build.

```
1  class SparseSet {
2   public:
3    SparseSet(unsigned int maxElements) {
4      m_capacity = maxElements;
5      m_size = 0;
6
7      // Allocate vector memory by resizing it to its final size.
8      m_sparse.resize(maxElements);
9      m_dense.resize(maxElements);
10   };
11
12  private:
13   std::vector<unsigned int> m_sparse;
14   std::vector<unsigned int> m_dense;
15   unsigned int m_capacity;
16   unsigned int m_size;
17 };
```

Listing 6.1: Initial SparseSet class.

```
1  assert((index < m_capacity) && "The element is out of bounds.");
```

Listing 6.2: Assert functionality implemented in the SparseSet class.

6.2 Insert Function

The insert function stores the new integer element (6) at the current `size` index (0) in the dense array. An indirection is established by storing the used dense index (0) at the element index (6) in the sparse array. The size is then incremented and points to the next empty slot in the dense array (1).

```
1  bool insert(unsigned int element) {
2    if (contains(element)) {
3      return false;
4    }
5
6    m_dense[m_size]    = element;
7    m_sparse[element] = m_size;
8    m_size++;
9
10   return true;
11 }
```

Listing 6.3: SparseSet *insert* function.

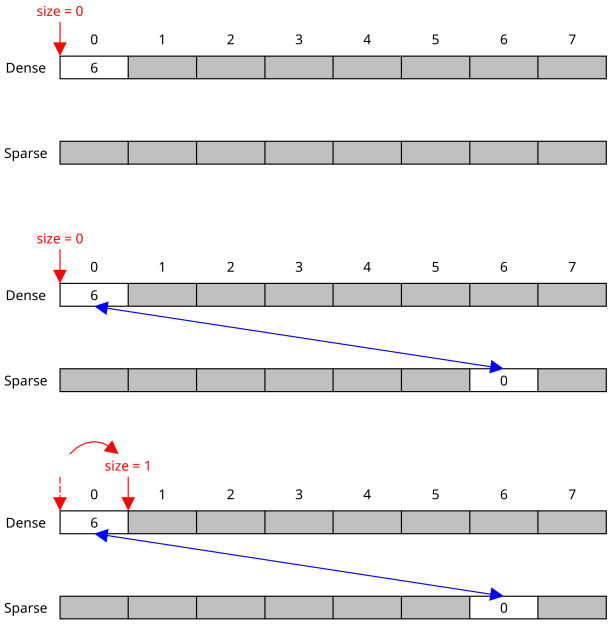

Figure 6.1: Sparse set insert function algorithm.

6.3 Remove Function

By decrementing the `size` (2), it now points to the last valid element in the dense array (5). This element will substitute the removed element (6) and is placed at the same index in the dense array (0). To restore the indirection, the value of the moved element in the sparse array (5) is set to its new position in the dense array (0).

```
1  bool remove(unsigned int element) {
2    if (!contains(element)) {
3      return false;
4    }
5
6    m_size--;
7    unsigned int movedElement = m_dense[m_size];
8    unsigned int denseIndex   = m_sparse[element];
9    m_dense[denseIndex]       = movedElement;
10   m_sparse[movedElement]    = denseIndex;
11
12   return true;
13 }
```

Listing 6.4: SparseSet *remove* function.

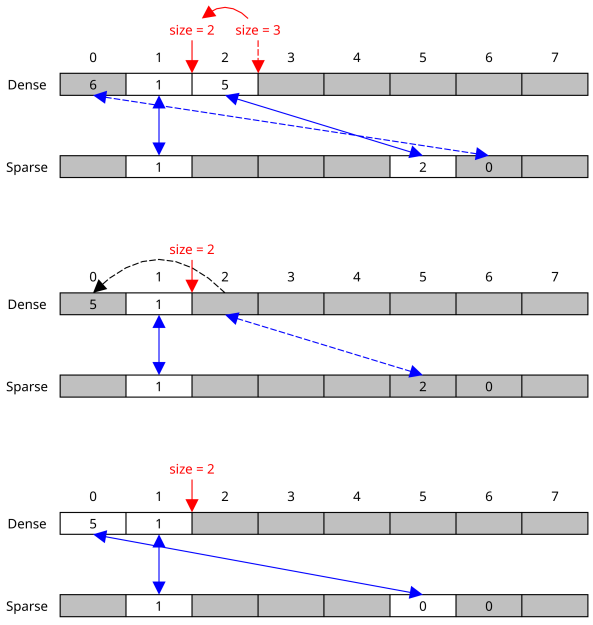

Figure 6.2: Sparse set remove function algorithm.

6.4 Contains Function

The membership of an element in the sparse set can be evaluated with the help of the `size` index and the indirection or mutual referencing scheme. Every element in the dense array below the `size` index is valid and therefore has a reference pointing into the sparse array. Using the value of the potentially contained element as an index into the sparse array, yields the index into the dense array at which the actual value of the element has to be stored. Otherwise the mutual referencing is invalid and the element is not contained in the sparse set. It even holds true after multiple insert and remove operations were performed (see Figure 6.3). Only elements pointed at with the blue double-sided arrows are contained in the set. Others may still have an index from prior operations, but not the right reference or a value above the `size` index. It is also worth mentioning that there is no need to check for elements in the set while iterating. Traversing the dense array using the `size` as a limitation is enough.

```
1 bool contains(unsigned int element) {
2   return (m_sparse[element] < m_size) &&
3          (m_dense[m_sparse[element]] == element);
4 }
```

Listing 6.5: SparseSet *contains* function.

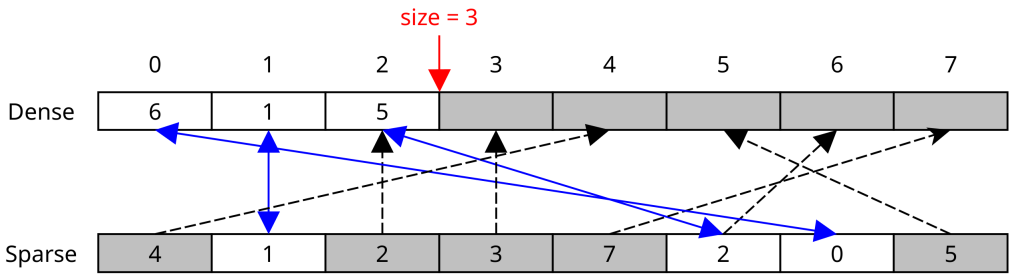

Figure 6.3: A sparse set containing elements with and without membership in the set.

6.5 Helper Functions

A handful of further functions were added to the SparseSet class and include getter for the size and the capacity, as well as iterators to conveniently iterate through the dense array. The sparse set can be cleared with a single assignment and without the need to invalidate all elements. An additional getter for the dense array index of a specific element was added and is used in the memory management of the ECS (see chapter 7).

```
1  unsigned int capacity() {
2    return m_capacity;
3  }
4
5  void clear() {
6    m_size = 0;
7  }
8
9  unsigned int size() {
10   return m_size;
11 }
12
13 const std::vector<unsigned int>::iterator begin() {
14   return m_dense.begin();
15 }
16
17 const std::vector<unsigned int>::iterator end() {
18   return m_dense.begin() + m_size;
19 }
20
21 unsigned int getDenseIndex(unsigned int element) {
22   return m_sparse[element];
23 }
```

Listing 6.6: SparseSet helper functions.

6.6 Conclusion

This chapter introduced a compact sparse set implementation. Its efficient handling of integer numbers is the basis for the upcoming implementation of an Entity Component System, where it is used to manage the integer valued entities and to align the component memory in the same dense way (see chapter 7).

7 Entity Component System

The *Entity Component System (ECS)* is a design pattern used in software architectures and is well known in the field of video game development. It follows the data-oriented design philosophy to achieve high performance when rendering complex systems and worlds within video games. This chapter starts with brief introductions of the traditional *object-oriented* and *data-oriented* design approaches, followed by the concepts behind ECS. After that, an ECS implementation and its individual parts are shown. It makes use of the formerly introduced sparse set (see chapter 6) and is intentionally kept minimal to stay within the scope of this work. This means that there will only be a reduced feature set without extra functionalities expected from a video game engine, like event and input handling or graphics rendering.

7.1 Object-Oriented Design

In *Object-Oriented Design (OOD)* objects are used to describe and solve problems. Classes represent the objects of the real world, with variables defining their attributes and functions their behaviors. A core concept in OOD is inheritance. Starting from a base class (*vehicle*), specified versions are derived (*car*, *truck*, *motorcycle*) and extend the base class with new attributes and different behaviors. Polymorphism ensures that functions with the same name fulfill the same expected tasks in a specified form in all derived classes. Since a class encapsulates both, data and logic, it is a compact way to abstract problems from the real world and express them within code. This makes it easier to formulate comprehensible solutions and intentions to other developers and the compiler. Knowledge about the machine code or underlying hardware is not strictly necessary, which facilitates the creation of programs using *Object-Oriented Programming (OOP)*.

7.2 Data-Oriented Design

In contrast to the Object-Oriented Design, the *Data-Oriented Design (DOD)* tries to break up the ties between function code and data [52]. Data is treated separately and more carefully. The data flow has to be considered and how the data will be processed, if it needs to be writable or even immutable. Information is represented by simple values, which means that data can be stored as *Plain Old Data (POD)* in generic data structures. By sorting the elements of a structure in descending order of their size, padding bytes used for memory alignment can be omitted and memory usage improved. Additionally, these structures are organized in large blocks of contiguous memory, which can easily be processed sequentially. Instead of packing all information in one object, maybe even reference other objects, DOD uses small POD structures and compose multiple of them together. The relationship can be described as *is-a* (object) in case of OOD and *has-a* (component) in DOD [16].
A homogeneous and continuous memory alignment has another benefit called *spatial locality*. This means the data lies next to each other and can be prefetched without any interruptions by the CPU. It also favors *temporal locality*, where the CPU can finish calculations in one go without flushing intermediate results, because a variable is missing or the cache is full. This may lead to improved performance and less cache misses with no extra optimizations, just by choosing a different data layout. This is especially interesting because CPUs have developed faster in higher clock speeds than in increased memory access. Reaching the boundaries of multi-level CPU caches (L1, L2 or L3) results in noticeable performance dips and switching to *RAM* is even worse [13].
After creating the data structures, the functions can be implemented. By design, these are minimal systems and execute only small and stateless tasks to reduce the transformations applied to the data. Once again, the CPU cache is ideally used, because it operates with one function in the instruction cache on many data in a row. Such small systems are also well suited to be parallelized. Furthermore, the separation between data and behavior offers great modularity. Tests, modifications and optimizations can easily be applied to individual systems or single data, without affecting the other parts.

7.3 ECS Architecture

The next sections feature a stepwise explanation and realization of the individual parts of the ECS architecture. As mentioned before, only a basic set of features is implemented to cover the current needs. Event handling or graphics rendering is not build-in yet, but modularity makes it easy to implement them in the future.
After the analysis of WONDER showed that a reimplementation is necessary, examining the data and its flows indicated some possible optimizations by putting the data closer together (see chapter 4). This already hinted in the direction of a data-oriented design. Further research on this topic led to the architecturally fitting approach of ECS. The data is kept in *components*, which are aligned in tight blocks of pre-allocated memory. This stabilizes performance by preventing memory allocations during runtime, at the cost of a higher memory usage. The behavior is handled separately from the data within modular and simple kept *systems*. Objects are created by the composition of components and identified using *entities*. Although coming from the field of game development, the concept is universally applicable, as long as it fits the problem. Strictly speaking, WONDER may be considered as a basic game engine, which is reduced to its audio and networking parts and uses the JACK audio callback to drive a game loop, where all the WFS computations are processed in real-time.
Besides books [52] [16], the research on the topics of *Data-Oriented Programming (DOP)* and ECS included a variety of helpful sources. Some insightful conference talks and presentations can be found here: [1] [35] [48] [29] [66] [43]. An excerpt of numerous blogs by indie game developers is listed here: [32] [9] [36] [12] [7] [42] [67]. Furthermore, various ECS libraries and implementations were investigated and compared:

- **EnTT** (C++) [10]
- **EntityX** (C++) [65]
- **anax** (C++) [34]
- **Artemis-odb** (Java) [39]

Overall, the basic message is very similar. The concept of Data-Oriented Design takes up a key role everywhere, while ECS can be considered as just another viable solution for a specific problem, here mostly the development of a performant game engine.

7.4 Entity

The entity is a key to identify an object. It is as simple as an integer value. To identify the components that belong to an entity, a bit-mask can be attached to it. The bit-width corresponds to the amount of available component types. In table form, every column represents an entity bit-mask and marks its separate parts (see Table 7.1). This kind of *object creation through composition* helps to prevent the known diamond problem, which occurs in OOP when multiple inheritances are used to create a new specified object [53] (see Figure 7.1). In this example, the class *Platypus* needs to be a mammal, but it also lays eggs. The inherited function *eat()* from both superior classes cause an ambiguity problem. Using an entity instead, every available type of component can be added to create an individual object without any dependencies (see Figure 7.2).

	0	1	2	...	n-1
Component A	×		×		×
Component B	×	×	×		
Component C		×	×		×
Component D	×		×		

Table 7.1: Object composition using the entity as index into the component arrays.

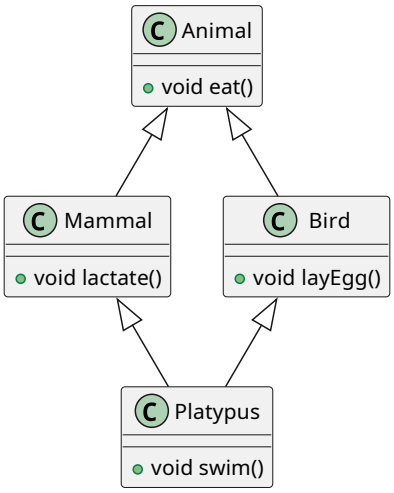

Figure 7.1: Diamond problem when using multiple inheritance.

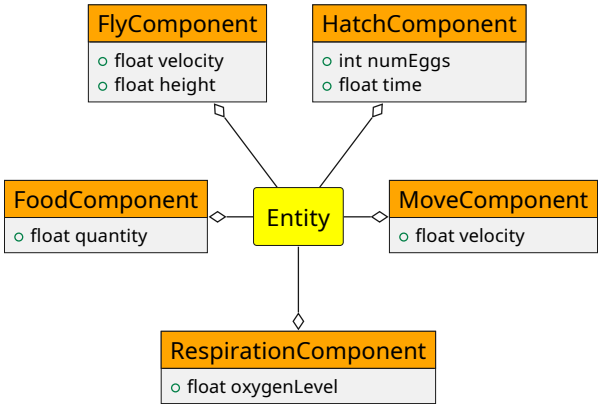

Figure 7.2: Object creation through component composition.

7.5 Component

A component is a generic data structure (POD) and consists of simple data types without any behavior or functions. In C++ it is best described by a *struct*, which has only public members and allows direct access (see Listing 7.1). It is stored in dense, contiguous and pre-allocated blocks in memory. Contrary to OOD, the objects in DOD are aligned in *Structure of Arrays (SoA)* and not in *Array of Structures (AoS)* (see Figure 7.3). This improves iteration speed, CPU cache usage, prefetching and allows easier vectorization and parallelization. Calculating the 3D position (`float, float float`) of an object in OOP, would require to load the whole object into the cache, even though the 12 bytes for the position would be sufficient. The depicted white spaces are other parts of the object, which takes up 40 bytes of memory. Besides the position, valuable cache memory is wasted and triggers cache misses whenever the 64 bytes of a L1 CPU cache line are used up. Here, already after two position retrievals. In DOP, a total of five positions fit into the same cache line and the CPU is able to easily prefetch the subsequent rest.

```
1 struct Animation {
2   Vector3D position;
3   Vector3D rotation;
4   Vector3D scale;
5   Vector3D rgbColor;
6 };
```

Listing 7.1: Typical component body with simple data types and no behavior.

