Institut für Visualisierung und Interaktive Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2319

Flexibler Soundserver für clusterbasierte VR

Alexander Derheim

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Dr. techn. h. c. Thomas Ertl
Betreuer:	Dr. Guido Reina Dipl.-Phys. Gregor Mückl
begonnen am:	14. Februar 2011
beendet am:	16. August 2011
CR-Klassifikation:	C.3 (Signal processing systems), D.4.1 (Synchronization), E.1 (DATA STRUCTURES)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Das Visualisierungsinstitut der Universität Stuttgart nahm 2010 ein einzigartiges Visualisierungslabor in Betrieb, dessen Herzstück eine Powerwall mit extrem hoher Auflösung ist. Für die Erzeugung und Darstellung des Bildmaterials auf dieser Powerwall kommen 64 Rechner zum Einsatz. Die Kommunikation der Rechner erfolgt über ein schnelles Netzwerk. [1]

Software für die Visualisierung auf einem solchen System, kann nicht einfach gekauft werden [1]. Dies gilt auch für die Soundverarbeitung, zumindest aufgrund mangelnder Flexibilität. Der Cluster aus 64 Rechnern benötigt allerdings eine Schnittstelle, die die anfallenden Soundanfragen entgegennimmt, filtert, den Umgebungsgegebenheiten entsprechend mischt und die zum System zugehörende Surround-Sound-Anlage treibt.

Somit widmet sich diese Arbeit der Entwicklung eines **flexiblen Soundservers für Clusterbasierte VR**. Der Schwerpunkt dieser Arbeit ist die interne Sound-Verarbeitung und Organisation.

Aufgabenstellung

Die Anforderung an den Soundserver sind [2]:

- Mehrkanalausgabe des Sounds über die proprietäre ASIO Schnittstelle, welche eine effiziente Soundverarbeitung, hohen Datendurchsatz, Synchronisation, geringe Verzögerungszeiten und Hardware seitige Erweiterbarkeit bietet [3].
- Beliebig viele gleichzeitig abspielbare Soundfiles. Die Anzahl soll nur durch die Rechenleistung eingeschränkt sein. Es sollen mindestens die Formate Ogg Vorbis und PCM-WAVE unterstützt werden.
- Mehrere Abspielanfragen sollen samplegenau parallel gestartet werden können.
- Platzierung der Sounds im 3D-Raum. Hier war der Wunsch dem Soundserver pro Sound einen Volume-Vektor zuordnen zu können.
- Filterketten sollen zur Laufzeit aufgebaut und Abspielanfragen zugeordnet werden können.
- Für jeden Ausgabekanal soll eine zusätzliche Filterkette verwendet werden, um Ausgabe spezifische Filterung durchzuführen, wie z.B. Kanallautstärke und Ausgleich der Hörer- bzw. Lautsprecher-Position.
- Die einzelnen Filter sollen mono arbeiten. Die Ein- bzw. Ausgabe soll entweder auf Zeit oder Frequenz spezialisiert sein.
- Geforderter Mindestfiltersatz:

  - Delay-Filter z.B. zum Ausgleich der Hörerposition
  - FFT Transformations- und Rücktransformations-Filter mit überlappenden Fenstern. Benötigt für Filter, die im Frequenzbereich arbeiten.
  - Multiband-Equalizer-Filter
  - Filter für Phasenverschiebung
- Um die Erweiterbarkeit zu gewährleisten soll ein einheitliches Filter-Interface verwendet werden.
- Der Soundserver soll einen austauschbaren Zeitgeber benutzen, dessen Implementierung zunächst auf der Systemzeit basieren soll. Dieser austauschbare Zeitgeber wird benötigt um die Clients und den Server zu synchronisieren z.B. anhand einer globalen Referenzzeit.
- Der Ausbau der Architektur zur Ansteuerung über Ethernet soll zumindest entworfen werde.

Gliederung

Diese Arbeit ist wie folgt gegliedert:

Kapitel 2 auf der nächsten Seite – Existierende Lösungen: Gibt einen Überblick über Designs existierender Lösungen, die mit der Aufgabenstellung verwand sind.

Kapitel 3 auf Seite 13 – Flexibler Soundserver für clusterbasierte VR: Fasst den Soundserver und dessen Komponenten zunächst in einer Übersicht zusammen, geht daraufhin Anhand der Komponenten auf das Design ein. Die entsprechenden Grundlagen werden neben den entsprechenden Komponenten behandelt.

Kapitel 4 auf Seite 63 – Test-Client: Geht kurz auf den, für die Erprobung des Netzwerkverhaltens und Fernsteuerung des Soundservers verwendeten, Test-Client ein.

Kapitel 5 auf Seite 65 – Zusammenfassung und Ausblick: Fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

Anhang A auf Seite 67 – Benchmarks: Zeigt das Laufzeit- und Skalierungs-Verhalten auf verschiedenen Systemen anhand Benchmarks auf.

Anhang B auf Seite 69 – GUI-Übersicht: Geht auf Systemvoraussetzungen, Konfiguration und Bedienung des Soundservers und Test-Clients ein.

Existierende Lösungen 2

Im Folgenden wird auf die bereits existierende Lösungen eingegangen, welche für die hier geforderten Aufgaben potenziell entweder als Teillösungen oder Gesamtlösungen in Frage kommen. Die Auflistung der (Teil-)Lösungen erhebt keinen Anspruch auf Vollständigkeit, vielmehr stellt sie einen Ausschnitt verfügbarer (Teil-)Lösungen dar.

2.1. NMM — Network-Integrated Multimedia Middleware

Die Network-Integrated Multimedia Middleware (NMM) ist eine Software-Architektur, die das Entwickeln von verteilten Multimedia-Anwendungen vereinfachen will. Das Hauptfeature von NMM ist, verteilte Geräte zu virtuellen Geräten zusammenschließen zu können. Zum Beispiel kann eine Audio-Ausgabe dadurch mit einfachen Mitteln auf mehrere Geräte gleichzeitig verteilt werden. Ein im Netz sich befindlicher Rechner wandelt und vermittelt diese Daten. [4]

Nach der Einsicht der Features und Plug-ins Beschreibungen [5] wird folgendes angenommen: Die Software-Architektur basiert weitgehend auf Plug-ins, die sich zu einem Gesamtkonzept zusammenschließen. Das bedeutet, dass die Funktionalität an viele Gegebenheiten angepasst werden kann, unter anderem auch zum Einsatz auf verschiedenen Betriebssystemen. Neue Funktionen können durch Plug-ins eingefügt werden. Die Architektur ist weitgehend auf Streaming von A/V-Daten ausgelegt. Mehrere Streams können hoch qualitativ synchronisiert werden.

Da ein hierzu verwendender Client keine Sound-Streams sondern Anfragen verschickt, ist die Streaming-Natur von NMM im ersten Augenblick ungeeignet. Falls nicht schon ein Plug-in existiert, wäre das eventuell durch ein Plug-in umgehbar. Hierzu wäre ein, durch das Plug-in angebundene Gerät notwendig, welches auf Anfrage Sounds streamt. Die Anfragen selber sind entweder auch durch ein Plug-in realisierbar oder müssen extern außerhalb des Konzepts realisiert werden. Desweiteren ist unbekannt in wie weit Sounds, d.h. hier Streams sich samplegenau synchronisieren lassen. In wieweit das Mischen der Sounds möglich ist, ist auch nicht ersichtlich, so bleibt die Frage, ob ein Mono-Sound im 3D-Raum platziert werden kann, also ob eine gewichtete Mischung in mehrere Kanäle möglich ist. Viele der angeforderten Features können durch Implementierung von Plug-Ins erfüllt werden, falls nicht bereits derartige Plug-ins existieren.

Das grundlegende Problem dieser Lösung ist, die zu erwartenden Latenzzeiten. Schon aufgrund, dass für die Realisierung der Soundanfragen eventuell ein Zwischengerät benötigt wird, können die Netzwerklatenzen je nach Implementierung oder Einsatz verdoppelt werden. Es ist anzunehmen, dass für den gedachten Einsatz, das NMM zwar synchronisieren kann, aber Latenzen dort weitgehend unkritisch sind. Weiterhin ist unklar wie auf die Soundhardware zugegriffen wird. Das hier in der Arbeit verwendete ASIO greift im Prinzip

direkt auf die Soundhardware zu. Somit wird mit ASIO auch der eventuell stark Latenz behaftete Windows-Mixer umgangen, der auch unter Umständen, durch unaufgefordertes Resampling und ähnliches, Samples verfälscht [6]. Eine weitere offene Frage ist, wie schnell oder flexibel das Konzept auf das Verbinden und Lösen von vielen Clients reagiert. Denn das Konzept steht hier 64 individuellen Clients entgegen. Die Synchronisation der Clients muss zudem nicht nur für die Soundverarbeitung vollzogen werden, sondern außerhalb eines Sound-Konzepts auch für die gemeinschaftliche Erzeugung von 3D-Bildern.

2.2. VLC media player

Nach Web-Recherchen (unter anderem [7]) wird davon ausgegangen, dass das Konzept von VLC sich ähnlich verhält wie das des NMM aus Abschnitt 2.1 auf der vorherigen Seite – NMM — Network-Integrated Multimedia Middleware. Mit ähnlich ist hier der Bezug zu den gestellten Aufgaben, Problemen und Lösungsansätzen gemeint. Erwartet wird zudem ein geringerer Funktionsumfang, also Mehraufwand für Implementierung von Plug-ins. Weiterhin erweckt das Konzept den Eindruck einer stärkeren Spezialisierung auf rein Player- bzw. Stream-Verhalten.

2.3. GStreamer

Wie der Name schon sagt, ist auch dieses Framework auf das Annehmen und die Verarbeitung von Streams ausgelegt. GStreamer hat auch vielversprechende Features [8]. So ist anzunehmen, dass einige hier geforderten Aufgaben bereits abgedeckt sein könnten. Und ist ebenfalls durch Plug-ins erweiterbar. So wird genannt, dass verschiedene Sample Formate unterstützt werden, also auch das in dieser Arbeit verwendete Float-Format. Es wird auch genannt das Mehrkanalkonfigurationen unterstützt werden, allerdings müsste untersucht werden, ob diese für die Volume-Vektor Verarbeitung einsetzbar oder flexibel genug sind. Um weitere geforderte Funktionen zu erhalten, wäre es eventuell möglich geeignete Plug-ins zu entwickeln. Auch solche um das Streaming von den Clients aus zu umgehen, denn das Streaming kann allgemein höhere Latenzen verursachen und erhöht zudem die Netzwerkbelastung, was auch zu höheren und zu noch unberechenbareren Latenzen führen kann. Das zum Einsatz kommende Netzwerk ist zwar hoch performant, dennoch für das 3D-Rendering steht von vornherein eine hohe Netzwerklast fest. Zudem müsste jeder einzelne Client im Falle von Streaming, die gleichen Sounddaten zwischenlagern oder über einen weiteren Netzwerkschritt weiterleiten. Da unter Windows wohl DirectShow für die Soundausgabe verwendet wird, sind somit auch z.B. die Nachteile des Windowsmixers zu erwarten.

2.4. PulseAudio

Auch PulseAudio bietet Serverfunktionalität für Streaming und ist dem GSteamer ähnlich, in Bezug auf gestellte Anforderungen, Probleme und Lösungsmöglichkeiten. Unter Linux wird hier zwar beschrieben, dass direkt auf die Soundhardware zugegriffen wird, allerdings wird unter Windows auf DirectSound zurückgegriffen. Dies unterliegt potenziell den gleichen Nachteilen des Windowsmixers. Es wird in der Quelle [9] betont, dass PulseAudio die Anwendungen von der tatsächlichen Hardware trennt, durch eine Abstraktion dieser. Allerdings wird versucht den Anwendungen mehr Einfluss auf die Verarbeitung der Streams zu geben. Es wird auch darauf hingeweisen, dass viele Anwendungen und Bibliothen PulseAudio direkt unterstützen.

2.5. DirectShow

Mit Hilfe von DirectShow ist es möglich Anwendungen für Soundverarbeitung zu implementieren. Mit DierectShow können viele bereits existierende Filter und Codecs verwendet werden, welche Graphen basiert zusammen geschaltet werden können [10]. Es ist zu erwarten, dass hier der WinowsMixer zwischen DierectShow und Soundausgabe geschaltet ist und somit eventuell hohe Latenzzeiten oder geringe Sampletreue auftreten können.

Eine direkt auf DirectShow aufbauende Software kann schwer oder gar nicht auf andere Betriebssysteme portiert werden. Das hier in der Arbeit verwendete ASIO ist zwar auch Windows gebunden, allerdings verhindert das Design des hier implementierten Soundservers nicht, dass das Ausgabemodul durch ein anderes ersetzt wird. Da ansonsten alle anderen verwendeten Bibliotheken auf mehreren Plattformen lauffähig sind, kann der Soundserver mit einem angepassten Backend auch auf Linux portiert werden. Da es einen experimentelen Wine Treiber für Linux gibt [6], wäre der Soundserver eventuell auch auf dem momentanen Stand unter Linux lauffähig.

Flexibler Soundserver für clusterbasierte VR

3

In diesem Kapitel wird auf die Grundlagen und die Architektur des implementierten Soundservers eingegangen. Der Soundserver ist in mehrere Module aufgeteilt, welche für die jeweilige Funktionalität zuständig sind. Solange die Schnittstellen der Module beibehalten oder entsprechend angepasst werden, sind die einzelnen Module weitgehend unabhängig voneinander modifizierbar oder auch austauschbar. Module bzw. Modulgruppen (z.B. FilterChain), welche am Fluss der Soundverarbeitung beteiligt sind, arbeiten in eigenständigen vom Hauptthread unabhängigen Threads. Die Kommunikation über Ethernet ordnet sowohl dem TCP/IP-Server als auch jedem einzelnen verbundenen Client einen individuellen Thread zu (Abschnitt 3.12 auf Seite 58 – Comm-Interface). Somit arbeitet der Soundserver hoch parallel. Bei großer Anzahl gleichzeitig abgespielter Sounds oder verbundenen Clients profitiert dieser deutlich von den modernen Multicore-CPUs (Anhang A auf Seite 67 – Benchmarks).

Die Module der Soundverarbeitung werden über Buffer miteinander verbunden. Allerdings Module bzw. Modulgruppen, die über eigenen Thread verfügen, benutzen diese Buffer um auf die vorangehende und nachfolgende Module zu warten, also um sich mit diesen zu synchronisieren. Somit haben die Buffer abgesehen von ihrer eigentlichen Funktion, Daten zwischen zu speichern, die Aufgabe als Verbindungs- und Synchronisations-Glieder zu dienen und werden somit zweckdienlich als Sync-Buffer bezeichnet (Abschnitt 3.6 auf Seite 24 – Sync-Buffer).

Im Folgenden wird eine Übersicht über die Architektur des Soundserver gegeben. Es wird vorausgesetzt, dass der Leser gewisses Grundverständnis über Abtastung und Quantisierung kontinuierlicher Signale mitbringt, welche hier nur knapp angesprochen werden (Abschnitt 3.2 auf Seite 16 – Grundlagen). Verständnis über Fouriertransformation wird nicht vorausgesetzt, da hier verwendete Sachverhalte weitgehend anschaulich erklärt werden (Abschnitt 3.8.1 auf Seite 33 – FFT). Ein Grundwissen über Fouriertransformation ist aber bei diesem komplexen Thema dennoch hilfreich. Viele Graphiken enthalten Elemente aus der objektorientierten Programmierung, setzen aber keine tiefe Kenntnis dieser Materie voraus. Vielmehr soll damit das Verständnis über die jeweilige Komponente und deren Schnittstelle zu Nachbarkomponenten gefördert werden.

3.1. Soundserver Übersicht

Die Abbildung 3.1 stellt die grundlegenden Komponenten des Soundservers in einer Übersicht dar. Die Anordnung der Komponenten ist weitgehend in drei Zeilen gegliedert. Komponenten der ersten Zeile gehören zu den soundverarbeitenden Einheiten und sind durchgängig miteinander verbunden. Komponenten der zweiten Zeile stellen Steuerstrukturen

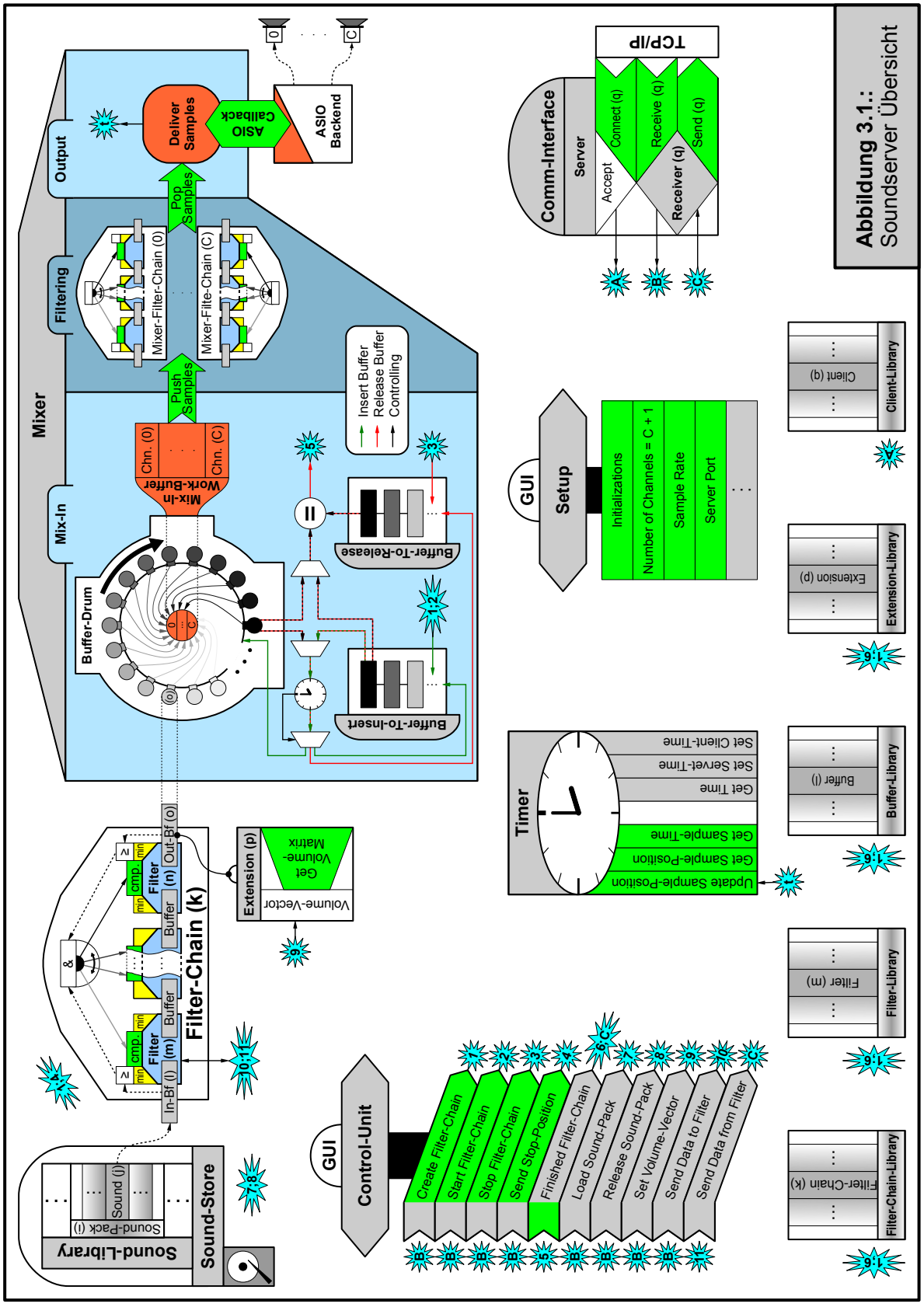
Abbildung 3.1.:
Soundserver Übersicht
Mixer
Output
Filtering
Mix-In
Deliver Samples
ASIO Callback
ASIO Backend
Pop Samples
Push Samples
Mixer-Filter-Chain (0)
Mixer-Filte-Chain (C)
Insert Buffer
Release Buffer
Controlling
Mix-In Work-Buffer
Chn. (0)
Chn. (C)
Buffer-Drum
Buffer-To-Release
Buffer-To-Insert
Comm-Interface
Server
Accept
Connect (q)
Receive (q)
Receiver (q)
Send (q)
TCP/IP
GUI
Setup
Initializations
Number of Channels = C + 1
Sample Rate
Server Port
Client-Library
Client (q)
Extension-Library
Extension (p)
Timer
Set Client-Time
Set Servet-Time
Get Time
Get Sample-Time
Get Sample-Position
Update Sample-Position
Buffer-Library
Buffer (i)
Filter-Library
Filter (m)
Filter-Chain-Library
Filter-Chain (k)
Control-Unit
Create Filter-Chain
Start Filter-Chain
Stop Filter-Chain
Send Stop-Position
Finished Filter-Chain
Load Sound-Pack
Release Sound-Pack
Set Volume-Vector
Send Data to Filter
Send Data from Filter
Filter-Chain (k)
Extension (p)
Get Volume-Matrix
Volume-Vector
In-Bf (i)
Filter (m)
Buffer
Filter (n)
Out-Bf (o)
Sound-Store
Sound-Library
Sound-Pack (i)
Sound (j)

und Schnittstellen zur Außenwelt, für die Komponenten der ersten Zeile, dar. Die letzte Zeile widmet sich den Verwaltungseinheiten, welche für die zur Laufzeit zu erstellenden und freizugebenden dynamischen Objekte zuständig sind. Die beschrifteten Sterne zeigen, durch eindeutigen Inhalt und zugehörigen Ursprungspfeilgrafiken, Einflussnahme einzelner Komponenten auf andere. Grau hinterlegte Felder beim Timer und Ein- bzw. Austritts-Felder der Control-Unit zeigen noch nicht vorhandene, verwendbare oder noch außerhalb der Spezifikation stattfindenden Funktionalitäten. Hauptsächlich liegt es am fehlendem Gegenpart, insbesondere aufgrund des Comm-Interfaces, also der Ethernet-Schnittstelle. Diese liegt wie gefordert als Entwurf dieser Arbeit bei (Abschnitt 3.12 auf Seite 58 – Comm-Interface). Allerdings wurden für Tests des grundlegenden Laufzeitverhaltens, Grundstrukturen des Comm-Interfaces bereits implementiert. Im Folgenden werden die in der Übersichtsgrafik dargestellten Komponenten kurz beschrieben. Anschließend wird auf die Komponenten und deren Grundlagen näher eingegangen.

Sound-Store ist dafür zuständig Sound-Files von einer externen Quelle in den Arbeitsspeicher zu laden oder zu dekodieren. Weiterhin beinhaltet er die, funktional zu den anderen Libraries in die dritte Zeile gehörende, Sound-Library. Diese verwaltet die geladenen Sound-Packs und stellt den ersten Schritt für jede Soundanfrage dar. (Abschnitt 3.4 auf Seite 22 – Sound-Store; Abschnitt 3.5 auf Seite 23 – Libraries)

Filter-Chain ist ein filtereinschließendes Konstrukt, welches im eigenständigem Thread die Berechnungen der Filter anstößt und die Randbedingungen überwacht. Sowohl die Filter an sich als auch die Filter-Chain verbinden sich untereinander und zu anderen Nachbareinheiten über sogenannte Sync-Buffer. (Abschnitt 3.7 auf Seite 30 – Filter-Chain; Abschnitt 3.8 auf Seite 31 – Filter; Abschnitt 3.6 auf Seite 24 – Sync-Buffer)

Extension ist eine Erweiterung für Sync-Buffer. Extension – Volume-Vector rüstet die Sync-Buffer mit der Volume-Vektor-Verarbeitung auf, welche für den Mixer benötigt wird. (Abschnitt 3.6.5 auf Seite 29 – Extension)

Mixer mischt die gefilterten Sounds, unter Berücksichtigung der Volume-Vektoren, in die Ausgangskanäle. Der Mixer arbeitet in drei Stufen. Die erste Stufe, der Mix-In, ist für das eigentliche Mischen der Sounds zuständig. Stufe zwei filtert die Ausgangskanäle um z.B. durch Delays die verschiedenen Entfernungen zwischen Hörer und Lautsprechern auszugleichen. Die letzte Stufe stellt eine Schnittstelle für ein Soundausgabemodul bereit und informiert den Timer über aktuelle Sampleposition und Zeit. Zu erwähnen ist, dass die zu mischenden Sounds als Ausgangs-Sync-Buffer der entsprechenden Filter-Chain angemeldet werden. Sync-Buffer, die anzumelden sind, werden in die Buffer-To-Insert Warteschlange abgelegt, hingegen Sync-Buffer, die wieder abgemeldet werden sollen, kommen in die Buffer-To-Release Warteschlange. Sync-Buffer die aktuell gemischt werden liegen in einem Ring, zweckmäßig Buffer-Drum genannt. Die Verwaltung der Warteschlangen und des Rings übernimmt die Mix-In Logik. (Abschnitt 3.9 auf Seite 48 – Mixer)

ASIO-Backend initialisiert, verwaltet und füttert den ASIO-Sound-Treiber. Durch ASIO-Callbacks werden dem Mixer die gemischten Sounddaten entnommen. (Abschnitt 3.10 auf Seite 54 – ASIO-Backend)

Control-Unit stellt eine abstrakte Schnittstelle zur Soundverarbeitung bereit. Sie bedient sowohl sekundäre Komponenten, wie das Comm-Interface, als auch die Außenwelt über die GUI [11]. Primäre Aufgabe der Control-Unit ist, aus wenigen Angaben fertige Filter-Chains zu stricken, diese am Mixer an und abzumelden und sich anschließend um die Entsorgung nicht mehr benötigter Filter-Chains zu kümmern. Sekundär soll sie sich um alle anderen modulübergreifenden internen und externen Anfragen kümmern. Die Control-Unit stellt ein programmiertechnisches Werkzeug dar und wird in dieser Arbeit nicht weiter behandelt, da die grundsätzlichen Auswirkungen dieser Einheit direkt aus der Übersichtsabbildung 3.1 ersichtlich sind.

Timer ist die Anlaufstelle für alle zeitbezogenen Anfragen. Unter anderem um jeder Sampleposition einen Zeitstempel zu zuordnen und umgekehrt. Die Synchronisation zwischen Server und Clients soll der Timer in der Endfassung entweder direkt übernehmen oder entsprechende noch zur Verfügung zu stellende Synchronisations-Werkzeuge benutzen. (Abschnitt 3.11 auf Seite 56 – Timer)

Setup ist für die Initialisierung der Anwendung zuständig, welche über eine Konfigurations-Datei gesteuert wird. Weiterhin stellt das Setup globale Eigenschaften und Methoden für Umgebungsinformationen und Einstellungen bereit. Hier kann über die GUI [11] Einflussnahme auf die Einstellungen und die Konfigurations-Datei ermöglicht werden. Aufgrund den trivialen Aufgaben, wird auf diese Komponente auch nicht weiter eingegangen.

Comm-Interface soll die TCP/IP Schnittstelle zu den Clients sein und kann auch zur Fernsteuerung des Soundservers ausgebaut werden. (Abschnitt 3.12 auf Seite 58 – Comm-Interface)

Libraries , Sound-Library eingeschlossen, sind Verwaltungskonstrukte der entsprechenden Objekte. Sie sind für die Vergabe eindeutiger IDs, Zugriffsmöglichkeiten, Speicherung oder Erstellung der Objekte und dessen Freigabe bzw. Zerstörung zuständig. (Abschnitt 3.5 auf Seite 23 – Libraries)

3.2. Grundlagen

Die Abbildung 3.2 links zeigt das Ergebnis einer möglichen Abtastung eines kontinuierlichen Signals. Das kontinuierliche Signal ist als graue Kurve und das Abtastergebnis als rote Stufenfunktion dargestellt. Die Quelle [12] kann als einstieg in die Abtastthematik verwendet werden und bietet weiterführende Literaturhinweise.