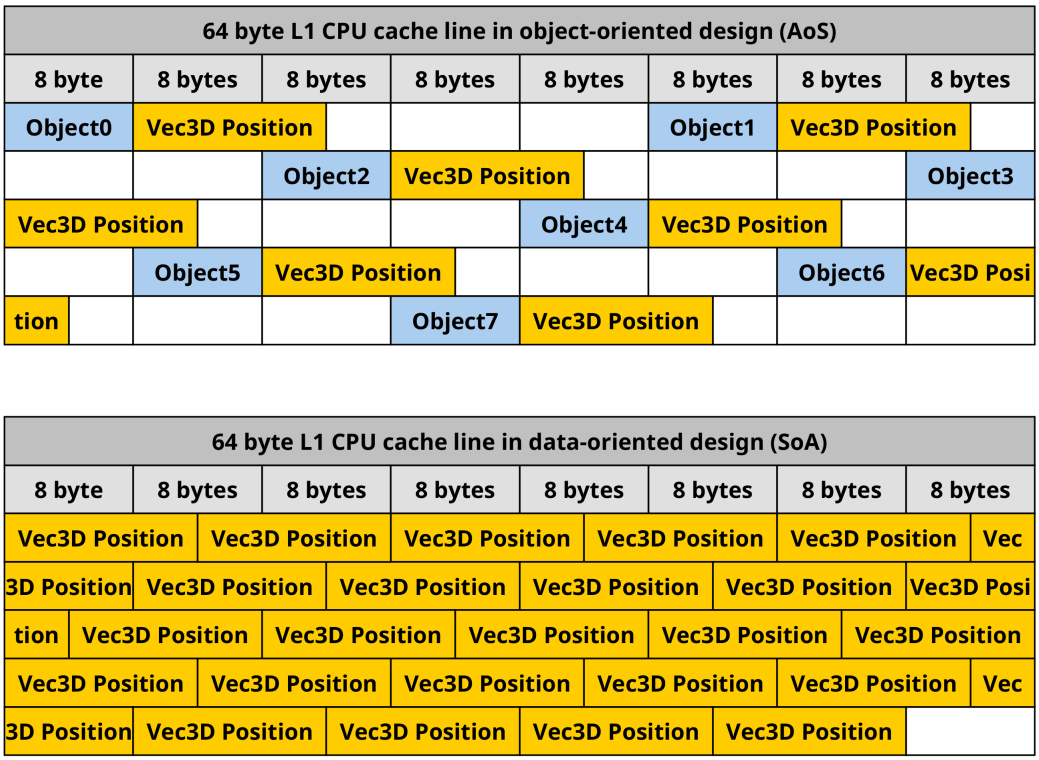

Figure 7.3: Exemplary memory layout between OOD (AoS) and DOD (SoA).

7.6 System

A system iterates over components and contains the logic of the program. The operations are kept basic for better understanding and to facilitate parallelization. By performing manipulations on the data, the components change their state and the program gets behavior. The computations can be triggered by a timer in a game loop, external events or a callback mechanism. Just like components, systems are CPU cache friendly by being loaded once into the instruction cache and then applied on many components in a row. This also helps with serialization, since the component data is already available as a contiguous block of memory. As seen with entities, a bit-mask can be used to filter a set of entities with certain components required for the logic of a system. This way, entities with different compositions but one common component can still be used by the same system. For instance, different entities can be moved by a movement system, as long as they posses a position component (see Figure 7.4).

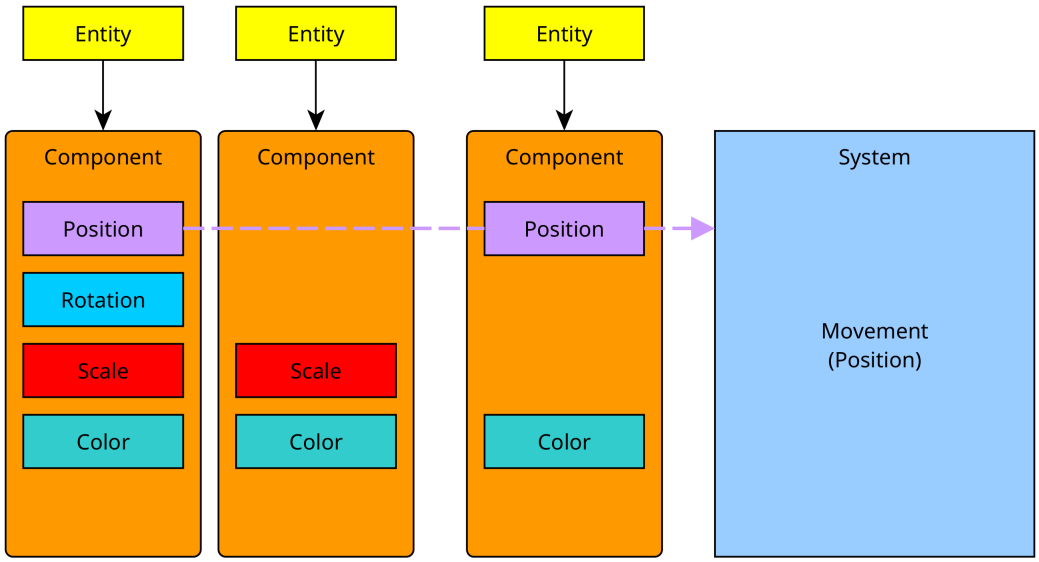

Figure 7.4: A system for entity movement using only components of type *Position*.

7.7 Types Header File

Before introducing the classes of the ECS, a naming schema is introduced in the header file *Types.hpp* to conveniently access and identify entities, components, systems and the component bit-masks. Also, fixed maximum numbers are used throughout the program to pre-allocate memory at the start and avoid further allocations during runtime.

```
1 using EntityID                    = unsigned int;
2 const EntityID MAX_ENTITIES       = 1024;
3
4 using ComponentID                 = unsigned int;
5 const ComponentID MAX_COMPONENTS = 32;
6 using ComponentMask               = std::bitset<MAX_COMPONENTS>;
7
8 using SystemID                    = unsigned int;
9 const SystemID MAX_SYSTEMS        = 32;
```

Listing 7.2: Type alias and maximums used for memory pre-allocations in the ECS.

7.8 EntityManager Class

The EntityManager is used to manage entities and their component bit-mask. Even though an entity just represents an integer number, it needs some logic to take care of creating and destroying entities. A *First In First Out (FIFO)* queue is being utilized to reuse entities, while a sparse set helps to avoid duplicates. For readability, a recurring `assert()` call is omitted in the upcoming functions (see Listing 7.4).

```
1  class EntityManager {
2   public:
3    EntityManager() {
4      // Initialize the queue with all possible EntityIDs.
5      for (EntityID index = 0; index < MAX_ENTITIES; index++) {
6        m_availableEntityIDs.push(index);
7      }
8
9      // Allocate vector memory by resizing it to its final size.
10     m_componentMasks.resize(MAX_ENTITIES);
11   }
12
13  private:
14   // A FIFO queue, which holds all available EntityIDs.
15   std::queue<EntityID> m_availableEntityIDs;
16
17   // A vector with component masks for EntityID.
18   std::vector<ComponentMask> m_componentMasks;
19
20   // A SparseSet to avoid creating / destroying an EntityID multiple times.
21   SparseSet<EntityID> m_entitySet{MAX_ENTITIES};
22
23   // The number of currently used entities.
24   std::size_t m_numActiveEntities{0};
25 };
```

Listing 7.3: Initial EntityManager class to manage entities and their bit-masks.

```
1  assert((index < MAX_ENTITIES) && "The EntityID is out of bounds.");
```

Listing 7.4: Assert functionality implemented in the EntityManager functions.

7.8.1 Create Entity Function

The create function returns an available entity from the queue and also stores it in the sparse set to check for it later during removal.

```
EntityID createEntity() {
  // Remove an available EntityID from the front of the queue.
  if (!m_availableEntityIDs.empty()) {
    EntityID index = m_availableEntityIDs.front();
    m_availableEntityIDs.pop();

    // Also insert the retrieved EntityID into the SparseSet.
    if (m_entitySet.insert(index)) {
      m_numActiveEntities++;
      return index;
    }
  }

  // Return an invalid value if no EntityIDs are left.
  return MAX_ENTITIES;
}
```

Listing 7.5: EntityManager *createEntity* function.

7.8.2 Destroy Entity Function

The destroy function uses the sparse set to remove only valid entities and puts them for reuse back into the queue. Also, the component bit-mask is reset for reuse.

```
void destroyEntity(EntityID index) {
  // Check if the given EntityID exists in the SparseSet before removal.
  if (m_entitySet.remove(index)) {
    // Reuse the destroyed EntityID by storing it at the back of the queue.
    m_availableEntityIDs.push(index);

    // Reset the component mask of the destroyed EntityID.
    m_componentMasks[index].reset();

    m_numActiveEntities--;
  }
}
```

Listing 7.6: EntityManager *destroyEntity* function.

7.8.3 Helper Functions

Helper functions for the component mask of an entity are available, as well as a function to retrieve the number of active entities to stay within the predefined ECS boundaries.

```
void setEntityComponentMask(EntityID index, ComponentMask mask) {
  // Set the component mask for the given EntityID.
  m_componentMasks[index] = mask;
}

ComponentMask getEntityComponentMask(EntityID index) {
  // Return the component mask for the given EntityID.
  return m_componentMasks[index];
}

std::size_t getNumActiveEntities() {
  return m_numActiveEntities;
}
```

Listing 7.7: EntityManager helper functions.

7.9 ComponentPool Class

The ComponentPool class is a generic buffer to store a predefined number of components inside a memory pool. Without having knowledge about the actually used component type and size, the internal buffer is designed around the idea to just allocate enough memory. Using type *char*, which is exactly one byte in size, the buffer can correctly be instantiated with a `sizeof()` call by the caller, who knows the actual size of the type (see Listing 7.8). This info is then used by the constructor to allocate a memory block for the maximum number of entities, i.e. one component per entity. A destructor frees the memory on destruction. Since the type is still unknown, the getter for a component uses a *void* pointer to return the starting address for the component of the specified entity. Again, the caller knows the actual type and can apply a static cast to ensure a correct memory access. For readability, some `assert()` calls were omitted in the code listing.

```
1  class ComponentPool {
2   public:
3    ComponentPool(std::size_t componentSize):m_componentSize{componentSize} {
4      m_componentBuffer = new char[m_componentSize * MAX_ENTITIES];
5    }
6
7    ~ComponentPool() {
8      if (m_componentBuffer != nullptr) {
9        delete[] m_componentBuffer;
10       m_componentBuffer = nullptr;
11     }
12   }
13
14   inline void* getComponent(EntityID index) {
15     return (m_componentBuffer + (index * m_componentSize));
16   }
17
18   std::size_t getComponentSize() {
19     return m_componentSize;
20   }
21
22  private:
23   // Type char (1 byte in size) is used as a generic memory placeholder.
24   char* m_componentBuffer{nullptr};
25
26   // The size of a single component in bytes.
27   const std::size_t m_componentSize;
28 };
```

Listing 7.8: ComponentPool class to store components in a memory pool.

7.10 ComponentManager Class

The component pools, each for every type, are managed by the ComponentManager class. A sparse set for each component type ensures the same dense memory layout as already used for entities. Templates help to instantiate different types. A static *ComponentID*, generated for each type, serves as an index to access an individual component pool.

```
class ComponentManager {
 public:
  ComponentManager() {
    // Allocate vector memory by resizing it to its final size.
    m_componentPools.resize(MAX_COMPONENTS, nullptr);
    m_entitySets.resize(MAX_COMPONENTS, MAX_ENTITIES);
  }

  ~ComponentManager() {
    for (std::size_t i = 0; i < m_componentPools.size(); i++) {
      if (m_componentPools[i] != nullptr) {
        delete m_componentPools[i];
        m_componentPools[i] = nullptr;
      }
    }
  }

  // Generate / Return a static and unique ComponentID for each type.
  template<typename T>
  inline ComponentID getComponentID() {
    static ComponentID s_componentID = m_numComponents++;
    return s_componentID;
  }

 private:
  // ComponentPool pointers, each for a different component type.
  std::vector<ComponentPool*> m_componentPools;

  // A SparseSet for each component pool ensures a dense memory layout.
  std::vector<SparseSet<EntityID>> m_entitySets;

  // A counter to generate an unique ComponentID for each component type.
  ComponentID m_numComponents{0};
};
```

Listing 7.9: Initial ComponentManager class to manage component pools.

7.10.1 Assign Component Function

The function to assign a component to an entity first retrieves the *ComponentID* for this particular component type, deduced from the template type *T*. In case the component type is new, a component pool is allocated for it. Next, the logic of a sparse set is applied to the assignment of the component. By successfully inserting the given entity index into the sparse set of a component type, a dense layout of entities is ensured. Now, the dense index of the given entity can be used to create the new component at the same spot in the component pool to mimic the dense layout of the entities in the sparse set. The address at the dense index in the component pool is retrieved and a new component is created at the same spot with the help of the *placement new* operator [45].

```
1  template<typename T>
2  T* assignComponent(EntityID index) {
3    // Generate / Retrieve the unique ComponentID as pool index.
4    ComponentID compIdx = getComponentID<T>();
5
6    // Check if component is a new type and allocate a memory pool.
7    if (m_componentPools[compIdx] == nullptr) {
8      m_componentPools[compIdx] = new ComponentPool(sizeof(T));
9    }
10
11   if (m_entitySets[compIdx].insert(index)) {
12     // Insertion succeeded, EntityID is valid and in the SparseSet.
13     // Component can be assigned to EntityID index.
14     // Apply logic of a sparse set to organize components in the pool.
15     // Use placement new to create a new component at the dense index.
16     EntityID denseIdx = m_entitySets[compIdx].getDenseIndex(index);
17     T* cPtr = new (m_componentPools[compIdx]->getComponent(denseIdx)) T();
18
19     // Return the address to the created component for the given EntityID.
20     return cPtr;
21   }
22
23   return nullptr;
24 }
```

Listing 7.10: ComponentManager *assignComponent* function.

7.10.2 Remove Component Function

The function to remove a component assigned to an entity first fetches the *ComponentID* for this particular component type, deduced from the template type *T*. This is used as an index into the entity sparse set of this component type. If the entity, from which the component should be removed, can successfully be removed from the sparse set, there is a new dense layout in the sparse set without the removed entity. This dense layout is now applied to the component pool, the same way it is done in the sparse set implementation (see section 6.3). Therefore, the last valid component in the component pool is copied to the same location, where the other component was previously removed. This fills the occurred gap and ensures a dense memory layout in the component pool.

```
1  template<typename T>
2  void removeComponent(EntityID index) {
3    // Generate / Retrieve the unique ComponentID as pool index.
4    ComponentID compIdx = getComponentID<T>();
5
6    if (m_entitySets[compIdx].remove(index)) {
7      // Removal succeeded, EntityID removed from dense in the SparseSet.
8      // Reorganize the component pool layout to match the sparse set.
9      // Use char pointers, since component pool buffer is of type char.
10     EntityID endIdx = m_entitySets[compIdx].size();
11
12     EntityID denseIdx =
13       m_entitySets[compIdx].getDenseIndex(index);
14
15     char* srcBegin =
16       (char*)(m_componentPools[compIdx]->getComponent(endIdx));
17
18     char* srcEnd = srcBegin + sizeof(T);
19
20     char* dstBegin =
21       (char*)(m_componentPools[compIdx]->getComponent(denseIdx));
22
23     // Copy the component from the source to the destination index.
24     std::copy(srcBegin, srcEnd, dstBegin);
25   }
26 }
```

Listing 7.11: ComponentManager *removeComponent* function.

7.10.3 Get Component Function

The function to get a component for a given entity from the component pool first retrieves the *ComponentID* for this particular component type, deduced from the template type *T*. Then the sparse set for this component type is used to retrieve the dense index for the given entity. This can be used to access the component cool of this type to get a pointer for the address of the desired component. Before returning the pointer, it has to be casted to the actual component type *T*.

```
template<typename T>
T* getComponent(EntityID index) {
  // Generate / Retrieve the unique ComponentID as pool index.
  ComponentID compIdx = getComponentID<T>();

  // Get the pointer to the component and cast it to the actual type.
  EntityID denseIndex = m_entitySets[compIdx].getDenseIndex(index);
  T* cPtr = (T*)(m_componentPools[compIdx]->getComponent(denseIndex));

  // Return the address to the requested component for the given EntityID.
  return cPtr;
}
```

Listing 7.12: ComponentManager *getComponent* function.

7.10.4 Destroy Entity Function

In case an entity is destroyed globally, all assigned component types from all component pools have to be removed accordingly. To delete all components of a destroyed entity, the same logic has to be applied as in the remove component function (see subsection 7.10.2). The only difference is that there is no template and type deduction, but instead all valid component pools have to be iterated and the affected components consecutively removed.

```
void destroyEntity(EntityID index) {
  // Remove the components of a globally destroyed EntityID.
  for (std::size_t i = 0; i < m_componentPools.size(); i++) {
    if (m_componentPools[i] != nullptr) {
      if (m_entitySets[i].remove(index)) {
        // Removal succeeded, EntityID removed from dense in the SparseSet.
        // Reorganize the component pool layout to match the sparse set.
        // Use char pointers, since component pool buffer is of type char.
        EntityID endIndex = m_entitySets[i].size();
        EntityID denseIndex = m_entitySets[i].getDenseIndex(index);

        char* srcBegin =
          (char*)(m_componentPools[i]->getComponent(endIndex));

        char* srcEnd = srcBegin + m_componentPools[i]->getComponentSize();

        char* dstBegin =
          (char*)(m_componentPools[i]->getComponent(denseIndex));

        // Copy the component from the source to the destination index.
        std::copy(srcBegin, srcEnd, dstBegin);
      }
    }
  }
}
```

Listing 7.13: ComponentManager *destroyEntity* function.

7.11 System Class

The System class is a base class for logic implementations. The destructor is intentionally *virtual*, so derived classes have to clean up for themselves. A sparse set is inherited and contains all filtered entities the system is interested in.

```
class System {
 public:
  virtual ~System() = default;

  SparseSet<EntityID> m_entitySet{MAX_ENTITIES};
};
```

Listing 7.14: System base class to add behavior to the ECS.

7.12 SystemManager Class

The SystemManager handles all classes derived from the System base class and allows to access them by pointers of the base class. Templates are used to manage different system types. A static *SystemID* for each type is used as an index into the vectors of System pointers and component bit-masks. The base class provides a sparse set for entities.

```
class SystemManager {
 public:
  SystemManager() {
    // Reserve vector memory by resizing it to its final size.
    m_systems.resize(MAX_SYSTEMS, nullptr);
    m_componentMasks.resize(MAX_SYSTEMS);
  }

  ~SystemManager() {
    for (std::size_t i = 0; i < m_systems.size(); i++) {
      if (m_systems[i] != nullptr) {
        delete m_systems[i];
        m_systems[i] = nullptr;
      }
    }
  }

  // Generate / Retrieve the unique SystemID as vector index.
  template<typename T>
  inline SystemID getSystemID() {
    static SystemID s_systemID = m_numSystems++;
    return s_systemID;
  }

 private:
  // Pointers to System objects, each for a different system type.
  std::vector<System*> m_systems;

  // A vector with component masks for each SystemID.
  std::vector<ComponentMask> m_componentMasks;

  // A counter to generate an unique SystemID for each system type.
  SystemID m_numSystems{0};
};
```

Listing 7.15: Initial SystemManager class to manage systems.

7.12.1 Assign System Function

A new system is assigned by generating an unique *SystemID* for the new type and calling the constructor of the type. In case the system is already assigned, the return statement returns a casted pointer to the already available system, by using the static *SystemID*.

```
template<typename T>
T* assignSystem() {
  // Generate / Retrieve the unique SystemID as index into the vectors.
  SystemID systemID = getSystemID<T>();

  // A nullptr indicates a new system type, which has to be initialized.
  if (m_systems[systemID] == nullptr) {
    m_systems[systemID] = new T();
  }

  // Return a pointer to the system and cast it to the actual class T.
  return static_cast<T*>(m_systems[systemID]);
}
```

Listing 7.16: SystemManager *assignSystem* function.

7.12.2 Set System Mask Function

A component mask for a particular system type is set by first retrieving the *SystemID* of the type and than using it as an index into the vector of component masks to store the new bit-mask. This enables to filter the components, which are needed for the logic of the system. The sparse set of each system will only have entities with the required component combinations and ensures a fast iteration process.

```
template<typename T>
void setSystemComponentMask(ComponentMask mask) {
  // Generate / Retrieve the unique SystemID as index into the vectors.
  SystemID systemID = getSystemID<T>();

  // Set the mask for components this system is interested in.
  m_componentMasks[systemID] = mask;
}
```

Listing 7.17: SystemManager *setSystemComponentMask* function.

7.12.3 Update Entity Mask Function

After assigning a component to or removing it from an entity, the component bit-mask of the entity changes. Therefore, the component bit-mask of all systems has to be adjusted. This ensures, that only the smallest common set of entities is used during iterations. A logical *AND* operation is utilized to merge old and new component bit-masks.

```
void updateEntityComponentMask(EntityID index, ComponentMask mask) {
  ComponentMask systemMask;

  for (std::size_t i = 0; i < m_componentMasks.size(); i++) {
    systemMask = m_componentMasks[i];

    if ((mask & systemMask) == systemMask) {
      if (m_systems[i] != nullptr) {
        // Both component masks match, add the EntityID to the SparseSet.
        m_systems[i]->m_entitySet.insert(index);
      }
    } else {
      if (m_systems[i] != nullptr) {
        // The component masks do not match, remove the EntityID.
        m_systems[i]->m_entitySet.remove(index);
      }
    }
  }
}
```

Listing 7.18: SystemManager *updateEntityComponentMask* function.

7.12.4 Destroy Entity Function

If an entity is globally destroyed, it also has to be removed from all systems.

```
void destroyEntity(EntityID index) {
  // Remove a globally destroyed EntityID from all systems.
  for (std::size_t i = 0; i < m_systems.size(); i++) {
    if (m_systems[i] != nullptr) {
      m_systems[i]->m_entitySet.remove(index);
    }
  }
}
```

Listing 7.19: SystemManager *destroyEntity* function.

7.13 Scene

The Scene class represents a global coordinator for all other managers. It constructs and destructs them and passes all the functionality of the ECS to the functions of the respective managers. Additionally, the scene is handled as a global object in the *main* function. Other classes, for example all derived System classes that need access to the entities and components, include the global scene as an external variable. A public status variable (`isValid`) helps to gracefully shutdown the main program if something went wrong. The individual functions for entities, components and systems are shown in the next subsections and an overview is depicted in Figure 7.5.

```
1  class Scene {
2   public:
3    Scene() {
4      m_entityManager    = new EntityManager();
5      m_componentManager = new ComponentManager();
6      m_systemManager    = new SystemManager();
7    }
8
9    ~Scene() {
10     if (m_entityManager != nullptr) {
11       delete m_entityManager;
12       m_entityManager = nullptr;
13     }
14
15     if (m_componentManager != nullptr) {
16       delete m_componentManager;
17       m_componentManager = nullptr;
18     }
19
20     if (m_systemManager != nullptr) {
21       delete m_systemManager;
22       m_systemManager = nullptr;
23     }
24   }
25
26   // Public status of the scene.
27   bool isValid = true;
28
29  private:
30   // ECS managers.
31   EntityManager* m_entityManager;
32   ComponentManager* m_componentManager;
33   SystemManager* m_systemManager;
34 };
```

Listing 7.20: Initial Scene class to coordinate all other managers.

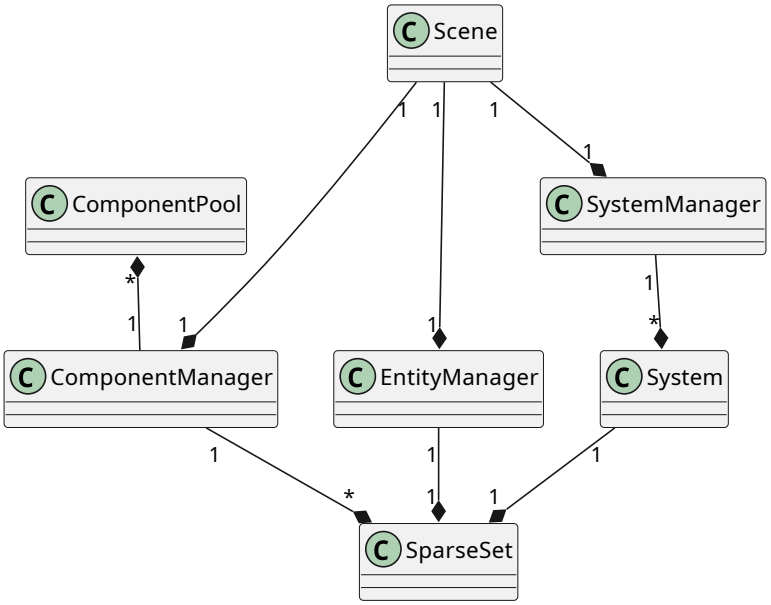

Figure 7.5: Overview of the Entity Component System classes.

7.13.1 Entity Functions

The entity functions of the Scene class include the creation of an entity and a possibility to retrieve the active entities. This is important if multiple entities are created with a loop that needs a stop criterion. The destruction of an entity is passed to all managers.

```
EntityID createEntityID() {
  return m_entityManager->createEntity();
}

void destroyEntityID(EntityID index) {
  m_entityManager->destroyEntity(index);
  m_componentManager->destroyEntity(index);
  m_systemManager->destroyEntity(index);
}

std::size_t getNumActiveEntities() {
  return m_entityManager->getNumActiveEntities();
}
```

Listing 7.21: Entity functions in Scene class.

7.13.2 Component Functions

The most important component functions of the Scene class are the ones for assigning and removing components. Here, the bit-masks have to be handled carefully and passed correctly from the ComponentManager to the managers for entities and systems.

```
template<typename T>
T* assignComponent(EntityID index) {
  T* componentPtr = m_componentManager->assignComponent<T>(index);

  ComponentMask mask = m_entityManager->getEntityComponentMask(index);
  mask.set(m_componentManager->getComponentID<T>(), true);
  m_entityManager->setEntityComponentMask(index, mask);

  m_systemManager->updateEntityComponentMask(index, mask);

  return componentPtr;
}

template<typename T>
void removeComponent(EntityID index) {
  m_componentManager->removeComponent<T>(index);

  ComponentMask mask = m_entityManager->getEntityComponentMask(index);
  mask.set(m_componentManager->getComponentID<T>(), false);
  m_entityManager->setEntityComponentMask(index, mask);

  m_systemManager->updateEntityComponentMask(index, mask);
}

template<typename T>
T* getComponent(EntityID index) {
  return m_componentManager->getComponent<T>(index);
}

template<typename T>
ComponentID getComponentID() {
  return m_componentManager->getComponentID<T>();
}
```

Listing 7.22: Component functions in Scene class.

7.13.3 System Functions

The system functions are for assigning new systems to the manager and to set their component bit-masks.

```
template<typename T>
T* assignSystem() {
  return m_systemManager->assignSystem<T>();
}

template<typename T>
void setSystemComponentMask(ComponentMask mask) {
  m_systemManager->setSystemComponentMask<T>(mask);
}
```

Listing 7.23: System functions in Scene class.

7.14 Conclusion

This chapter presented OOP and DOP and how the latter offers performance benefits, due to a more hardware and data oriented implementation approach. These concepts are also used in the ECS software design pattern, which was introduced thereafter. All individual ECS components were discussed and were followed by an implementation walkthrough. It was kept minimal and simple, showed how to utilize the sparse set for a dense memory layout and how to handle the separate parts within managers. Despite the small feature set, the implementation is fully functional and usable. The good thing is, that the modular structure makes it easy to extend the architecture with additional components and features in the future, like a functioning event handler or a graphics pipeline.

8 CoRGII

CoRGII (***Co****ntrolled* ***R****enderer,* ***G****raphical User Interface and* ***I****mmersive* ***I****nteraction*) is a project that started as an attempt to rework and improve the WFS related programs of the WONDER software suite, which are used in the I²AudioLab of the HAW Hamburg. WONDER consists of multiple programs, all designed to run on multiple computers in a local network to distribute the WFS computation workload. This works reasonably well in the I²AudioLab, but showed several anomalies over time. Since the code was basically unmaintained for over a decade and suffered from software erosion, the first improvement attempts focused on overhauling WONDER's build process, in order to use the latest versions of compilers and libraries. It helped to gain knowledge about the inner workings, but it also revealed some bugs, inconsistencies and partly finished code. This was the moment when the idea of CoRGII was conceived. The plan was to extract the WFS components, which were the only ones in use anyway, and to combine them into one single program. This would simplify maintenance and omit the process to ensure compatibility with the unused programs of the WONDER suite. Before starting the venture, the performance of the existing network and audio rendering components was measured to serve as a benchmark for future comparisons. However, this revealed more problems with WONDER's current software architecture (see chapter 4). CoRGII's task now shifted from uniting and improving WONDER's WFS components to an entire reimplementation process, since a redesign of the former architecture became necessary and inevitable. With performance in mind, a data-oriented design approach was chosen to introduce a more stringent separation of data and algorithms. At the same time, the handling of objects should be more expressive than just using POD structures, which is usually the case in data-oriented programming. The ECS software design pattern fits these requirements and forms the basis of CoRGII's new architecture (see chapter 7). CoRGII is currently still a work in progress, so this chapter will first start off by introducing the concepts behind the new components. This is followed by their partial implementation and development status. Finally, the remaining problems are discussed.

8.1 Architectural Structure

The naming of CoRGII is an acronym for its integrated functionalities. They combine basic requirements for the WFS setup in the I²AudioLab and adopted features from the WONDER software suite. Together they represent a network controlled audio renderer for interactive and real-time applications, written in C++. Unlike WONDER, CoRGII is developed as a single program with clearly separated and modular components, which are interconnected using the global *Scene* object and its underlying ECS as an intermediary. The concept behind each individual part of the software is described in this section, while the actual program code is presented in the upcoming sections. An overview of the new program structure is shown in Figure 8.1.

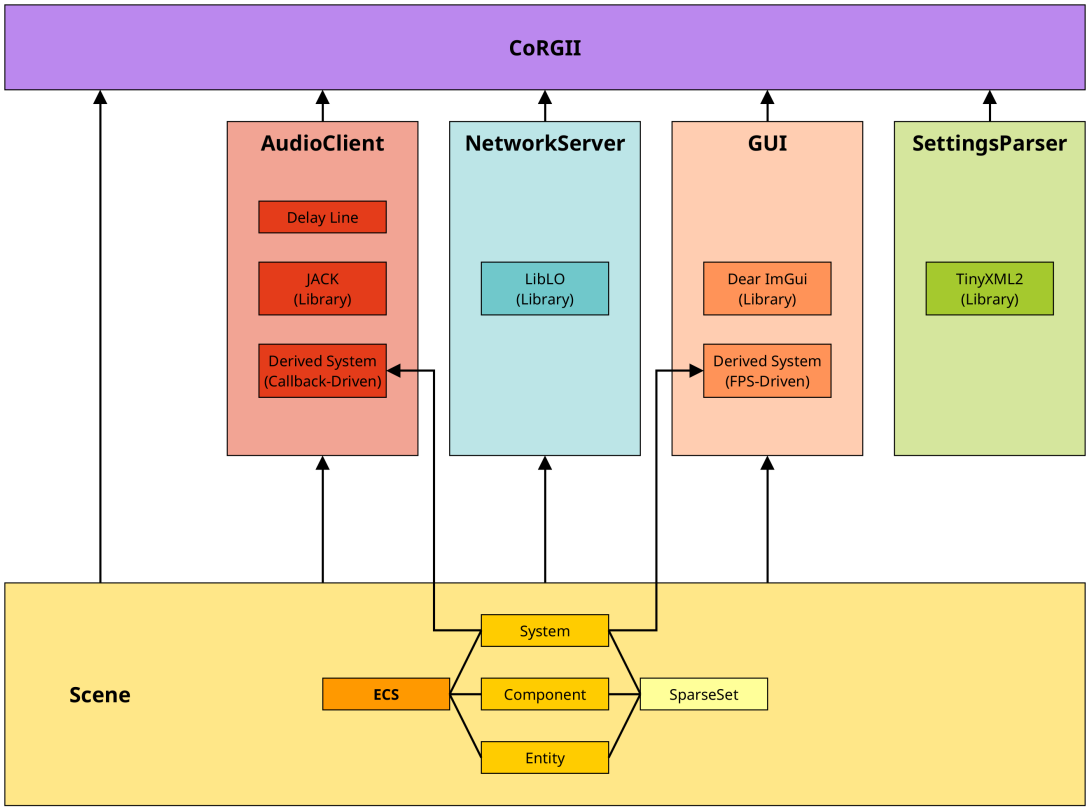

Figure 8.1: Overview of the CoRGII program and its components.

8.1.1 Audio Client

The basic structure of the audio client remains the same as in tWONDER. As before, the JACK client library is implemented to access the locally running, low-latency JACK audio server, which provides access to the sound card and its input and output ports. By registering a callback function, the client is called by a real-time thread of the JACK server, whenever there is audio to process. From the outside, CoRGII's audio inputs represent the number of virtual WFS sources and are externally supplied with sound, for example from a DAW on another computer. The audio outputs represent the number of loudspeakers connected to the sound card. JACK also has the benefit of virtual input and output ports. This is a convenient way to run simulations and process far more input and output buffers than physically offered by the sound card.
Compared with WONDER, CoRGII's delay line implementation receives a rework. Since chapter 4 revealed inefficient read and write functions, they are replaced by simpler and faster versions that were introduced in chapter 5. Another change affects the real-time command engine and its feature to process OSC commands with a duration parameter. It showed severe reliability issues due to its synchronization and therefore is removed completely. A comparable functionality can be achieved by using OSC *timetags*, scripts or automation tracks in a DAW.
The JACK callback process serves as an access point to the audio buffers and as a loop that is constantly updating the systems of the ECS. Rather than iterating over objects and calling their member functions, CoRGII uses derived systems from the ECS to access entities and modify the components assigned to them. To filter the individual entities required by a system, a component bit-mask is applied. This practice of granular systems instead of big member functions requires a break down of the program logic into smaller pieces, but it also facilitates parallelization. However, handling multiple, daisy-chained systems should be treated carefully, in case the order of execution is important. At this point, the audio client exhibits the most noticeable shift in the programming paradigm. Relying on pre-defined object functions is now replaced by self-defined systems, which adds an extra step during the initialization of the audio client.