Abgespeicherte Sound-Dateien basieren für gewöhnlich auf einer diskreten äquidistanten Zeitabtastung, wie auf der linken Seite der Abbildung zu sehen ist. Die zeitliche Auflösung der Abtastung wird als Abtast- oder Sample-Rate bezeichnet. Übliche Werte für Musik und Filmton sind 44,1 kHz und 48 kHz. Die Samplerate des Soundserver kann auf die vom ASIO-Treiber unterstützten Sampleraten frei eingestellt werden und arbeitet standardmäßig mit 48 kHz. An dieser Stelle sei auf das Nyquist-Shannon-Abtasttheorem hingewiesen. Dieses besagt, dass um ein abgetastetes Signal wieder in das ursprüngliche Signal beliebig genau

approximieren zu können, die Abtastfrequenz mindestens doppelt so hoch sein muss wie die höchste vorkommende Frequenz des Ursprungssignals [13]. Dieser Sachverhalt spielt bei der Frequenzfilterung eine Zentrale Rolle und wird wieder im Abschnitt 3.8.1 auf Seite 33 – FFT aufgegriffen. Somit trägt ein mit 48 kHz abgetastete Signal effektiv maximal 24 kHz an nutzbarer Bandbreite und schöpft das menschliche Hörvermögen mehr als aus.

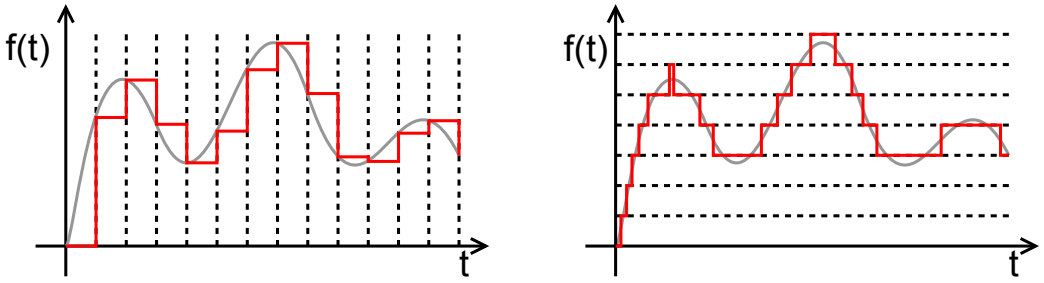

Abbildung 3.2.: Abtastung [12] links und Quantisierung [14] rechts

Die Quantisierung hingegen ordnet jedem kontinuierlichen Abtastwert einen diskreten Wert zu, siehe hierzu Abbildung 3.2 rechts. Somit kann das ursprünglich kontinuierliche Signal als ein Datenstrom kodiert oder als sogenanntes PCM (Puls-Code-Modulation) unkodiert digital verarbeitet, gespeichert oder durch Rückwandlung wiedergegeben werden. Als Einstieg in die Quantisierung kann die Quelle [14] verwendet werden. Für die Quantisierung der Samples gibt es viele Formate. Das gängigste Format ist 16 Bit Integer, welches unter andern bei Audio-CDs verwendet wird. Der Soundserver arbeitet intern mit 32 Bit Float und kann diese nativ aus PCM-WAVE Dateien laden. Im Gegensatz zu 16 Bit Integer ist bei 32 bit Float der Ausgleich von Quantisierungsfehlern, durch Dithering (siehe [15]) und ähnlichem, nicht notwendig, da Fließkommazahlen auch kleinere Werte und Unterschiede besser erfassen. Hinzu kommt, dass man mit Float bei vielen Zwischenrechen- bzw. Filterschritten mit einem genaueren Ergebnis rechnen kann, insbesondere aufgrund kleinerer Rundungsfehler.

3.3. Grundstrukturen

In diesem Abschnitt werden implementierte Grundstrukturen angesprochen, welche keine direkten Bestandteile der verwendeten Programmiersprache, also C++, sind. Ziel war es schnelle, nicht allzu überladene, dennoch flexible Werkzeuge für viele Komponenten des Soundservers zur Verfügung zu haben. Geschwindigkeit wurde erreicht durch implizite oder explizite Vorgaben von Wertebereichen, welche dann effiziente Rechenschritte zuließen. Weiterhin wurde gezielt auf manche Überprüfungen von Zugriffsverletzungen verzichtet, da ansonsten insbesondere z.B. beim samplegenauen Zugriff, die Performance drastisch reduziert werden würde. Flexibilität wurde durch Design und Ausnutzung der Template- und Vererbungs-Funktionalität von C++ erreicht. Weiterhin wurde auf notwendige Synchronisation von konkurrierenden Threads geachtet oder wo es nicht unbedingt notwendig war, wurde aus Performance Gründen unter Beachtung dieses Umstandes auf Synchronisation verzichtet. Teilweiser Verzicht auf Zugriffssicherheit und unnötige Synchronisation sollte

bei gewissenhafter Programmierung und Kenntnis der Rahmenbedingungen keine großen Schwierigkeiten bereiten.

3.3.1. Dynamic-Array

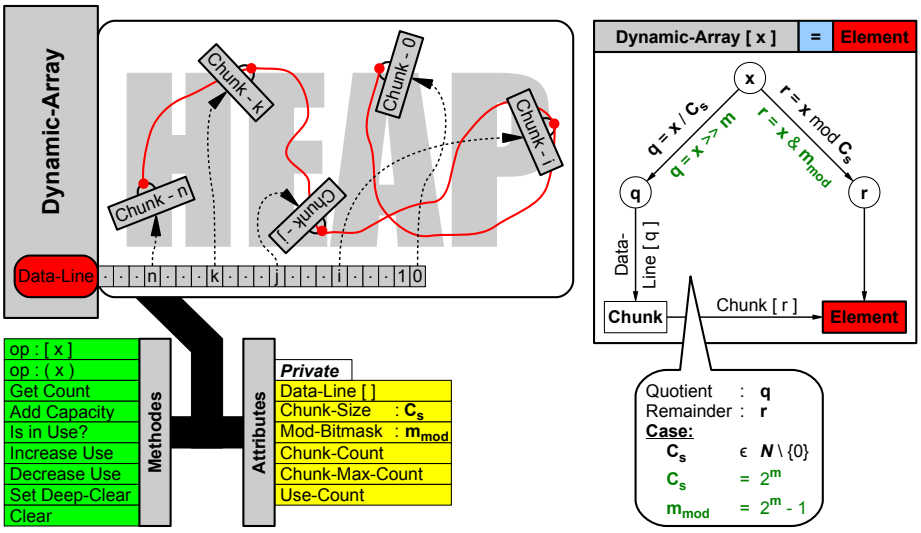

Abbildung 3.3.: Dynamic-Array

Die erste vorgestellte Grundstruktur ist das Dynamic-Array. Die Grundideen sind in Abbildung 3.3 skizziert. Es handelt sich um ein dynamisches Array, welches während der Laufzeit, nach seinem Erstellen, seine Größe ändern kann. Das Design sieht vor, dass eine Änderung der Größe, keine Referenzen auf die Datenelemente entzieht. Dies wird, wie im Bild anhand der roten Kurve oder bildlich gesprochen der roten Schnur erreicht. Diese Schnur, als Data-Line bezeichnet, ist eine logische Zwischenschicht, ein von außen transparentes Array auf Referenzen. Die eigentlichen Daten des Dynamic-Arrays liegen in gleich großen Blöcken, Chunks wild über die Halde verteilt. Die Data-Line spannt diese Blöcke zu einem fortlaufenden virtuellem Array auf, d.h. bildlich wird die Schnur in richtiger Reihenfolge durch gefädelt und gespannt. An sich ist die Data-Line ein gewöhnliches, begrenztes Referenz-Array. Dieses muss beim Erreichen der Kapazität vergrößert werden, was durch Neuaufbau des Referenz-Arrays bewerkstelligt wird. Die effektive Größe des Dynamic-Arrays ist nur durch den endlichen Wertebereich der Indizierung und der Halden-Verwaltung d.h. Kapazität beschränkt.

Je nach Größe der Datenblöcke und gewähltem Erweiterungsverhalten des Referenz-Arrays, geschieht der Neuaufbau vergleichsweise sehr selten. Dieser Neuaufbau des Data-Line-Arrays, ist für konkurrierende, reine lesende Vorgänge transparent. Für Schreibvorgänge innerhalb der aktuellen Größe gilt dies ebenfalls. Gleichzeitige Schreib- und Lesevorgänge auf dieselbe Zelle müssen von außen synchronisiert werden, ebenso wie Größenänderung und gleichzeitige Lese- oder Schreibvorgänge allgemein.

Das Zuordnen des richtigen Elements zu einem Index, ist rechts in der Abbildung 3.3 skizziert. Der schwarze Fall stellt den allgemeinen Fall dar, die Idee an sich. Der grüne Fall stellt die Hauptidee der Laufzeitoptimierung dar. Da die Datenblöcke also Chunks alle gleichgroß gewählt sind, ist das Auffinden des gesuchten Elements recht einfach. Den richtigen Index der Data-Line erhält man durch ganzzahlige Division von Eingabe-Index durch die Größe der Datenblöcke. Das Element des referenzierten Daten-Blocks mit dem Index vom Rest der Division entspricht schließlich dem gesuchten Element. Wählt man allerdings die Blockgröße so, dass sie einer Zweierpotenz entspricht, lässt sich die Division und Findung des Restes zu einer bitweisen Schiebe- und einer bitweisen Und-Operation reduzieren. Bitweises Schieben nach rechts um m Stellen entspricht einer Division von 2^m, was den ersten Schritt der Elementauffindung bedeutet. Weiterhin stellt die Zahl 2^m binär eine Eins mit m Nullen dar. Durch einmalige Subtraktion um, eins erhält man binär m Einsen. Somit kann die Modulo-Operation, um den Divisionsrest zu finden, auf eine bitweise Und-Operation reduziert werden.

Es wurden zwei Zugriffsoperatoren implementiert, einmal der Standard-Array-Operator "[ ]" und der Funktions-Operator "( )" . Der Standard-Operator verhält sich nach außen hin wie jedes andere Array auch. Der Funktions-Operator wurde mit derselben Funktionalität implementiert, allerdings erweitert er zusätzlich die Array-Größe bei Überschreitung der aktuellen Größe automatisch. Dieses Verhalten erfordert einen höheren Aufwand und spiegelt sich entsprechend in der Performance wieder.

Von den ersten Versuchen mit dem allgemeinen Fall bis hin zur aktuellen Lösung konnte die Geschwindigkeit etwa verdreifacht werden, allerdings nicht nur ausschließlich durch diese Optimierung. Im Anhang A auf Seite 67 – Benchmarks wird die Performance unter anderem mit dem C++ Float Array verglichen. Der limitierende Faktor ist hier die verwendete For-Schleife, allerdings sollte jede kleinste Berechnung, die auf die Daten zurückgreift, unverhältnismäßig höheren Aufwand mit sich bringen. Somit war das Dynamic-Array je nach Zugriffsart, soweit überhaupt mit einem statischen Array vergleichbar, effektiv gleich schnell bis etwa doppelt so langsam, wie das C++ Float Array. Bei komplexeren Berechnungen sollte sich die Geschwindigkeit noch weiter relativieren.

Das Dynamic-Array wurde ursprünglich entwickelt primär, um beliebig große Sound-Daten im Arbeitsspeicher zu beherbergen und auch für den Einsatz als Buffer in den Verarbeitungsstufen. Allerdings wurde der Einsatz als Buffer wieder verworfen, denn es gab kein Vorteil dieses als Buffer einzusetzen, die einmal erstellt eine feste Größe haben. Allerdings abgesehen von dem Einsatz als Sound-Speicher, kommen die Dynamic-Arrays direkt und indirekt an vielen Stellen des Soundserver zum Einsatz. In Filter-Chains und im Mixer werden sie durch ihre Template-Natur als Arrays auf Filter, Sync-Buffer und mehr eingesetzt. Als Template und mittels Vererbung wurde die Funktionalität der Dynamic-Arrays erweitert und sie kommen als Libraries für Sounds, Filter-Chains usw. zum Einsatz (Abschnitt 3.5 auf Seite 23 – Libraries). Auch wenn die Entwicklung und Optimierung des Dynamic-Arrays hätte vermieden werden können, durch Verwendung von vorgefertigten Bibliotheken, konnte durch dessen Entwicklung nicht nur ein angepasstes und flexibles Werkzeug erstellt werden, sondern wurde auch wertvolles Know-How gewonnen, um den Soundserver unter laufzeitkritischen Aspekten zu implementieren. Tatsächlich wurde der

Versuch unternommen ein vorgefertigtes komplexes dynamische Array für die genannten Zwecke zu verwenden. Allerdings hatte sich dieses bei einer Größenänderung oft komplett neu aufgebaut, was sehr laufzeitkritisch sein kann und zudem während dem Neuaufbau etwa doppelten Speicher verbrauchte. Hinzu kam, dass im reininitialisierungs Fall Referenzen entzogen wurden und somit ein Einsatz in einer Library erschwert wurde.

Links unten in der Abbildung 3.3 ist eine Liste von Methoden und Attributen aufzufinden, die zum einen die Schnittstelle des Dynamic-Arrays aufzeigen soll und zum anderen wichtige interne, für die Grafik oder Verständnis relevante, Daten enthält. Diese Liste ist an die objektorientierte Programmierung angelehnt und wird sich weiterhin in dieser Arbeit bei vielen Abbildungen wieder finden. Diese ist als eine Verständnishilfe und zugleich als Klartext-Dokumentation der Schnittstelle gedacht. Auf diesen Methoden- und Attributlisten wird im weiteren nur bei Bedarf eingegangen.

3.3.2. Queue

Die nächste Grundstruktur ist die in Abbildung 3.4 dargestellte doppelt verkettete Liste. Diese als Queue bezeichnete Liste wird im Mixer zweifach als Warteschlange verwendet, einmal als Buffer-To-Insert- und dann als Buffer-To-Release-Warteschlange (siehe Abbildung 3.1 auf Seite 14). Desweiteren wird die Queue auch in den Libraries verwendet (Abschnitt 3.5 auf Seite 23 – Libraries). Sie hat in den Libraries die Aufgabe freigewordene IDs zu katalogisieren, um erneutes freigeben dieser IDs zu ermöglichen.

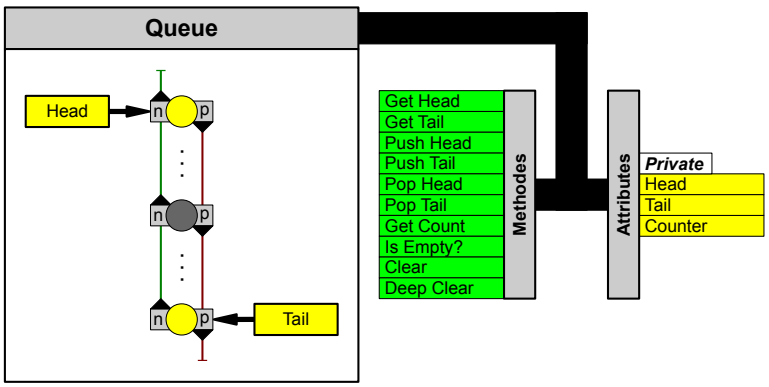

Abbildung 3.4.: Queue

Die Ausrichtung der Bezeichnungen dieser Queue ist an das FIFO-Prinzip angelehnt, d.h. das erste Element mit dem der Queue-Head gefüttert wird, ist auch das erste Element, das am Queue-Tail wieder aufgefangen werden kann. Somit bedeuten die internen Verweise Next (n), jedes gespeicherten Elements, dass sie auf das chronologisch nächste am Head eingefügte Element verweisen. Entsprechend umgekehrt gilt das für die internen Verweise Previous (p).

Abgesehen von den internen an FIFO-Prinzip orientierten Bezeichnungen, ist die Queue in der Verwendung völlig frei. So können Elemente ohne Einschränkung, sowohl am Head als auch am Tail, eingefügt oder entnommen, also ge-pushed bzw. ge-popped, werden. Somit ist die Queue unter anderem auch als Stack, d.h. LIFO-Prinzip, einsetzbar.

Da auch die Queue als Template implementiert wurde, kann die Queue als Warteschlange für nahezu alle Arten von Elementen spezialisiert werden. Für die Entnahme von Elementen wird ein Counter zur Verfügung gestellt, dieser sollte bei der Entnahme von Elementen überprüft werden. Ansonsten quittiert eine verhungerte Schlange mit einer Exception den Dienst. Falls ein Anwendungseinsatz eine parallele Entnahme von beiden Enden vorsieht, sollte beachtet werden, dass zwischen Counter-Abfrage und Elemententnahme, das Element eventuell bereits von der anderen Seite entnommen wurde. Folglich ist hier eine externe Synchronisation notwendig, um dem Verhungern der Schlange vorzubeugen.

3.3.3. Ring

Der in Abbildung 3.5 dargestellte Ring ist die letzte entwickelte Grundstruktur. Abgesehen von der Funktionalität eines Rings ist die Implementierung des Rings an die Queue angelehnt und kann somit auch beliebige Arten von Elementen beherbergen. Der als Buffer-Drum bezeichnete Ring hat eine essentielle Aufgabe in der Mix-In-Stuffe des Mixers (siehe Abbildung 3.1 auf Seite 14). Siehe hierzu auch den Abschnitt 3.9 auf Seite 48 – Mixer.

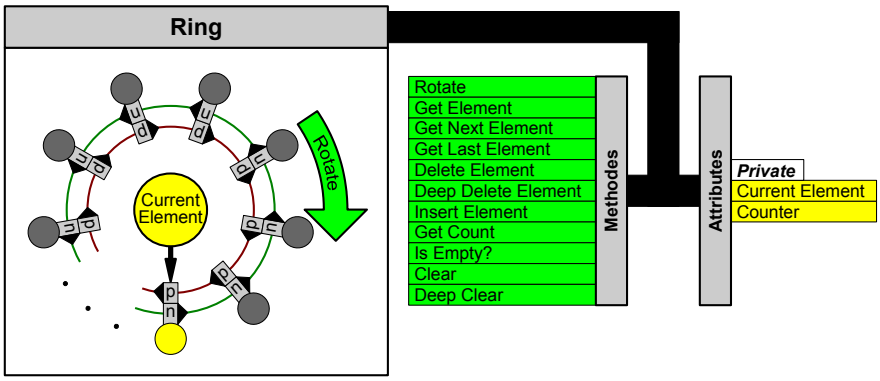

Abbildung 3.5.: Ring

Das Ringkonstrukt zeigt stets auf ein aktuelles oder leeres Element. Durch Rotieren zeigt das Konstrukt auf das nächste in der Ringliste gespeicherte Element. Aus der Ringliste kann das aktuelle, vorherige oder nächste Element abgefragt werden. Beim Entnehmen von Elementen, kann allerdings das aktuelle oder das vorherige Element entnommen werden. Mit “Deep Delete Element“ ist gemeint, dass das gespeicherte Element eine Referenz auf ein Objekt ist und dieses beim Löschen mit zerstört wird. Beim Einfügen eines neuen Elements kann ausgewählt werden, ob dieses an der aktuellen Spitze des Rings oder an der letzten Stelle stehen soll.

3.3.4. LogBOX

Im Zuge der Implementierung des ASIO-Backends wurde eine Möglichkeit gesucht, die vielen anfallenden ASIO-Rückmeldungen schnell und übersichtlich für das Debugging darzustellen oder zu sichern. Mit Hilfe der verwendeten GUI-Bibliothek [11] wurde die sogenannte "LogBOX" in einem Texteditor ähnlichen Fenster realisiert. Anschließend wurde die LogBOX für den allgemeinen Einsatz global in den Soundserver eingebunden. So können laufzeitbezogene Informationen direkt auf dem Bildschirm dargestellt werden, was bei einer Fehlersuche im Quellcode und beim Setup des Servers oder seiner Umgebung hilfreich sein kann. Durch anpassen der Einstellungen kann ferner erreicht werden, dass der Soundserver Log-Dateien abspeichert. Da die GUI-Umgebung "allergisch" auf Synchronisationsversuche reagiert, wurde auf Synchronisation verzichtet, aus diesem Grund kann es beim parallelen Schreiben in die LogBOX zu Darstellungs- oder Inhalts-Anomalien kommen, eventuell sind Crashs nicht auszuschließen. Als Abhilfe könnte hier eine Nachrichtenwarteschlange zwischen geschaltet werden, basierend z.B. auf der hier in Abschnitt 3.3.2 auf Seite 20 – Queue vorgestellten Warteschlange.

3.4. Sound-Store

Den ersten Schritt der Soundverarbeitung stellt der Sound-Store dar. Wie in Abbildung 3.6 skizziert, ist er dafür zuständig mit externen Sound-Daten umzugehen. Durch Angabe einer speziell formatierten Text-Datei wird der Sound-Store dazu veranlasst, das in der Datei beschriebene, komplette Sound-Pack in den Arbeitsspeicher zu Laden oder zu Dekodieren. Auf dem momentanen Stand unterstützt der Sound-Store zwei Audio-Formate. Unterstützt wird das verlustbehaftete OGG-Vorbis Format und diverse unkomprimierte WAVE-PCM Formate. OGG-Vorbis wird mit Hilfe des zugehörigen Open-Source-SDK in den Arbeitsspeicher dekodiert [16]. Die WAVE-Dateien werden nach Auswertung des RIFF-Headers direkt in den Arbeitsspeicher geladen oder umkonvertiert [17]; [18].

Der implementierte WAVE-Loader unterstützt die gängigen und theoretisch möglichen PCM-Integer Formate, die von 8 Bit bis hin zu 64 Bit Präzision reichen. Desweiteren werden 32 Bit und 64 Bit Float-Formate unterstützt. Die Sample-Formate werden alle in das intern verwendete 32 Bit Float-Format konvertiert [19]. Da bei der Konvertierung der Bit-Tiefen, kein Ausgleich von Quantisierungsfehlern stattfindet, ist das 32 Bit Float-Format das optimale Sample-Format für den Soundserver. In diesem Fall können die Sounds durch Vorverarbeitung noch weiter für den Einsatz klanglich optimiert werden, ohne dass beim Laden Konvertierungsfehler verursacht werden.

Jeder einzelne geladene Soundkanal des gesamten Sound-Packs wird jeweils in einem separaten Dynmic-Array abgelegt (Abschnitt 3.3.1 auf Seite 18 – Dynamic-Array). Die nun als dynamisches Array vorliegenden Mono-Sounds werden pro Sound-Pack in eine Separate Library mit eindeutigen IDs gepackt, in Abbildung 3.6 als Sound-Pack bezeichnet. Dabei wird darauf geachtet, dass die einzelnen Kanäle von Mehrkanal-Sound-Dateien einen fortlaufenden zusammengehörenden ID-Block bekommen. Die Absicht dieses Verhaltens

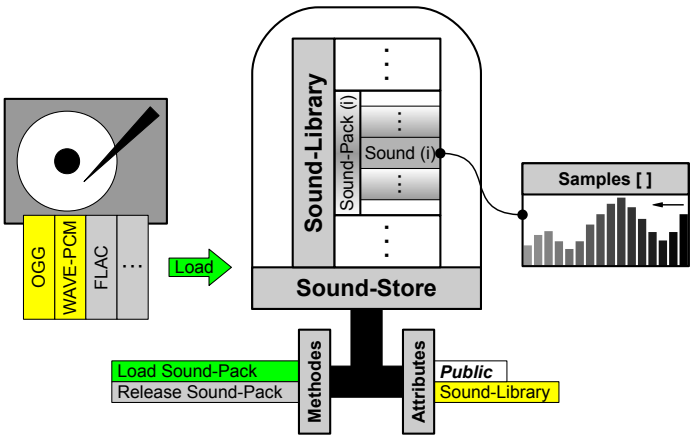

Abbildung 3.6.: Sound-Store

ist der, dass um ein Mehrkanal-Sound abzuspielen, es reichen soll den ersten Kanal zu referenzieren, um daraufhin folgende Kanäle durch entsprechende Offsets zu erhalten. D.h. um ein Mehrkanal-Sound abzuspielen reicht es seine ID und die Kanalanzahl im Sound-Pack zu kennen, anstatt z.B. bei 8-Kanal-Sound mit 8 verschiedenen IDs herum zu hantieren.

Die erstellten Sound-Pack-Libraries werden hingegen selber in einer umschließenden Library zusammengefasst. Diese sind in der Abbildung 3.6 als Sound-Library bezeichnet. Für die Soundverarbeitung ist die Sound-Library eine zentrale Komponente und deswegen über den Sound-Store hinweg direkt verwendbar. Auf einen Sound kann über eine zweifache Indizierung zugegriffen werden, ähnlich einem 2D-Array, also im Prinzip mit Sound-Library[Sound-Pack-ID][Sound-ID]. Der Zugriff auf einzelne Samples geschieht über eine weitere Indizierungsstufe, also im Prinzip als 3D-Array. Allerdings jeden einzelnen Sample so zu referenzieren, würde die Performance sinnlos reduzieren. Um Performance zu sichern reicht es eine Referenz des abzuspielenden Sounds bei der Sound-Library anzufragen und diese dann als ein 1D-Array zu verwenden.

3.5. Libraries

Dieser Abschnitt knüpft direkt am Abschnitt 3.4 auf der vorherigen Seite – Sound-Store an. Die Funktionalität der Sound-Library wurde auch für andere Libraries übernommen und ist im Grunde bis auf individuelle Anpassungen identisch. Die Schnittstelle wurde in Abbildung 3.7 skizziert.

Wie im Abschnitt 3.3.1 auf Seite 18 – Dynamic-Array beschrieben, basieren die Libraries auf dem Dynamic-Array. Die durch "Allocate next free Element" eingefügte Elemente, werden als Referenz in dem zugrundeliegenden Dynamic-Array abgelegt. Es wird immer stets der nächste Freigewordene oder noch unbenutzte Index für neue Elemente reserviert. Der nächste freie Index wird der Warteschlange "Free Elements" entnommen. Wird bei einer

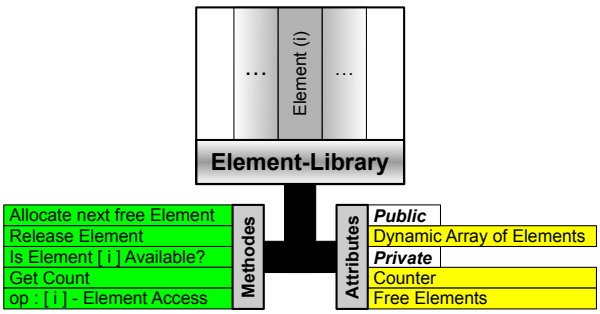

Abbildung 3.7.: Libraries

Entnahme die Schlange leer, so wird anhand des Counters der nächste noch nicht im Array enthaltene Index in diese Schlange eingefügt. Somit steht immer ein freies Element für die nächste Reservierung bereit. Wird ein Element durch "Release Element" freigegeben, so wird dessen ID in die "Free Elements" Warteschlange eingefügt. Da immer nur unbenutzte Indizes in der "Allocate next free Element" Warteschlange liegen, erhält man bei der Entnahme eines Index eine eindeutige ID. Die Libraries dienen somit als Verwaltungsstrukturen für Filter-Chains, Filter, Sync-Buffer, deren Extensions und wie schon besprochen für Sound-Daten (Abschnitt 3.4 auf Seite 22 – Sound-Store). Im Zuge des Ausbaus der Netzwerk-Funktionalität wird, wie in Übersichtsabbildung 3.1 auf Seite 14, eine weitere Library hinzu kommen. Eine um die angemeldeten Clients zu verwalten.