8.1.2 Network Server

The network server in CoRGII inherits multiple features that are present in WONDER's central control unit cWONDER and the audio renderer tWONDER. Both programs implement the LibLO OSC library and have a similar handling of OSC messages, which is why the set of overlapping and useful features is combined. The publish-subscribe architecture integrated in cWONDER is adapted and enables clients in the same network to connect to a central OSC server and automatically get updates when changes happen to the scene. This also ensures backward compatibility with programs developed for WONDER, since they can connect to CoRGII's OSC server in the same way. OSC offers further compatibility by allowing messages received on different OSC paths to trigger the same function. As a result, CoRGII can simultaneously listen for messages on old WONDER OSC paths and redirect them to call its own equivalent functions.
A new addition to CoRGII is a multicast support, which shifts the distribution workload from the publish-subscribe server to the network hardware itself and keeps all members of the multicast group up-to-date about changes in the scene. The network behavior of the OSC server can be changed via command line arguments.
Besides loudspeakers, the WFS setup in the I²AudioLab consists of a tracking system that tracks targets inside the listening area. To represent trackable objects, additional message paths for a *tracker* has been added to CoRGII's OSC server. A *listener*, as a special case of a trackable object, was added as well, since WFS calculations depend on it as a reference point. Both trackable objects also improve CoRGII's suitability for future use cases involving AR, MR and VR. The *Point-Of-View (POV)* or direction of a trackable object is currently realized with angles in degree for azimuth and elevation. This simplified approach might be replaced or extended by a quaternion representation in upcoming iterations of the implementation, since it is very common in computer graphics and a safer way to create object rotations without causing a gimbal lock. Regarding the tracking system installed in the I²AudioLab, an extra parser program could be beneficial. At the moment, the tracking software only distributes Euler angles as matrices of strings into a multicast group. Converting these strings to quaternions and resending them as OSC messages, would lead to a tighter integration with CoRGII.

8.1.3 Settings Parser

The settings parser unites the configurations of all components. Former settings from cWONDER and especially tWONDER, with all of its loudspeaker configuration files, are now combined into a single settings file. The format is still based on Extensible Markup Language (XML) but the parsing library was changed from libxml++2.6 to *TinyXML-2*[1]. Its feature set is smaller but it has no dependencies with other libraries and only consists of two files.
The layout and attributes of the settings file are extended to reflect the changes in CoRGII. Each component now has its own set of attributes. A GUI may use the room dimensions and the refresh rate to render the scene. The network server attributes are extended with multicast and network protocol support. Trackers and a listener are added as trackable objects. The number of JACK audio ports and some WFS parameters are directly configurable. Sound sources are now also part of the settings file and receive a *gain* attribute to better control their volume without being restricted to a simple *on/off* state. The gain attribute was also added to the loudspeaker attributes and can be used to apply a tapering window. This is helpful to reduce the truncation effect known from WFS (see subsection 3.4.2). To simplify managing multiple distributed audio renderers with their own sets of different loudspeakers, the settings file subdivides the lists of loudspeakers by hostnames. While parsing, the settings parser first determines the hostname of the current computer and then only parses the loudspeaker list assigned to its hostname.
Some features that were removed involve the project management in cWONDER. The transmission of the scene to xWONDER, using XML strings packed into OSC messages, was only a temporary solution and never improved. A general function to receive the status of the scene via OSC can act as a replacement. Exporting the scene to a XML file was also removed due to the new attributes schema. Future export functions should include the settings and loudspeaker attributes as well, in case a tapering window was applied to the loudspeaker gains. All attributes of the current settings file are listed in section A.4 and the XML representation can be found in section A.5.

[1] https://github.com/leethomason/tinyxml2

8.1.4 Graphical User Interface

The CoRGII GUI is still in its planning phase, since it has the lowest priority of all components. It will replace xWONDER and the aim is to switch from the Qt GUI framework to *Dear ImGui*[2], an immediate mode GUI. This type of GUI is mostly used in video game engines and complements CoRGII's embedded ECS, which also originates from video game development. The advantage of this combination is the common data storage among all parts of CoRGII. Furthermore, Dear ImGui supports a wide variety of graphics libraries and APIs, which means a lot of freedom, compatibility and platform choice during the development process. Depending on the usage scenarios, it is imaginable to implement virtual reality controls next to a standard desktop integration. Switching between the different graphic modes may be achieved via command line arguments.
Although a final implementation still lies in the future, there is an intermediate solution by using Processing as an external program to control CoRGII and visualize the objects of the scene. It was already used successfully while performing tests and benchmarks on WONDER's delay line and real-time command queue implementation (see chapter 4). Processing is easy to use for rapid prototyping and offers a plugin system to extend its functionalities. This makes it possible to create an immediate GUI with *controlP5* [49] (see Figure 8.2), use the OSC protocol with *oscP5* [50] and parse the settings file with the Java XML support. Basically xWONDER, sparse set and ECS, but all in one IDE.

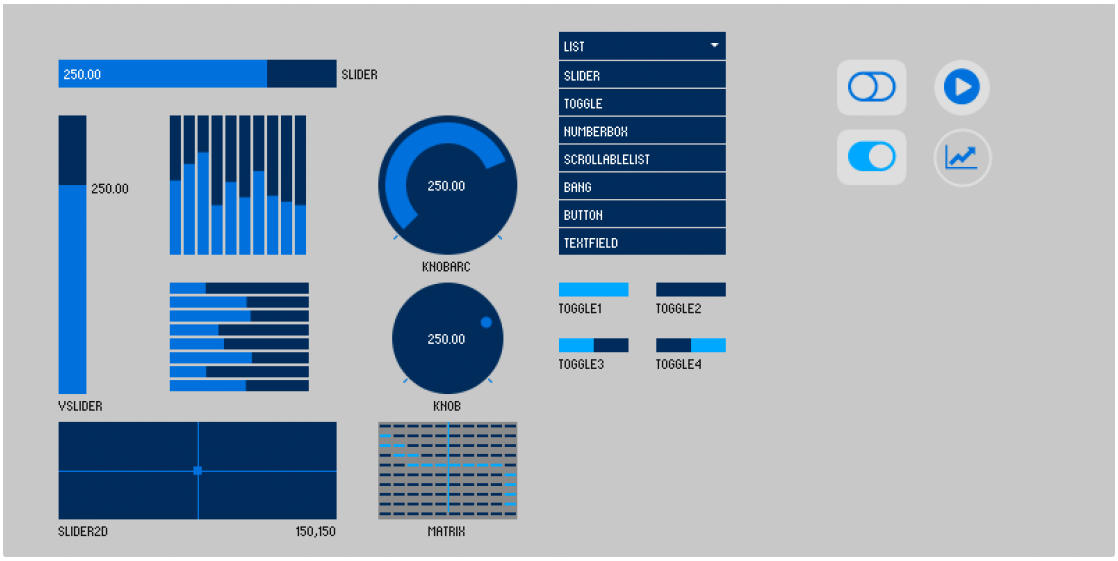

Figure 8.2: Exemplary GUI elements from *controlP5* plugin used in Processing [49].

[2] https://github.com/ocornut/imgui (Last accessed on: March 20, 2023)

8.1.5 Scene

The scene is the foundation of the data management in CoRGII. It incorporates the ECS and sparse set to provide a dense and CPU cache friendly memory layout. A link between the data encapsulated in the scene and the audio, network and GUI parts is established by using derived systems. These systems can be updated by callbacks (JACK), incoming messages (OSC) or a graphics renderer (GUI) to manipulate the data, which consists of entity IDs and components attached to them. Data locking mechanisms are currently omitted, since read and write access happens in distinct systems. Even if intermediate values occur, for example when a source position is changed via OSC message in between a JACK callback, the new values will be taken over in the next callback at the latest.

8.1.6 Build System

The development started out with a simple *Makefile* but has adopted the *CMake*[3] build system over time. Currently, the only supported OS is Linux, but future compatibility with Windows and MacOS is planned. The tool *pkg-config* is utilized to find and include CoRGII's libraries: JACK, LibLO and TinyXML-2. An excerpt of the CMake script file *CMakeLists.txt* can be found in section A.2.

8.1.7 Testing

The individual components were rudimentary tested during their development. However, some tests were not viable due to cache emulation restrictions combined with JACK's real-time callback thread. Therefore, future test scenarios may include additional unit and component testing. The tools used for testing were *perf stat* and *Valgrind*[4].

Obtain cache, cycles, instructions, etc. via perf stat:

```
$ perf stat -d -d -d ./program
```

Valgrind cache simulation:

```
$ valgrind -tool=cachegrind ./program
```

Valgrind memory leak check, e.g. for destructors:

```
$ valgrind -leak-check=full -show-leak-kinds=all -s ./program
```

[3] https://cmake.org (Last accessed on: March 20, 2023)
[4] https://valgrind.org (Last accessed on: March 20, 2023)

8.2 AudioClient Class

The *AudioClient* class is contained in a single header file (*AudioClient.hpp*). For a better readability, recurring `assert()` calls or exception throws are omitted in the upcoming code listings. The initial version is shown in Listing 8.1 and contains the member variables, getter functions and a singleton pattern. This programming pattern deletes the *copy* and *move* constructors, as well as the respective assignment operators, to ensure only a single instance of the audio client exists.

```
1  class AudioClient {
2   public:
3    unsigned int getNumInputs() const { return m_numJackClientInputs; }
4    unsigned int getNumOutputs() const { return m_numJackClientOutputs; }
5    unsigned int getBufferSize() const { return m_bufferSize; }
6    unsigned int getSampleRate() const { return m_sampleRate; }
7    unsigned int getNumCallbacks() const { return m_numCallbacks; }
8
9   private:
10   // Remove copy/move constructor and assignment operator for a singleton.
11   AudioClient(const AudioClient& other)             = delete;
12   AudioClient(AudioClient&& other)                  = delete;
13   AudioClient& operator=(const AudioClient& other) = delete;
14   AudioClient& operator=(AudioClient&& other)       = delete;
15
16   // JACK client ports, buffers and parameters.
17   std::vector<DelayLine*> m_delayLines;
18   std::vector<jack_port_t*> m_jackInputPorts;
19   std::vector<jack_port_t*> m_jackOutputPorts;
20   std::vector<float*> m_inputBuffers;
21   std::vector<float*> m_outputBuffers;
22   jack_client_t* m_jackClient;
23   unsigned int m_numJackClientInputs{0};
24   unsigned int m_numJackClientOutputs{0};
25   unsigned int m_bufferSize{0};
26   unsigned int m_sampleRate{0};
27   unsigned int m_numCallbacks{0};
28 };
```

Listing 8.1: Initial AudioClient class.

The destructor and JACK's callbacks for processing audio and react to a shutdown of the JACK server are listed in Listing 8.2. A static redirection of the C-style JACK functions is needed, because otherwise ISO C++ forbids to register a non-static member function of the AudioClient class as a function for JACK's callback mechanism. Hence, static member functions of the AudioClient class are registered for the JACK callbacks, which redirect the callbacks to the non-static member functions. This is realized by utilizing the available `void` pointer parameter *arg*. It enables to pass anything to JACK's callback functions and then cast it to a specific type, in this case a pointer of type `AudioClient` to access its (non-static) member variables (see line 24 and 28 in Listing 8.2).

```
1  ~AudioClient() {
2    // Disconnect this JACK client from the JACK server and shut it down.
3    if (m_jackClient != nullptr) {
4      jack_client_close(m_jackClient);
5      m_jackClient = nullptr;
6    }
7
8    // Free memory allocated by the delay lines.
9    for (DelayLine* delayLine : m_delayLines) {
10     delete delayLine;
11   }
12 }
13
14 void shutdown() {
15   std::cerr << "The JACK server shut down or removed this client.\n";
16   std::exit(1);
17 }
18
19 int process(const jack_nframes_t nframes) noexcept {
20   // JACK callback function. The code is covered in its own subsection.
21 }
22
23 static void staticJackShutdown(void* arg) {
24   return (static_cast<AudioClient*>(arg)->shutdown());
25 }
26
27 static int staticJackProcess(const jack_nframes_t nframes, void* arg) {
28   return (static_cast<AudioClient*>(arg)->process(nframes));
29 }
```

Listing 8.2: AudioClient *destructor* and JACK's callback functions.

8.2.1 Constructor

The constructor of the AudioClient class is covered separately due to the amount of code. It is more extensive because it consists of a wrapper around JACK's C-style API. Again, some simplifications are applied to restrict the code length and improve readability. Overall, the code for the constructor is divided into ten steps.
The first step is to connect the JACK client with the server. Normally, the JACK server (*jackd* for JACK daemon) is expected to already be running in the background. However, an option is added to avoid this step and automatically start a server if it is not running yet (see line 15 in Listing 8.3). After a connection is established, the second step is to register the (static) callback functions of the client, which should be called by the JACK server. One for a graceful shutdown of the client if the server stops working and one to process the audio samples of a callback frame (see line 38 and 42 in Listing 8.3).
The next two steps (3 and 4) initialize the vectors for JACK's input and output ports and the vectors with the input and output buffers, which hold the audio samples of the sound card during a callback frame (see Listing 8.4).
In the subsequent two steps (5 and 6) the input and output ports are registered at the JACK client. These are needed to connect the input side of the client with the sound card inputs and the output side of the client with the sound card outputs (see Listing 8.4).
Before activating the JACK client in step number 8, the vector with the delay lines is initialized in step number 7. Each audio input gets a delay line and each delay line stands for a virtual WFS sound source (see Listing 8.5). A fixed size of 65536 samples, or roughly 1.3 s of delay using a sample rate of 48 kHz, is chosen for brevity and is normally set via a variable from the settings file.
The activation of the JACK client in step 8 starts the audio processing callback by the JACK server and is required to be able to connect the input and output ports of the client with the input and output ports of the sound card, which is done in the last two steps, number 9 and 10 (see Listing 8.6 and Listing 8.7).
The audio processing function is omitted and showed separately (see subsection 8.2.2). Handling of the derived ECS systems is also postponed (see subsection 8.2.3). Normally, their declaration would be part of the private member variables and the initialization would take place in the constructor or a function that is called inside the constructor. The initialization must always happen before activating the JACK client in step 8, since this starts the callback of the audio processing function, which directly accesses the systems.

```
1  AudioClient(unsigned int numInputs = 2, unsigned int numOutputs = 2,
2              bool startJackServer = false)
3    : m_numJackClientInputs{numInputs}, m_numJackClientOutputs{numOutputs} {
4
5    // Define the name of the JACK client.
6    std::string clientName = "CoRGII";
7
8    // Specify a JACK server name when using multiple JACK servers.
9    std::string serverName = ""; // Default is blank.
10
11   // Adjust the start behaviour of the JACK client.
12   jack_options_t jackOptions;
13
14   if (startJackServer) {
15     // Try to start a JACK server based on the .jackdrc file.
16     jackOptions = JackNullOption;
17   } else {
18     // Connect to an already running JACK server. Default behaviour.
19     jackOptions = JackNoStartServer;
20   }
21
22   // Variable with status of the JACK client after connection attempts.
23   jack_status_t jackStatus;
24
25   // 1. Open a connection to the JACK server.
26   m_jackClient = jack_client_open(clientName.data(),
27                                   jackOptions,
28                                   &jackStatus,
29                                   serverName.data());
30
31   // The connection attempt of the JACK client failed.
32   if (m_jackClient == nullptr) {
33       std::exit(1);
34   }
35
36   // 2. Tell JACK server to call the static shutdown and process functions.
37   //    Pass this AudioManager object (via void pointer) as an argument.
38   jack_on_shutdown(m_jackClient,
39                    staticJackShutdown,
40                    static_cast<void*>(this));
41
42   jack_set_process_callback(m_jackClient,
43                             staticJackProcess,
44                             static_cast<void*>(this));
45
```

Listing 8.3: AudioClient *constructor* 1/5 (lines 1-45).

```
46    // 3. Initialize the (virtual) input and output ports.
47    //    Allocate and resize the vector memory to its final size.
48    m_jackInputPorts = std::vector<jack_port_t*>(m_numJackClientInputs);
49    m_jackInputPorts.resize(m_numJackClientInputs);
50
51    m_jackOutputPorts = std::vector<jack_port_t*>(m_numJackClientOutputs);
52    m_jackOutputPorts.resize(m_numJackClientOutputs);
53
54    // 4. Initialize the audio input and output buffers.
55    //    Allocate and resize the vector memory to its final size.
56    m_inputBuffers = std::vector<float*>(m_numJackClientInputs);
57    m_inputBuffers.resize(m_numJackClientInputs);
58
59    m_outputBuffers = std::vector<float*>(m_numJackClientOutputs);
60    m_outputBuffers.resize(m_numJackClientOutputs);
61
62    // 5. Name and register the input ports of the JACK client.
63    for (unsigned int i = 0; i < m_numJackClientInputs; i++) {
64      // Use JACKs port naming starting with a "1" instead of the typical "0".
65      std::string portName = "Input_" + std::to_string(i + 1);
66      m_jackInputPorts[i]  = jack_port_register(m_jackClient,
67                                                portName.data(),
68                                                JACK_DEFAULT_AUDIO_TYPE,
69                                                JackPortIsInput,
70                                                0);
71    }
72
73    // 6. Name and register the output ports of the JACK client.
74    for (unsigned int i = 0; i < m_numJackClientOutputs; i++) {
75      // Use JACKs port naming starting with a "1" instead of the typical "0".
76      std::string portName = "Output_" + std::to_string(i + 1);
77      m_jackOutputPorts[i] = jack_port_register(m_jackClient,
78                                                portName.data(),
79                                                JACK_DEFAULT_AUDIO_TYPE,
80                                                JackPortIsOutput,
81                                                0);
82    }
83
```

Listing 8.4: AudioClient *constructor* 2/5 (lines 46-83).

```
84    // 7. Initialize the delay lines for all input ports.
85    //    Allocate and resize the vector memory to its final size.
86    m_delayLines = std::vector<DelayLine*>(m_numJackClientInputs, nullptr);
87    m_delayLines.resize(m_numJackClientInputs);
88
89    for (DelayLine* delayLine : m_delayLines) {
90      delayLine = new DelayLine(65536); // Buffer for 2^16 samples.
91    }
92
93    // 8. Activate the JACK client. This starts the real-time callback.
94    //    This is also a prerequisite to be able to connect the ports next.
95    int result = jack_activate(m_jackClient);
96
97    // Result code 0 means success, otherwise a non-zero value is returned.
98    if (result == 0) {
99      std::cout << "The JACK client was activated and is running.\n";
100   } else {
101     std::cerr << "The JACK client could not be activated.\n";
102     std::exit(1);
103   }
104
105   // Retrieve the sample rate and the buffer size of the active JACK client.
106   m_sampleRate = jack_get_sample_rate(m_jackClient);
107   m_bufferSize = jack_get_buffer_size(m_jackClient);
108
```

Listing 8.5: AudioClient *constructor* 3/5 (lines 84-108).

```
109   // 9. Connect the sound card inputs with the JACK client input ports.
110   //    JACK calls them "output ports" (JackPortIsOutput)
111   //    because they are readable by the JACK client.
112   unsigned int numSoundCardInputs = 0;
113   const char** input;
114   const char** soundCardInputs = jack_get_ports(
115                                    m_jackClient,
116                                    nullptr,
117                                    nullptr,
118                                    JackPortIsPhysical | JackPortIsOutput);
119
120   // Count the number of physically available sound card input ports.
121   if (soundCardInputs != nullptr) {
122     for (input = soundCardInputs; *input; input++) {
123       numSoundCardInputs++;
124     }
125   } else {
126     std::cerr << "The sound card has no input ports (capture).\n";
127     std::exit(1);
128   }
129
130   // Iterate over all virtual inputs of the JACK client and
131   // connect them with all available phyical sound card inputs.
132   for (unsigned int i = 0; i < m_numJackClientInputs; i++) {
133     if (jack_port_name(m_jackInputPorts[i]) != nullptr) {
134       if (soundCardInputs[i] != nullptr) {
135         // IMPORTANT: Keep the order of "source" and "destination" in mind.
136         // Sound card (source) is connected with JACK client (destination).
137         int result = jack_connect(m_jackClient,
138                                   soundCardInputs[i],
139                                   jack_port_name(m_jackInputPorts[i]));
140
141         // Result code 0 means success, otherwise an error occured.
142         if (result != 0) {
143           std::cout << "Unable to connect sound card input port ";
144           std::cout << (i + 1) << " with the JACK client.\n";
145         }
146       }
147     }
148   }
149
150   // Free the memory allocated by the jack_get_ports(...) call.
151   jack_free(soundCardInputs);
152
```

Listing 8.6: AudioClient *constructor* 4/5 (lines 109-152).

```
153   // 10. Connect the sound card outputs with the JACK client output ports.
154   //     JACK calls them "input ports" (JackPortIsInput)
155   //     because they are writable by the JACK client.
156   unsigned int numSoundCardOutputs = 0;
157   const char** output;
158   const char** soundCardOutputs = jack_get_ports(
159                                     m_jackClient,
160                                     nullptr,
161                                     nullptr,
162                                     JackPortIsPhysical | JackPortIsInput);
163
164   // Count the number of physically available sound card output ports.
165   if (soundCardOutputs != nullptr) {
166     for (output = soundCardOutputs; *output; output++) {
167       numSoundCardOutputs++;
168     }
169   } else {
170       std::cerr << "The sound card has no output ports (capture).\n";
171       std::exit(1);
172   }
173
174   // Iterate over all virtual outputs of the JACK client and
175   // connect them with all available phyical sound card outputs.
176   for (unsigned int i = 0; i < m_numJackClientOutputs; i++) {
177     if (jack_port_name(m_jackOutputPorts[i]) != nullptr) {
178       if (soundCardOutputs[i] != nullptr) {
179         // IMPORTANT: Keep the order of "source" and "destination" in mind.
180         // JACK client (source) is connected with sound card (destination).
181         int result = jack_connect(m_jackClient,
182                                   jack_port_name(m_jackOutputPorts[i]),
183                                   soundCardOutputs[i]);
184
185         // Result code 0 means success, otherwise an error occured.
186         if (result != 0) {
187           std::cout << "Unable to connect sound card output port ";
188           std::cout << (i + 1) << " with the JACK client.\n";
189         }
190       }
191     }
192   }
193
194   // Free the memory allocated by the jack_get_ports(...) call.
195   jack_free(soundCardOutputs);
196 }
```

Listing 8.7: AudioClient *constructor* 5/5 (lines 153-196).

8.2.2 Audio Processing Function

The audio processing function is called by the JACK server whenever there are new audio samples available to be processed. Using a sample rate of 48 kHz and a buffer size of 128 samples per callback frame, it is executed 375 per second or once every 2.67 ms. This continuity is used to drive the ECS systems and generate the behavior of the program. An additional callback counter serves as a reference for time based calculations and represents the elapsed time, measured in callbacks.
The audio processing function is still a work in progress and consists of three major steps, which are listed in a simplified form in Listing 8.8. First, the `float` pointers for the (readable) input buffers of the sound card are acquired (see line 3). Afterwards, the new audio data can be manipulated. In this case, it is simply copied into the corresponding input delay lines (see line 8). Calculations and updates of the ECS systems are also performed during this step. Finally, the `float` pointers for the (writable) output buffers of the sound card are acquired (see line 26) and the manipulated audio data is copied into them.
The steps on how to render virtual WFS sound sources are described in lines 24-39. In a purely object-oriented approach, this implementation would cause no problems. However, here are multiple paradigms that collide with each other. The C-style JACK API with its object-oriented C++ wrapper in the AudioClient class, the object-oriented DelayLine class and the data-oriented and component based ECS. This combination led to some unpredictable problems, which are responsible for the current working status and have to be further investigated. To utilize the advantages of both, OOP and DOP, a meaningful middle course between both approaches has to be chosen. Some insights and considerations:

1. The number of delay lines is static and bound to the inputs of the sound card.
2. The number of loudspeakers is static and bound to the outputs of the sound card.
3. The number of JACK's input and output ports is static after activating the client.
4. Delay lines implemented in ECS would be a removable component at any time.
5. Adding/Removing any audio components would very likely cause audible artifacts.
6. The components of source and loudspeaker entities are embedded in the ECS.
7. The delay line does not know the ECS, if it is not a derived system.
8. The ECS always iterates over all entities and has an iterative but no direct access.
9. Using vectors or sparse sets as intermediary would enable direct/index-based access.
10. Maybe some kind of event handling system is needed after all.

```
int process(const jack_nframes_t nframes) noexcept {
  // Get the pointers to the buffers of all input channels.
  for (unsigned int i = 0; i < m_jackInputPorts.size(); i++) {
    m_inputBuffers[i] = (float*)(jack_port_get_buffer(m_jackInputPorts[i],
                                                      nframes));

    // Write samples from input channels into the corresponding delay line.
    m_delayLines[i]->write(m_inputBuffers[i], nframes);
  }

  // Update all ECS systems.
  m_wfsPlaneSourceGainSystem->update();
  m_wfsPointSourceGainSystem->update();
  m_wfsFocusedSourceGainSystem->update();
  m_wfsDelaySystem->update();
  m_velocitySystem->update(m_numCallbacks);

  //
  // [WORK IN PROGRESS]
  //
  // OOP approach to read from delay lines into audio output buffers.
  //

  // Iterate over all loudspeakers (audio output buffers).
  for (unsigned int i = 0; i < m_jackOutputPorts.size(); i++) {
    m_outputBuffers[i] = (float*)(jack_port_get_buffer(m_jackOutputPorts[i],
                                                       nframes));

    // Iterate over all WFS sources (delay lines).
    for (unsigned int j = 0; j < m_jackInputPorts.size(); j++) {
      // 1. Calculate the delay between source[j] and loudspeaker[i].
      // 2. Calcualte the gain between source[j] and loudspeaker[i].
      // 3. Use delay and gain to read samples from delayLine[j] and
      //    sum up the samples in the current audio output buffer[i].
    }

    // Audio output buffer[i] (loudspeaker[i]) now contains
    // a mix including all WFS sources[0-j] (delayLines[0-j]).
  }

  // Increment the number of callbacks used in time based calculations.
  m_numCallbacks++;

  return 0;
}
```

Listing 8.8: AudioClient *process* function.

8.2.3 ECS System Handling

Individual ECS systems are derived from the System base class (see Listing 8.9). The inheritance is kept to a minimum and only the destructor is `virtual`, in case the derived class allocates memory that should be freed on destruction. In the Scene class the SystemManager is then able to successfully destroy all derived systems by calling the destructor of the System base class (see section 7.12). The `init()` function is optional and helps with initialization tasks outside of the constructor. The `update()` function utilizes the sparse set member variable from the base class, which contains entities filtered by a component bit-mask (see line 14). features the algorithm of the system and is called during the JACK callback. Both functions are not prescribed in the System base class to avoid inheritance, but they are used as a loose guideline for derived systems.

```
1  extern Scene scene; // Global scene object (ECS) located in main program.
2
3  class VelocitySystem : public System {
4   public:
5    VelocitySystem()  = default;
6    ~VelocitySystem() = default; // Add own destructor if needed.
7    void init() {}               // Add initialization if needed.
8
9    void update(int sampleRate, int bufferSize, int numCallbacks) {
10     // Calculate the elapsed time in seconds.
11     float currentTimestamp = (bufferSize / sampleRate * numCallbacks);
12
13     // Iterate over all entities matching the component bit-mask.
14     for (auto const& entity : m_entitySet) {
15       // Get the components of the current entity.
16       Position* pos = scene.getComponent<Position>(entity);
17       Velocity* vel = scene.getComponent<Velocity>(entity);
18
19       Vector3D distVec  = (pos->oldPosition - pos->newPosition);
20       float distance    = distVec.length();
21       float deltaTime   = (currentTimestamp - pos->oldTimestamp)
22       pos->oldTimestamp = currentTimestamp;
23       vel->velocity     = distance / deltaTime; // Meters per second.
24     }
25   }
26 };
```

Listing 8.9: Exemplary VelocitySystem class.

Inside the AudioClient class the derived system is added as a member variable in line 32 (see Listing 8.10). The initialization happens in the AudioClient constructor by assigning the system to the global scene object in line 7. A component bit-mask is added as an entity filter in lines 10-13 and the additional initialization function is called line 16. At this point, the system is ready to be updated by the JACK callback in line 28. A dummy entity with the components used by the VelocitySystem is created in lines 19-24.

```
1  #include "Systems/VelocitySystem.hpp" // Contains global scene object.
2
3  class AudioClient {
4   public:
5    AudioClient(unsigned int numInputs, unsigned int numOutputs, ...) {
6      // Assign the VelocitySystem to the scene (ECS).
7      m_velocitySystem = scene.assignSystem<VelocitySystem>();
8
9      // Create a component bit-mask for the VelocitySystem.
10     ComponentMask mask;
11     mask.set(scene.getComponentID<Position>());
12     mask.set(scene.getComponentID<Velocity>());
13     scene.setSystemComponentMask<VelocitySystem>(mask);
14
15     // Initialize the VelocitySystem.
16     m_velocitySystem->init();
17
18     // Create a new entity.
19     EntityID velEntity = scene.createEntityID();
20
21     // Assign components to the entity that are used in the VelocitySystem.
22     Position* pos = scene.assignComponent<Position>(velEntity);
23     Velocity* vel = scene.assignComponent<Velocity>(velEntity);
24     vel->velocity = 0; // Set current velocity to 0.
25   } // AudioClient constructor.
26
27   int process(const jack_nframes_t nframes) noexcept {
28     m_velocitySystem->update(m_sampleRate, m_bufferSize, m_numCallbacks);
29   }
30
31  private:
32   VelocitySystem* m_velocitySystem;
33 };
```

Listing 8.10: Handling and updating the VelocitySystem.

8.3 NetworkServer Class

The *NetworkServer* class is incorporated into a single header file (*NetworkServer.hpp*). Its initial version is shown in Listing 8.11 and contains the destructor, a validity function in case of a shutdown, the member variables and a singleton pattern to ensure that only a single instance of the network server is running. The initialization and general usage of the OSC handlers (lines 22-23) is examined separately (see subsection 8.3.2). Implementing the OSC handlers to properly interact with the ECS is still a work in progress. To improve the readability, parts of the code are simplified or shortened.

```
1  class NetworkServer {
2   public:
3    ~NetworkServer() {
4      if (m_oscServer != nullptr) {
5        delete m_oscServer;
6        m_oscServer = nullptr;
7      }
8    }
9
10   bool isValid() const { return m_isValid; }
11
12  private:
13   // Remove copy/move constructor and assignment operator for a singleton.
14   NetworkServer(const NetworkServer& other)            = delete;
15   NetworkServer(NetworkServer&& other)                 = delete;
16   NetworkServer& operator=(const NetworkServer& other) = delete;
17   NetworkServer& operator=(NetworkServer&& other)      = delete;
18
19   std::vector<lo::Address> m_renderRelayClients;
20   std::vector<lo::Address> m_visualRelayClients;
21   lo::ServerThread* m_oscServer;
22   void initStandaloneHandlers();
23   void initRelayHandlers();
24
25   std::string m_remoteHost;
26   unsigned int m_remotePort{0};
27   unsigned int m_localPort{0};
28   bool m_isValid{false};
29 };
```

Listing 8.11: Initial NetworkServer class.