3.6. Sync-Buffer

Die Sync-Buffer sind die alles verbindende Einheiten, der internen Soundverarbeitung. Mit einem gemeinsamen Interface gibt es für verschiedene Aufgaben jeweils angepasste Varianten dieser Sync-Buffer. Auf die Varianten wird im Abschnitt 3.6.4 auf Seite 28 – Varianten näher eingegangen. Sync-Buffer haben drei wichtige Aufgaben. Zum einen dienen sie für die soundverarbeitende Recheneinheiten, im Allgemeinen Filter, als Zwischenspeicher. Zum anderen dienen sie als Synchronisationsmittel zwischen dem Eingangs- und Ausgangs-Thread, also z.B. für Filter-Chain und Mix-In. Und zuletzt tragen und verwalten sie Informationen für das Timing der Sounds. Abbildung 3.8 fasst die funktionalen Ideen der Sync-Buffer zusammen. Hier wird auch die vorgesehene Zusammenarbeitsweise der zu verbindenden Komponenten aufgezeigt.

3.6.1. Buffering

Der obere Teil der Abbildung widmet sich dem Buffering-Fall. Der Kern ist hier ein gewöhnliches Array, welches von einem Zugriffs-Model umschloßen wird. Tatsächlich greifen die Filter aus Performance Gründen direkt auf dieses Array zu, allerdings sollen sie nur auf die Bereiche zugreifen, die ihnen vom Zugriffsmodel als sicher anvertraut werden. Im Prinzip

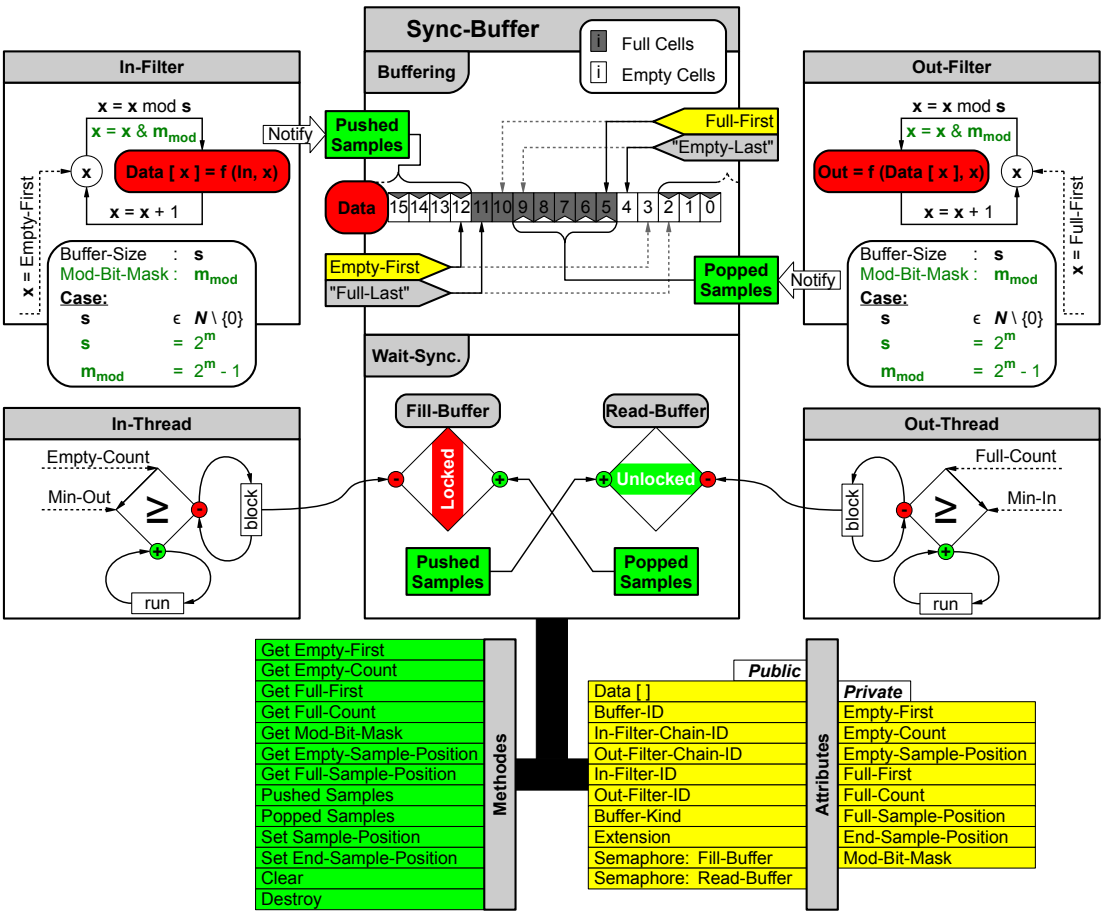

Abbildung 3.8.: Sync-Buffer

handelt es sich um einen Ring-Puffer fester Größe. Graue Felder des Arrays in der Abbildung stellen gefüllte Zellen dar, also Samples die vom Filter am Ausgang gelesen werden können. Weiße Felder symbolisieren leere Zellen, als solche die vom Filter am Eingang befüllt werden können. Die grauen und weißen Dreiecke (“Pfeile“) deuten den nächsten Zustand an, also nach einem Schreibe- (graue Dreiecke) oder Lesevorgang (weiße Dreiecke).

Der lesende Filter am Ausgang kennt die “Full-First“ Buffer-Position, d.h. das erste zu lesende Sample. Durch den Full-Counter kennt er indirekt die “Full-Last“ Buffer-Position. Innerhalb diesen Bereichs kann der Out-Filter völlig sicher Samples auslesen. Der Worstcase wäre hier lediglich, dass der schreibende In-Filter währenddessen weiter Zellen gefüllt hat, ohne dass der Out-Filter es in seinem aktuellen Zyklus mitbekommen hat. Natürlich darf der In-Filter, wenn er einmal Buffer-Positionen als gefüllt markiert hat nicht mehr auf diese zugreifen, was auch kein konstruktiven Sinn machen würde. Ist der Out-Filter in seinem Zyklus mit dem Auslesen eins Buffer-Bereichs fertig, quittiert er das über die Methode “Popped-Samples“ am Sync-Buffer und gibt im gleichen Zug die Anzahl ausgelesener Samples an. Somit werden diese als leere Zellen markiert, d.h “Full-First“ verschoben und die Counter angepasst. Der

Schreibende In-Filter sieht sie dann spätestens in seinem nächsten Zyklus als leere Zellen und kann diese ebenfalls sicher befüllen. Entsprechend gilt das Ganze analog, für den Fall leere Zellen zu füllen und kann anhand der Grafik und der Beschreibung des lesenden Falls nachverfolgt werden.

Betrachtet man z.B. in der Abbildung das Ablaufdiagramm des Out-Filters so wird gezeigt wie die Filter intern auf dem Sync-Buffer arbeiten sollen. Die Ringeigenschaft des Sync-Buffers wird dadurch erreicht, dass beim Überschreiten der Buffer-Grenze der Zugriffs-Index wieder auf Null gesetzt werden soll, um weiter fortfahren zu können, also ein typisches Modulo-Verhalten. Da die Buffer auf die Größen von 2^n beschränkt sind, gelten dieselben Optimierungsmöglichkeiten wie beim Dynamic-Array (Abschnitt 3.3.1 auf Seite 18 – Dynamic-Array). D.h. wie der, in der Sync-Buffer-Abbildung in grün hervorgehobene Fall, wird die teure Modulo-Operation zu einer bitweisen Und-Operation reduziert und kann samplegenau auf den Index x angewendet werden. Um die Größenlimitierung der Sync-Buffer effizient aufzuheben oder im Fall, dass das verwendete Verfahren gar langsamer ist, werden schnelle Hardware Modulo-Zähler gebraucht. Ob diese im Befehlssatz heutiger Prozessoren enthalten sind und zur Verfügung stehen, ist zum Erstellungszeitpunkt dieser Arbeit unbekannt. Je nach Realisierung des Zählers können eventuell Zählstanderhöhungen mit Schrittweiten größer als eins, zum Verlust der Moduloeigenschaft führen. Was beim momentanen Design der Pushed- und Popped-Samples Methoden der Fall wäre.

Ein alternatives Buffering-Verfahren wäre Dubble-Buffering gewesen. Hier kann aber erst wieder nach einem Bufferswitch, im gegenüber liegenden Teil des Buffers, gelesen bzw. geschrieben werden. Das bedeutet der schreibende In-Filter kann die andere Hälfte des Buffers erst wieder beschreiben, wenn der lesende Out-Filter seinen Teil komplett ausgelesen und geswitcht hat. Es gibt Filter die eine gewisse feste Anzahl von Samples lesen bzw. schreiben müssen. Passt hier die Buffergröße nicht exakt, kann ein Bufferswitch nicht durchgeführt werden oder der Filter muss denn Rest zwischenspeichern, um Nachschub bekommen zu können. Benötigen zwei Filter verschiedene feste Buffergrößen, so ist das zusätzliche Zwischenspeichern in den Filtern, selbst durch Größenanpassung der Buffer, nicht zu umgehen. Aufgrund dessen wäre auch ein samplegenaues Mischen der Sounds erschwert worden. Beim ersten Mix-In-Fenster kann es sogar passieren, dass nur ein einziges Sample des neu einzumischenden Sounds in das Mix-In-Fenster einzumischen ist. Mit dem hier verwendeten Verfahren, bleibt Auslese- und Schreibmenge völlig flexibel, solange kein Versuch unternommen wird, über die Kapazitäten hinaus zu lesen oder zu schreiben. Die Sync-Buffer-Größen werden bei der Erstellung der Filter-Chains und der Mixer-Filter-Chains weitgehend automatisch gesetzt. Die minimalen Anforderungen der angrenzenden Filter werden aufaddiert. Liegt die Gesamtanforderung dabei unterhalb der doppelten Größe, der vom ASIO-Backend verwendeten Buffer, so wird die Größe der Sync-Buffer entsprechend aufgestockt. Hiermit soll die Wahrscheinlichkeit erhöht werden, dass zwischen zwei ASIO-Zugriffsfenstern im Durchschnitt an einem Sync-Buffer, je nach Filterart, nicht mehr als einmal gelesen und einmal geschrieben wird.

3.6.2. Synchronization

Die In- und Out-Filter rechnen vorgesehener Weise in verschiedenen Threads. Somit muss z.B. der Thread, der den Out-Filter enthält, warten, bis der Sync-Buffer genug gefüllt wurde, um vom Out-Filter gelesen werden zu können. Dies wird hier als Wait-Synchronization bezeichnet und wird im zweiten Teil der Abbildung 3.8 dargestellt. Hierbei spielen das Pushen und Poppen der Samples im Buffering-Falls und die Semaphoren "Fill-Buffer" und "Read-Buffer" eine zentrale Rolle. Das Puschen und Poppen schaltet überkreuzt die "Read-Buffer" und "Fill-Buffer" Semaphoren frei, was man bildlich auch als Cross-Synchronization bezeichnen könnte. Genauer gesagt, werden z.B. vom Out-Filter die Samples als gelesen markiert, so wird dem schreibenden In-Thread mittels "Popped-Samples" über das Semaphore "Fill-Buffer" implizit signalisiert, dass sich was am Füllstand des Sync-Buffers geändert hat. So wird der schreibende In-Thread wieder reaktiviert, falls er aufgrund von zu vollem Sync-Buffer in den Wartemodus gegangen ist. Im Ablaufdiagramm des Out-Threads ist zu sehen wie dieser selbst in den Warte Modus geht. Zu Beginn jedes Zyklus wird geprüft, ob die geforderte Mindestanzahl an Samples bereits im Sync-Buffer vorliegt. Ist das der Fall, so läuft der Thread seinen Zyklus durch. Andernfalls hängt er sich am "Read-Buffer" Semaphore auf, bis er vom schreibenden In-Thread wieder an diesem Semaphore reaktiviert wird, was den Namen "Hang-Synchronization" verdienen würde. Auch hier gilt analoge Betrachtungsweise des In-Thread-Falls.

Allerdings muss ein einmal freigeschaltetes Semaphore zweimal gesperrt werden, um den eigenen Thread aufzuhängen. Das ist, im ersten Augenblick, eine lästige Tatsache und mit momentanen Mitteln nicht zu ändern. Dennoch löst diese Tatsache sogar das Problem, welches auftritt, falls zwischen der Abfrage des Füllstandes und dem eigenen Aufhängen, eine durch den Konkurrenz-Thread verursachte Füllstandänderung passiert. Dies hätte im Allgemeinen dann eine völligen Blockade zur Folge. Dieser Fall tritt hier aber nicht ein, denn fragt ein Thread den Füllstand mit einem negativen Ergebnis ab, versucht er sich zwar aufzuhängen, was ihm aber erst beim zweiten negativen Anlauf gelingt. Ändert sich jetzt der Füllstand zwischen den Versuchen durch den Gegen-Thread, so wird der zweite Aufhängeversuch erst überhaupt nicht unternommen. Da hier also auf eine Freischaltung immer zwei blockierende Anfragen folgen müssen, bekommt der blockierende Thread spätestens beim zweiten Versuch den geänderten Füllstand mit. Geschieht die Füllstandänderung nach dem Aufhängen, so ist es im ursprünglichem Sinne der Synchronisation und der blockierte Thread wird wieder reaktiviert. Geschieht die Füllstandänderung sogar vor dem ersten Versuch zu blockieren, so ist es ganz im Sinne der Performance, da weniger Synchronisationsschritte durchwandert werden.

Obwohl Critical-Section etwas schnellere Synchronisation bietet, war der Einsatz hier nicht möglich. Die Kern-Idee funktionierte bei diesem Verfahren nicht, d.h. ein aufgehängter Thread konnte nicht mehr durch ein Fremd-Thread reaktiviert werden. Ansonsten brachte bei allen anderen Synchronisationsstellen ein Wechsel von Semaphore auf Critical-Section etwa 30% höhere Leistung. Der Leistungsschub war größtenteils im Mix-In-Thread vorzufinden. Da der Mix-In-Thread bei vielen angemeldeten Sounds bei weitem den größten Aufwand hat, sollte ein theoretischer Wechsel von Semaphore auf Critical-Section keinen nennenswerten,

effektiven Leistungsschub bringen. Hinzu kommt, dass bei Verwendung rechenaufwendiger Filter der Synchronisationsanteil verschwindend gering werden sollte. Besteht der Wunsch die Synchronisation mit wenig Aufwand zu beschleunigen, so kann die verwendete ASIO-Buffer-Größe erhöht werden. Diese hat automatisch den Effekt, dass alle internen Buffer entsprechend vergrößert werden und somit weniger oft Synchronisationsschritte vollzogen werden. Die Performance steigt zwar dadurch, allerdings aber auch die Latenz.

3.6.3. Timing

Die letzte Grundfunktionalität der Sync-Buffer widmet sich dem Timing. Um die Sounds an der richtigen Stelle zu mischen, benötigt der Mixer eine geeignete Zeitangabe. Da ein individueller Sound dem Mixer, als der ihm zugehörige End-Sync-Buffer bekannt ist, lag es nahe die notwendigen Timinginformationen den Sync-Buffern zu überlassen. Die Timinginformationen sind simple Samplepositionsangaben, d.h. Informationen, die besagt an welcher Stelle das aktuelle Sample z.B. im Mixer anzuordnen ist. Den Samplepositionen kann durch den Timer (Abschnitt 3.11 auf Seite 56 – Timer) ein eindeutiger Zeitpunkt zugeordnet werden. Die Realisierung dieser Funktion enthält im Prinzip zwei gekoppelte Samplezähler. Der erste Zähler enthält die Positionsangabe für das “Full-First“ Sample und gibt somit über den Timer dessen Zeitpunkt an. Der zweite Zähler ist mit dem ersten mittels des aktuellen Füllstandes gekoppelt und gibt somit die Position des “Full-First“ Samples an. Die Aktualisierung der Zähler geschieht implizit beim Puschen bzw. Poppen der Samples. Hiermit können Filter-Stufen auf zeitbezogene Informationen zurück greifen, was hauptsächlich im Mixer benötigt wird und zwar beim Mischen der Sounds und bei der Synchronisierung mit der Ausgabe, also dem ASIO-Backend (Abschnitt 3.10 auf Seite 54 – ASIO-Backend). Im Abschnitt 3.9 auf Seite 48 – Mixer wird weiter auf die Verwendung und die Gründe der Samplepositionsangaben eingegangen und zwar in den Unterabschnitten Mix-In und Output. Eine Erweiterung wäre hier eine zusätzliche Zeitangabe und würde die Positionsangaben zu einem Zeitstempel erweitern. So wäre es möglich, bei Konsistenzverlust der Samplepositionen, diese durch den Timer wieder in konsistente Form zu bringen. Konsistenzverlust tritt z.B. dann auf, wenn der ASIO-Treiber seine Samplepositionen zurück setzt, was im Normalfall nicht passieren sollte. Allerdings würde die zusätzliche Zeitangabe, den hier vermiedenen, höheren Rechenaufwand bedeuten. Die notwendige fortlaufende Zeitpunktaktualisierung, wäre durch ein samplegenauen Offset auf längere Zeit großen Rundungsschwankungen unterlegen. Somit müsste bei jeder Aktualisierung zumindest in regelmäßigen Abständen, der Timer abgefragt werden. Was, bei Verwendung vieler Sounds oder langen Filter-Chains, beim Timer zu Synchronisationsstau führen kann und somit dieser, samt den Filter-Chains, die Funktion nicht ordnungsgemäß erfüllen kann.

3.6.4. Varianten

Die Sync-Buffer verbinden alle in der Soundverarbeitung vorkommende Komponenten miteinander. Da nicht alle Verbindungen die gleichen Anforderungen oder Gegebenheiten besitzen, gibt es die Sync-Buffer in verschiedenen Varianten. Allerdings ist dieser Umstand

nach außen hin transparent, da sie alle auf demselben Grund-Interface basieren. Somit ist es auch möglich mehrere Filter-Chains nicht nur parallel an den Mixer anzuhängen, sondern auch mehrere in Reihe zu schließen. Theoretisch wäre es somit möglich jedem Filter eigene Filter-Chain, also einen Thread mit etwas Overhead, zuzuordnen. Im folgenden werden die verschiedenen Varianten und deren Einsatzzwecke aufgezeigt:

Static-Sync-Buffer ist die Standardausführung eines Sync-Buffers, mit der bisher besprochenen Funktionalität. Static betont hier nur den Umstand, dass der Buffer eine feste, zum Erstellungszeitpunk wählbare, Größe hat.

Static-Not-Sync-Buffer bringt kein implizites Synchronisationsverhalten mit sich. Die Synchronisationsmechanismen können dennoch von außen manuell benutzt werden. Diese Variante eines Sync-Buffers kommt innerhalb von Filter-Chains zwischen den Filtern zum Einsatz. Da die Filter innerhalb ihrer Filter-Chain seriell arbeiten, wird hier keine Synchronisation gebraucht und es kann somit, der hier unnötige Synchronisationsoverhead ausgelassen werden.

Complex-Not-Sync-Buffer entspricht dem Static-Not-Sync-Buffer mit dem Unterschied, dass dieser zum Einsatz zwischen frequenzbasierten Filtern kommt, die mit komplexen Zahlen rechnen.

Self-Fill-Static-Sync-Buffer füllt sich bei einem "Popped-Samples" Aufruf wieder selbst mit Samples auf. Dieser Sync-Buffer wird an der Spitze der Soundverarbeitung verwendet. Die Samples zum wieder Auffüllen werden aus dem zugeordnete Sound von der Sound-Library geladen.

3.6.5. Extension

Die Sync-Buffer können mittels sogenannten Extensions um Funktionalität erweitert werden. So ist es auch möglich eine Extension zu entwickeln, damit bestimmte angrenzende Filter z.B. untereinander kommunizieren können. Also könnte z.B. ein Metadatenaustausch damit realisiert werden.

3.6.5.A. Extension – Volume-Vector

Eine Art Metadatenaustausch wurde bereits durch die Volume-Vector-Extension realisiert. Hierbei erhält der Mixer zu den konsumierten Samples Lautstärkeinformationen für einzelne Ausgabekanäle. Somit wird aus einem gefilterten Mono-Sound ein im 3D-Raum platzierter Sound.

Die Abbildung 3.9 zeigt den Funktionsumfang dieser Extension. Aus dem aktuell zugeordneten Volume-Vektor wird eine Volume-Matrix repliziert, die die angeforderten Samplebreite hat. Somit kann der Mixer mit samplegenauen Lautstärkeangaben den zugehörigen Sound einmischen. Dies ermöglichte einen stufenlosen Fade-In und Fade-Out zu realisieren. Innerhalb des Fade-In und des Fade-Out wird der Volume-Vektor mit einem Faktor zwischen

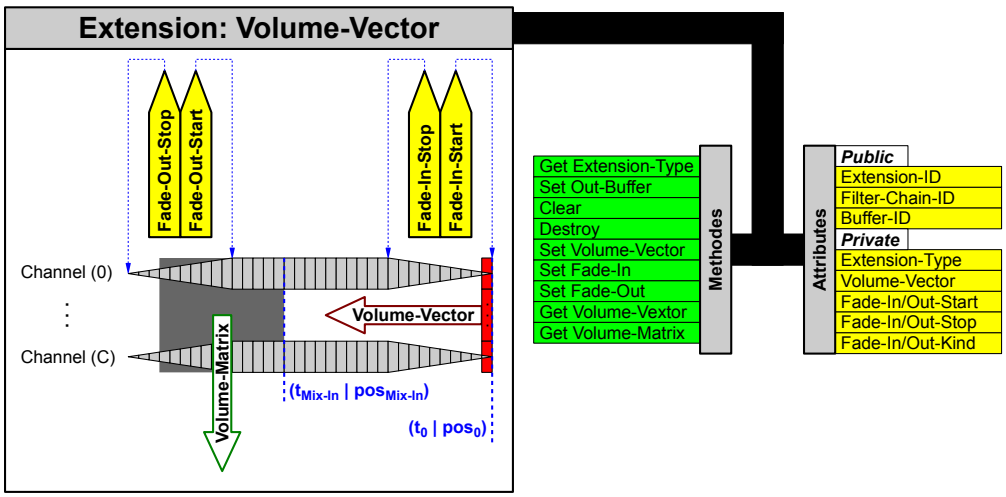

Abbildung 3.9.: Extension – Volume-Vector

null und eins entweder linear oder quadratisch skaliert in der Volume-Matrix abgelegt. Zwischen dem "Fade-In-Stop" und dem "Fade-Out-Start" wird der Volume-Vektor in die Volume-Matrix kopiert. Die Fade-In und Fade-Out Grenzen sowohl auch der Volume-Vektor, können jederzeit abgeändert werden und wirken spätestens ab dem nächsten Mix-In Zyklus des Mixers. Wie mit Hilfe der Volume-Matrizen gemixt wird, wird in Abschnitt 3.9 auf Seite 48 – Mixer behandelt.

3.7. Filter-Chain

Filter-Chains bilden nach der Sound-Library die nächste Verarbeitungsstufe. Eine Filter-Chain, wie in Abbildung 3.10 skizziert, ist im Grunde nur ein Container für Filter und Sync-Buffer, die zu einem Sound gehören. Allerdings mit Mechanismen, die die Buffer zwischen den internen Filtern, beim Einfügen neuer Filter, automatisch erzeugen. Zusätzlich besitzt eine aktive Filter-Chain einen Thread, der die Berechnung der einzelnen Filter anstößt. Liegen am Eingang nicht genug Samples vor oder ist am Ausgang nicht genug Platz vorhanden, so geht die Filter-Chain, wie im Abschnitt 3.6.2 auf Seite 27 – Synchronization beschrieben, in den Wartemodus, bis sie wieder reaktiviert wird.

Die momentane Ablaufstrategie sieht vor, dass zuerst versucht wird den letzten Filter auszuführen, damit bei Erfolg die nächste Stufe bereits weiter arbeiten kann, während die restlichen Filter ausgeführt werden. Liegen am Eingang des letzten Filters zu wenig Samples vor, so wird die Filterkette vom ersten Filter bis zum letzten Filter durchwandert. Falls die Filter etwa die gleiche Menge an Samples verarbeiten, kann beim nächsten Durchlauf vom letzten Filter an gearbeitet werden. Dies gilt auch für die nachfolgenden Durchläufe, da bei jedem Durchlauf wieder Samples am Eingang entnommen werden. Hiermit sollte die Reaktionszeit am Ausgang im Durchschnitt verbessert werden. Dies kann aber eventuell zu

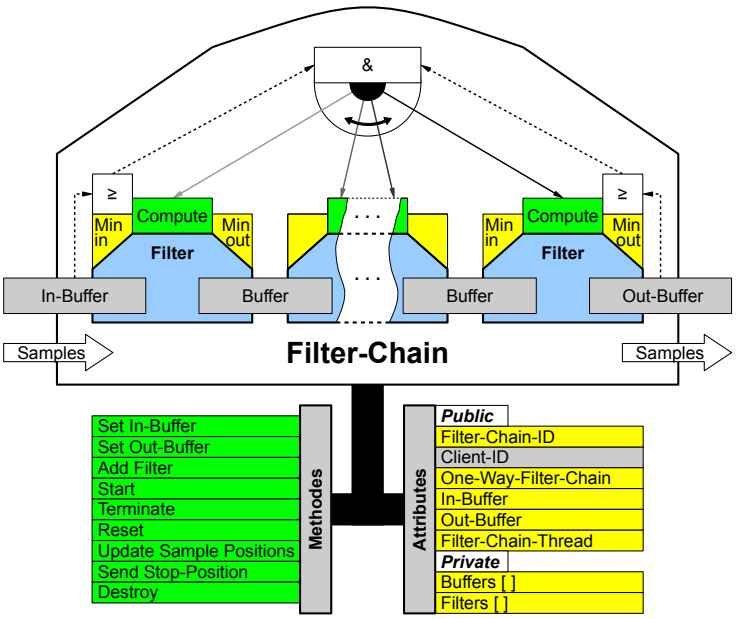

Abbildung 3.10.: Filter-Chain

recht unterschiedlichen Laufzeiten einer Filter-Chain führen. Falls die Filter-Chains an ihre Leistungsgrenzen stoßen, können an dieser Stelle verschiedene Strategien erprobt werden, insbesondere wenn komplexere Filterreihen zum Einsatz kommen.

Wie bei den Methoden, der abgebildeten Filter-Chain zu sehen ist, bietet eine Filter-Chain auch Möglichkeiten an Samplepositionen der Sync-Buffer anzupassen. So muss bei einer Änderung der Mischposition, nicht an jedem Buffer von Hand modifiziert werden. Vielmehr geschieht die Positionsänderung mit Hilfe der Filter, so kann ein Filter, der das Zeitverhalten verzehrt, die Positionsangaben entsprechend modifizieren. Dieses Verhalten wird momentan benötigt, da die Positionsangaben nicht nur am letzten Sync-Buffer für den Mix-In gebraucht werden, sondern der erste Buffer meldet, anhand den Positionen, weiter an welcher Sampleposition der abgespielte Sound zu Ende ist.

3.8. Filter

Dieser Abschnitt widmet sich dem verwendeten Filter-Interface, welches in Abbildung 3.11 grafisch dargestellt ist. Der Hauptbestandteil jedes Filters ist die Compute-Methode. Wie der Name schon andeutet, ist diese für die eigentlichen Berechnungen, also Filterfunktion verantwortlich. Beim Starten der Berechnung kann angegeben werden, ob der Filter nur eine bestimmte Anzahl von Samples berechnen soll, oder er die Menge selber frei wählen soll. Allerdings lässt das Design gänzlich offen, ob und wie viele Samples er tatsächlich verarbeitet.

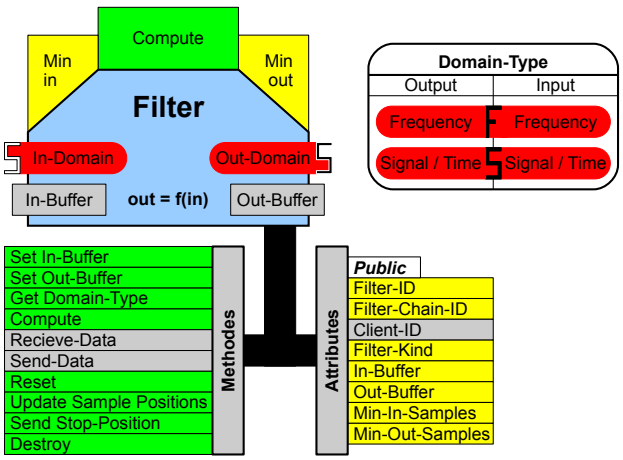

Abbildung 3.11.: Filter

Hier ist beim Design der Filter zu beachten dass der Filter nicht grundlos keine Samples verarbeitet, bzw. dass dadurch kein Aufhängen der Filter-Chain verursacht wird. Durch "Min-In" und "Min-Out" gibt der Filter an, wie viel Samples mindestens am Eingang anliegen müssen bzw. wie viel Platz am Ausgang zur Verfügung stehen muss, um eine Berechnung erfolgreich durchführen zu können. Diese Angaben sind insbesondere beim Ersten und Letzten Filter einer Filter-Chain notwendig. Durch diese Angaben kann sich die Filter-Chain mit anderen Komponenten synchronisieren. Ist eine Angabe zu niedrig, d.h. der Filter rechnet trotz scheinbar erfüllten Voraussetzungen nicht, geht die Filter-Chain in ungebremstes Busy-Waiting über. Das ist aus dem Ablaufdiagramm der Synchronisation in der Abbildung 3.8 auf Seite 25 ersichtlich. Und zwar meldet der Filter die Bedingung sei erfüllt, also blockiert die Filter-Chain nicht und versucht die Berechnung zu starten, welche aber kein Effekt hat und die Bedingung immer noch erfüllt bleibt. Dies wiederholt sich solange bis der Filter, aufgrund geänderter Füllstände, eventuell doch eine Berechnung durchführt. Dieses ungebremste Busy-Waiting hat zufolge, dass auf einer Dualcore-CPU schon zwei solche Filter-Chains zu massiven Sound-Aussetzern führen können.

Ausgangs- und Eingans-Sync-Buffer des Filters tragen Positionsangaben der in ihm gepufferten Samples. Das Interface sieht vor, dass der Filter die Positionsangaben zwischen Ausgang und Eingang, seinem Modell nach, konsistent hält. Dies geschieht zum einen halbautomatisch (halb-, da vom Filter beeinflussbar) durch pushen und poppen von Samples an den Sync-Buffern zum anderen können extern Updates angefragt werden.

Durch "Receive-Data" und "Send-Data" soll den Filtern eine Kommunikationsmöglichkeit mit z.B. Netzwerk-Clients gegeben werden. Hierbei sollen rohe Daten zwischen den Parteien ausgetauscht werden, um die Datenaufbereitung haben sich die Parteien selbst zu kümmern. Damit bleibt das Interface allgemein Nutzbar, egal welcher Filter wirklich darunter liegt. Momentan implementierte Filter tauschen ihre Daten allerdings noch über spezialisierte Methoden aus. Im Verlauf des Ausbaus der Netzwerk-Funktionalität sollte die hier vorgestellte Interface-Erweiterung eingearbeitet werden.

Es gibt zwei vorgesehene Datenformate, auf denen die Filter rechnen. Zum einen für reelle Zahlen und zum anderen für komplexe Zahlen. Reelle Zahlen, hier auch Floats genannt, werden für den Zeit- bzw. Signal-Bereich der Sound-Daten verwendet. Dies ist der bisher besprochene Fall, der unter anderem beim Laden, Abspielen und samplebasierter Filterung der Sound-Daten verwendet wird. Für Filterung im Frequenz-Bereich ist das zweite Datanformat, d.h. Komplexe-Zahlen, erforderlich. Filter geben an in welchen der beiden Bereiche sie arbeiten, genauer gesagt geben sie es getrennt für Ein- und Ausgabe an. Somit werden alle vier Varianten gedeckt, also Filterung im Zeitbereich (Signal/Signal), im Frequenzbereich (Frequency/Frequency), Filterung mit Hintransformation (Signal/Frequency) und Filterung mit Rücktransformation (Frequency/Signal). Dieser Sachverhalt ist in der Abbildung 3.11 mittels grafischer Schlüssel dargestellt, mit S-Schlüssel für Signal- bzw. Zeitbereich und F-Schlüssel für Frequenzbereich. Beim Zusammenstellen der Filter überprüft die Filter-Chain ob die Schlüssel passen und ignoriert den neuen falschen Filter zusammen mit einer negativen Rückmeldung.

3.8.1. FFT

Fast immer liegen die zu filternden Signale als Daten vor, welche aus dem Zeitbereich kommen, also Zeitbezogene Funktionswerte darstellen. Filter die im Zeitbereich filtern, können diese Eingangssignale direkt verwenden und das Filterergebnis liegt meist auch im Zeitbereich. Somit können Signale direkt gefiltert werden und das Ergebnis direkt weiter verwendet werden. Allerdings sind für manche Aufgaben Zeitbezogene Filter ineffektiv oder schlicht weg ungeeignet. Besonders Frequenz veränderndes Filtern stellt sich im Zeitbereich als schwierig dar. So ist oft die Filterung nicht besonders flexibel, ferner ist meist entweder das Laufzeitverhalten schlecht oder das Ergebnis ungenau. Basiert ein Filter auf Faltung zweier Funktionen, so muss er beide geeignet integrieren, was einen hohen Rechenaufwand inne hat. Diese Probleme sind aber im Frequenzbereich viel effizienter zu lösen. So ist die Faltung zweier Zeitbezogener Funktionen im Frequenzbereich eine Multiplikation deren Frequenzspektren. Dies gilt auch bei der Soundverarbeitung. Um z.B. ein Tiefpass im Zeitbereich zu realisieren wird eine entsprechende Filterfunktion benötigt, mit der dann das zu filternde Signal gefaltet wird. Zudem erfordert das Anpassen des Tiefpass-Filters, eine neue Filterfunktion. Im Frequenzbereich reicht eine maskenhafte Filterfunktion, um durch Multiplikation mit dem im Frequenzbereich liegenden Signals eine Filterung durchzuführen. Ein Extrembeispiel wäre eine ideale Tiefpass-Filterfunktion, die bis zur Grenzfrequenz alle Frequenzen mit eins Multipliziert und den Rest mit Null.

Da nun eben z.B. die Sounddaten aus dem Zeitbereich kommen, ist eine Filterung im Frequenzbereich ohne Zwischenschritte nicht möglich. Das bedeutet die Daten müssen in den Frequenzbereich transformiert werden und anschließend nach der Filterung wieder in den Zeitbereich zurück transformiert werden. Für die Transformation in den Frequenzbereich kommt im Allgemeinen die sogenannte Fourier-Transformation zum Einsatz und für die Rücktransformation entsprechend die inverse Fourier-Transformation. Allerdings liegen digitalisierte Werte, wie Sounddaten, diskret vor. Hierfür kommt dann die “Diskrete Fourier-Transformation“ in Frage. Der Unterschied ist, dass die normale bzw. kontinuierliche

Fourier-Transformation eine Integration kontinuierlicher Signale vornimmt und die diskrete eine Summation diskreter Daten. Sowohl die kontinuierliche als auch die diskrete Fourier-Transformation sind sehr rechenintensiv. Allerdings durch Anwendung der “Devide and Conquer“ Strategie, konnten effizientere Algorithmen erstellt werden. Eine solche optimierte diskrete Fourier-Transformation wird als “Schnelle Fourier-Transformation“ oder meist als FFT bezeichnet, was an das englische Wort “fast“ angelehnt ist. Für mathematischen Einstieg können die Quellen [20], [21], [22] und [23] aus dem Literaturverzeichnis verwendet werden, die Quellen verweisen teilweise auf weiterführende Literatur.

Für die Implementierung der Frequenzfilterung wurde die FFTW Bibliothek [24] verwendet. Diese stellt eine Hardware spezifische, hochoptimierte FFT bereit. Die bei FFTW verwendeten Datentype und Formate entsprechen den üblichen hierfür verwendeten. So lehnen sich auch die im Soundserver verwendete Buffer und Filter an diese an, was weitere unnötige Konversionen oder Transformationen vermeidet. Um die Funktionsweise der Frequenzfilter aufzuzeigen, werden im folgenden Kernpunkte der FFT und Frequenzfilterung intuitiv behandelt. Anhand der Abbildung 3.12 wird eine intuitive Auffassung der FFT dargestellt.

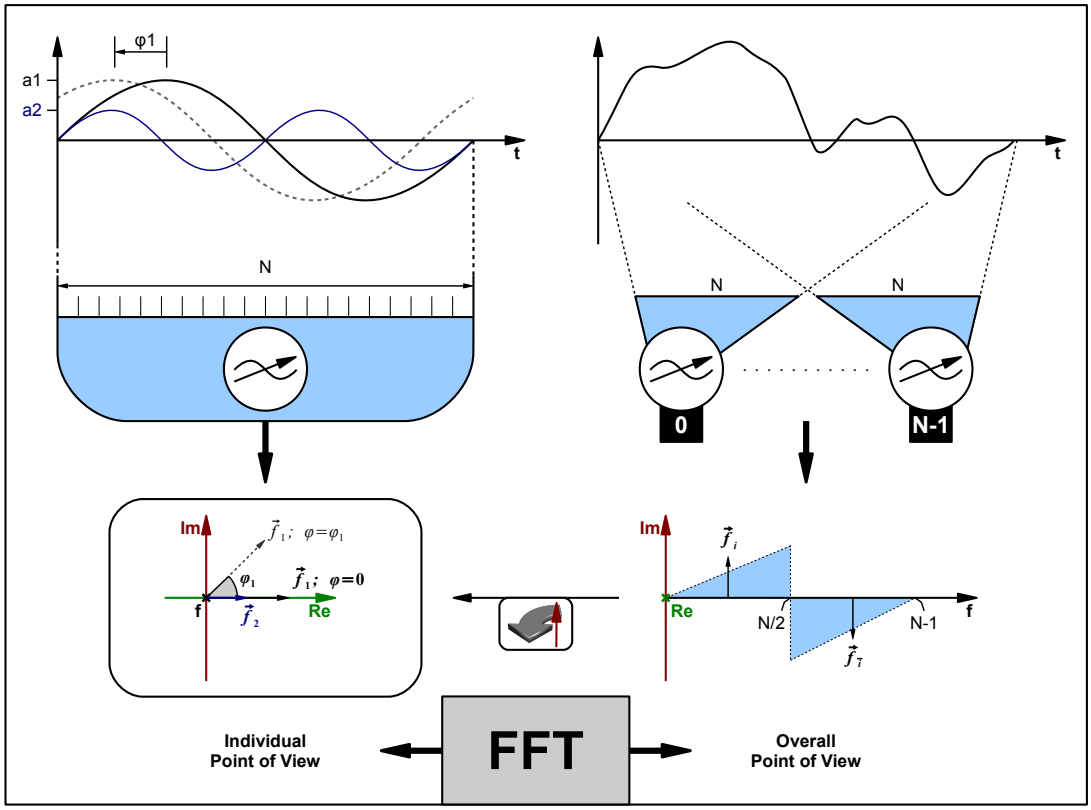

Abbildung 3.12.: FFT – Intuitiv

Auf Sounddaten bezogen, transformiert eine FFT stets ein Zeitfenster. In der Abbildung ist das Zeitfenster N-Sample breit. Anhand des schwarz eingefärbten Sinussignals, wird gezeigt wie ein Frequenzvektor der FFT zustande kommt. Ein FFT-Vektor wird als komplexe Zahl

aufgefasst und kann dadurch im Polarkoordinatensystem der repräsentierten Frequenz eine Amplitude und Phase zuordnen. Diese Angaben sind notwendig um ein Sinus durch wenige Parameter zu beschreiben, also wieder rekonstruierbar zu machen. So reichen die Angaben Frequenz, Amplitude und Phase aus, um ein Sinus eindeutig zu parametrisieren.

Der dargestellte schwarze Sinus entspricht ferner der FFT Grundfrequenz, d.h. Periodendauer entspricht dem Zeitfenster und ist somit stellvertretend für die erste Frequenz einer FFT. Also würde dieser Sinus transformiert werden, so würde die Amplitude der ersten FFT-Frequenz dem abgetasteten Sinus entsprechen. Da der dargestellte Sinus zudem phasengleich mit dem Abtastfenster ist, würde der FFT-Vektor eine Phase von null aufweisen. Der für den schwarzen Sinus stellvertretende FFT-Vektor $\vec{f}_1$, mit Phase $\varphi = 0$, ist unter dem Sinus-Schaubild im Querschnitt der FFT dargestellt. Der FFT-Vektor $\vec{f}_1$, mit Phase $\varphi = \varphi_1$ entspricht dem grau gestrichelten Sinus. Der grau gestrichelte Sinus eilt dem schwarzen Sinus um die Phase φ_1 vor, was als Drehung des Vektors $\vec{f}_1$ um Phase φ_1 resultiert. Ein, hier blau eingefärbter, Sinus mit doppelter Grundfrequenz und halber Amplitude ist als FFT-Vektor $\vec{f}_2$ dargestellt, Da der Periodenursprung dieses Sinus auch mit dem Zeitfenster beginnt, ist seine Phase gleich Null. Und da die Länge des Vektors der Amplitude entspricht, ist er auch halb solange wie $\vec{f}_1$. Damit sollte ein grundlegendes Verständnis zwischen den Werten des Zeitbereichs und des Frequenzbereichs aufgebaut worden sein.

Die Transformation eines Sinus zu einem Vektor, soll anhand der Struktur zwischen den beiden Schaubildern verdeutlicht werden. Diese stellt eine Art Frequenzfilter oder intuitiver gesagt einen “Sinusdetektor“ dar. Da die FFT eine diskrete Fourier-Transformation ist werden pro Zeitfenster N-Proben vom Detektor analysiert. Da hier digitalisierte Sounddaten gefiltert werden sollen, liegen diese Proben hier bereits als Samples vor. Die N-Proben “vergleicht“ der Detektor quasi mir einer ihm zugrunde liegenden “Referenzschwingung“ . Genauer gesagt werden die zum Sinuskreis, der zu untersuchenden Schwingung, gehörende Kreisvektoren untersucht. Ein solcher “Sinusdetektor“ mit zwei verschiedenen Einstellungen, ist in Abbildung 3.13 dargestellt. Im linken Fall wird jedem der N Eingängen ein Phasendreher vorangestellt, der den jeweiligen zu untersuchenden Kreisvektor eines Samples als Vorverarbeitungsschritt geeignet dreht. Durch individuelle Einstellung dieser Phasendrehungen, wird ein Detektor auf seine Referenzfrequenz eingestellt. Die Phasendreher werden so ausgelegt, dass wenn eine zu untersuchende Schwingung sich der Referenzschwingung in Phase und Frequenz gleicht, folgendes gilt. Die gedrehten Kreisvektoren lassen sich zu einem Gesamtvektor aufaddieren, der die Phase Null und die N-fache Amplitude der untersuchten Schwingung aufweist. Betrachtet man nun den resultierenden Vektor so erkennt man, dass bereits dieser dem zu erwartenden Frequenzvektor bis auf die Länge gleicht, also durch Normalisierung mit N in diesen übergeht. Übrigens ist das der Grund warum bei FFTW normalisiert werden muss, weil diese Bibliothek die zur FFT gehörende Normalisierung aus Effizienzgründen dem Entwickler überlässt.

Liegt nun eine Phasenverschobene, ansonsten zur Referenzschwingung identische Schwingung vor, addieren sich die vorverarbeiteten Kreisvektoren zu einem phasenverschobenen resultierenden Vektor, der auch hier dem zugehörigen Frequenzvektor entspricht. Eine entsprechende Schwingung mit niedriger Amplitude hat auch kürzere Kreisvektoren, also reflektiert der resultierende Vektor auch diesen Sachverhalt wieder. Ein “Sinusdetektor“

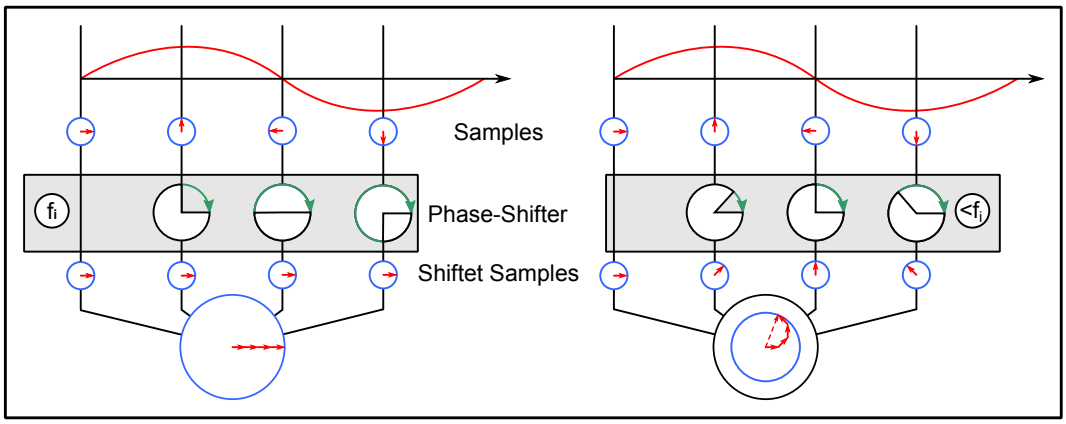

Abbildung 3.13.: FFT-Detektor: Vektorisiert und abstrakt weitgehend aus [25] übernommen

reagiert allerdings nicht nur ausschließlich auf eine exakte Frequenz, sondern abfallend auch auf nahe gelegene Frequenzen. Ein möglicher Fall ist in Abbildung 3.13 rechts dargestellt. Hier sind die Phasendreher so eingestellt, dass sie sich auf eine Referenzschwingung mit niedrigerer Frequenz als die der zu prüfenden Schwingung beziehen. Nun addieren sich die Kreisvektoren nicht mehr in einer Linie, sondern bilden einen Bogen und verursachen einen kürzeren resultierenden Vektor, d.h. auch einen kürzeren Frequenzvektor. Je weiter sich die Frequenzen der Referenzschwingung und der zu prüfenden Schwingung unterscheiden, desto weiter geht der resultierende Kreisvektor gegen null.

Da nun, wie in Abbildung 3.12 rechts dargestellt, eine FFT aus N Frequenzvektoren der N individuellen Detektoren hervorgeht (auch Spektrum genannt), ist nun folgendes ersichtlich. Eine untersuchte Schwingung verursacht somit im Spektrum nicht nur bei der ihr entsprechenden Frequenz ein "Ausschlag" sondern auch auf abnehmende weise in der Umgebung des Spektrums. Diese Frequenzstreuung spielt im Abschnitt 3.8.2 auf Seite 39 – Fensterfunktion eine zentrale Rolle und wird dort näher erläutert. Weiterhin, wie in der Abbildung oben rechts skizziert, zerlegt die FFT anhand der Detektoren ein, bis auf Einschränkungen, beliebiges Signal in seine Frequenzkomponenten. Dies ist möglich, da ein beliebiges Signal, durch Fortsetzung mit null ins minus und plus Unendliche, sich aus unendlich ausgedehnten Sinusschwingungen zusammen setzen lässt. Da periodische Signale sich auf ihre Grundperiodendauer beschränken lassen, kann hier auf Unendlichkeit verzichtet werden. Somit bedeuten Einschränkungen hier, dass die FFT davon ausgeht, dass ein Periodisches Gesamtsignal transformiert wird, welches ebenfalls eine Zentrale Bedeutung im Abschnitt 3.8.2 auf Seite 39 – Fensterfunktion spielt und dort näher behandelt wird. Unter anderem wird dort auch behandelt, wie ein beliebiges Signal für die FFT quasi periodisch gemacht werden kann.

Nun da die FFT aus N Detektoren besteht und jeder Detektor stellvertretend einer Frequenz steht, würde dies das Nyquist-Shannon-Abtasttheorem [13] verletzen. Die hier beschriebene FFT kann suggerieren, dass mit N-Abtastwerten eine FFT mit N-Frequenzen entsteht, also dass hier die Abtastrate auch der maximal detektierbaren Frequenz entspricht, was eine

Verletzung des Abtasttheorem wäre. Allerdings, wie man in Abbildung 3.12 rechts unten angedeutet sieht, geht ab der $\frac{1}{2}N + 1$ -ten Frequenz das Spektrum aus einer Spiegelung der ersten Hälfte des Spektrums hervor. Somit enthält ein reelles transformierte Signal redundante Frequenzen und der Sound entspricht einem reellen Signal. Somit kann bei der Speicherung und Verarbeitung eines reellen transformierten Signals, der Teil mit "negativen Frequenzen" weggelassen werden. Dieser wird allerdings wieder für die Rücktransformation und manche andere Berechnungen benötigt und kann wieder rekonstruiert werden. Zu beachten ist, dass die Spiegelung der zugehörigen Frequenz durch umgekehrte Phasendrehung hervorgeht, also kein direktes Negieren des Frequenzvektors und steht somit dem eigentlichen Frequenzvektor konjugiert komplex gegenüber, entsprechen also als Summe einem auf reellen Achse liegenden Vektor. Dies ist übrigens auch der Grund warum der komplexe Wertebereich bei einer Rücktransformation wieder in einen reellen Bereich resultiert. Also darf bei der Filterung der konjugiert, komplexe Charakter nicht verletzt werden.

Dass die Abtastfrequenz mindestens doppelt so hoch sein muss, wie die im Signal maximal vorkommende Frequenz, kann anhand der Sinusgleichung einer Schwingung verdeutlicht werden:

$$y(x) = a \sin(2\pi f x + \varphi)$$

Um nun eine bestimmte Schwingung durch die Funktion darstellen zu können, werden drei Parameter benötigt, einmal die Amplitude a dann die Frequenz f und schließlich die Phasenverschiebung φ. Die Frequenz an sich ist die Laufvariable der FFT, also kann jedem spektralen Wert bzw. Vektor eine solche Gleichung zugeordnet werden:

$$y_i(x) = a_i \sin(2\pi f_i x + \varphi_i)$$

Somit bleiben zwei unbekannte Parameter übrig. Liegt nun eine Schwingung z.B. als Ansammlung von Samples vor, kann diese mathematisch mit obiger Gleichung beschrieben werden. Hier sieht man deutlich den Vorteil des Frequenzbereichs, denn anstatt den Sinus mit N-Samples abzubilden, genügt es hier die drei Parameter zu wissen. Um eine Gleichung mit zwei Unbekannten zu lösen benötigt man ein Gleichungssystem mit zwei linear unabhängigen Gleichungen. Dieses Gleichungssystem erhält man durch Einsetzen zweier Sample-Proben:

$$\begin{aligned} \mathbf{y_1} &= a_i \sin(2\pi f_i \mathbf{x_1} + \varphi_i) \\ \mathbf{y_2} &= a_i \sin(2\pi f_i \mathbf{x_2} + \varphi_i) \end{aligned}$$

Eine Lösung dieses Gleichungssystems liefert die fehlende Amplitude a_i und Phase φ_i. Also werden, pro zu untersuchende Frequenz zwei Samples gebraucht, somit ist die höchste untersuchbare Frequenz halb so groß wie die Samplerate und wird in Literatur meist als die Nyquist-Frequenz bezeichnet. Es werden aber N-Detektoren für die FFT verwendet, davon liefern nur N/2 viele Detektoren die eigentlichen Frequenzen. Was mit den restlichen passiert, kann man sich anhand einer Filmaufnahme vorstellen, die ebenfalls diskret Bilder aufnimmt.

Nimmt eine Videokamera z.B. eine Propeller auf, der sich beginnt zu drehen und kontinuierlich schneller wird, so sieht man dann den Propeller sich erst in eine Richtung drehen. Irgendwann sieht man ihn nahezu stehen und wenn die Geschwindigkeit sich noch weiter steigert, beginnt er sich scheinbar rückwärts zu drehen. Hat die Kamera eine Bildrate von 50 Bildern Pro Sekunde, so ist anzunehmen, dass bei der Nyquist-Frequenz von 25 Rotationen pro Sekunde, der scheinbare Propeller Stillstand zu beobachten ist. Zwischen 25 und 50 Rotationen pro Sekunden, sollte ein sich rückwärts drehender Propeller zu beobachten sein, d.h. eine Spiegelung seiner tatsächlichen Bewegung. Also ist das restliche FFT Spektrum, wie besprochen, eine Spiegelung des Hauptspektrums.

Beim Propeller Beispiel ist indirekt noch ein weiterer Effekt zu beobachten, der eine weitere Eigenschaft der FFT andeutet. Dreht sich der Propeller stetig noch schneller, so sieht man den Propeller immer wieder seine Drehrichtung ändern, also deutet es auf eine periodische Wiederholung hin. Tatsächlich ist das FFT Spektrum periodisch und das Spiegelspektrum findet sich ebenso im negativen Frequenzbereich. So kann derselbe Fall auch von -N/2 bis + N/2 betrachtet werden, darum werden die Frequenzen des Spiegel-Spektrum oft auch als negative Frequenzen bezeichnet. Die -N/2 bis + N/2 Betrachtungsweise ist die, die häufig in der Literatur gewählt wird, die von null bis N ist oft bei Implementierung von Algorithmen zu finden. Um die letzte Propeller Betrachtung zurechtzurücken, sei hier noch darauf eingegangen, was mit der indirekten Beobachtbarkeit, des periodischen Verhaltens gemeint ist. Um den periodischen Sachverhalt erfassen zu können, müsste die Kamera ihre Aufnahmerate nach oben verschieben, also theoretisch z.B. mit 50 Hz starten und mit 100 Hz aufhören, was hier nicht möglich ist. Die höheren Spiegelungen wirken dennoch zum Teil in dem unteren Bereich. Da die Abtast- bzw. Aufnahmerate eventuell gar viel niedriger ist, als die abgetastete Rotationsfrequenz, dehnen sich die periodischen Spektren über ihre Grenzen in die Nachbar Spektren hinaus. Es kann davon ausgegangen werden, dass aus diesem Grund die Silhouette des Propellers eventuell mehrfach zu sehen ist.

Um auf die Grundidee einzugehen, warum die eigentliche FFT schneller ist als die hier noch im Grunde vorliegende DFT, wird hier wieder an die Phasendreher erinnert. Jeder Frequenzvektor des Spektrums entspricht einem Sinusdetektor, jedem dieser Detektoren sind N Phasendreher zugeordnet. Dies bedeutet, dass ein und dasselbe Sample durch viele Phasendreher gedreht wird. Durch geschickte Anordnung der Phasendreher und Ausnutzung der Zwischenergebnisse, kann auf eine Vielzahl davon verzichtet werden. Dieser, als “Devide and Conquer“ bezeichnete Sachverhalt macht aus einem DFT-Algorithmus einen FTT-Algorithmus.