8.3.1 Constructor

The constructor of the NetworkServer class is covered in its own subsection due to the amount of code. Based on the given parameters and the desired functionality, it offers several combinations to initialize the embedded OSC server (see Listing 8.12).
The default configuration creates a local OSC server (see line 9). Additionally, a switch has been added to show how to select the network protocol, either TCP or UDP, via a given parameter (see line 12 and 14).
The next combination is a local OSC server using the remote port of a relay server and subscribing to its *render* stream (see line 18). This ensures that all subscribers in a distributed network environment have the same system status and receive all their data via OSC messages from a central relay server. Splitting the streams into a *render* and *visual* portion helps to supply the subscribers only with data relevant to them.
The third option is to start a local relay server using the remote port, so other programs can connect and subscribe to its streams. Their OSC addresses are saved in vectors, which are pre-allocated in line 32 and 33. In contrast to the other cases, the initialization of the OSC handlers is realized with a call to `initRelayHandlers()` (see line 35). The only difference between the OSC handlers in the regular `initStandaloneHandlers()` function is a loop at the end of every handler, which distributes the incoming OSC message to all subscribers of the render or visual stream.
Finally, there is an option to join a multicast group, which comprises a self sustained distributed network. The OSC messages are distributed via the network hardware itself and there is no need for a central server (see line 36).
All of these options allow multiple combinations between the network server and the audio client. The following list shows the possibilities and WONDER equivalents, which can be recreated using CoRGII:

- Local NetworkServer and AudioClient (standalone tWONDER).
- Local NetworkServer, remote relay server, AudioClient (distributed tWONDER).
- Relay NetworkServer and AudioClient (cWONDER and tWONDER).
- Relay NetworkServer (cWONDER).
- Multicast NetworkServer and AudioClient (CoRGII).

```
NetworkServer(int localPort, std::string remoteHost, int remotePort,
              std::string protocol, bool actAsRelay = false,
              bool useRelay = false, bool useMulticast = false) {

  m_localPort  = localPort;
  m_remoteHost = remoteHost;
  m_remotePort = remotePort;

  if (!actAsRelay && !useRelay && !useMulticast) {
    // 1. Start a local OSC server.
    if (protocol == "TCP") {
      m_oscServer = new lo::ServerThread(m_localPort, LO_TCP, nullptr);
    } else {
      m_oscServer = new lo::ServerThread(m_localPort, LO_UDP, nullptr);
    }

    initStandaloneHandlers();
  } else if (!actAsRelay && useRelay && !useMulticast) {
    // 2. Start a local OSC server and subscribe to a remote relay server.
    m_oscServer = new lo::ServerThread(m_localPort);

    // Send a connect/subscribe OSC message to the remote relay server.
    lo::Address remoteAddress(m_remoteHost, m_remotePort);
    remoteAddress.send("/corgii/relay/connect/render", "");

    initStandaloneHandlers();
  } else if (actAsRelay && !useRelay && !useMulticast) {
    // 3. Start a local OSC relay server using the remote port.
    m_oscServer = new lo::ServerThread(m_remotePort);

    // Pre-allocate vector memory for 16 clients in subscription streams.
    m_renderRelayClients.reserve(16);
    m_visualRelayClients.reserve(16);

    initRelayHandlers();
  } else if (!actAsRelay && !useRelay && useMulticast) {
    // 4. Start a local OSC server and connect to a multicast group.
    m_oscServer = new lo::ServerThread(m_remoteHost, m_remotePort,
                                       nullptr, nullptr nullptr);

    initStandaloneHandlers();
  }

  m_oscServer->start(); // Start the OSC server.
}
```

Listing 8.12: NetworkServer *constructor*.

8.3.2 OSC Handler Initialization

The OSC handlers can be initialized in two ways. Both are shown in Listing 8.13 and take place in the constructor of the NetworkServer class. In this exemplary case it is assumed that the OSC server (`m_oscServer`) is already initialized but not started yet. The first and more common variant of adding a method handler is shown in line 6 and 7. The `add_method()` call requires multiple arguments, starting with a *path*, which is the destination for the OSC messages (`/CoRGII/source/pos`). It is followed by a ordered parameter list of the data types packed inside the OSC message (`ifff`: short for `integer`, `float`, `float`, `float`). Both, path and parameter list of an incoming OSC message, have to match in order to trigger the handler. The actual function is listed next (`posHandler`). It is a static member function and enables accessing non-static member variables by passing a `this` pointer, which is the last argument in the list. An example on how to access member variables is shown in line 41. The `this` pointer is passed as a `void` pointer inside the `user_data` argument. Casting it back to a `NetworkServer` pointer allows to access the member variable `m_isValid`.
The second variant to initialize an OSC handler is shown in line 9 and 22. This time, instead of specifying a separate function, a lambda function is declared directly. Accessing the data of an OSC message is shown in lines 13-16. Member variables are also accessible by using the `this` pointer inside the lambda capture `[this]` (see line 24).
The concept of OSC offers an easy way to maintain compatibility with old programs developed for WONDER. As long as the parameter list is the same, it is enough to add two handlers, both specifying the same function, but using two different paths, one for CoRGII and another for WONDER, to achieve the same behavior. An additional function to map the old WONDER OSC paths onto CoRGII's new functions, maybe even repack old OSC messages with deviating parameters, would allow CoRGII to serve as a full replacement for WONDER.
The only problem, which was already discussed in the audio processing function of the AudioClient class (see subsection 8.2.2), is the integration of the ECS. Simply transferring the new concepts into an existing architecture led to some unexpected obstacles. As a result, the network server suffers from the same problem as the audio client, with the main culprit being the iterative access approach of the ECS. Without direct access, a simple position change of a sound source would require to iterate over all available sound sources to find the one specified in an OSC message. This is very inefficient and a better solution is needed to link the ECS with the other components of CoRGII.

```cpp
class NetworkServer {
 public:
  NetworkServer(...) {
    // Add static member functions as source position and exit handler.
    // Passing "this" as the last argument makes it accessible as user_data.
    m_oscServer->add_method("/CoRGII/source/pos", "ifff", posHandler, this);
    m_oscServer->add_method("/CoRGII/exit", nullptr, exitHandler, this);

    // Create a source position handler using a lambda function.
    m_oscServer->add_method("/CoRGII/source/pos", "ifff",
    [](lo_arg** argv, int argc, void* data) {
      // Retrieve the values from the OSC message.
      std::cout << "Position of source ID " << *(&argv[0]->i) << ": (";
      std::cout << *(&argv[1]->f) << ", ";
      std::cout << *(&argv[2]->f) << ", ";
      std::cout << *(&argv[3]->f) << ")\n";

      // The data argument can be casted to an OSC message object.
      lo::Message msg = static_cast<lo::Message>(data);
    });

    // Create an exit handler using a lambda function and capture "this".
    m_oscServer->add_method("/CoRGII/exit", nullptr,
    [this](lo_arg** argv, int argc, void* data) {
      m_isValid = false;
    });
  }

 private:
  lo::ServerThread* m_oscServer;
  bool m_isValid{false};

  static int posHandler(const char* path, const char* types, lo_arg** argv,
                        int argc, void* data, void* user_data) {
    // Same code as in the lambda version of the position handler.
    return 0;
  }

  static int exitHandler(const char* path, const char* types, lo_arg** argv,
                         int argc, void* data, void* user_data) {
    static_cast<NetworkServer*>(user_data)->m_isValid = false;
    return 0;
  }
};
```

Listing 8.13: Exemplary OSC handler initialization.

8.4 SettingsParser Class

The *SettingsParser* class is contained in a single header file (*SettingsParser.hpp*). It is intended to act as a data vault and in contrast to the other components of CoRGII, the settings parser is detached from the ECS. The startup arguments and all settings acquired from the XML file are stored with public access to avoid a lot of unnecessary getters, while the parsing process is private and encapsulated in multiple stages inside the constructor. A singleton pattern ensures that only one instance of the settings parser is active. Parts of the code are shortened or simplified to improve the readability. The parsing of every XML attribute is omitted due to the massive amount of similar code. Since the TinyXML-2 API is well documented and easy to understand, it is recommended to take a look at the examples in the documentation. A list with all attributes in use can be found in section A.4 and the complete settings XML file is listed in section A.5.

```
1  class SettingsParser {
2   public:
3    ~SettingsParser() = default;
4
5    // All parsed settings are stored in public variables or vectors.
6
7   private:
8    // Remove copy/move constructor and assignment operator for a singleton.
9    SettingsParser(const SettingsParser& other)            = delete;
10   SettingsParser(SettingsParser&& other)                 = delete;
11   SettingsParser& operator=(const SettingsParser& other) = delete;
12   SettingsParser& operator=(SettingsParser&& other)      = delete;
13
14   void parseArguments(int argc, char* argv[]);
15
16   std::string parseHostname();
17
18   // Each individual parser is private and called via the constructor.
19 };
```

Listing 8.14: Initial SettingsParser class.

8.4.1 Constructor

The SettingsParser constructor starts with retrieving the hostname (see subsection 8.4.3). Then the startup arguments of the main program are parsed and used to load the settings XML file (line 6 and 18). Parsing the different categories of attributes is done in separate functions by passing a pointer of the XML root node (see line 25). Each function takes care of different checks to ensure the parsed data is within valid ranges.

```
1  SettingsParser(int argc, char* argv[]) {
2    // Retrieve the hostname of this computer.
3    hostname = parseHostname();
4
5    // Parse the command line arguments from the main function.
6    parseArguments(argc, argv);
7
8    if (hostname.empty()) {
9      std::exit(1); // Unable to retrieve hostname of this computer.
10   }
11
12   // Transform the hostname to lowercase before reading the Settings.xml.
13   std::transform(hostname.begin(), hostname.end(),
14                  hostname.begin(), ::tolower);
15
16   // Load the settings from the XML file into the XML document object.
17   tinyxml2::XMLDocument xmlDoc;
18   tinyxml2::XMLError xmlError = xmlDoc.LoadFile(settingsFile.c_str());
19
20   if (xmlError != tinyxml2::XML_SUCCESS) {
21     std::exit(1); // Failed to open the XML file.
22   }
23
24   // Get the root node "CoRGII" inside the XML file with the settings.
25   tinyxml2::XMLNode* xmlRootPtr = xmlDoc.FirstChildElement("CoRGII");
26
27   if (xmlRootPtr == nullptr) {
28     std::exit(1); // Failed to read the root node of the XML file.
29   }
30
31   // Parse the settings inside the XML file in separate stages.
32 }
```

Listing 8.15: SettingsParser *constructor*.

8.4.2 Startup Arguments Parser

The startup arguments parser processes the parameters of the main program, which are passed via the command line during startup. This enables to control the behavior of CoRGII's various components. Switches are used to activate the audio client or a future GUI and to change the kind of the network server. A list with all available command line arguments can be found in section A.3.

```
1  void parseArguments(int argc, char* argv[]) {
2    while (true) {
3      int result = getopt(argc, argv, "n:f:jsrmgh");
4
5      if (result == -1) {
6        // Reached the end of the parameter list.
7        break;
8      }
9
10     switch (result) {
11       case 'n':
12         // Retrieve the argument for the hostname.
13         hostname = optarg;
14         break;
15
16       case 'f':
17         // Retrieve the argument for the path and name of the settings file.
18         settingsFile = optarg;
19         break;
20
21       case 'j':
22         // Disable the JACK audio client.
23         useAudioClient = false;
24         break;
25
26       case 's':
27         // Act as a network relay server.
28         actAsRelayServer  = true;
29         useNetworkRelay   = false;
30         useMulticastGroup = false;
31         break;
32
```

Listing 8.16: SettingsParser *parseArguments* function 1/2 (lines 1-32).

```
33        case 'r':
34          // Connect to a remote network relay.
35          actAsRelayServer  = false;
36          useNetworkRelay   = true;
37          useMulticastGroup = false;
38          break;
39
40        case 'm':
41          // Connect the OSC server with a multicast group.
42          actAsRelayServer  = false;
43          useNetworkRelay   = false;
44          useMulticastGroup = true;
45          break;
46
47        case 'g':
48          // Start with the optional graphical user interface (GUI).
49          useGUI = true;
50          break;
51
52        case 'h':
53          // Print help with all available arguments.
54          std::cout << "\nThe command line arguments are:\n\n";
55          std::cout << "-n (name of this host computer)\n";
56          std::cout << "-f (file path and name to the Settings.xml file)\n";
57          std::cout << "-j (disable the JACK audio client)\n";
58          std::cout << "-s (act as a network relay server)\n";
59          std::cout << "-r (connect to a remote network relay)\n";
60          std::cout << "-m (connect to a multicast group)\n";
61          std::cout << "-g (start with a graphical user interface)\n";
62          std::cout << "-h (print this help text)\n\n";
63          std::exit(0);
64
65        default:
66          break; // Unknown argument.
67      }
68    } // while
69
70    // Print any remaining and unsupported arguments.
71    while (optind < argc) {
72      std::cout << argv[0] << ": unsupported option -- '";
73      std::cout << argv[optind++] << "'\n";
74    }
75  }
```

Listing 8.17: SettingsParser *parseArguments* function 2/2 (lines 33-75).

8.4.3 Hostname Parser

The hostname parser uses the environment variables of the OS to retrieve the current hostname and to help processing individual loudspeaker lists in the settings XML file, which are separated by a hostname attribute. Macros are used to distinguish between different OS versions and variable names.

```
1  std::string parseHostname() {
2    char* tempName = nullptr;
3    std::string hostname;
4
5  #if defined(WIN32) || defined(_WIN32) || defined(_WIN64)
6    tempName = getenv("COMPUTERNAME");
7
8    if (tempName != nullptr) {
9      hostname = tempName;
10     tempName = nullptr;
11   }
12 #else
13   tempName = getenv("HOSTNAME");
14
15   if (tempName != nullptr) {
16     hostname = tempName;
17     tempName = nullptr;
18   } else {
19     // Allocate a char buffer for the hostname.
20     tempName = new char[512];
21
22     // Calling gethostname() returns a 0 on success.
23     if (gethostname(tempName, 512) == 0) {
24       hostname = tempName;
25     }
26
27     // Free the char buffer for the hostname.
28     delete[] tempName;
29     tempName = nullptr;
30   }
31 #endif
32
33   return hostname;
34 }
```

Listing 8.18: SettingsParser *parseHostname* function.

8.5 Main Program

The *main* program unites all previously discussed components of CoRGII and initializes them in different stages. For a better readability, the code is shortened and some checks and functions are omitted. Among them is a signal handler, which captures interrupt signals like the keyboard shortcut `Ctrl + C` and makes it possible to shutdown the main program gracefully. Therefore, the audio client and network server are placed on a global scope (see lines 2-3) to access them in the `cleanup()` function (see lines 5-15). This avoids memory leaks, in case an interrupt occurs or the program quits.
The first step in the main function is to parse all settings (see line 19 in Listing 8.20). These are used to initialize the audio client in line 23 and the network server in line 48. To prevent the main function from unintentionally exiting, the main thread is constantly timed out in a *while*-loop, which checks the scene's validity (see lines 55-57). Since the scene is accessible on a global scope, its public member variable `isValid` can be modified either by the OSC server, when it receives a *quit* OSC message or by the audio client, if the JACK server issues a shutdown callback. If this is the case, the loop is exited and the `cleanup()` function destroys the audio client and network server. The global scene object is automatically destroyed when the main program quits.

```
1  Scene scene;
2  AudioClient* audioClient;
3  NetworkServer* networkServer;
4
5  void cleanup() {
6    if (audioClient != nullptr) {
7      delete audioClient;
8      audioClient = nullptr;
9    }
10
11   if (networkServer != nullptr) {
12     delete networkServer;
13     networkServer = nullptr;
14   }
15 }
16
```

Listing 8.19: Main program of CoRGII 1/2 (lines 1-16).

```
17 int main(int argc, char* argv[]) {
18   // Parse startup arguments and settings file.
19   SettingsParser settings(argc, argv);
20
21   // Initialize the audio client.
22   if (settings.useAudioClient) {
23     audioClient = new AudioClient(settings.numJackInputs,
24                                   settings.numJackOutputs,
25                                   settings.useOnlyPhysicalPorts,
26                                   false); // Don't start a JACK server.
27
28     if (audioClient != nullptr) { scene.isValid = true; }
29   }
30
31   // Initialize the network server.
32   std::string remoteHost  = "";
33   std::string protocol    = settings.networkProtocol;
34   unsigned int remotePort = 0;
35   unsigned int localPort  = settings.localPort;
36   bool actAsRelay         = settings.actAsRelayServer;
37   bool useRelay           = settings.useNetworkRelay;
38   bool useMulticast       = settings.useMulticastGroup;
39
40   if (useRelay) {
41     remoteHost = settings.relayHost;
42     remotePort = settings.relayPort;
43   } else if (useMulticast) {
44     remoteHost = settings.multicastGroup;
45     remotePort = settings.multicastPort;
46   }
47
48   networkServer = new NetworkServer(localPort, remoteHost, remotePort,
49                                     protocol, actAsRelay, useRelay,
50                                     useMulticast);
51
52   if (networkServer != nullptr) { scene.isValid = true; }
53
54   // Check scene every second and keep running as long as it is valid.
55   while (scene.isValid) {
56     std::this_thread::sleep_for(std::chrono::seconds(1));
57   }
58
59   cleanup();
60   return 0;
61 }
```

Listing 8.20: Main program of CoRGII 2/2 (lines 17-61).

8.6 Conclusion

This chapter introduced the architectural ideas behind CoRGII, the composition of the program and the structure of its components. The evolution of the development was discussed from its finding stage up to the actual implementation. This process was accompanied by numerous code examples to emphasize the intention of an understandable and well written documentation. At the same time, some unforeseeable obstacles were revealed, when trying to combine the data-oriented ECS with the object-oriented audio client and network server. Differences in object handling and iteration caused issues that require a solution, which can link both approaches but without losing their individual benefits. Even though the current status of the program is partially a work in progress, it was still possible to pinpoint the problems to tackle in the future.

9 Conclusions

I started this thesis by introducing the subject of WFS and discussed its driving functions, capabilities and limitations. The WONDER software suite was shown as a solution to render such an audio technique. However, its code base is dated and unmaintained, so a benchmark was conducted to check the performance of the core components. The results showed how critical the state of WONDER's inner workings is and motivated me to rethink its current implementation and to start developing a new and more efficient software architecture instead of partially fixing the old one. First, I reworked the delay line code. Further inspecting WONDER's object management and data flows suggested a data-oriented design. In search of a practicable solution, I came across ECS and its data-oriented way to manage objects as a composition of components. It is accompanied by a sparse set, which is used to achieve a dense memory layout. Both go hand in hand and represent the core elements of CoRGII, my reimplementation approach of WONDER. After realizing all these new components and finally combining them with a newly implemented audio renderer and network server, the knowledge was gained that one does not simply mix different software designs, hoping to achieve automagically a better performance.

On my journey through this thesis, I built and documented the foundation of a network controlled audio renderer. My emphasis was to show the ongoing results in an expressive and easy to follow manner. The biggest challenge of this work was the amount of code and components that had to be replaced. Bringing all the individual and working parts together in the last step of the reimplementation process, unfortunately resulted in some compatibility issues, which is why CoRGII is currently still a work in progress.

An important thing I learned throughout this whole project was the fresh perspective on software problems using a data-oriented and component based programming approach. In the beginning, it feels unusual and less intuitive. The reason is, that traditional object-oriented programming works more like the real world and modeling problems with objects is easier. But in the end, it is a matter of the right tool for the right job and the answer to which approach is the best is - as always - it depends.

10 Future Work

The implementation of CoRGII is not finished yet, which is why it is still a work in progress. Since it consists of multiple parts, the work that has to be done in the future, can be distributed onto different components. Additionally, not all components have the same priority. The most important components are currently the audio client, the network server and the ECS in between. Their problem is the different ways they use to iterate over objects and entities. The ECS always iterates over all entities, while the audio client and the network server need an index-based access to data. Since the scene interconnects all parts of CoRGII, it might be used as a global mapper. Sparse sets or vectors could store the entity ID for a static index value.
Despite a low priority, something that might also help with this problem, could be an embedded event handling system in the ECS. Events, like the change of a source position, could be routed to the right entity inside the event handling system.
The time-varying delay line implementation suffers from a problem, regarding the read pointer and the delay growth factor. If the read pointer reached the correct delay value, but the delay growth factor is still non zero, the read pointer could overshoot the correct delay value, causing audible artifacts.
Some refinement is needed in the WFS driving functions. A division by zero is possible during the calculations, when a source is too close to a loudspeaker. To mitigate this possibility, a minimal distance between a source and a loudspeaker could be introduced or a very small *epsilon* value could be added to their distance. A WFS pre-equalization filter is also something that has to be implemented in the future. *FFTW*[1] is a fitting library for this task.
A support for quaternions could be added to the settings parser. Currently, the rotation of a tracker or listener is implemented with azimuth and elevation angles, which could suffer from a gimbal lock. A quaternion only needs four variables to represent the rotation (x, y, z, w), but the implementation is not trivial and it is recommended to use an existing mathematical or graphical library.

[1] https://www.fftw.org Last accessed on: March 20, 2023)

The priority for an embedded GUI in CoRGII, using the Dear ImGui library, is very low at the moment. Actually, Processing turned out to be a very versatile tool to develop an immediate GUI with support for OSC and XML parsing. Since Dear ImGui has a lot of graphics APIs to choose from, it might be beneficial to pick and familiarize with a well known and platform-independent one, instead of trying to support most of them.
So far, no performance improving features are present in CoRGII. Multi-threading could be implemented by hand, using C++ threads, or by adding a handful of *pragmas* and utilize *OpenMP*[2]. Despite being relatively easy to implement, OpenMP might need some fine tuning regarding its thread scheduler, since it is possible that the creation and destruction of threads uses more time than the actual calculation they are intended for. Another way to parallelize calculations is to use *Single Instruction Multiple Data (SIMD)* or *Advanced Vector Extension (AVX)* intrinsics to vectorize code. Manual loop unrolling might also help to increase performance.
Testing in CoRGII is currently neglected, but should be improved soon by creating own test scenarios or directly implementing a C++ unit testing framework.
An addition to the already planned WFS pre-equalization filter might be a further FIR filter, which simulates the air absorption and its effects on the frequencies of the sound. This would enhance the realism of the sound source simulation, but also complicate the implementation and increase the computation load. Therefore, this feature should be regarded as optional.

[2] https://www.openmp.org (Last accessed on: March 20, 2023)

A Appendix

A.1 Software

List of software used in the making of this work (Last accessed on: March 20, 2023).

- **Arch Linux:** Operating system.
  URL: https://archlinux.org
- **CMake:** Build tool.
  URL: https://cmake.org
- **GIMP:** GNU Image Manipulation Program.
  URL: https://www.gimp.org
- **GNU Compiler Collection:** C++ build tool chain.
  URL: https://gcc.gnu.org
- **GNU Octave:** Scientific Programming Language.
  URL: https://octave.org
- **Hotspot:** Linux perf GUI for performance analysis.
  URL: https://github.com/KDAB/hotspot
- **Inkscape:** Vector graphics tool.
  URL: https://inkscape.org
- **JACK:** JACK Audio Connection Kit library.
  URL: https://jackaudio.org
- **LaTeX & TeXstudio:** Document Preparation System & Editor for LaTeX.
  URL: https://www.latex-project.org
  URL: https://www.texstudio.org

- **LibLO:** C / C++ Open Sound Control library.
  URL: https://liblo.sourceforge.net
- **ocenaudio:** Audio editor with FFT analyzer.
  URL: https://www.ocenaudio.com
- **PlantUML QEditor:** Tool for class diagrams.
  URL: https://github.com/jalbersol/plantumlqeditor
- **Processing:** IDE for creating images, animations and interactions.
  URL: https://processing.org
- **Pyzo:** Python IDE.
  URL: https://pyzo.org
- **Qt Creator:** C++ and Qt IDE.
  URL: https://www.qt.io/product/development-tools
- **sfs-matlab:** Sound Field Synthesis Toolbox for Matlab.
  URL: https://sfs-matlab.readthedocs.io/en/latest
- **sfs-python:** Sound Field Synthesis Toolbox for Python.
  URL: https://sfs-python.readthedocs.io/en/latest
- **TinyXML-2:** C++ XML parser library.
  URL: https://github.com/leethomason/tinyxml2
- **Valgrind:** Tool suite for debugging and profiling Linux programs.
  URL: https://valgrind.org
- **yEd Graph Editor:** Tool for graphics and diagrams.
  URL: https://www.yworks.com/products/yed

A.2 CoRGII CMake Script

CoRGII uses the *CMake* build system. The *CMakeLists.txt* script file is listed here.

```
1  # Requirements for the CMake version.
2  cmake_minimum_required(VERSION 3.9...3.23)
3
4  # Project name, description and programming language.
5  project(corgii
6    VERSION 1.0.0
7    DESCRIPTION "Controlled Renderer, Graphical User Interface and Immersive
   ↪  Interaction"
8    LANGUAGES CXX
9  )
10
11 # The tool "pkg-config" is required to find all necessary packages.
12 find_package(PkgConfig REQUIRED)
13
14 # Find JACK Audio Connection Kit, Lightweight OSC (liblo) and TinyXML2.
15 pkg_check_modules(JACK REQUIRED jack)
16 pkg_check_modules(LIBLO REQUIRED liblo)
17 pkg_check_modules(TINYXML2 REQUIRED tinyxml2)
18
19 # Create list with all source files.
20 set(SOURCES
21   src/Main.cpp
22   Settings.xml
23 )
```

Listing A.1: CMake script file for CoRGII 1/3 (lines 1-23).

```
24
25  # Create list with all header files.
26  set(HEADERS
27    # Main
28    include/AudioClient.hpp
29    include/NetworkServer.hpp
30    include/SettingsParser.hpp
31
32    # ECS
33    include/ECS/ComponentManager.hpp
34    include/ECS/ComponentPool.hpp
35    include/ECS/EntityManager.hpp
36    include/ECS/Scene.hpp
37    include/ECS/SparseSet.hpp
38    include/ECS/System.hpp
39    include/ECS/SystemManager.hpp
40    include/ECS/Types.hpp
41
42    # Components (excerpt)
43    include/Components/Position.hpp
44    include/Components/Velocity.hpp
45
46    # Systems (excerpt)
47    include/Systems/MovementSystem.hpp
48    include/Systems/VelocitySystem.hpp
49
50    # Math
51    include/Math/Vector2D.hpp
52    include/Math/Vector3D.hpp
53  )
54
55  # Define executable name and add source files.
56  add_executable(corgii ${SOURCES} ${HEADERS})
57
58  # Add include folder with header files.
59  target_include_directories(corgii PRIVATE
60    include
61  )
```

Listing A.2: CMake script file for CoRGII 2/3 (lines 24-61).

```
62
63 # Add libraries.
64 target_link_libraries(corgii PRIVATE
65   ${JACK_LDFLAGS}
66   ${LIBLO_LDFLAGS}
67   ${TINYXML2_LDFLAGS}
68 )
69
70 # Add compiler options. Use different options for debug and release build.
71 target_compile_options(corgii PRIVATE
72   -Werror -Wfatal-errors -Wpedantic -Wall -Wextra
73   $<$<CONFIG:DEBUG>:-g -O0>
74   $<$<CONFIG:RELEASE>:-g -march=native -mtune=native -flto=auto -O3>
75   ${JACK_CFLAGS}
76   ${LIBLO_CFLAGS}
77   ${TINYXML2_CFLAGS}
78 )
79
80 # Set the C++ compiler version and disable non-standard C++ extensions.
81 set_target_properties(corgii PROPERTIES
82   CXX_STANDARD 17
83   CXX_STANDARD_REQUIRED YES
84   CXX_EXTENSIONS NO
85 )
```

Listing A.3: CMake script file for CoRGII 3/3 (lines 62-85).

A.3 Command Line Arguments

The following command line arguments can be used at startup.

- **-n**: name of this host computer
- **-f**: file path and name to the Settings.xml file
- **-j**: disable the JACK audio client
- **-s**: act as a network relay server
- **-r**: connect to a remote network relay
- **-m**: connect to a multicast group
- **-g**: start with a graphical user interface
- **-h**: print list with arguments

A.4 Settings File Attributes

Room settings:

- length (in meters)
- width (in meters)
- height (in meters)

GUI settings:

- refresh_rate (in hertz)

NetworkServer settings:

- local_port (OSC server port)
- relay_host (IP address of remote OSC server)
- relay_port (port of the remote OSC server)
- multicast_group (multicast group address)
- multicast_port (multicast group port)
- network_protocol (TCP or UDP)

AudioClient settings:

- jack_inputs (number of virtual JACK inputs (sources))
- jack_outputs (number of virtual JACK outputs (speakers))
- physical_ports> (true or false, use only physically available ports)

Renderer (WFS) settings:

- wfs_pre_delay (WFS pre-delay in seconds)
- wfs_max_delay (WFS maximum delay in seconds (delay line size))

Listener settings and attributes:

- num_listeners (integer, only one listener is currently supported)
- listener attributes:
  - ID (integer)
  - name (string, currently limited to 32 characters)
  - pos_x (float)
  - pos_y (float)
  - pos_z (float)
  - pov_azimuth (float, between -180.0 and 180.0 degrees)
  - pov_elevation (float, between -180.0 and 180.0 degrees)
  - colorR (integer, between 0 and 255)
  - colorG (integer, between 0 and 255)
  - colorB (integer, between 0 and 255)

Tracker settings and attributes:

- num_trackers (integer, maximum number of trackers)
- tracker attributes:
  - ID (integer)
  - name (string, currently limited to 32 characters)
  - pos_x (float)
  - pos_y (float)
  - pos_z (float)
  - pov_azimuth (float, between -180.0 and 180.0 degrees)
  - pov_elevation (float, between -180.0 and 180.0 degrees)
  - colorR (integer, between 0 and 255)
  - colorG (integer, between 0 and 255)
  - colorB (integer, between 0 and 255)

Source settings and attributes:

- num_sources (integer, maximum number of sources)
- source attributes:
  - ID (integer)
  - name (string, currently limited to 32 characters)
  - pos_x (float)
  - pos_y (float)
  - pos_z (float)
  - type (integer)
  - angle (float, between 0.0 and 360.0 degrees)
  - gain (float, between 0.0 and 1.0)
  - colorR (integer, between 0 and 255)
  - colorG (integer, between 0 and 255)
  - colorB (integer, between 0 and 255)

Speaker settings and attributes:

- hostname attribute:
  - num_speakers (integer, number of upcoming speakers for this host)
- speaker attributes:
  - ID (integer)
  - pos_x (float)
  - pos_y (float)
  - pos_z (float)
  - norm_x (float, between -1.0 and 1.0)
  - norm_y (float, between -1.0 and 1.0)
  - norm_z (float, between -1.0 and 1.0)
  - gain (float, between 0.0 and 1.0)

A.5 Settings File XML Format

The settings file (*Settings.xml*), which is used by CoRGII at startup, is listed below.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2
3  <CoRGII>
4
5    <!-- Room settings. -->
6    <Room>
7      <length>4.989</length>
8      <width>5.7125</width>
9      <height>2.5</height>
10   </Room>
11
12   <!-- GUI settings. -->
13   <GUI>
14     <refresh_rate>60</refresh_rate>
15   </GUI>
16
17   <!-- NetworkServer settings. -->
18   <Network>
19     <local_port>9000</local_port>
20     <relay_host>192.168.3.1</relay_host>
21     <relay_port>58100</relay_port>
22     <multicast_group>224.72.65.87</multicast_group>
23     <multicast_port>9000</multicast_port>
24     <network_protocol>UDP</network_protocol>
25   </Network>
26
27   <!-- AudioClient settings. -->
28   <Audio>
29     <jack_inputs>4</jack_inputs>
30     <jack_outputs>4</jack_outputs>
31     <physical_ports>false</physical_ports>
32   </Audio>
33
34   <!-- Renderer (WFS) settings. -->
35   <Renderer>
36     <wfs_pre_delay>0.022</wfs_pre_delay>
37     <wfs_max_delay>2.0</wfs_max_delay>
38   </Renderer>
```

Listing A.4: CoRGII settings XML file format 1/2 (lines 1-38).