Die hier beschriebenen Sachverhalte wurden durch drei Quellen inspiriert, d.h. unter Einbezug von Vorwissen frei interpretiert. Im Literaturverzeichnis unter [25], [26] und [20] zu finden. Die Betrachtungsweise anhand der hier bezeichneten Sinus- bzw. Frequenz-Detektoren ist angelehnt an die Quellen [26] und [25]. Die Beschreibung anhand der Phasendreher ist an den Ausführungen der Quelle [25] orientiert. Es wurde versucht das komplexe Thema weitgehend anschaulich zu erklären, was unter Umständen durch die starken Vereinfachungen oder Beispiele zu mathematischen Ungereimtheiten führen könnte. Für mathematischen, übersichtlichen Einstig in die FFT-Thematik kann die Quelle [20] zurate genommen werden.

3.8.2. Fensterfunktion

Der vorherige Abschnitt 3.8.1 auf Seite 33 – FFT behandelte die grundlegenden Ideen und Eigenschaften der FFT. Hierbei wurde bereits implizit die erste Fensterfunktion verwendet. Es wurde davon gesprochen, dass die FFT stets ein Zeitfenster transformiert. Gibt man dem Zeitfenster an jeder seiner Stellen z.B. den wert eins vor, so entspricht dieses Zeitfenster einer Rechteck-Fensterfunktion. Eine Fensterfunktion wird angewandt, indem sie eins zu ein mit den Samples multipliziert wird, welche sich im Bereich des Zeitfensters befinden. Aufgrund seines auf die Multiplikation neutralen Verhaltens, verändert das Rechteck-Fenster die Samples nicht. Somit kann die besprochene FFT, als FFT mit der Rechteck-Fensterfunktion angesehen werden. Durch Einführung anderer Fensterfunktionen kann entscheidend Einfluss auf das Transformationsverhalten einer FFT genommen werden. Eine Anschauung ist, dass das Frequenzspektrum eine Ansammlung von Frequenzfiltern ist, dessen Frequenz filternde Eigenschaften der zugrundeliegenden Fensterfunktion zuzusprechen sind [26].

Im Abschnitt 3.8.1 auf Seite 33 – FFT wurde gezeigt, dass eine transformierte Schwingung nicht nur dem ihr entsprechenden Frequenzausschlag verursacht, sondern abfallend auch in der spektralen Nachbarschaft. Anders ausgedrückt, reagiert eine “Spektrale Linie“ also Filter oder Detektor, auch auf Schwingungen, die eine ähnliche Frequenz, wie die Spektrallinie selbst aufweisen. Diese Streuung der Frequenzen ist der zugrunde liegenden Fensterfunktion und ihrer Größe zuzuschreiben. Weiterhin wurde erläutert, dass die FFT von periodischen Signalen ausgeht. Allgemein ist das zu transformierende Signalstück aber nicht periodisch fortsetzbar. Da die FFT dennoch davon ausgeht, entstehen dadurch Unstetigkeitsstellen, welche im Spektrum weitere Störfrequenzen zur Folge haben.

Die FFT unterliegt Einschränkungen in der Frequenzauflösung und Erfassung schwacher Signale [26]. Die Frequenzauflösung besagt, wie viele separate Frequenzbereiche die FTT auflösen kann. Ferner besagt sie, wie gut ein Signal abgebildet wird, das zwischen zwei Spektrallinien sich bewegt. Bei der Erfassung schwacher Signale, ist die Hauptschwierigkeit die, dass beim Vorhandensein von starken Signale, die schwachen Signale von den starken überdeckt werden können. Hier spielt die Streuung von Frequenzen, auch Leck-Effekt genannt eine große Rolle. Auch die, durch periodische Unstetigkeiten der Grenzstellen eines Zeitfensters verursachten, Störfrequenzen tragen zum Leck-Effekt bei.

Die Frequenzauflösung lässt sich primär durch die Größe des Zeitfenster beeinflussen. Allerdings bedeutet ein größeres Fenster auch einen höheren Rechenaufwand, welcher bei der FFT nicht so ins Gewicht fällt wie bei der DFT. Zudem hat man unter Umständen zum Transformationszeitpunkt nicht beliebig viele Samples zur Verfügung bzw. große Zeitfenster könnten eine nicht vertretbare Latenz mit sich bringen. Die Auflösung eines Signals, dass eigentlich zwischen zwei Spektrallinien anzusiedeln ist, kann zudem durch eine geeignete Fensterfnktion beeinflusst werden. Die Erfassung von schwachen Signalen gegenüber starken, kann ebenfalls durch das Design der Fensterfunktion beeinflusst werden. Auch das Problem nicht periodisch fortsetzbarer Signale lässt sich durch eine Fensterfunktion lindern.

So kann eine Fensterfunktion viele angesprochene Probleme lindern, hat allerdings das Problem zur Folge, dass durch hervorheben einer Eigenschaft, eine andere verschlechtert

wird. Somit ist die Wahl oder das Design einer Fensterfunktion immer ein Trade-off und von Fall zu Fall werden unterschiedliche Fensterfunktionen benötigt. So hat ein Rechteckfenster, durch seine schmalbandige Transformierte, die wohl beste Frequenzauflösung unter den Fensterfunktionen, verursacht aber sehr viele Leckfrequenzen, da diese nicht die aperiodischen Signale der FFT als periodische verkauft. So hat eine im Zeitbereich schmale und an den Enden gegen null strebende Fensterfunktion einen guten Leckcharakter bezüglich den Fenstergrenzen. Diese verursacht aber eine schlechte Frequenzauflösung, da ihre Transformierte recht breit ist und viele Frequenzen sich dadurch überschneiden. Somit werden hier zwar wenig Störfrequenzen verursacht, aber aufgrund schlechter Frequenzauflösung ein hohes Maß an Frequenzstreuung.

Wie schon verraten wurde, besteht der Trick darin ein nicht periodisch fortsetzbares Signal quasi periodisch zu machen, indem es mit einer Fensterfunktion multipliziert wird, die an den Enden weich gegen null geht. Da beide Enden im Idealfall Null sind, gibt es keine Unstetigkeiten beim aneinander Reihen des identischen Fensters. Um eine möglichst hohe Frequezauflösung zu erreichen, sollte der mittlere Bereich des Fensters recht breit sein. Ist der mittlere Bereich so breit, dass die Flanken zu den Enden hin sehr steil sind, so können durch den recht scharfen Übergang Störfrequenzen eingeführt werden. In Abbildung 3.14 sind zwei Fensterfunktionen abgebildet. Oben das Rechteck-Fenster, was einer FFT ohne Fensterfunktion entspricht und unten das momentan im Soundserver verwendete “von-Hann-Fenster“ . Das linke Schaubild stellt jeweils den Zeitbereich dar und das rechte den Frequenzbereich. Zu beachten ist, dass im Frequenzbereich eine logarithmisch Amplitudenskala in dB verwendet wurde. Zu sehen ist, dass das Rechteck-Fenster eine schmalere Spitze hat, d.h. sie kann zwei nahe gelegene Frequenzen besser auflösen. Allerdings sieht man auch, dass das Rechteck-Fenster verglichen zum von-Hann-Fenster sehr flach abfällt. Somit wird beim Rechteck-Fenster mehr Rauschen aus dem gesamten Spektrum übernommen.

Das von-Hann-Fenster wurde ausgewählt, da es zum einen ein guten Trade-off für Sounddaten verspricht und zum anderen sich gut in die Implementierung einreihte. Die zugrunde liegende Sinusfunktion ist zudem intuitiv verständlich. Das wieder zusammensetzen der zurück transformierten gefensterten Samples wird in Abschnitt 3.8.5 auf Seite 43 – FFT-Transformation-Filters angesprochen. Die Abbildung 3.17 aus dem genannten Abschnitt stellt in der Dialogblase das Zusammensetzen zweier Fenster symbolisch dar. Hierbei sind die Fenster je um eine halbe Breite versetzt. Da dem von-Hann-Fenster eine Kosinusfunktion zugrunde liegt, bedeutet die halbe Breite eines halben Kosinus eine Phasenverschiebung von $frac\pi 2$. Somit kann man das Zusammensetzen als Addition von cos und sin ansehen. Eine mögliche Funktionsgleichung für das von-Hann-Fenster ist:

$$H(x) = \cos^2(\tfrac{\pi x}{N} x) = \tfrac{1}{2}[1 + \cos(\tfrac{2\pi x}{N} x)] \text{ [28]}$$

Da nun die Addition zweier verschobenen Zeitfenster als Addition von cos und sin angesehen werden kann, erhält man:

$$\cos^2(\tfrac{\pi x}{N} x) + \sin^2(\tfrac{\pi x}{N} x) = 1$$

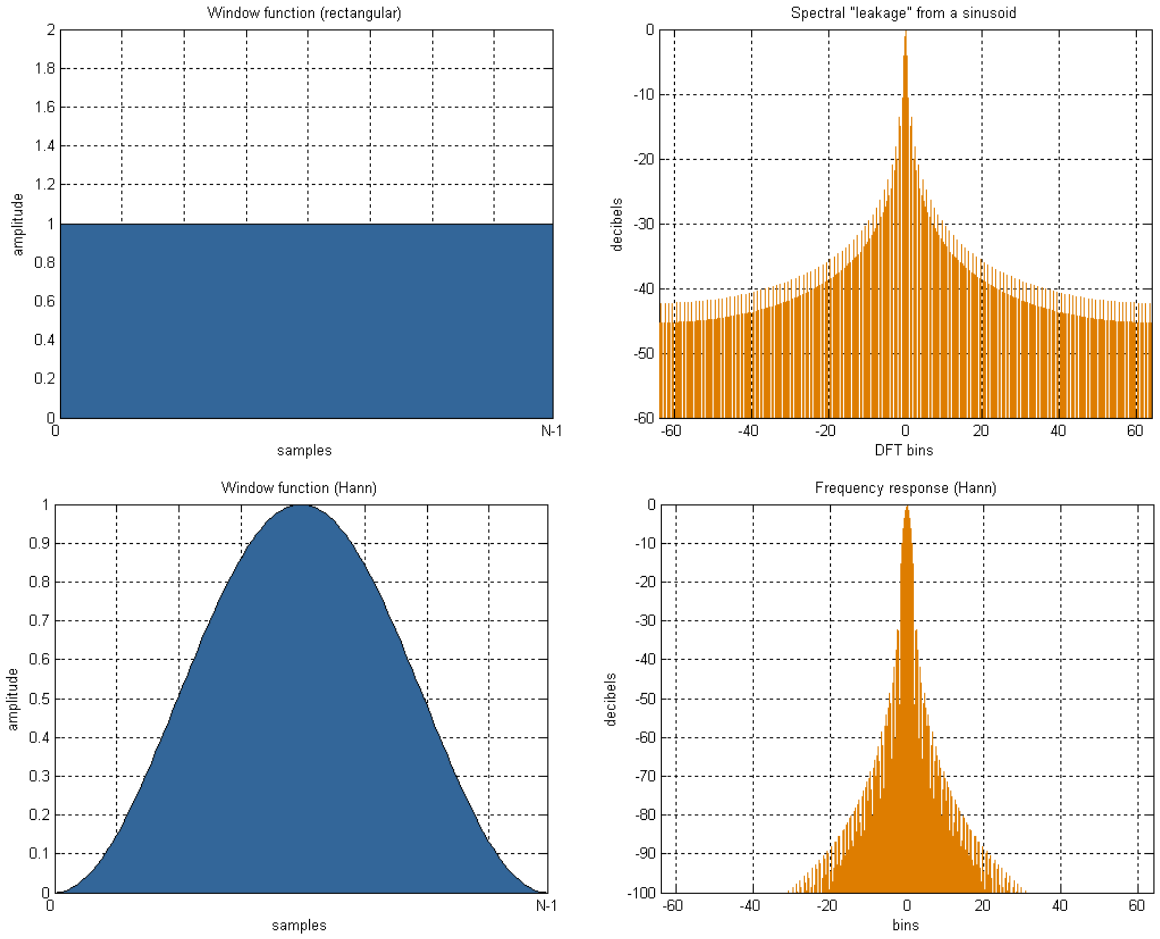

Abbildung 3.14.: FFT-Fensterfunktionen [27]

Entspricht also der Abbildung 3.17. D.h. die einzelnen von-Hann-Fenster addieren sich geschickt angeordnet zu eins und haben rechnerisch im Zeitbereich keinen Effekt und müssen deswegen auch nicht weiter herausgefiltert werden.

3.8.3. Blocking-Delay-Filter

In den nächsten Abschnitten werden die implementierten Filter vorgestellt. Der erste vorgestellte Filter wird als "Blocking-Delay-Filter" bezeichnet und seine Funktionsweise in Abbildung 3.15 dargestellt.

Blocking bedeutet hier, wie in der Grafik angedeutet, dass die Samples am Eingang solange angehalten werden bis das gesetzte Delay-Kontingent aufgebraucht wurde. Das Aufbrauchen des Delay-Kontingents wird erreicht, indem in die Ausgabe Stille geschrieben wird und das Kontingent dementsprechend heruntergesetzt wird. Ist Verzögerungszeit des Sounds erreicht, d.h. das Delay-Kontingent gleich null, verhält sich der Filter wie ein Null-Filter,

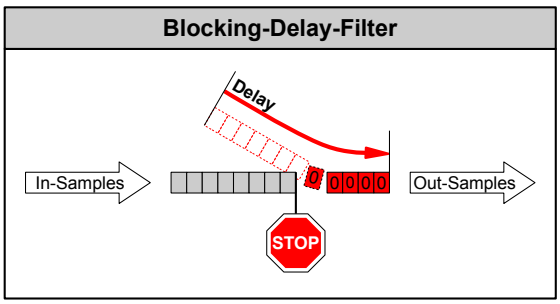

Abbildung 3.15.: Blocking-Delay-Filter

also kopiert lediglich die Eingabe in die Ausgabe. Dieser Filter verbraucht somit keinen zusätzlichen Speicher um Daten zwischen zu speichern. Das ist auch sehr praktisch für große, feste oder nur nach oben zu ändernde Delays. Das Delay kann jederzeit nach oben erhöht werden, was in der Soundausgabe entsprechende Stille bedeutet. Der Filter kann auf dem jetzigen Stand keine Delays wieder abbauen. Da der Filter ohne Zwischenspeicherung die Delays erzeugt, gibt es nur eine Möglichkeit diese Funktionalität zu erreichen. Der Filter müsste dann, das negative Delay-Kontingent durch Abstoßung oder durch Resampling der Eingabe abarbeiten.

Es gibt kritische Fälle, bei denen dieser Filter aufgrund seines blockierenden Charakters nicht eingesetzt werden soll bzw. darf. Der Einsatz in den Sound-Filter-Chains ist soweit unproblematisch, allerdings nur solange keine Funktionen dazukommen, die eine direkte oder indirekte Kommunikation zwischen Filter-Chains oder Filtern einführen. Der Einsatz dieses Filters in den Mixer-Filter-Chains ist hingegen schädlich und sollte vermieden werden. Im Allgemeinen, falls ein Delay-Filter im Mixer benötigt wird, werden pro Ausgabekanal oder Kanalgruppe unterschiedliche Delays gesetzt. Durch das blockierende Verhalten wird dadurch erreicht, dass am Mix-In die Out-Buffer nicht mehr Synchron beschrieben werden können. Dies bedeutet, wenn der blockierende Delay-Unterschied zweier Kanäle größer als der Arbeitspuffer des Mix-Ins ist, so wird der Mix-In theoretisch dauerhaft blockiert bzw. kann nicht fehlerfrei arbeiten.

3.8.4. Non-Blocking-Delay-Filter

Der in Abbildung 3.16 skizzierte “Non-Blocking-Delay-Filter“ kann hingegen im Mixer eingesetzt werden. Dieser, wie der Name schon sagt, blockiert weder die Eingabe noch die Ausgabe. Das Delay wird durch Zwischenspeicherung der Eingabe erreicht. Hierbei wird zuerst ein delaybreites Array angelegt, das mit Stille gefüllt ist. Nun wird pro Sample ein Sample aus dem Delay-Array in die Ausgabe geschrieben, in dieselbe Delay-Array-Position wird daraufhin ein Sample aus der Eingabe geschrieben. Dasselbe Prozedere wird am nächsten Delay-Sample durchgeführt und so weiter. Somit ist das erste Sample aus der Eingabe erst nach einmal durchwandern des Delay-Arrays hörbar bzw. wirkend und damitist das Delay-Verhalten erreicht. Das Delay ist hier jederzeit änderbar und zwar nicht nur nach

obenn, sonder auch wieder bis auf null reduzierbar. Zwischen dem Delay-Wechsel wird ein neues Delay-Array erstellt, das alte Array soweit möglich in das neue kopiert und dann entsorgt. Ist das neue Delay kürzer als das alte so geht ein teil des zwischengespeicherten Sounds verloren, anders herum wird es mit Stille aufgefüllt.

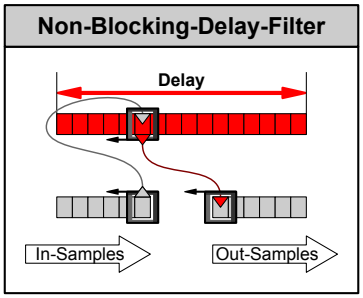

Abbildung 3.16.: Non-Blocking-Delay-Filter

Das Delay-Verhalten dieses Filters ist nach außen hin völlig transparent, da er immer synchron eine gleiche Anzahl von Samples liest und schreibt. Es kann lediglich angegeben werden, dass dieser das Delay an die Samplepositionen der Sync-Buffer weiter gibt. Es ist eventuell empfehlenswert bei sehr großen Delays das Standard-Array durch das Dynamic-Array aus Abschnitt 3.3.1 auf Seite 18 – Dynamic-Array einzusetzen. Hier wäre es auch möglich das Delay In-Space zu ändern, d.h. ohne zu es kopieren, falls man z.B. das Delay erst an einem Rücksprungpunkt ändert.

3.8.5. FFT-Transformation-Filters

Um im Frequenzbereich filtern zu können, müssen die im Zeitbereich liegenden Samples, wie in Abschnitt 3.8.1 auf Seite 33 – FFT besprochen, in den Frequenzbereich transformiert werden und anschließend nach der Filterung wieder zurück transformiert werden, um abgespielt werden zu können. Somit widmet sich dieser Abschnitt den beiden in Abbildung 3.17 dargestellten Transformationsfiltern.

Das “S2F-Transform“ steht für Signal-to-Frequenzy Transformation und somit für die Hintransformation. Entsprechen steht “F2S-Transform“ für die Rücktransformation. Für die Hin- und Rücktransformation werden überlappende Fenster verwendet. Als Fensterfunktion kommt momentan das, in Abschnitt 3.8.2 auf Seite 39 – Fensterfunktion erwähnte, von-Hann-Fenster zum Einsatz. Die Fensterfunktion an sich wird nicht bei jedem Zyklus neu berechnet, sondern liegt einmal beim Anwendungsstart berechnet, als ein Array zur effiziente Nutzung bereit. Unten in Abbildung 3.17 ist die Idee, wie sich die Fenster wieder zusammensetzen lasse, in einer Dialogblase dargestellt. Addiert man zwei aufeinanderfolgende von-Hann-Fenster so ergibt deren Überlappung den Faktor eins. Da zu jedem Zeitpunk zwei von-Hann-Fenster aufaddiert werden, ergibt sich ein Dauerhafter eins Faktor, somit haben die Fenster, multipliziert mit Samples, keinen verzehrenden Effekt im Zeitbereich.

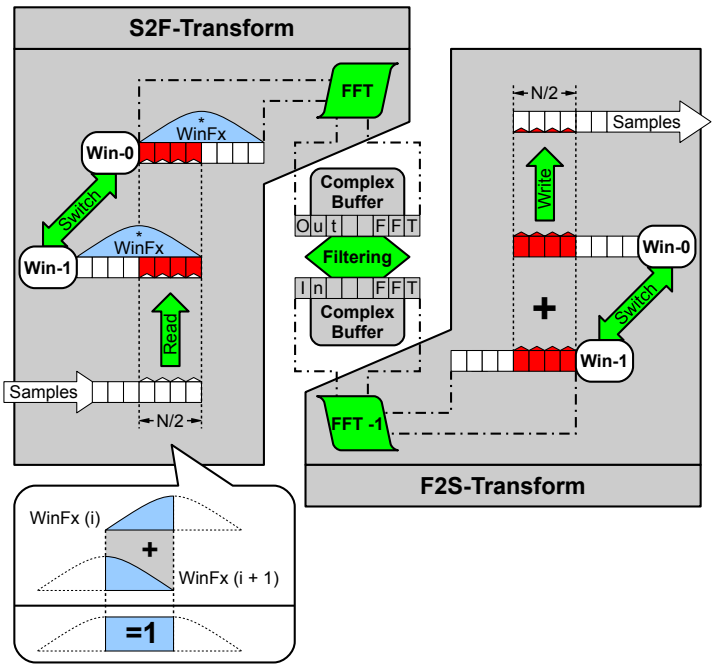

Abbildung 3.17.: FFT-Transformation-Filters

Die Transformation in den Frequenzbereich geschieht in zwei Schritten. Der S2F-Transform-Filter beinhaltet zwei Fensterarrays. Diese Arrays haben jeweils eine Breite von N Samples. N ist hier die FFT-Transformations-Größe. Der erste Schritt befüllt diese Fenster mit Samples. Der zweite Schritt transformiert das volle der beiden Fenstern in den Frequenzraum und gibt das Ergebnis, als komplexe Zahlen, an den entsprechenden Ausgangs-Buffer weiter.

Das Befüllen der Fenster geschieht wie folgt: Der Eingabe werden $N/2$ Samples entnommen und in den überlappenden Bereich der beiden Fenster geschrieben. Beim Schreiben werden allerdings jeweils die zugehörigen Fensterfunktionen mit den Samples multipliziert. Daraufhin kann die Transformation des oberen Fenster vollzogen werden. Switched man jetzt die beiden Fenster und befüllt sie wieder auf gleiche Weise, so erkennt man leicht, dass das obere Fenster mit den alten und neuen Samples befüllt ist und zwar in chronologisch richtiger Reihenfolge. Aus diesem Grund reicht es in jedem Zyklus nur $N/2$ anstatt N Samples zu lesen und erhält trotzdem für die Transformation notwendige N Samples. Das Switchen der Fenster geschieht allerdings am Anfang jedes Zyklus und wurde aus anschaulichen Gründen anders herum angedeutet.

Die Transformation wird mit Hilfe der hoch optimierten Bibliothek FFTW [24] durchgeführt. Die Fenster und Ausgangs-Buffer wurden so konzipiert, dass FFTW direkt aus dementsprechenden Fenster lesen und in den Ausgags-Buffer schreiben kann. Initialisierung von FFTW kann je nach Einstellung sehr viel Zeit in Anspruch nehmen, so wird die Bibliothek, genau wie die Fensterfunktionen, nur einmal beim Anwendungsstart initialisiert. Um FFTW parallel in mehreren Threads nutzen zu können, stehen spezielle Befehle zur Verfügung.

Die nun in den Frequenzbereich transformierten Samples können nun mit auf Frequenz spezialisierten Filtern gefiltert werden. Im Abschnitt 3.8.6 – FFT-Amplitude/Phase-Filters werden zwei solche auf FFT-Fenster basierte Filter vorgestellt.

Die anschließende Rücktransformation im F2S-Transform-Filter, verläuft analog. Zu Beginn eines Zyklus werden die Fenster des Filter geswitcht. Daraufhin wird das anliegende FFT-Fenster nur in das untere Fenster rücktransformiert. Danach wird der überlappende Bereich der beiden Fenster, wie in der Dialogblase dargestellt, aufaddiert und anschließend in den Ausgangs-Buffer geschrieben.

Spielt man gedanklich den gesamten Vorgang beim ersten Gesamtdurchlauf durch, so merkt man, dass bei der Rücktransformation das erstemal nur Stille ausgegeben wird. Hier wird also eine Latenz von $N/2$ Samples verursacht. Um dem entgegen zu wirken wird die erste Rücktransformation nicht in den Ausgabe-Buffer geschrieben, sonder erst ab der zweiten. Also wird beim ersten Durchlauf die doppelte CPU-Leistung und doppelter Speicherdurchsatz verbraucht. Allerdings, da die Filter zwischen den Zyklen im Verhältnis zur Rechenzeit sich lange im Wartemodus befinden, sollte der einmalige Aufwand nur bei höherer Systembelastung relevant sein.

Der Vorteil von überlappenden Fenstern ist nicht nur, dass sie den parasitären Effekten im Frequenzbereich entgegen wirken, sondern wie hier gezeigt auch halb so große Buffer und somit halbe Latenzen verursachen. Oder anders gesehen bei gleichen Buffer-Größen, kann man, im Gegensatz zum Fall ohne überlappende Fenster, doppelt so große FFT-Fenster verwenden, was die Frequenzgenauigkeit insbesondere im unteren Bereich deutlich erhöhen sollte.

3.8.6. FFT-Amplitude/Phase-Filters

Für Frequenzfilterung wurden zwei Filter implementiert. Zum Einen der “FFT-Amplitude-Filter“ der die Amplituden der einzelnen Frequenzen filtert und somit unter anderem als Equalizer eingesetzt werden könnte. Zum Anderen der “FFT-Phase-Filter“ der die Phasenlage einzelner Frequenzen filtert. Da beide Filter, bis auf den Effekt, nahe zu identisch sind, werden sie hier zusammen behandelt.

Die Abbildung 3.18 fasst beide Filter zusammen. Die einzelnen Frequenzen liegen als komplexe Zahlen vor. Die negativen Frequenzen liegen hierbei verschoben in der zweiten Hälfte der Buffer. Der Gleichanteil liegt an Stelle null und die Nyquist-Frequenz an Stelle $N/2$, siehe hierzu auch Abschnitt 3.8.1 auf Seite 33 – FFT. Die Schaubilder in der Abbildung zeigen ein Beispiel anhand einer Frequenz $\vec{f}_i$. Die zugehörige negative Frequenz wird als $\vec{f}_{\bar{i}}$ bezeichnet. Das linke Schaubild zeigt einen Querschnitt des Hauptschaubilds, anhand dessen die Filterung veranschaulicht werden soll. Beide Filter haben die Filterkoeffizienten, als jederzeit abänderbare, Arrays vorliegen. Der Amplitudenfilter multipliziert lediglich jeden Frequenzvektor $\vec{f}_i$ mit dem zugehörigen Koeffizienten a_i. Der Phasenfilter rotiert jeden Frequenzvektor $\vec{f}_i$ um die ihm zugehörige Phasenverschiebung φ_i. Hier ist aber zu beachten, dass die zugehörige negative Frequenz $\vec{f}_{\bar{i}}$ in Gegenrichtung gedreht werden muss,

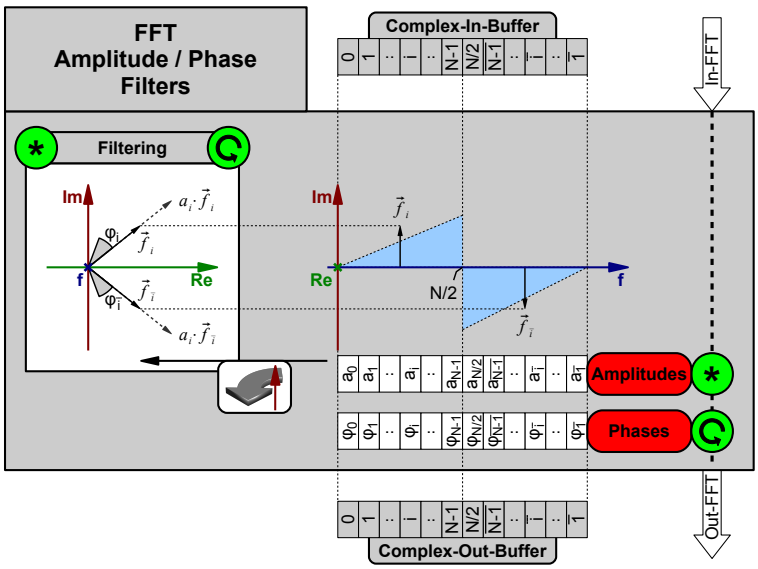

Abbildung 3.18.: FFT-Amplitude-Phase-Filters

also ist $\varphi_{\bar{i}} = -\varphi_i$. Die Rotation wird mittels einer Rotationsmatrix durchgeführt, allerdings werden die Sinus- und Cosinuswerte nur einmal pro Koeffizientenänderung berechnet und zwischengespeichert. So kann die Rotation im Normalfall auf je zwei Float-Multiplikationen und zwei Float-Additionen bzw. Subtraktionen reduziert werden.

Da die Koeffizienten der Filter frequenzgenau eingestellt werden können, kann der Amplitudenfilter nicht nur als $N/2$ bandiger Equalizer eingesetzt werden, sondern theoretisch auch für Faltung z.B. mit einem zweiten Signal. Hierzu wären aber evtl. Anpassungen bei der Koeffizientenübergabe notwendig, unter anderem um beide Signale zu synchronisieren. Im Zuge des Netzwerkausbaus sollten die Filter evtl. Koeffizienten z.B. auch aus Splinekurven errechnen können. Dies würde die Netzwerkbandbreite drastisch entlasten. Zudem zu scharfe Koeffizientenübergänge erzeugen teilweise gut hörbare Artefakte.

3.8.7. Mixer-Filter

Der letzte "implementierte" Filter ist der "Mixer-Filter" . Hierbei handelt es sich eher um einen Zusammenschluss mehrerer Filter zu einem. Die Idee war, dem Mixer einen schnellen und latenzarmen Gesamtfilter zur Verfügung zu stellen. Mit diesem Filter kann pro Ausgabekanal individuell ein Delay gesetzt werden, im Frequenzbereich sowohl Amplituden als auch Phasen gefiltert werden und sowohl Kanallautstärke als auch gesamt Lautstärke reingerechnet werden.

Der Filterverbund ist in Abbildung 3.19 dargestellt. Dadurch dass nach außen hin nur ein In-Buffer und ein Out-Buffer existieren, wird die Latenz des Mixers drastisch gesenkt, denn

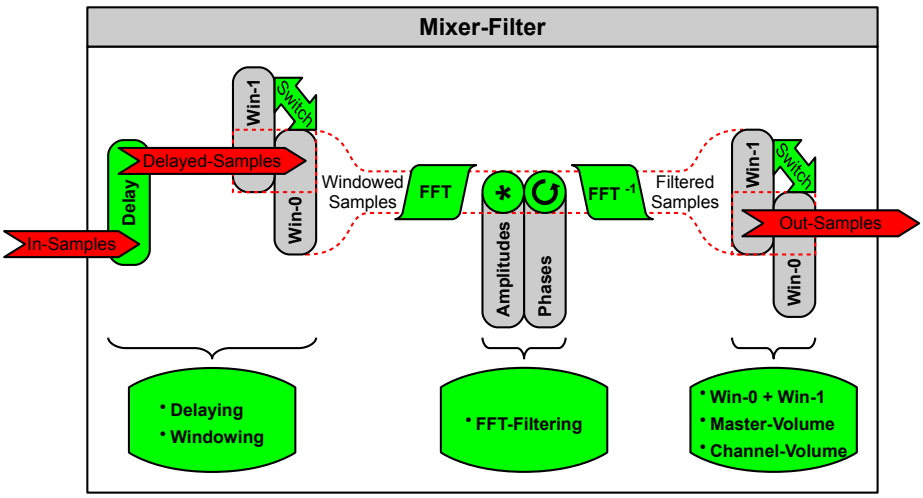

Abbildung 3.19.: Mixer-Filter

jedes im Mixer zwischengespeicherte Sample erhöht die Latenz zwischen dem frühest möglichen Mix-In und der Soundausgabe. Weiterhin ließen sich einfache Laufzeitoptimierungen durchführen. So konnten mehrere Schleifen zusammengefasst werden. Es konnte der Delay-Filter und das Befüllen der Fensterfunktionen in einer Schleife zusammengefasst werden. Vielmehr schreibt der Delay-Filter direkt in die Fenster des Transformations-Filters. Die Amplituden- und Phasen-Filterung wird auch in einem Zug durchgeführt. Analog wird auch die Ausgabe durch eine Schleife gefüttert. In der Schleife der Ausgabe werden zusätzlich noch die Lautstärken reingerechnet. Da FFTW keine Normalisierung durchführt wird diese auch in der Ausgabe mit erledigt. Durch Generalisierung des Mixer-Filters, wäre dieser auch in den Sound-Filter-Chains einsetzbar.

3.8.8. Signalgenerator

Im Zuge der Filterimplementierung wurde ein simpler Signalgenerator erstellt. Dieser ist im Grunde ein modifizierter F2S-Transform-Filter, der in Abschnitt 3.8.5 auf Seite 43 – FFT-Transformation-Filters behandelt wurde. Die Anbindung des Generators geschieht ebenso als Filter einer Sound-Filter-Chain. Der Effekt ist, dass die Filter-Chain nach außen hin keine Samples am Eingang verbraucht, sondern ausschließlich welche am Ausgang erzeugt.

Die generierende Funktion wurde dadurch erreicht, dass der zu Grunde liegende F2S-Transform-Filter keine eingehenden FFT-Fenster zurück in den Signalraum transformiert, sondern ein vorgegebenen FFT-Block. Die einzelnen Frequenzvektoren des Blocks können, unter Angabe einer Amplitude und Phase, beliebig von außen gesetzt werden. Da die hier verwendete FFT-Blockgröße sich an der Samplerate des Soundservers orientiert, liefert ein Frequenzvektor an der Stelle 100 mit Amplitude 1.0 und Phase null eine 100 Hz Sinusschwingung mit der Amplitude von 1.0. Die “rücktransformierten“ FFT-Blöcke reihen sich

außerdem nahtlos aneinander an. Dies kann leicht nachvollzogen werden. Wenn man sich eine beliebige, vom Generator erzeugbare Schwingung, erstmal mit Phase null vorstellt, so ist die Frequenz einer solchen Schwingung ein vielfaches der Schwingung mit 1 Hz. Da 1 Hz die Grundfrequenz des FFT-Pakets ist, beginnt das erste zurücktransformierte Sample des FFT-Blocks mit der ersten Nullstelle der Schwingung und das letzte Sample endet quasi mit der dritten Nullstelle, also mit einer Periode pro FFT-Block. Alle vielfachen dieser Grundfrequenz mit Phase null haben ebenso eine Nullstelle und dieselbe Steigung an den Transformationsgrenzen. Eine Aneinanderreihung der Rücktransformationen hat also in diesem Fall die nahtlose Fortsetzung der Schwingungen zur Folge. Das dies auch für beliebige " feste " Phasen gilt ist klar, den in jeder aneinandergereihten Rücktransformation ist die entsprechende Schwingung um die gleiche Phase verschoben. Also wird auch hier die Schwingungen nahtlos fortgesetzt. Lediglich bei Änderung der Frequenzvektoren kann es zu einem Amplituden oder Phasenbruch kommen, was ein hörbares Knacksen zwischen zwei FFT-Blöcken verursachen kann.

3.9. Mixer

Das Herzstück der Soundverarbeitung, bzw. die alles am Fluss haltende Pumpe, ist der Mixer. Der in Abbildung 3.20 dargestellte Mixer arbeitet in drei Phasen. Die Erste Phase, welche als "Mix-In" bezeichnet wird, bildet den Kern des Mixers und läuft in einem eigenständigen Thread ab. "Filtering" ist die zweite Phase und ist für die Filterung der Ausgabekanäle zuständig. Schließlich ist "Output" dafür verantwortlich, dem für die Soundausgabe zuständigen Modul, Sounddaten zur Verfügung zu stellen. Die beschrifteten Sterne beziehen sich auf die Soundserver-Übersichtsgrafik 3.1 auf Seite 14 und stellen Anschlusspunkte zu andere Modulen dar.

3.9.1. Mix-In

Die Funktionsweise des Mix-In kann ebenfalls in drei Phasen eingeteilt werden. Die Erste Phase kümmert sich um die Verwaltung, der zu mischenden Sounds, die Zweite um das eigentliche Mischen und die Dritte um das Versorgen der Mixer-Filter-Chains mit Nachschub.

3.9.1.A. Sounds-Verwaltung

Die Verwaltung der Sounds wird mit Hilfe drei Strukturen realisiert. Die Hauptstruktur ist ein Ring, welcher in Abschnitt 3.3.3 auf Seite 21 – Ring behandelt wird. Dieser, als "Buffer-Drum" bezeichnet, enthält die aktuellen und in Zukunft zu mischenden Sounds. Genauer gesagt ist hier und im Weiteren als Sound, die Referenz des Ausgangs-Sync-Buffers, von der zum Sound gehörenden Filter-Chain, gemeint. Die anderen zwei Strukturen sind die, in

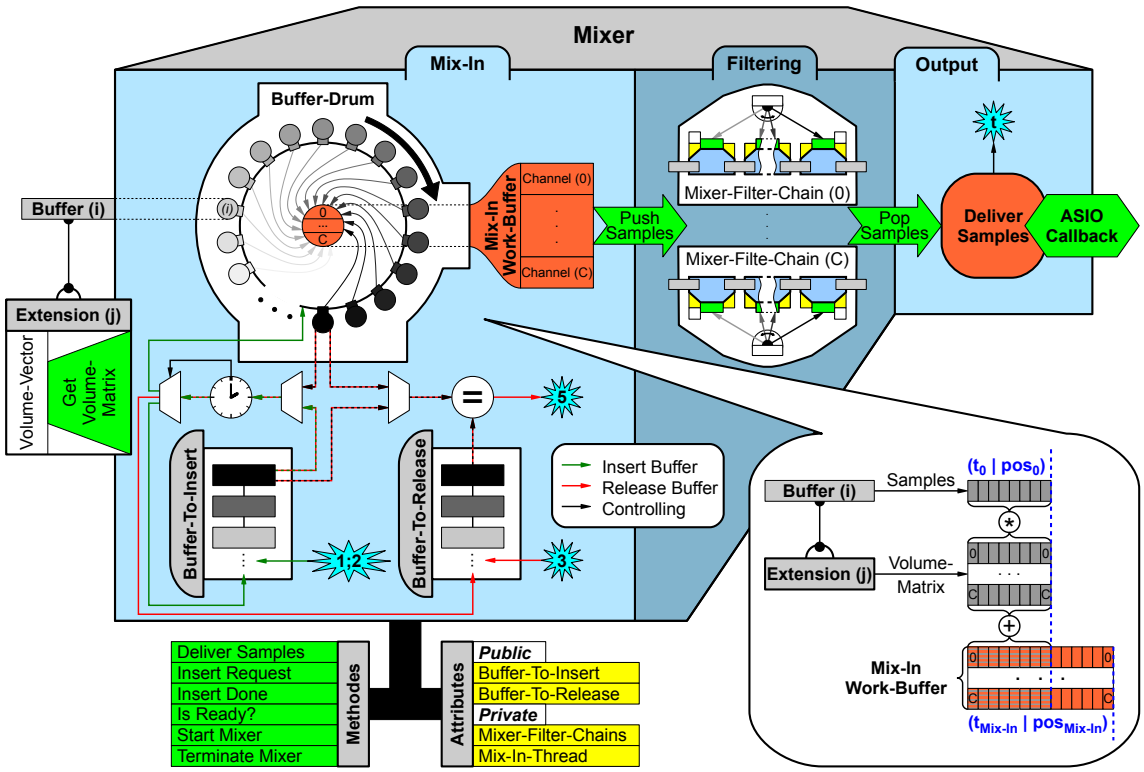

Abbildung 3.20.: Mixer

Abschnitt 3.3.2 auf Seite 20 – Queue vorgestellten, Warteschlangen. Dabei enthält "Buffer-To-Insert" die noch zum richtigen Zeitpunkt in die Buffer-Drum zu übernehmenden Sounds. "Buffer-To-Release" enthält hingegen Sounds, also wie gesagt Referenzen der Sounds, die der Mixer wieder abstoßen soll.

Durch die Methode "Insert Request" wird der Mix-In quasi angehalten. Die Methode liefert zusätzlich noch die nächste Mix-In-Sampleposition. Diese Position gibt quasi den Zeitpunkt an, wann ein angemeldeter Sound frühestens hörbar ist. Dieser Zeitpunkt kann verwendet werden, um einen Sound so schnell wie möglich abspielen zu lassen, d.h. falls ein bestimmt getimeter Zeitpunkt unwichtig ist. Zu den Samplepositionen ist allgemein folgendes noch zu sagen. Mit Hilfe des, in Abschnitt 3.11 auf Seite 56 – Timer behandelten Timers, kann der Positionsangabe ein Zeitpunkt zugeordnet werden. Dies gilt für alle im Mixer verwendeten Samplepositionen. Werden die Samplepositionen konsistent weiter geleitet, so gilt das auch innerhalb der Sound-Filter-Chains. Bei der Mischung gilt die Sampleposition des Sounds, also Buffers, als der Zeitpunk für das Mischen. Das bedeutet durch Updaten dieser Sampleposition des Sound-Buffers, wird z.B. von außen der Mix-In-Zeitpunkt des Sounds gesteuert.

Ist der Mix-In durch Insert Request quasi angehalten, können beliebig viele Sounds an der Buffer-To-Insert-Warteschlange angemeldet werden. Beliebig, wird hier nur durch die

Rechenleistung und allgemein durch Rechnerressourcen begrenzt. Desweiteren können Sounds zur Abmeldung an der Buffer-To-Release-Warteschlange angemeldet werden. Ist das Anmelden beendet, wird das durch die Methode “Insert Done“ quittiert und der Mix-In kann sein Dienst wieder aufnehmen.

Die Phase der Verwaltung im Mix-In jongliert zu Beginn jedes Mix-In-Zyklus mit den in den Verwaltungsstrukturen angemeldeten Sounds. D.h. wie in der Mixer-Abbildung 3.20 umrissen, werden die einzufügenden Sounds auf ihre Mix-In-Kriterien überprüft. Liegt der Zeitpunkt in der Vergangenheit, so werden Samples solange abgestoßen, bis der Sound wieder Synchron ist. Ist der Buffer des Sounds dabei aufgebraucht, wird der Sound bis zum nächsten Zyklus wieder am Schluss der Buffer-To-Insert-Warteschlange eingefügt. Liegt der anzumeldende Sound innerhalb oder über dem Mix-In-Zeitfenster wird dieser in die Buffer-Drum verschoben, d.h. zum Mischen freigegeben.

Die Verwaltung, der in der Buffer-Drum bereits enthaltenen Sounds, wird während dem Mischen erledigt, also eigentlich während Phase zwei, d.h. zu Beginn des Mischvorgangs des jeweiligen Sounds. Hierbei werden Sounds die über dem Mix-In-Zeitfenster liegen ignoriert bis sie im Zeitfenster liegen. Liegt ein Sound unter dem Zeitfenster, wird auch hier versucht Samples abzustoßen bzw. der Sound wird wieder in die Buffer-To-Insert-Warteschlange verschoben. Meldet ein Sound, dass er die End-Sampleposition erreicht hat, wird seine Filter-Chain gestoppt und der Sound in die Buffer-To-Release-Warteschlange verschoben.

Die Verwaltungsschritte bezüglich freizugebender Sounds, also Sounds der Buffer-To-Release-Warteschlange, werden nicht explizit durchgeführt. Die Verwaltung wird implizit an der entsprechenden Stelle mit erledigt, um das Suchen zu ersparen. So wird der Sound abgestoßen , während seine Mix-In-Kriterien überprüft werden, falls er an der Spitze der Buffer-To-Release-Warteschlange liegt. Desweiteren wird ein Sound vor seinem Mix-In Versuch abgestoßen, falls er an der Spitze der Buffer-To-Release-Warteschlange liegt. Wird während einem Mix-In-Zyklus die nicht leere Buffer-To-Release-Warteschlange verändert, so wird davon ausgegangen, dass ein Sound abzumelden ist, der nicht im Mixer enthalten ist. Dementsprechend wird die Buffer-To-Release-Warteschlange gestutzt. Der hier unternommene Versuch das Buffer-Release-Verhalten zu beschleunigen könnte noch Fehler aufweisen oder Fehler an anderen Stellen verursachen und sollte eventuell überdacht werden. Allerdings sollten mögliche Probleme erst auftauchen, wenn viele Sounds gleichzeitig abgestoßen werden sollen. Im Betrieb wurden keine ersichtlichen Fehler bemerkbar.

3.9.1.B. Mixing

Die zweite Phase des Mix-In ist, wie gesagt das eigentliche Mischen der Sounds. Der Versuch die Sounds direkt in die Ausgangsbuffer zu mischen, erwies sich als schwierig, fehleranfällig, synchronisationsproblematisch und kaum debuggbar. Aus diesem Grund kommt der sogenannte “Mix-In–Work-Buffer“ als Zwischenspeicher zum Einsatz.

Das Mischen wird in der Mixer-Abbildung 3.20 unten rechts, in der Dialogblase grafisch aufgezeigt. Während einem Mix-In-Zyklus, wird nur dann gemischt, wenn der zu mischende Sound innerhalb des Mix-In-Zeitfensters liegt, also falls er sich innerhalb der momentanen

zeitlichen Grenzen des Mix-In–Work-Buffers befindet. Ansonsten werden, wie besprochen, Samples abgestoßen oder der Mix-In überspringt diesen Sounds, bis der Sound innerhalb eines Mix-In-Zeitfensters liegt. Sind die Voraussetzungen erfüllt, wird der Sound samplegenau, also an der richtigen Stelle des Work-Buffers, eingemischt. Der Work-Buffer selbst ist eine Reihe von Buffern, simplen Arrays, die zu Beginn jedes Mix-In-Zyklus gelöscht werden. Jeder dieser Unterbuffer ist stellvertretend für einen Ausgabekanal des Mixers, also in der Grafik von Kanal 0 bis Kanal *C*.

Jedem Sound bzw. seinem Buffer muss eine Volume-Vektor-Berechnungseinheit zugeordnet sein. Diese ist, in Form der in Abschnitt 3.6.5.A auf Seite 29 – Extension – Volume-Vector besprochenen Extension, bei der Erstellung der Sound-Filter-Chain, anzuhängen. Diese Extension liefert auf Anfrage eine geeignete Volume-Matrix, basierend auf dem ihr zugewiesenen Volume-Vektor. Dies macht den Mixer gegenüber Änderungen an der Volume-Vektor-Verarbeitung unempfindlicher. Also ist damit nicht nur samplegenaue Fade-In und Fade-Out möglich, sondern generell samplegenaue Volume-Vektor Verarbeitung, ohne Modifikation am Mixers. Mit diesem Hilfsmittel ist das Mischen, mathematisch ausgedrückt, nichts weiter als sampleweise Skalarmultiplikation mit dem zum Sample gehörenden Spaltenvektor der Volume-Matrix und anschließender Addition der zugehöriger Spaltenvektoren des Ergebnisses mit dem Work-Buffer.

Beim "Aufaddieren" vieler Sounds muss prinzipiell keine Skalierung der Sounds durchgeführt werden. Bildlich erklärt ist das Aufaddieren zweier Sounds vergleichbar mit zwei abgespielten Sounds über nebeneinander stehende Lautsprecher, denn hier überlagern bzw. addieren sich die Schallwellen entsprechend. Signaltechnisch gesprochen überlagern sich die Signale, d.h. Sounds auch hier. Das hat zur Folge, dass positive und negative Flanken statistisch gleich oft vorkommen. Die Tatsache, dass die Sounds im Allgemeinen, über viele Samples gesehen, keinen Gleichanteil haben, untermauert die Addierbarkeit der Sounds. Dies macht die Addition also Überlagerung im Mittel unkritisch. Hörtechnisch konnte, innerhalb als normal angesehenen Parameter, kein übersteuern des Sounds festgestellt werden. Dies schließt allerdings nicht aus, dass bei ungünstiger Konstellation der überlagerten Sounds, ein vereinzeltes übersteuern auftreten kann. Eine Möglichkeit dem Problem entgegen zu wirken, wäre Nachverarbeitung des Work-Buffers, d.h. im Falle einer (drohenden) Übersteuerung, die Ausgabe vorzugsweise über einen kurzen Zeitraum auf geeignete Art zu dämpfen. Da zwischen jeder Recheneinheit die Samplepositionen bekannt sind, wäre auch ein quasi in "Zukunft" blickender Übersteuerungsschutz möglich. Und zwar liegen im Allgemeinen zwischen Mix-In und ASIO-Backend bis zu mehrere ms Sound. Merkt nun der Mix-In eine bereits vorliegende Übersteuerung, reicht es wenn er ab diesem Zeitpunkt die Ausgabe entsprechend zeitlang dämpft. Damit kein abruptes leiser werden zu hören ist, kann der Mix-In die Position der Übersteuerung mixerweit bekannt geben. Denn beim Versorgen der Buffer des ASIO-Backends kann anhand des bekannten Übersteuerungszeitpunktes, quasi in weiser Voraussicht, das Anfangsstück der Übersteuerung gedämpft werden.

3.9.1.C. Push

Die letzte Phase des Mix-Ins wartet auf die anschließenden Mixer-Filter-Chains, bis deren Eingangs-Buffer alle mindestens das Mix-In-Fenster fassen können. Daraufhin wird der Mix-In–Work-Buffer in die Buffer der Mixer-Filter-Chains, wie in der Mixerabbildung 3.20 angedeutet, “gepusht“ also kopiert.

3.9.1.D. Auswertung

Das hier vorgestellte Mischen, über den Mix-In–Work-Buffer, hat gegenüber dem direkten Mischen eine höhere Latenz bzw. einen längeren Samplepfad.Gegenüber dem direkten Mischen kommt hier ferner der Kopierschritt vom Work-Buffer in die Ausgabe hinzu. Da aber beim direkten Mischen viel Verwaltungsaufwand getrieben werden muss, um pro Kanal und Sound in die richtige Position mischen zu können, ist trotz des Kopierschritts nicht zu erwarten, dass das Mischen über den Mix-In–Work-Buffer schlechtere Laufzeit erzielt. Bedenkt man das hoch parallele Gesamtkonzept bietet dieses Verfahren sogar Laufzeitvorteile. So kann die Ausgabe, d.h. Mixer-Filter-Chains, Sounddaten verarbeiten, also Platz für den Mix-In schaffen, während der Mix-In noch auf die Eingabe wartet oder diese mischt. Und im nächsten Schritt kann die Eingabe bereits weiter Sounddaten filtern, während der Mix-In auf die Ausgabe wartet oder in diese schreibt. Der reale Fall dürfte sich zwischen den beiden Fällen bewegen. Anzunehmen ist, dass hier fast ausschließlich auf die Ausgabe gewartet wird. Dies wird angenommen, da der Mixer und die Sound-Filter-Chains allgemein schneller rechnen sollten, als die Soundausgabe Daten braucht, andernfalls wäre der Soundserver an der Auslastungsgrenze. Der hier verwendete indirekte Mix-In sollte zudem auf kurze Auslastungen flexibler reagieren können.

Das Mischen mit Hilfe der Volume-Matrizen kann auf den erste Blick rechenintensiver erscheinen, als das Mischen mit einem Volume-Vektor. Der einzige Mehraufwand ist hier allerdings, die Matrizen an sich, also hauptsächlich Erstellung der Matrizen, was im Allgemeinen nur die Replikation der Volume-Vektoren erfordert. Der eigentliche Rechenaufwand bleibt der gleiche, es sind genau so viele Multiplikationen und Additionen zu tätigen wie bei der Mischung mit einem Volume-Vektor.

3.9.2. Filtering

Die zweite Phase des Mixers, als “Filtering“ bezeichnet, hat die Aufgabe ausgangsspezifische Filterung durchzuführen. Im Endeffekt ist somit die Soundausgabe jedes einzelnen “Lautsprechers“ individuell filterbar. Zum Filtern werden hier dieselben Filter-Chains verwendet, wie die bei der Filterung individueller Sounds. Somit können auch im Prinzip dieselben Filter eingesetzt werden. Allerdings gibt es hier eine technische Einschränkung. Die Filter-Chains müssen so bestückt werden, dass sie quasi synchron an allen Eingängen die gleiche Samplemenge konsumieren. Entsprechend gilt das auch für die Ausgänge. Mit quasi ist hier gemeint, dass sie zwar zeitlich unterschiedlich schnell rechnen dürfen, dennoch der zeitliche

Skalierungsfaktor identisch sein muss. Ist das nicht der Fall, so laufen die Filter-Chains auseinander. Wird die Varianz größer als der Mix-In–Work-Buffer, so wird der Mix-In bzw. die Filter-Chains blockiert bzw. funktionieren nicht mehr fehlerfrei. Der in Abschnitt 3.8.3 auf Seite 41 – Blocking-Delay-Filter beschriebene "Blocking-Delay-Filter" ist ein Beispiel hierfür. Verwenden zwei Kanäle diesen Filter mit unterschiedlichen Delays, so führt das blockierende Verhalten des Filter zu der beschriebenen Varianz der Samples. Dieser Sachverhalt schließt Resampling-Filter nicht generell aus, es muss lediglich gelten, dass die Resampling-Faktoren, Stauverhalten und Sample-Sprünge aller Mixer-Filter-Chains gleich sind. Der besprochene Mixer-Filter (Abschnitt 3.8.3 auf Seite 41 – Blocking-Delay-Filter) ist genau für diesen Einsatz konzipiert. Besteht jede Mixer-Filter-Chain aus diesem Filter, so verhält er sich bezogen auf die Samplevarianz auf allen Kanälen gleichartig, unabhängig von gewählten Delays oder anderen Filtereigenschaften.

3.9.3. Output

Die letzte Phase des Mixers wird als "Output" bezeichnet und bieten eine Schnittstele für die ASIO-Callbacks. Siehe hierzu auch Abschnitt 3.10 auf der nächsten Seite – ASIO-Backend. Die Hauptaufgabe des Outputs ist die geeignete Befüllung der von ASIO-Callbacks überreichten Buffer. Zusätzlich wird an dieser Stell zugleich der Timer aktualisiert.