```
39
40    <!-- Listener settings. -->
41    <Listeners>
42      <num_listeners>1</num_listeners>
43      <listener ID="0" name="Listener #000" pos_x="0.0" pos_y="0.0"
      ↪  pos_z="0.0" pov_azimuth="0.0" pov_elevation="0.0" colorR="255"
      ↪  colorG="0" colorB="0" />
44    </Listeners>
45
46    <!-- Tracker settings. -->
47    <Trackers>
48      <num_trackers>256</num_trackers>
49      <tracker ID="0" name="Tracker #000" pos_x="0.0" pos_y="0.0" pos_z="0.0"
      ↪  pov_azimuth="0.0" pov_elevation="0.0" colorR="255" colorG="0" colorB="0"
      ↪  />
50    </Trackers>
51
52    <!-- Source settings. -->
53    <Sources>
54      <num_sources>256</num_sources>
55      <source ID="0" name="Source #000" pos_x="-1.25" pos_y="0.75" pos_z="2"
      ↪  type="1" angle="0" gain="1.0" colorR="255" colorG="0" colorB="0"   />
56
57      <source ID="1" name="Source #001" pos_x="-1.25" pos_y="0.75" pos_z="2"
      ↪  type="1" angle="0" gain="1.0" colorR="0" colorG="255" colorB="0"   />
58
59      <source ID="2" name="Source #002" pos_x="-1.25" pos_y="0.75" pos_z="2"
      ↪  type="1" angle="0" gain="1.0" colorR="0" colorG="0" colorB="255" />
60
61      <source ID="3" name="Source #003" pos_x="-1.25" pos_y="0.75" pos_z="2"
      ↪  type="0" angle="0" gain="1.0" colorR="255" colorG="0" colorB="255" />
62    </Sources>
63
64    <!-- Speaker settings. -->
65    <!-- NOTE: Speaker lists are separated by the hostname of a computer. -->
66    <!-- NOTE: The speaker ID number should be unique! -->
67    <Speakers>
68      <dell num_speakers="2"> <!-- Lowercase hostname & speakers. -->
69        <speaker ID="0" pos_x="0.75000"  pos_y="0.50000" pos_z="0.00000"
      ↪  norm_x="0.00000" norm_y="-1.00000" norm_z="0.00000" gain="1.00000" />
70
71        <speaker ID="1" pos_x="-0.75000" pos_y="0.50000" pos_z="0.00000"
      ↪  norm_x="0.00000" norm_y="-1.00000" norm_z="0.00000" gain="1.00000" />
72      </dell>
73    </Speakers>
74  </CoRGII>
```

Listing A.5: CoRGII settings XML file format 2/2 (lines 39-74).

Bibliography

[1] Mike Acton. Data-Oriented Design and C++. In *CppCon 2014*. CppCon, 2014. URL https://github.com/CppCon/CppCon2014/tree/master/Presentations/Data-Oriented%20Design%20and%20C%2B%2B. (Last accessed on: March 20, 2023).

[2] Jens Ahrens. *Analytic Methods of Sound Field Synthesis*. Springer Science & Business Media, 2012 edition, 2012. ISBN 978-3-642-25742-1.

[3] Jens Ahrens and Sascha Spors. Reproduction of moving virtual sound sources with special attention to the Doppler Effect. In *124th Convention of the Audio Engineering Society*, Amsterdam, The Netherlands, May 2008.

[4] Jens Ahrens and Sascha Spors. On the secondary source type mismatch in wave field synthesis employing circular distributions of loudspeakers. In *127th Convention of the Audio Engineering Society*, New York, USA, October 2009.

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

[6] Augustinus J. Berkhout. A holographic approach to acoustic control. *Journal of the Audio Engineering Society*, 36:977–995, December 1988.

[7] Nikhil Bhargav. Cache-Friendly Code. https://www.baeldung.com/cs/cache-friendly-code, October 2009. (Last accessed on: March 20, 2023).

[8] Preston Briggs and Linda Torczon. An Efficient Representation for Sparse Sets. *ACM Letters on Programming Languages and Systems*, 2(1–4):59–69, March 1993. ISSN 1057-4514.

[9] Michele Caini. ECS back and forth (Part 1 - Introduction). https://skypjack.github.io/2019-02-14-ecs-baf-part-1, February 2019. (Last accessed on: March 20, 2023).

[10] Michele Caini. EnTT (C++ ECS library) [cdee000]. https://github.com/skypjack/entt, February 2023. (Last accessed on: March 20, 2023).

[11] Carola Christoffel. *Modifikation der Software einer Wellenfeldsyntheseanlage zur Wiedergabe fokussierter Quellen in Abhängigkeit der Zuhörerposition.* Bachelor's thesis, Hochschule für Angewandte Wissenschaften, 2014.

[12] David Colson. How to make a simple entity-component-system in C++. https://www.david-colson.com/2020/02/09/making-a-simple-ecs.html, February 2020. (Last accessed on: March 20, 2023).

[13] Intel® Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Intel® Corporation, January 2023. URL https://cdrdv2.intel.com/v1/dl/getContent/671488. (Last accessed on: March 20, 2023).

[14] Ivica Crnkovic. Component-based software engineering - new challenges in software development. In *Proceedings of the 25th International Conference on Information Technology Interfaces, 2003. ITI 2003.*, pages 9–18, 2003. doi: 10.1109/ITI.2003.1225314.

[15] Erbes, Vera and Spors, Sascha. Influence of the Listening Room on Spectral Properties of Wave Field Synthesis. In *German Annual Conference on Acoustics (DAGA)*, pages 1057–1060, Kiel, Germany, March 2017.

[16] Richard Fabian. *Data-Oriented Design: Software Engineering for Limited Resources and Short Schedules.* Richard Fabian, Stockport, UK, first edition, 2018. ISBN 978-1-916-47870-1.

[17] Gergely Firtha. *A Generalized Wave Field Synthesis Framework with Application for Moving Virtual Sources.* PhD thesis, Budapest University of Technology and Economics, 2019.

[18] Gergely Firtha, Péter Fiala, Frank Schultz, and Sascha Spors. Improved Referencing Schemes for 2.5D Wave Field Synthesis Driving Functions. *IEEE Transactions on Audio, Speech, and Language Processing*, 25(5):1117–1127, May 2017.

[19] Firtha, Gergely and Fiala, Peter. Wave Field Synthesis of Moving Sources with Retarded Stationary Phase Approximation. *Journal of the Audio Engineering Society*, 63(12):958–965, December 2016.

[20] Wolfgang Fohl. The Wave Field Synthesis Lab at the HAW Hamburg. In *Sound - Perception - Performance*, chapter 10, pages 243–255. Springer Science & Business Media, 2013 edition, 2013. ISBN 978-3-319-00107-4.

[21] Wolfgang Fohl and Eva Wilk. Enhancements to a Wave Field Synthesis System to Create an Interactive Immersive Audio Environment. In *3rd Int. Conf. on Spatial Audio*. VDT, September 2015.

[22] Fontana, Tiago and Netto, Renan and Livramento, Vinicius and Guth, Chrystian and Almeida, Sheiny and Pilla, Laércio and Güntzel, José Luís. How Game Engines Can Inspire EDA Tools Development: A Use Case for an Open-Source Physical Design Library. In *Proceedings of the 2017 ACM on International Symposium on Physical Design*, ISPD '17, page 25–31, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346962. doi: 10.1145/3036669.3038248. URL https://doi.org/10.1145/3036669.3038248.

[23] Matthias Geier, Jens Ahrens, and Sascha Spors. The SoundScape Renderer: A Unified Spatial Audio Reproduction Framework for Arbitrary Rendering Methods. In *124th Convention of the Audio Engineering Society*, Amsterdam, The Netherlands, May 2008. Audio Engineering Society (AES).

[24] Marvin Geitner. *Erweiterung der Wellenfeldsynthese-Software WONDER um 3D-Quellenpositionen und zusätzliche Lautsprecherkomponenten.* Bachelor's thesis, Hochschule für Angewandte Wissenschaften, 2015.

[25] Martin Hansen. *Positionierung von Klangquellen einer Wellenfeldsynthese-Anlage mit Hilfe eines Audio-Plugins.* Bachelor's thesis, Hochschule für Angewandte Wissenschaften, 2014.

[26] Stefan Heidtmann. Implementation and Evaluation of Optimization Strategies for Audio Signal Convolution. Master's thesis, Hochschule für Angewandte Wissenschaften Hamburg, 2019.

[27] Christiaan Huygens. *Traité de la lumière.* A Leide, Chez Pierre vander Aa (marchand libraire), 1690. URL https://doi.org/10.5479/sil.294285.39088000545160. (Last accessed on: March 20, 2023).

[28] Julius O. Smith III. *Physical Audio Signal Processing.* W3K Publishing, 2010. ISBN 978-0-9745607-2-4. URL http://books.w3k.org. (Last accessed on: March 20, 2023).

[29] Tim Johansson. Job System & Entity Component System. In *Game Developers Conference 2018 (GDC)*. GDC, 2018. URL https://www.gdcvault.com/play/1024839/Job-System-Entity-Component-System. (Last accessed on: March 20, 2023).

[30] Birger Kollmeier, Thomas Brand, and Bernd Meyer. *Perception of Speech and Sound*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-49127-9.

[31] Lange, Patrick and Weller, Rene and Zachmann, Gabriel. Wait-free hash maps in the entity-component-system pattern for realtime interactive systems. In *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 1–8, 2016. doi: 10.1109/SEARIS.2016.7551583.

[32] Noel Llopis. Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP). https://gamesfromwithin.com/data-oriented-design, December 2009. (Last accessed on: March 20, 2023).

[33] Baalman Marije A.J., Torben Hohn, Simon Schampijer, and Thilo Koch. Renewed architecture of the sWONDER software for Wave Field Synthesis on large scale systems. In *Proc. of the 5th Int. Linux Audio Conference (LAC07)*, Berlin, Germany, March 2007.

[34] Miguel Martin. anax (C++ ECS library) [48b36fe]. https://github.com/miguelmartin75/anax, February 2023. (Last accessed on: March 20, 2023).

[35] Scott Meyers. CPU caches and why you care. In *code::dive Conference 2014*. code::dive, 2014. URL https://www.youtube.com/watch?v=WDIkqP4JbkE. (Last accessed on: March 20, 2023).

[36] Austin Morlan. A simple entity component sytem (ECS) [C++]. https://austinmorlan.com/posts/entity_component_system, June 2019. (Last accessed on: March 20, 2023).

[37] Malte Nogalski. Acoustic Redirected Walking with Auditory Cues by Means of Wave Field Synthesis. Master's thesis, Hochschule für Angewandte Wissenschaften Hamburg, 2015.

[38] Malte Nogalski and Wolfgang Fohl. Acoustic redirected walking with auditory cues by means of wave field synthesis. In *2016 IEEE Virtual Reality (VR)*, pages 245–246. IEEE, 2016.

[39] Adrian Papari. Artemis-odb (Java ECS library) [51628b3]. https://github.com/junkdog/artemis-odb, February 2023. (Last accessed on: March 20, 2023).

[40] Pesterev, Aleksey and Zeldovich, Nickolai and Morris, Robert T. Locating Cache Performance Bottlenecks Using Data Profiling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 335–348, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605585772. doi: 10.1145/1755913.1755947. URL https://doi.org/10.1145/1755913.1755947.

[41] Will C. Pirkle. *Designing Audio Effect Plugins in C++ - For AAX, AU, and VST3 with DSP Theory*. Routledge, New York, second edition, 2019. ISBN 978-0-429-95432-0.

[42] Player Unknown Productions. Entity Component System explained - the bedrock of our large-scale world simulation. https://playerunknownproductions.net/news/entity-component-system, March 2022. (Last accessed on: March 20, 2023).

[43] Aras Pranckevičius. Entity Component Systems & Data Oriented Design. In *Unity Training Academy 2018-2019 (No. 3)*. Unity Training Academy, 2018. URL https://aras-p.info/texts/files/2018Academy%20-%20ECS-DoD.pdf. (Last accessed on: March 20, 2023).

[44] Rudolf Rabenstein, Marcus Renk, and Sascha Spors. Limiting Effects of Active Room Compensation using Wave Field Synthesis. In *118th Convention of the Audio Engineering Society*, Barcelona, Spain, May 2005.

[45] C++ Reference. Placement new operator. https://en.cppreference.com/w/cpp/language/new#Placement_new, February 2023. (Last accessed on: March 20, 2023).

[46] C++ Reference. sleep_for function. https://en.cppreference.com/w/cpp/thread/sleep_for, February 2023. (Last accessed on: March 20, 2023).

[47] Joshua D. Reiss and Andrew McPherson. *Audio Effects - Theory, Implementation and Application*. CRC Press, Boca Raton, Florida, first edition, 2014. ISBN 978-1-466-56029-1.

[48] Vittorio Romeo. Implementation of a component-based entity system in modern C++. In *CppCon 2015*. CppCon, 2015. URL https://github.com/CppCon/

CppCon2015/tree/master/Tutorials/Implementation%20of%20a%20component-based%20entity%20system%20in%20modern%20C%2B%2B. (Last accessed on: March 20, 2023).

[49] Andreas Schlegel. ControlP5, a GUI library for the programming environment Processing, March 2015. URL https://doi.org/10.5281/zenodo.16290.

[50] Andreas Schlegel. oscP5: An OSC library for java and the programming environment Processing, March 2015. URL https://doi.org/10.5281/zenodo.16308.

[51] Frank Schultz. *Sound Field Synthesis for Line Source Array Applications in Large-Scale Sound Reinforcement*. PhD thesis, Universität Rostock, 2016. (Last accessed on: March 20, 2023).

[52] Yehonathan Sharvit. *Data-Oriented Programming: Reduce Software Complexity*. Manning Publications Co., Shelter Island, New York, first edition, 2022. ISBN 978-1-617-29857-8.

[53] Ghan Bir Singh. Single versus Multiple Inheritance in Object Oriented Programming. *SIGPLAN OOPS Messenger*, 6(1):30–39, January 1995. ISSN 1055-6400.

[54] Slay, Tylor and Spitzer, Grace B. and Bass, Robert B. Proposed Application for an Entity Component System in an Energy Services Interface. In *2022 IEEE Conference on Technologies for Sustainability (SusTech)*, pages 177–180, 2022. doi: 10.1109/SusTech53338.2022.9794252.

[55] Julius Smith, Stefania Serafin, Jonathan Abel, and David Berners. Doppler Simulation and the Leslie. In *Proc. of the 5th Int. Conference on Digital Audio Effects (DAFx-02)*, Hamburg, Germany, September 2002.

[56] Sascha Spors. Extension of an Analytic Secondary Source Selection Criterion for Wave Field Synthesis. In *123th Convention of the Audio Engineering Society*, New York, USA, October 2007.

[57] Sascha Spors and Rudolf Rabenstein. Spatial Aliasing Artifacts Produced by Linear and Circular Loudspeaker Arrays used for Wave Field Synthesis. In *120th Convention of the Audio Engineering Society*, Paris, France, May 2006.

[58] Sascha Spors, Herbert Buchner, and Rudolf Rabenstien. A novel approach to active listening room compensation for wave field synthesis using wave-domain adaptive filtering. In *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 4, 2004.

[59] Spors, Sascha and Ahrens, Jens. Analysis and Improvement of Pre-equalization in 2.5-dimensional Wave Field Synthesis. In *128th Convention of the Audio Engineering Society*, London, UK, May 2010. ISBN 978-0-937803-74-5.

[60] Spors, Sascha and Rabenstein, Rudolf and Ahrens, Jens. The Theory of Wave Field Synthesis Revisited. In *124th Convention of the Audio Engineering Society*, Amsterdam, The Netherlands, May 2008.

[61] Spors, Sascha and Schultz, Frank, and Rettberg, Till. Improved Driving Functions for Rectangular Loudspeaker Arrays Driven by Sound Field Synthesis. In *German Annual Conference on Acoustics (DAGA)*, Aachen, Germany, March 2016.

[62] E. W. Start. *Direct Sound Enhancement by Wave Field Synthesis*. PhD thesis, Technische Universiteit Delft, 1997.

[63] Szyperski, Clemens and Gruntz, Dominik and Murer, Stephan. *Component Software - Beyond Object-oriented Programming*. Pearson Education, Amsterdam, 2002. ISBN 978-0-201-74572-6.

[64] Dimitris Theodoropoulos, Catalin Bogdan Ciobanu, and Georgi Kuzmanov. Wave Field Synthesis for 3D Audio: Architectural Prospectives. In *Proceedings of the 6th ACM Conference on Computing Frontiers*, CF '09, page 127–136, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584133.

[65] Alec Thomas. EntityX (C++ ECS library) [3989cf4]. https://github.com/alecthomas/entityx, February 2023. (Last accessed on: March 20, 2023).

[66] Unity. Understanding Data Oriented Design for Entity Component Systems. In *Game Developers Conference 2019 (GDC)*. GDC, 2019. URL https://www.youtube.com/watch?v=0_Byw9UMn9g. (Last accessed on: March 20, 2023).

[67] Unity. ECS for Unity. https://unity.com/ecs, 2023. (Last accessed on: March 20, 2023).

[68] Wiebusch, Dennis and Latoschik, Marc Erich. Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems. In *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 25–32, 2015. doi: 10.1109/SEARIS.2015.7854098.

[69] Wierstorf, Hagen and Hohnerlein, Christoph and Spors, Sascha and Raake, Alexander. Coloration in Wave Field Synthesis. *Journal of the Audio Engineering Society*, 55(8):5–3, August 2014.

[70] Eugen Winter. *Analyse und Reimplementierung einer plattformunabhängigen Software für das 3D-Audio-Rendering.* Bachelor's thesis, Hochschule für Angewandte Wissenschaften Hamburg, 2016.

[71] Fiete Winter and Sascha Spors. On Fractional Delay Interpolation for Local Wave Field Synthesis. In *European Signal Processing Conference*, pages 2415–2419, Budapest, Hungary, August 2016.

[72] Winter, Fiete and Spors, Sascha. A Comparison of Sound Field Synthesis Techniques for Non-Smooth Secondary Source Distributions. In *German Annual Conference on Acoustics (DAGA)*, pages 1463–1466, Aachen, Germany, March 2016.

[73] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall, 1976. ISBN 978-0-13-022418-7.

[74] Udo Zölzer. *Digital Audio Signal Processing.* John Wiley & Sons, New York, second edition, 2008. ISBN 978-0-470-68002-5.

[75] Udo Zölzer. *DAFX - Digital Audio Effects.* John Wiley & Sons, New York, second edition, 2011. ISBN 978-0-470-97967-9.

Glossary

cWONDER WONDER's central network control unit to coordinate OSC commands.

HAW Hamburg Hamburg University of Applied Sciences, Germany.

I²AudioLab Laboratory for interactive and immersive audio systems at HAW Hamburg.

tWONDER WONDER's time delay based audio renderer for wave field synthesis.

xWONDER WONDER's graphical user interface (GUI).

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original