Die Callbacks stellen nicht nur die zu befüllenden Buffer zur Verfügung, sondern überreichen bei der Anfrage zusätzliche Informationen. So wird unter anderem mitgeteilt, an welcher ASIO internen Sampleposition der erste zu überreichende Sample steht. Da sich die Soundserver Samplepositionen an eben diese ASIO internen Samplepositionen ausrichten, kann überprüft werden ob die vorliegenden Samples synchron mit der Ausgabe sind. Die verwendeten Sync-Buffer, aus Abschnitt 3.6 auf Seite 24 – Sync-Buffer, führen einen impliziten Samplezähler mit sich, der z.B. bei Initialisierung auf die ASIO-Sampleposition aktualisiert werden kann. Der Zählerstand der ersten vollen und ersten leeren Position der Sync-Buffer wird implizit immer durch "Pushed-Samples" und "Popped-Samples" entsprechend erhöht. Somit ist die Überprüfung, ob die gemischten Samples mit ASIO synchron sind, ein simpler Vergleich der Zählerstände aller Kanäle. Dass die auszugebenden Samples asynchron werden können, passiert für gewöhnlich nur, wenn die Soundverarbeitung bzw. das System oder ASIO über kurz oder lang überlastet wird oder Fehlfunktionen auftreten. Der ASIO-Treiber an sich bricht, laut Spezifikation, bei Zeitüberschreitung die Anfrage ab, um dann die nächste dem Timing entsprechende Anfrage zu starten. Passiert es also dass die auszugebenden Samples hinterher hängen, werden Samples abgestoßen. Im Falle dass die Soundverarbeitung generell oder beim Abstoßen der asynchronen Samples nicht nachkommt, wird der entsprechende Ausgangsbuffer nicht befüllt, also mit Stille belassen. Passiert aus unerklärlichen Gründen, dass die gemischten Samples in die Zukunft zeigen, so werden erst wieder Samples ausgegeben wenn die Samplepositionen der ASIO-Anfrage übereinstimmen. Somit ist auf dementsprechenden Kanal erst wieder Sound zu hören wenn die Synchronität wieder hergestellt wurde. Die ASIO-Anfrage wird vom Mixer-Output sofort bedient, also nicht wie im bisherigen Design auf die Eingabe gewartet bis diese genug Samples zur Verfügung gestellt hat. Entsprechende Wartefunktionen wurden erprobt und

können bei Bedarf wieder reaktiviert werden. Die subjektive Erfahrung war hier, dass es keine wirklichen Vorteile brachte, da die Soundverarbeitung im Allgemeinen bereits vor jeder ASIO-Anfrage mit ihren Berechnungen fertig ist, bzw. durch die verwendete automatisch Buffergröße bis zu insgesamt zwei ASIO-Anfragen gepuffert werden. Es wird auch vermutet, dass dieses Warteverhalten im Callback-Thread bei manchen ASIO-Treibern zu Instabilität führte, trotz Versuchen das Callback-Zeitfenster nicht zu überschreiten.

Zusätzlich zu den ASIO internen Samplepositionen teilen die ASIO-Callbacks die dazugehörigen Zeitstempel mit. Die Sampleposition und der Zeitstempel bilden ein Paar, das dem Timer für Zeitsynchronisationsmaßnahmen übergeben wird. Die vom ASIO bereitgestellten Zeitstempel basieren auf der lokalen Systemzeit. Soweit es bekannt ist, ist es nicht möglich, dass ASIO einen anderen Zeitstempel liefert, als den der auf der Systemzeit basiert. Da der Soundserver eben ein Server für nicht lokale Clients darstellen soll, wird eine geeignete Zeitsynchronisation erfordert. Auf diese Problematik wird dann in Abschnitt 3.11 auf Seite 56 – Timer eingegangen.

3.10. ASIO-Backend

Das implementierte ASIO-Backend stellt den Abschluss der vom Soundserver beeinflussbaren Soundverarbeitung dar. Die Aufgaben sind unter anderem auf Anfrage einen ASIO-Treiber zu suchen, zu initialisieren, zu starten und zu stoppen. Zur Initialisierung gehört auch dazu Callback-Funktionen zur Verfügung zu stellen, d.h. ASIO stellt im Grunde leere Rümpfe, bzw. damit eine Schnittstelle, für die vom Entwickler zu programmierenden eigentlichen ASIO-Callbacks zur Verfügung.

Die für die Soundausgabe implementierte Callback-Funktion bildet die Gegenstelle für den Output des in Abschnitt 3.9 auf Seite 48 – Mixer besprochenen Mixers. Hierbei wird, je nach ASIO-Treiber, entweder manuell der dem momentanen Zyklus entsprechende Zeitstempel (Zeit; Sampleposition) über ASIO Routinen abgefragt oder das ASIO liefert den Zeitstempel implizit mit. Die Angaben werden in ein natives, im Allgemeinen dafür vorgesehene, 64-Bit Format konvertiert. Für die Ausgabekanäle wird ein geeigneter leerer Buffer vorbereitet. Daraufhin wird der Mixer, unter Angabe des Zeitstempels, angewiesen die Buffer zu füllen. Da das Sampleformat, das vom ASIO akzeptiert wird, sich von Treiber zu Treiber unterscheidet und eine Einstellmöglichkeit weder dokumentiert wurde noch gefunden werden konnte, werden die Samples in das vom Treiber akzeptierte Format konvertiert und anschließend in die Treiber-Buffer kopiert. Die Konvertierung an sich wird mit Hilfe von ASIO zur Verfügung gestellter Routinen bewerkstelligt. Hat ASIO den gesamten Vorgang nicht unterbrochen, da keine Zeitüberschreitung vorlag, ist das Funktionsende für das Backend zugleich das Ende des momentanen Zyklus. Bei vorzeitiger Unterbrechung spielt allerdings das ASIO den zum Unterbrechungszeitpunkt vorliegenden Stand der Treiber-Buffer ab. In dem Fall entweder noch Stille oder undefinierte Anzahl kopierter Samples. Einzelne Werte können sogar undefiniert sein, falls das Kopieren nicht Sampleweise atomar ist. Würde ein ASIO-Treiber native 32 Bit Floats akzeptieren, so wäre es möglich die gesamte Soundverarbeitung, von Laden der Sounds bis hin zum Treiber bzw. Hardware, mit 32 Bit Floats zu realisieren.

Allgemein wird vom ASIO-Backend versucht die vom Treiber gewünschte Buffer-Größe auf einen 2^n Wert einzustellen. Der Grund ist, dass die internen Zwischen-Buffer auf 2^m Größen limitiert sind. Sie werden automatisch mindestens doppelt so groß wie die ASIO-angelegt. In dem Fall bedeutet dies, dass $2 * 2^n = 2^m$ also $2^(n+1)$ ist und somit der Größenlimitierung der Buffer entspricht. Dadurch müssen weniger oft sehr kleine Fragmente von Sample-Blöcken gefiltert bzw. verarbeitet werden. Es wird an der Stelle empfohlen, bei Wiedergabeproblemen, die Puffergröße im ASIO-Treiber auf den Wert von 512 einzustellen. Dieser Wert hat den Hintergrund, dass bei Verwendung der implementierten Frequenzfilter eine voreingestellte FFT-Größe von 1024 gewählt wurde. Da die FFT-Größe dank der überlappenden Fenstern nur die halbe Buffer-Last verursacht, wirken sich die FFT-Transformationen auf die Buffer mit 512 Samples aus. Somit ist eine durchschnittliche eins zu eins Abarbeitung der Sampleblöcke möglich, d.h. in Schrittweiten von 512 Samples. Die FFT-Größe an sich ist variabel einstellbar und zwar in der zum Server gehörenden Konfigurationsdatei, welche beim Start der Anwendung ausgewertet wird. Falls eine geringere Latenzzeit gewünscht wird, kann eine kleinere FFT und ASIO-Buffer Größe eingestellt werden. Ist die Soundausgabe nicht robust genug, kann auf Kosten der Latenz ein größerer Wert verwendet werden. Bei einer Samplerate von $48kHz$ verursachen 512 Samples eine Latenzzeit von $512/48kHz = 10,7ms$. Die gesamte Latenz eines anzumeldenden Sounds, also zwischen Mix-In und verlassen der Soundkarte, unter Verwendung des Mixer-Filter im Mixer, setzt sich wie in Tabelle 3.1 gezeigt zusammen. Das Beispiel in der Tabelle bezieht sich auf die vorgeschlagenen bzw. momentan verwendete Standardwerte. Zu beachten ist, dass die Latenz des ASIO-Treibers auch höher liegen kann, als die Länge eines ASIO-Buffers. Dies ist vom Treiber abhängig und kann mittels ASIO-Routinen abgefragt werden, vorausgesetzt der verwendete Treiber liefert zuverlässige Werte. In Tabelle 3.2 wurde eine Reihe von Latenzen anhand der Formel und verschiedener Einstellungen erstellt. Grau markierte Zeilen deuten Einstellungen an, die im Bezug auf Design und zu erwartende Verhalten der Treiber als vernünftig angesehen werden. Die Tabelle kann zur Hilfe genommen werden, um die Einstellungen des Soundservers und ASIO-Treibers den Gegebenheiten anzupassen. Dabei gilt je niedriger die eingestellten Werte sind, desto kürzer ist die Latenzzeit aber damit auch geringer die Robustheit. Zu bedenken ist, das hier eine komplette Frequenzfilterung im Mixer realisiert wurde und diese die Latenzzeiten aufgrund Prinzip bedingter FFT-Fenster vergleichsweise recht hoch hält. Falls eine Latenzzeit von ca. 10 ms oder gar unter 10 ms benötigt wird, kann dies theoretisch, durch Verwendung von simpleren Filtern im Mixer zusammen mit minimalen Einstellungen, erreicht werden. Falls auf Ausgabefilterung komplett verzichtet wird, könnte überlegt werden die Mixer-Filter-Chains “kurzschließbar “ zu machen, d.h. zwischen Mix.In und Ausgabe wären im Mixer nur noch einer Reihe von Buffern. Somit wäre die minimalste theoretisch mögliche Latenz ein vielfaches von 64 Samples, da eine Einstellung der ASIO-Buffer unter 64 Samples vom Soundserver nicht unterstützt wird. Also wären es 64 Samples für ASIO-Treiber, $2 \cdot 64$ für einen Zwischen-Buffer und 64 für den Mix-In, also zusammen 256 Samples was eine Latenzzeit von 5.3 ms ergibt. Dies ist ein theoretischer eventuell nicht erreichbarer Wert, bzw. könnte eine stabile Soundausgabe damit nicht erreicht werden.

Im Prinzip ist das ASIO-Backend austauschbar. Bis auf die Namensgebung hier und in der Implementierung steht dem im Grunde nichts im Weg. Ein alternatives Backend muss lediglich die Schnittstelle bieten, die benötigt wird. Unter anderm muss es sich initialisieren

f	:=	48.000 Hz	Samplerate
s	:=	512 Samples	Größe des ASIO-Buffers
s_f	:=	1024 Samples	FFT-Fenstergröße
s_{ef}	=	$\frac{1}{2} * f_s$	Effektive Buffer-Last von FFT
A_l	$\geq$	s	Mindestlatenz des ASIO-Treibers
$MFout_l$	=	$\max(2 \cdot s, s + f_{es})$	Latenz am Ausgang der Mixer Filterung
$MFin_l$	=	$\max(2 \cdot s, f_{es} + mix)$	Latenz am Eingang der Mixer Filterung
Mix	=	s	Latenz im Mix-In
l_{ges}	$\geq$	$A_l + MFout_l + MFin_l + Mix$	Gesamte Latenz
	=	3072 Samples	
t_{ges}	=	l_{ges} / f	Gesamte Latenzzeit
	=	64,0 ms	

Tabelle 3.1.: Latenz Berechnung

lassen und mit dem Soundserver z.B. Samplerate und Kanalanzahl aushandeln. Die Sample-Transfer-Schnittstelle erwarten vom Backend lediglich ein Zeitstempel und zum Befüllen 32 Bit Float-Buffer, sowie ein Callback-Verhalten.

3.11. Timer

Für die zeitabhängige Anmeldung und das Samplegenaue Mischen der Sounds beim Mixer, wird ein geeignetes Verfahren gebraucht. Die Soundverarbeitung an sich arbeitet durchwegs mit Samplepositionen bzw. Zählern. Eine Anmeldung am Mixer wird auch durch eine Samplepositionsangabe bewerkstelligt. Da außerhalb der Soundverarbeitung oder gar über das Netzwerk mit den Samplepositionen nicht gearbeitet werden kann, wird ein Timer benötigt. Der Timer hat die Hauptaufgabe jeder Positionsangabe eine exakte Zeitangabe zuzuordnen. Der Timer muss also bei Angabe einer Zeit, die zugehörige Sampleposition liefern und umgekehrt. Mit der Position(Zeit) "Funktion" kann ein Sound zeitlich samplegenau im Mixer platziert werden. Die "Funktion" Zeit(Position) wäre wiederum z.B. für Zeit bezogene Feedbacks an Clients nötig. Weiterhin ist eine geeignete Zeitsynchronisation zwischen dem Soundserver und seinen Clients notwendig. Denn ansonsten, wenn die internen Zeiten der Parteien zu unterschiedlich sind, wird der angefragte Sound deutlich zum falschen Zeitpunkt Abgespielt. Falls die Anfragen auf lokalen, zudem gänzlich asynchronen Systemzeiten basieren, wäre auch eine Varianz von Sekunden, Minuten oder gar Jahren möglich. Abbildung 3.21 stellt ein Grundgerüst bzw. Interface eines für den Soundserver geeigneten Timers dar.

Da während dieser Arbeit weder Synchronisationsmittel noch die verwendete Synchronisationsart der Clients fest stand, ist die Implementierung des Timers nur auf das Notwendigste

ASIO-Buffer	FFT-Fenster	Latenz / Samples	Latenz / ms
1024	256	6144	128,0
512	256	3072	64,0
256	256	1536	32,0
128	256	768	16,0
64	256	512	10,7
1024	512	6144	128,0
512	512	3072	64,0
256	512	1536	32,0
128	512	1024	21,3
64	512	768	16,0
1024	1024	6144	128,0
512	1024	3072	64,0
256	1024	2048	42,7
128	1024	1536	32,0
64	1024	1280	26,7

Tabelle 3.2.: Latenz Liste

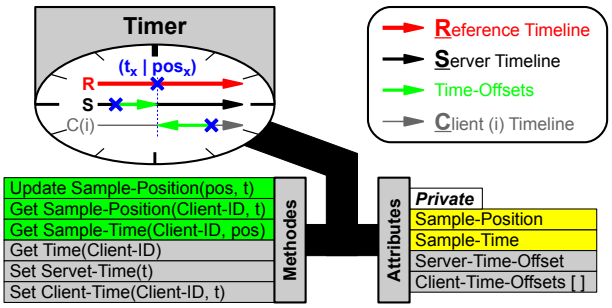

Abbildung 3.21.: Timer

begrenzt geblieben. Es wurde darauf geachtet, die Implementierung offen für verschiedene Möglichkeiten zu halten. Der hier vorgestellte Timer stellt weitgehend ein Entwurf dar, denn die bereits implementierten Teile des Timers basieren gänzlich auf lokale Systemzeit.

Wie Anfangs erwähnt wird eine Sampleposition zu Zeit Zuordnung gebraucht. Dies wird bereits, wie in Abschnitt 3.10 auf Seite 54 – ASIO-Backend erwähnt mit Hilfe der vom ASIO bereitgestellten Zeitstempeln erledigt. Hierfür wird vom Mixer bei jedem Callback-Aufruf die Timer Methode “Update Sample-Position“ aufgerufen. Das Problem ist hier, dass sich die Zeitangaben auf die lokale Systemzeit beziehen. Benötigt wird aber für Soundserver und Clients eine gemeinsame Zeit, also globale bzw. wie hier genannt Referenzzeit.

In der Timer-Abbildung ist ein mögliches allgemein gehaltenes Synchronisationsverfahren anhand eines Zeitdiagramms dargestellt. Sowohl Soundserver als auch Clients sollen sich direkt oder indirekt über eine Referenzzeit verständigen. Mit direkt ist hier gemeint, dass die abgebildeten Client-Offsets null sind, also dass die Clients auf die Referenzzeit synchronisiert sind. Somit ist entweder die Soundserver eigene Systemzeit die Referenzzeit, also Server Offset gleich null oder es ist ein Offset bekannt, um wie viel die Soundserverzeit von der Referenzzeit abweicht. Beim indirekten Fall sind die Anfragen der Clients nicht auf die Referenzzeit synchronisiert, aber dem Timer ist der individuelle Offset der Clients und eventuelle Offset des Soundservers bekannt. Somit können alle Anfragen und Positionsaktualisierungen auf die Referenzzeit bezogen werden.

Hier wurde oft gesagt, dass die Offsets dem Timer bekannt sein müssen. Dies erfordert gewisse Synchronisationstechniken, die über diese Arbeit hinausgehen und ein allgemeines Problem darstellen, welches außerhalb des Soundserver zu lösen wäre, da Zeitsynchronisation nicht nur für die Soundverarbeitung benötigt wird. Der Timer hier kann als eine Schnittstelle zur eigentlichen Synchronisationskomponente verstanden werden. Hierbei sollte diese Komponente den Timer regelmäßig bezüglich der Time-Offsets aktualisieren. Eine beliebig genaue Serversynchronisation ist unter Umständen nicht notwendig, den ein Sample hat bei 48 kHz eine Zeitspanne von etwa. 0.02 ms, d.h. eine höhere Synchronisationspräzision sollte spätestens ab da nicht mehr erforderlich sein. Im Normalfall sollten bis zu mehreren hundert daneben liegender Samples keine hörbare Abweichungen verursachen.

Als Beispiel für die Soundserver Synchronisation wäre folgendes Szenario denkbar. Eine externe Synchronisationseinheit oder eine Instanz dieser wird lokal auf dem Soundserver-Rechner ausgeführt. Die Einheit benutzt ein geeignetes Synchronisationsmittel, um im Bezug zur Referenzzeit zu stehen. Mit einem regelmäßigen punktuellen Abgleich mit der lokalen Systemzeit, ermittelt die Einheit, das dem Soundserver zuzuschreibende Time-Offset. Durch eine geeignete Kommunikationsschnittstelle wird das Offset stets dem Timer mitgeteilt. Als Kommunikationsmittel wäre auch eine Kommunikation über TCP/IP, d.h. dann über "localhost" denkbar und könnte durch Anpassung des Comm-Interfaces erreicht werden. Eine Variante davon wäre, wenn die Systemzeit durch die Synchronisationseinheit an sich mit der Referenzzeit synchronisiert werden würde. Sind die Clientanfragen ebenfalls, auf irgend eine Art und Weise, auf die Referenzzeit synchronisiert, so werden im Timer überhaupt keine Offsets benötigt und dieser kann dann komplett mit der Systemzeit d.h. "Referenzzeit" arbeiten.

Da der Timer bei Beibehaltung des internen Interfaces komplett austauschbar ist, ist theoretisch jegliches denkbare Synchronisationsverfahren implementierbar. Eine Einbindung als DLL für direkte Austauschbarkeit wäre auch denkbar.

3.12. Comm-Interface

Für die Kommunikation des Soundservers mit externen Komponenten, d.h. im Allgemeinen mit Clients, wird hier der Entwurf von dem "Comm-Interface" dokumentiert. Eine minimaler

Funktionsumfang wurde bereits implementiert, um grundlegendes Zeitverhalten des gesamten Designs subjektiv bewerten zu können, was in Kapitel 4 auf Seite 63 – Test-Client näher erläutert wird. Der prinzipielle Kommunikationsverlauf ist in Abbildung 3.22 dargestellt. Gestrichelte Pfeile bedeuten hier eine pro Verbindung stattfindende einmalige Anfrage oder Aktion. Durchgezogene Pfeile stellen den für die gesamte Verbindung offen stehenden Kommunikationskanal dar. Die Indizes "(i)" beziehen sich auf einen individuellen Client, d.h. auf dessen Kommunikationskanal. Das bedeutet, die mit "(i)" gekennzeichneten, Sachverhalte werde pro Verbindung angelegt oder durchgeführt. Für die bestehende Realisierung wurde "TCPComChannel" der "VISlib" Bibliothek [29] verwendet. Der hier vorgestellte Entwurf ist, wie in der Abbildung gekennzeichnet, ebenfalls an "TCPComChannel" angelehnt. Diese realisieren eine Kommunikation über das TCP/IP Protokoll, welches im Internet und Ethernet Verwendung findet.

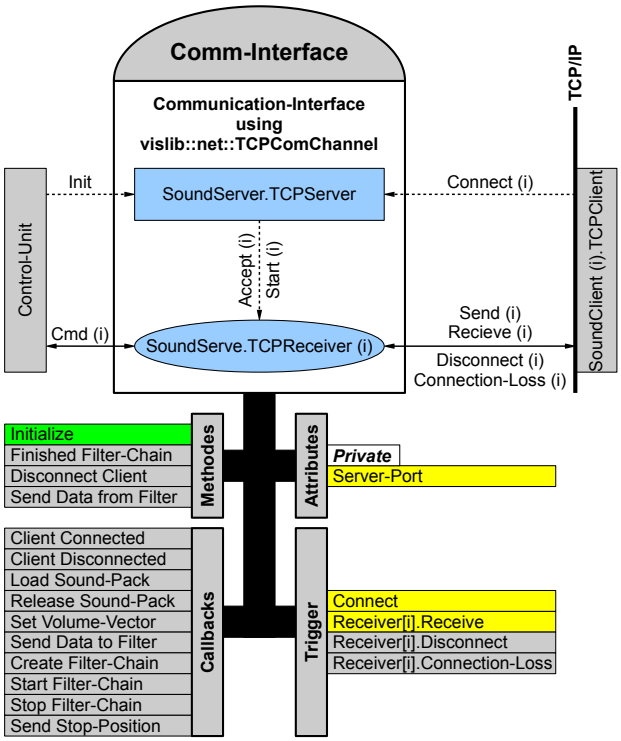

Abbildung 3.22.: Comm-Interface

So Verbindet sich ein Client mit dem Soundserver über die "Connect" Anfrage. Die nebenläufige Serverkomponente des Comm-Interfces akzeptiert die Verbindung. Beim Akzeptieren wird jeder Client an der Client-Library angemeldet, zudem wird jedem Client ein Receiver als Kommunikationspartner zur Verfügung gestellt. Der Receiver wird als eigenständiger Thread angelegt und gestartet, was im Grunde eine beliebige, durch Rechenleistung begrenzte, Anzahl von Clients ermöglicht. Ab Aktivierung des Receiver-Threads, bildet dieser die eigentliche Schnittstelle zwischen dem Client und dem Soundserver. Somit reicht der Receiver Client-Anfragen geeignet an die Control-Unit oder direkt an eine der Soundserver

Komponenten, wie dem Timer oder erledigt ein spezielle Aufgabe gar eigenständig. Für Anfragen oder Rückmeldungen in die andere Richtung, z.B. von der Control-Unit zum Client, ist der Receiver ebenfalls zuständig. Bei Trennung oder Verlust der Verbindung muss der Receiver entsprechend reagieren. So sind z.B. Sound-Packs, die nur von dem einen Client verwendet wurden, wieder frei zu geben. Für die meisten Funktionen soll die Control-Unit als Leitstelle verwendet werden. Ändert sich intern z.B. an der Soundverarbeitung etwas, reicht es dann den entsprechenden Teil der Control-Unit anzupassen, um die Schnittstelle zu anderen Komponenten, insbesondere Comm-Interface, weitgehend bewahren zu können.

Die Tabelle 3.3 listet einen möglichen Entwurf für einen Grundsatz an Nachrichten, die der Server und entsprechend die Clients verstehen sollen. Die Tabellenformatierung der "Byte – Structure" Spalten in der "Massege Contennd List" Tabbele entsprechen in Kurzform der "Massege Head" Tabelle. Hierbei entspricht die erste Zeile einer Bezeichnung der Byte-Belegung und kann als Variablennamen dienen. Die zweite Zeile nimmt Bezug auf den internen Datentyp und die letzte auf dessen Größe in Bytes. Eine Nachricht besteht aus einer Anordnung dieser Bytes, deren Anordnung der spaltenweise von links nach rechts angeordneten Zellen entspricht. Wie in der "Message Head" Tabelle gezeigt besteht eine Nachricht grundsätzlich aus drei gruppierten Byte-Folgen. Die erste gibt die gesamte Größe der Nachricht in Bytes an, damit das Comm-Interface weiß, wann eine Nachricht zu Ende ist, also ab wann die Nachricht verarbeitet werden kann. Die zweite Byte-Folge gibt die Art der Nachricht als eine eindeutige H-ID an, damit das Comm-Interface die der Nachricht entsprechenden Routinen durchführen kann. Schließlich enthält die letzte Byte-Folge den eigentlichen Inhalt der Nachricht. Auf diesen Inhalt nimmt die "Messege Contennd List" Tabelle Bezug.

Die "Massege Contennd List" Tabelle ist nach den H-IDs der Nachrichten sortiert und geht zeilenweise auf einzelne Nachrichten ein. Unter "Enum-ID" Wird im Grunde die Kommunikationsrichtung und zeitliche Abhängigkeit gezeigt. Der Bezug für die Kommunikationsrichtung ist der Soundserver, d.h. Request stellt eine Client-Anfrage dar und Answer die bestätigende Antwort vom Server. Also stehen beide in einem engen Zeitkontext und auf eine Anfrage hat eine Antwort zu folgen. Entweder ist die Antwort eine simple Bestätigung, dass die Nachricht empfangen wurde, oder gibt in Bezug auf die Anfrage spezialisierte Informationen weiter. Mit Feedback ist hier gemeint, dass der Server dem Client z.B. mitteilt das eine Abspielanfrage ihr Ende erreicht hat und steht nicht in einem engen Zeitkontext. Die Spalte Enum-ID gibt die vorgeschlagene interne Bezeichnung der Nachricht an. Unter Description wird kurz in Englisch die Funktion der Nachricht beschrieben. Schließlich stellt Byte – Structure die Byte weise Zusammensetzung der Nachricht dar.

Die letzten zwei Spalten gehen auf die direkte Kommunikation von einem Client mit einem Filter ein und umgekehrt. Hierbei dient der Server nur als Vermittler und gibt die rohen Daten einfach weiter. Die Interpretation wird dem Filter überlassen. Eine Interpretation durch das Comm-Interface würde verursachen, dass bei jedem neuen Filter oder Änderung eines Filter, die Routinen des Comm-Interfaces abgeändert oder erweitert werden müssten. Die Interpretation der Filter, kann ein dem hier entsprechendes Design verwenden. Somit wird für jeden Filter, als Entwurf, eine eigenen solche Kommunikationstabelle benötigt.

Message Head			
Label	MSGSize	MSGHeadID (H-ID)	MSGContent
Type	__int16	__int16	void
Size / Bytes	2	2	MSGSize – 4

Message Content List										
H-ID	Type	Enum-ID	Description	Byte – Structure						
0	Answer	MSG_StdAnswer	Standard response from Soundserver – whether success or fail	MSGHeadType __int16 2	ReqSucceeded bool / __int8 1					
1	Request	MSG_KeyCode	Keycoode (wxWidgets) of pressed Key without modifiers	KeyCode __int16 2	VolVecChannels __int8 1	VolVecElement[] float32 4VolVecChannels				
2	Request	MSG_VolumeVector	Update Volume-Vactor of a Filter-Chain	VolVecChainID __int16 2	VolVecChannels __int8 1	VolVecElement[] float32 4VolVecChannels				
3	Request	MSG_LoadSoundPack	Register corresponding sounds for use	StringSize __int16 2	String const char* 1 * StringSize					
3	Answer	MSG_LoadSoundPack	Gives ID of the registered Sound-Pack	SoundPackID __int16 2						
4	Request	MSG_Release_SoundPack	Client releases this Sound-Pack	SoundPackID __int16 2						
5	Request	MSG_createFilterChain	Just create a Filter-Chain	FilterSet[] __int16 2(FilterSet[0] + 1)						
6	Request	MSG_createFilterChain	Create a Filter-Chain linked with a registered sound	FilterSet[] __int16 2(FilterSet[0] + 1)	sndPCK_ID __int16 2	snd_ID __int16 2				
7	Request	MSG_createFilterChain	Create a Filter-Chain linked with a loaded sound and set an Volume-Vector	FilterSet[] __int16 2(FilterSet[0] + 1)	sndPCK_ID __int16 2	snd_ID __int16 2	sndPos unsigned __int64 8	VolVecChannels __int8 1	VolVecElement[] float32 4VolVecChannels	
8	Request	MSG_createFilterChain	Create a Filter-Chain linked with a loaded sound, set an Volume-Vector and setup to play (see startFilterChain).	FilterSet[] __int16 2(FilterSet[0] + 1)	sndPCK_ID __int16 2	snd_ID __int16 2	sndPos unsigned __int64 8	VolVecChannels __int8 1	VolVecElement[] float32 4VolVecChannels	
				StartTimeCode unsigned __int64 8	FadeInDuration unsigned __int64 8	FadeInKind __int16 2	StopTimeCode unsigned __int64 8	FadeOutDuration unsigned __int64 8	FadeOutKind __int16 2	
				Loop bool / __int8 1	OneWayFChain bool / __int8 1					
5	Answer	MSG_createFilterChain	Answers the given IDs of created Filter-Chain and filters	FilterChainID __int16 2	FilterIDs[] __int16 2 * (FilterIDs[0] +1)					
9	Request	MSG_startFilterChain	Start an existing Filter-Chain at a timecode with a setupable fade-in and fade-out	FilterChainID __int16 2	sndPos unsigned __int64 8	StartTimeCode unsigned __int64 8	FadeInDuration unsigned __int64 8	FadeInKind __int16 2		
				StopTimeCode unsigned __int64 8	FadeOutDuration unsigned __int64 8	FadeOutKind __int16 2	Loop bool / __int8 1	OneWayFChain bool / __int8 1		
10	Request	MSG_stopFilterChain	Stop a prior started Filter-Chain to absolute given stopTimeCode	FilterChainID __int16 2	StopTimeCode unsigned __int64 8	FadeOutDuration unsigned __int64 8	FadeOutKind __int16 2			
11	Request	MSG_signalStopFilterChain	Tries to stop Filterchain according to Delay-Filters	FilterChainID __int16 2	StopTimeCode unsigned __int64 8	FadeOutDuration unsigned __int64 8	FadeOutKind __int16 2			
12	Feedback	MSG_finishedFilterChain	Server signals a stopped / finished Filter-Chain	FilterChainID __int16 2	Destroyed bool / __int8 1					
13	Feedback	MSG_FilterChainReset	Any requested Filter-Chain is not valid anymore							
14	Feedback	MSG_SoundPackReset	Any registered FilterChain and Sound-Pack is not valide anymore							
15	Request	MSG_Disconnect	Client disconnect from Server. Related resources are free for dispose							
Filter Communication										
1000	Request	MSG_ClientToFilter	Send arbitrary data to filter. Data to be interpreted by filter	Size unsigned __int64 8	Data void Size					
1001	Request	MSG_FilterToClient	Send arbitrary data from filter. Data to be interpreted by client	Delay unsigned __int64 8	Data void Size					

Tabelle 3.3.: Massege-List

Test-Client 4

Eine Kommunikation über Netzwerk, kann hohe Latenzzeiten aufweisen. Zudem können Latenzeinzeiten stark variieren, was zur Folge hat, dass Datenpakete in falscher Reihenfolge bei der Gegenstelle ankommen können. Die Reihenfolge der Datenpakete ist bei dem Design des Soundserver weitgehend unkritisch. Datenpakete werden pro Client separat verarbeitet, also spielt die zeitliche Reihenfolge zwischen Client-Datenpaketen hier keine Rolle. Die Datenreihenfolge eines separaten Clients wird auf Anfrage-Antwort Basis implizit in richtiger Reihenfolge gehalten. Somit spielen nur die eigentliche Latenzzeiten eine Rolle bei der Kommunikation. Um erste akustische Eindrücke bezogen auf Latenzzeit zu erhalten, wurde eine Test-Client erstellt. Der Soundserver kann auf bestimmte Tastendrücke einen Sound abspielen. Mit einem entsprechenden Sound-Pack kann die Tastatur des Soundservercomputers als einer Art Klavier verwendet werden. Da der implementierte Test-Client über TCP/IP Key-Codes versenden kann, kann sowohl die Tastatur des Soundservers als auch die des Clients zu einem Klavier umfunktioniert werden. Somit kann subjektiv über die Server Tastatur die Latenzzeiten der Soundverarbeitung beurteilt werden und über die Tastatur des Clients zusätzlich die Latenzzeiten der zugrunde liegenden Kommunikation. Bei der Erprobung der Latenzen waren bei einer reaktionsstarken Soundverarbeitung, subjektiv keine negativ auffallenden Netzwerklatenzen wahrzunehmen.

Wird ein Soundclient z.B. dazu benutzt zwei kollidierende Objekte zu rendern und will diese durch einen lautstarken Knall untermauern, so gibt es grundsätzlich zwei Arten eine Soundanfrage zu starten. Das Senden der Soundanfrage kann der Client zum Zeitpunkt der eigentlichen Kollision schicken und ist somit allen Latenzen ausgeliefert, d.h. insbesondere den Netzwerk und Mixer Latenzen. Eine Kollision kann schon recht frühzeitig erkannt werden, da physikalische Objekte einer gewissen Trägheit unterliegen. Somit kann der Client unter Ausnutzung solcher in voraus erkennbaren Fälle eine Soundanfrage mit einem Zeitstempel starten. Solche Anfragen unterliegen dann nur der Treiber bzw. Hardware Latenz. Allerdings setzt es voraus, dass die Anfragen früh genug gestartet werden, also dass sie vor dem Latenzhorizont liegen. Die Treiber und Hardware Latenzen können in diesem Fall theoretisch auch ausgeglichen werden, falls dem Soundserver ein entsprechendes Offset mitgeteilt werden kann.

Desweiteren wurde bereits eine erste Nachricht aus der Kommunikationstabelle 3.3 implementiert. Mit dieser Nachricht können Volume-Vektoren an eine Filter-Chain übertragen werden. So wurde im Test-Client eine grafische Einheit implementiert, anhand der man intuitiv die Ausrichtung eines Sounds steuern kann. Hierbei wird das Fenster in $N-1$ Bereiche aufgeteilt, die Anordnung der Bereiche entspricht der prinzipiellen Richtung eines Lautsprechers. Abbildung 4.1 stellt den Fall für $N=8$ Kanäle dar, also den 7.1 Surround-Sound Fall. Der nicht abgebildete Kanal ist der für den Subfoofer zuständige LFE und entsteht hier als normalisierte Summe der anderen Kanäle. Es stehen dem Benutzer zwei Kreise zur Verfügung, mit denen er interagieren kann. Jeder Kreis ist stellvertretend einem Stereo-Soundkanal. Berührt ein Kreis den Bereich eines Lautsprechers, so wird dieser über den Lautsprecher hörbar. Der Kreisanteil, der im Lautsprecherbereich liegt, ist allerdings

stellvertretend für die Lautstärke des betreffenden Lautsprechers. Nicht nur der Anteil des Kreises beeinflusst die Lautstärke sondern auch die Entfernung vom Ursprung.

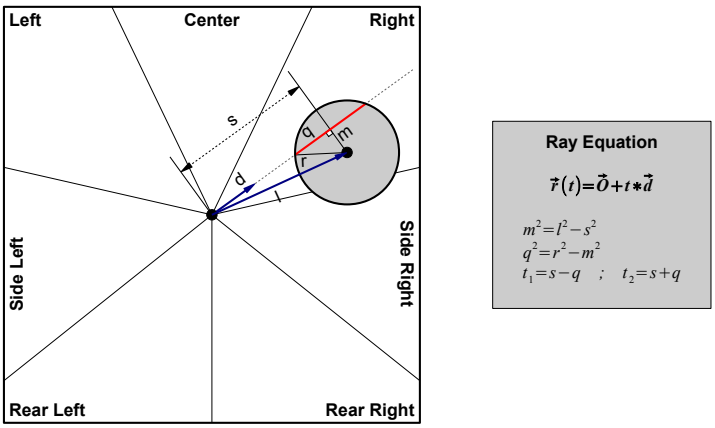

Abbildung 4.1.: Test-Client — Volume-Vector-Rendering: Aus [30] entnommen

Die Erzeugung eines Volume-Vektors wird durch eine Simple Strahlenverfolgung realisiert. Die Strahlen gehen vom Ursprung aus und decken rundum die gesamte Fläche ab. Durchwandert ein Strahl einen Kreis, so wird die durchwanderte Länge dementsprechenden Lautsprecherbereich dazu gezählt. Die somit entstandenen charakteristischen Werte bilden, geeignet normalisiert, einen Volume-Vektor. Dies wird für beide Stereo-Kanäle separat berechnet. Die in Abbildung 4.1 eingearbeitete Strahlengleichung stimmt nur für den dargestellten Fall. Fallunterscheidungen werden in der Quelle [30] behandelt. So wird bei der Implementierung auch nur ein Teil der Strahlengleichung verwendet. Für den zu berechnenden Effekt reicht es den Wert für q zu kennen.

Die Berechnung stimmt zwar nicht mit einer echten Volume-Vektor-Erzeugung überein, ist aber intuitiv, kanaldiskret und grafisch steuerbar. Somit kann die grafische Einheit dazu benutzt werden, um die Anlage zu testen. Z.b kann die Zuordnung der Kanäle im Soundserver und der Anlage damit überprüft werden. Durch geeignete Sounds kann auch subjektiv das prinzipielle Klangbild der Lautsprecher einer Anlage beurteilt oder eingestellt werden.

Zusammenfassung und Ausblick

5

Zusammenfassung

Im Zuge dieser Arbeit wurden die soundverarbeitenden Komponenten und Schnittstellen des "Flexiblen Soundserver für clusterbasierte VR" entworfen und implementiert. In Abschnitt 3.1 auf Seite 13 – Soundserver Übersicht wird eine Übersicht der entworfenen Komponenten gegeben, welche in Abbildung 3.1 graphisch dargestellt ist. Der Soundserver ist im Stande Sounddateien in den Arbeitsspeicher zu laden (Abschnitt 3.4 auf Seite 22 – Sound-Store) und diese beliebig oft, parallel und zum beliebigen Zeitpunkt wiederzugeben (Abschnitt 3.9 auf Seite 48 – Mixer). Um Sounds individuell zu filtern wurden diverse beliebig zusammenschaltbare Filter implementiert (Abschnitt 3.8 auf Seite 31 – Filter). Der Schwerpunkt der erstellten Filter lag bei der Filterung im Frequenzbereich. Für eine qualitative hochwertige und latenzarme Soundausgabe wird die ASIO-Schnittstele verwendet (Abschnitt 3.10 auf Seite 54 – ASIO-Backend). Ein Entwurf und erste Implementierungsansätze für die Netzwerkkommunikation waren ebenso Thema dieser Arbeit.

Die Implementierung setzt an vielen Stellen auf parallele Verarbeitung und profitiert von modernen Prozessoren. Für die Synchronisation der Treads wurden bestehende Mechanismen verwendet oder anhand dieser Synchronisationsabläufe entworfen (Abschnitt 3.6.2 auf Seite 27 – Synchronization). Generell wurde auf das Echtlaufzeitverhalten geachtet (z.B. Abschnitt 3.3.1 auf Seite 18 – Dynamic-Array und Abschnitt 3.6.1 auf Seite 24 – Buffering). Die Implementierung der Komponenten ist weitgehend modular, d.h. Komponenten sind individuell änderbar und austauschbar.

Ausblick

Das zugrunde liegende Soundserverdesign kann herangenommen werden um weiterführende Funktionen auf zu setzen. So ist es denkbar den Soundserer zu einer vollständigen 3D soundverarbeitenden Einheit auszubauen. Durch Erweiterung oder Anpassung weniger Grundstrukturen kann Streaming über Netzwerk oder Sounderfassung realisiert werden. Mit angepasstem Cacheverhalten, kann das Vorladen der Sounds stark eingeschränkt oder bei Verwendung moderner Massenspeicher weitgehend vermieden werden. Somit würde sowohl die Vorladezeit der Sound-Packs als auch der Arbeitsspeicherverbrauch minimiert werden.

Der implementierte Soundserver kann im Grunde als ein Streamprozessor angesehen werden. Und zwar ein Streamprozessor der n-Eingabeströme zu m-Ausgabeströme verarbeitet. Sowohl die Eingangsströme als auch die Ausgabeströme können individuelle Berechnungen

durchführen. Eine Generalisierung des Mix-Ins, würde eine beliebige Wandlung von n zu m Streams erlauben.

Eine völlig andere Sichtweise wäre ein Vergleich des Soundservers mit einem Neuron ([31] behandelt neuronale Netze). Bei einer Mono-Ausgabe könnte der Soundserver n durch Volume-Vektoren gewichtete Reize über Mix-In verarbeiten. Somit wäre der Soundserver bereits als ein simples Neuron einsetzbar. Durch die Ein- und Ausgabefilterung könnte ein Pre- bzw. Post-Processing der Reize ermöglicht werden, somit könnte z.B. auch eine verzögerte oder zeitlich verzerrte Rückkopplung für Lernfunktionen ermöglicht werden. Durch m Ausgabekanälen und geeigneten Volume-Vektoren wäre mit einem Soundserver ein m-fach Neuro möglich, d.h. m Neuronen die auf dieselben gefilterten Reize unterschiedlich reagieren. Durch die Volum-Vektoren wäre auch ein beliebig mit den Reizen gekoppeltes m-fach Neuron möglich. Mit ersetzen der Summation im Mix-In durch komplexere Funktionen, wäre ein komplexeres Neuron möglich.

Benchmarks

A

	Machine-0 / s	Machine-1 / s
Kind	Laptop	Desktop
CPU	Intel T720	AMD Phenom II X4 965
RAM	3 GB	4 GB
OS	Win 7 x64	Win 7 x64

Description	S	S
Dynamic-Array		
250 * 10^6 – Float-Cells 100 Times: Create(Size = 0); guardedFill; Delete	230	154
250 * 10^6 – Float-Cells 100 Times: Create(Size = full); guardedFill; Delete	145	100
250 * 10^6 – Float-Cells 100 Times: Create(Size = 0); 250*10^3*addSize(10^3) guardedFill; Delete	146	101
250 * 10^6 – Float-Cells; preCreat(Size = full) 100 Times: speedFill; Delete	73	62
250 * 10^6 – Float-Cells; preCreate(Size = full); preFill 100 Times: speedRead; Delete	24	14
50 * 10^6 – (Float*)-Cells 20 Times: Create(Size = 0); guardedFill; Delete	222	108
C++ Array		
250 * 10^6 – Float-Cells 100 Times: Create(Size = full); Fill; Delete	122	51
250 * 10^6 – Float-Cells; preCreate(Size = full); 100 Times: Fill; Delete	47	19
250 * 10^6 – Float-Cells; preCreate(Size = full); preFill 100 Times: Read; Delete	21	14
250 * 10^6 – Float-Cells; 100 Times: Create(Size = full);memset; Delete	98	46
Sound (Stereo-Out) (playback of unfiltred mono sounds)		
Parallel playback	500	1650
Parallel / Queued playback	170 / 1530	570 / 5130

Tabelle A.1.: Benchmark Ergebnisse

GUI-Übersicht

B

B.1. Soundserver

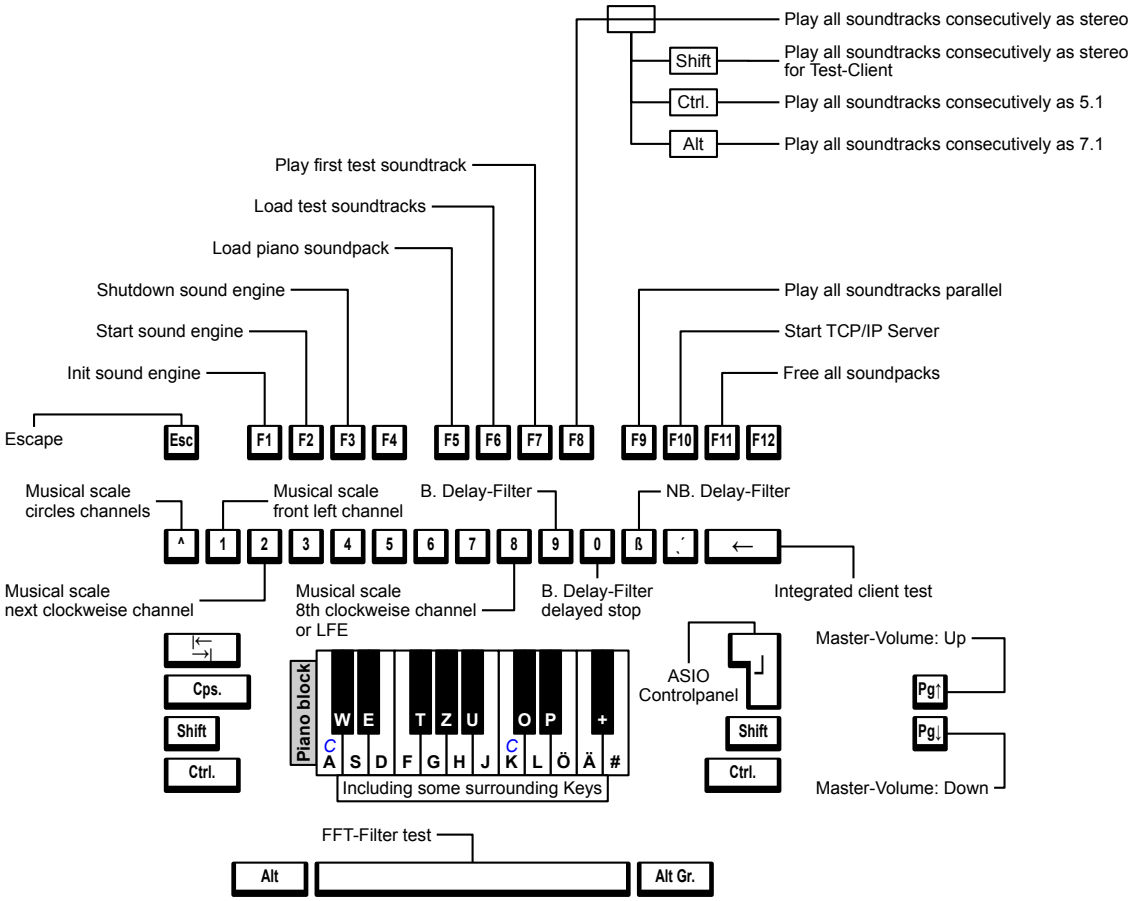

Abbildung B.1.: Soundserver — Keyboard

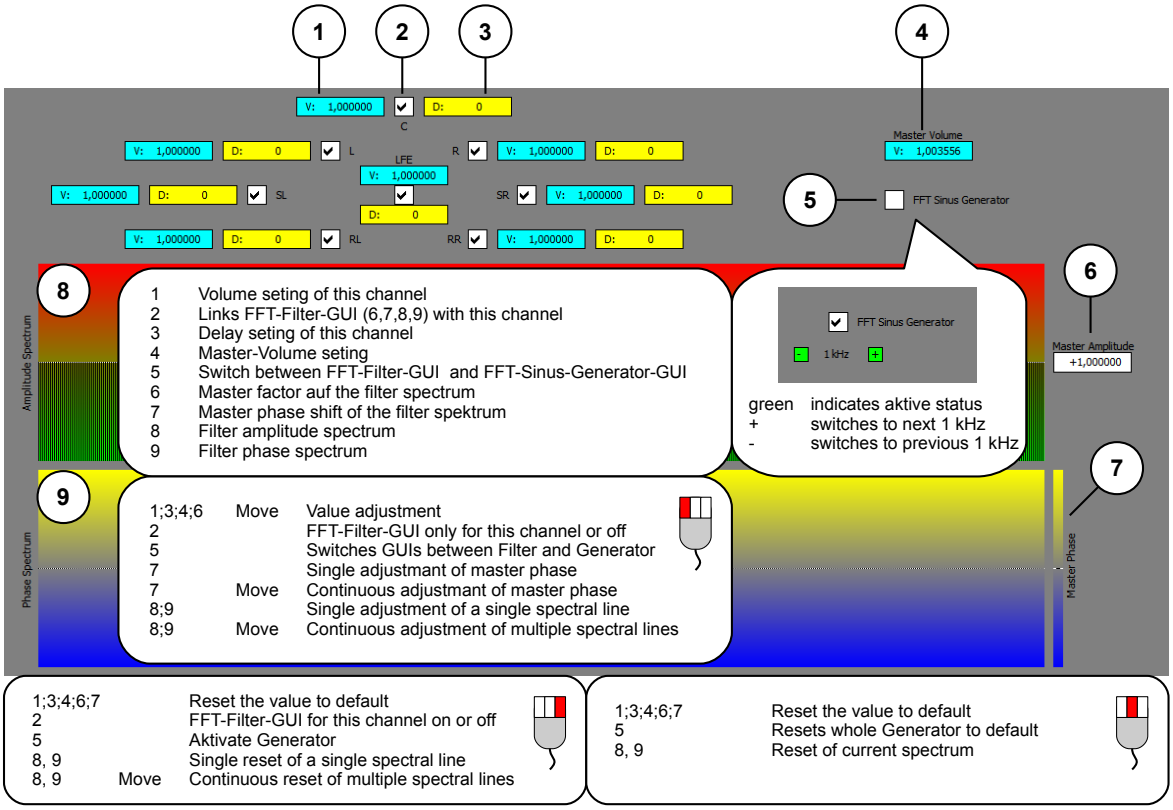

Abbildung B.2.: Soundserver — GUI

B.2. Test-Client

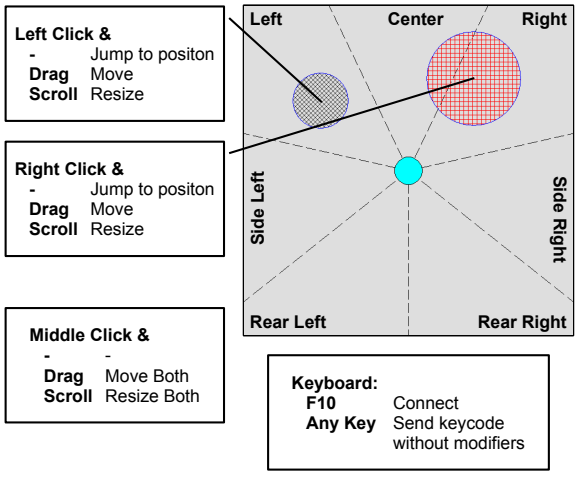

Abbildung B.3.: Test-Client — GUI

Literaturverzeichnis

[1] *Übergabe des Visualisierungsinstituts der Universität Stuttgart*. http://www.visus.uni-stuttgart.de/index.php?id=34&no_cache=1&tx_ttnews[tt_news]=50&cHash=3e2e12143428937da9c3f9026af0b554. Version: 11 2010 (Zitiert auf Seite 7)

[2] Reina, Guido: *Aufgabenbeschreibung*. 02 2011. – Liegt dem Vertag der Studienarbeit bei (Zitiert auf Seite 7)

[3] Scheffler, Stefan: *Steinberg Audio Streaming Input Output Specification - Development Kit 2.2*. http://www.steinberg.net/de/company/developer.html. Version: 02 2006 (Zitiert auf Seite 7)

[4] *Network-Integrated Multimedia Middleware*. http://de.wikipedia.org/wiki/Network-Integrated_Multimedia_Middleware. Version: 02 2010 (Zitiert auf Seite 9)

[5] *Features and Plug-ins of NMM*. http://www.motama.com/nmmdocs_features.html. Version: 04 2010 (Zitiert auf Seite 9)

[6] *Audio Stream Input/Output*. http://en.wikipedia.org/wiki/Audio_Stream_Input/Output. Version: 07 2011 (Zitiert auf den Seiten 10 und 11)

[7] *VLC media player*. http://de.wikipedia.org/wiki/VLC_media_player. Version: 07 2011 (Zitiert auf Seite 10)

[8] *GStreamer:features*. http://gstreamer.freedesktop.org/features/index.html (Zitiert auf Seite 10)

[9] *PulseAudio*. http://de.wikipedia.org/wiki/PulseAudio. Version: 08 2011 (Zitiert auf Seite 11)

[10] *DirectShow*. http://en.wikipedia.org/wiki/DirectShow. Version: 05 2011 (Zitiert auf Seite 11)

[11] *wxWidgets*. http://www.wxwidgets.org/ (Zitiert auf den Seiten 16 und 22)

[12] *Abtastung (Signalverarbeitung)*. http://de.wikipedia.org/wiki/Abtastung_(Signalverarbeitung). Version: 08 2011 (Zitiert auf den Seiten 16 und 17)

[13] *Nyquist-Shannon-Abtasttheorem*. http://de.wikipedia.org/wiki/Abtasttheorem. Version: 07 2011 (Zitiert auf den Seiten 17 und 36)

[14] *Quantisierung*. http://de.wikipedia.org/wiki/Quantisierung. Version: 07 2011 (Zitiert auf Seite 17)

[15] *Dithering (Audiotechnik)*. http://de.wikipedia.org/wiki/Dithering_(Audiotechnik). Version: 01 2011 (Zitiert auf Seite 17)

[16] *Vorbis audio compression*. http://xiph.org/vorbis/ (Zitiert auf Seite 22)

[17] *RIFF WAVE*. http://de.wikipedia.org/wiki/RIFF_WAVE. Version: 07 2011 (Zitiert auf Seite 22)

[18] Davidson, Nathan: *How to Load a Wave File*. http://www.cpp-home.com/tutorials/333_1.htm (Zitiert auf Seite 22)

[19] Souza, César: *Converting audio bit depths in C#*. http://crsouza.blogspot.com/2009/08/converting-between-different-audio.html. Version: 08 2009 (Zitiert auf Seite 22)

[20] Bourke, Paul: *Fast Fourier Transform*. http://paulbourke.net/miscellaneous/dft/. Version: 06 1993 (Zitiert auf den Seiten 34 und 38)

[21] *Fourier-Transformation*. http://de.wikipedia.org/wiki/Fourier-Transformation. Version: 07 2011 (Zitiert auf Seite 34)

[22] *Diskrete Fourier-Transformation*. http://de.wikipedia.org/wiki/Diskrete_Fourier-Transformation. Version: 07 2011 (Zitiert auf Seite 34)

[23] *Schnelle Fourier-Transformation*. http://de.wikipedia.org/wiki/Schnelle_Fourier-Transformation. Version: 07 2011 (Zitiert auf Seite 34)

[24] *FFTW*. http://www.fftw.org/ (Zitiert auf den Seiten 34 und 44)

[25] sprut: *Fast Fourier Transform (FFT)*. http://www.sprut.de/electronic/pic/16bit/dsp/fft/fft.htm. Version: 08 2009 (Zitiert auf den Seiten 36 und 38)

[26] Bore, Chris: *FFT window functions*. http://www.bores.com/courses/advanced/windows/files/windows.pdf (Zitiert auf den Seiten 38 und 39)

[27] *Window function*. http://en.wikipedia.org/wiki/Window_function. Version: 08 2011 (Zitiert auf Seite 41)

[28] Weisstein, Eric W.: *Hanning Function*. http://mathworld.wolfram.com/HanningFunction.html (Zitiert auf Seite 40)

[29] *VISlib™*. https://vis-web.informatik.uni-stuttgart.de/trac/vislib/wiki/WikiStart (Zitiert auf Seite 59)

[30] Mannl, Uwe: *Grundlagen zur effizienten Behandlung von Strahlen und Schnittpunktberechnungen im Rendering-Prozess*. http://www-gs.informatik.tu-cottbus.de/projektstudium2006/doku/Strahlen_in_der_CG.pdf. Version: 11 2006 (Zitiert auf Seite 64)

[31] *Artificial Neural Networks/Print Version*. http://en.wikibooks.org/wiki/Artificial_Neural_Networks/Print_Version. Version: 01 2008 (Zitiert auf Seite 66)

Alle URLs wurden zuletzt am 14.08.2011 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Alexander Derheim)