Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Darjush Bahreini

Implementation und Evaluation eines WFS-Streaming-Moduls für Unity

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Darjush Bahreini

Implementation und Evaluation eines WFS-Streaming-Moduls für Unity

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer: Prof. Dr. Wolfgang Fohl
Zweitgutachter: Prof. Dr. Martin Becke

Eingereicht am: 20. September 2018

Darjush Bahreini

Thema der Arbeit
Implementation und Evaluation eines WFS-Streaming-Moduls für Unity

Stichworte
Virtual Reality, VR, Unity, Objektbasiertes Audio, Wellenfeldsynthese, NAudio, C#, Audio-Streaming

Kurzzusammenfassung
Diese Arbeit behandelt die Konzipierung, Entwicklung und Evaluierung eines Audio-Streaming-Moduls für die Wellenfeldsyntheseanlage innerhalb der Unity-Umgebung. Das entwickelte Modul streamt monophone Audiodaten, die mit Objekten in der virtuellen Welt assoziiert sind, auf die Wellenfeldsyntheseanlage. Hierbei können beliebig viele virtuelle Schallquellen in der Unity-Umgebung genutzt werden. Neben einer eingehenden Betrachtung von genutzten Umgebungen, Frameworks und der zu Grunde liegenden Technologie wird des Weiteren der aktuelle Stand der Technologie und Forschung beleuchtet. Im Mittelpunkt dieser Arbeit stehen jedoch die Entwicklung sowie die Evaluation des entwickelten Systems.

Darjush Bahreini

Title of the paper
Implementation and evaluation of a WFS streaming module for Unity

Keywords
Virtual Reality, VR, Unity, Object based audio, Wave Field Synthesis, NAudio, C#, Audio streaming

Abstract
This thesis deals with the design, development and evaluation of a streaming module for the Wavefield Synthesis System within the unity environment. The realized system streams monophonic audio data, associated to objects from the virtual world, to the wave field synthesis unit.The number of virtual soundsources used within the Unity environment is not limited. Besides the detailed observation of the used environments, frameworks and basic technologies, the current state of technology and research is looked at. The central aspect of this work however is the development and the evaluation of the developed system.

Inhaltsverzeichnis

1. Einleitung **1**

2. Theoretische Grundlagen **3**
2.1. Der kanalbasierte Ansatz . . . 3
2.2. Der objektbasierte Ansatz . . . 4
2.2.1. Unity und der Umgang mit Sound . . . 5

3. Stand der Forschung **6**
3.1. Allgemeine Erkenntnisse . . . 6
3.2. Paradigmenwechsel . . . 7
3.3. Entwickelte Applikationen und Frameworks . . . 8

4. Technisches Umfeld **10**
4.1. Die WFS Anlage . . . 11
4.2. Die Netzumgebung . . . 11
4.2.1. ASIO . . . 13
4.3. Unity 3D und Mono . . . 14
4.4. NAudio und die .NET Umgebung . . . 15

5. Anforderungsanalyse **17**
5.1. Allgemeine Anforderungen, die mit Hinsicht auf das Arbeitsumfeld zu beachten sind: . . . 17
5.2. Abgrenzung der Systemsichten . . . 18
5.3. Use Cases Klassenbibliothek: . . . 19
5.3.1. Initialisieren und Starten des Moduls . . . 20
5.3.2. Anlegen und Entfernen von Schallquellen . . . 22
5.3.3. Abspielen und Stoppen von Schallquellen . . . 24
5.3.4. Positionsänderung von Schallquellen übermitteln . . . 26
5.3.5. Anpassen der Lautstärke . . . 27
5.4. Anforderungsprofil der Unity-Sicht: . . . 28
5.5. Nichtfunktionale Anforderungen an das System: . . . 28
5.5.1. Anforderungen an die Perfomanz . . . 28
5.5.2. Anforderungen an die Erweiterbarkeit, Wartbarkeit und Integration . . . 30
5.6. Requirements-Studie . . . 31
5.6.1. Ergebnisse: . . . 31
5.6.2. Daraus resultierende Anpassungen der Requirements: . . . 32

6. Konzeption und Implementation des Moduls **33**
6.1. Aufbau des genutzten Audioframeworks . . . 33
6.1.1. IWaveProvider . . . 33
6.1.2. WaveFormat . . . 34
6.1.3. WaveStream . . . 34
6.1.4. WaveFileReader . . . 35
6.1.5. RawSourceWaveStream . . . 35
6.1.6. AsioOut . . . 35
6.2. Übersicht der umgesetzten Klassen im Detail . . . 35
6.2.1. WFSStreamer . . . 35
6.2.2. WFSSoundObject . . . 37
6.2.3. WFSStreamerThread . . . 38
6.2.4. WFSManyChannelWaveProvider . . . 38
6.2.5. OSCComModule . . . 40
6.2.6. WFSWaveStreamArray . . . 40
6.3. Entwicklung des Unity-Scripts . . . 41
6.3.1. Übersetzung der Koordinatensystem bzw Umsetzung der Kamera . . . 42
6.3.1.1. Ohne Virtual-Reality-Brille . . . 42
6.3.1.2. Mit Virtual-Reality-Brille . . . 43

7. Technische Evaluation des entwickelten Moduls **44**
7.1. Die Testumgebung . . . 44
7.2. Laufzeitmessungen . . . 44
7.3. Speicherbelegung . . . 46
7.4. Analyse des Ressourcenverbrauchs der Unity-Umgebung . . . 47
7.5. Bewertung . . . 47

8. Zusammenfassung **48**
8.1. Fazit . . . 48
8.2. Probleme . . . 49
8.3. Ausblick und offene Requirements . . . 49

A. Anhang zur Requirements-Studie **52**
A.1. Aufgabenblatt der Requirements-Studie . . . 52
A.2. Fragebogen der Requirements-Studie . . . 54

Abbildungsverzeichnis

2.1. Produktion und Distribution beim kanalbasierte Ansatz 3
2.2. Produktion und Distribution beim objektbasierten Ansatz 4

3.1. Grundlegender Aufbau des FascinatE Audio Systems [OSS14] 8
3.2. Beispielhafte Anwendung von Pure Data 9

4.1. Aufbau der WFS Anlage an der HAW Hamburg. Quelle: Four Audio (2011). Anleitung zur Wellenfeldsynthese-Anlage. 10
4.2. Die grafische Nutzeroberfläche, xWonder mit einigen aktivierten Quellen . . . 12
4.3. Die Unity Entwicklungsumgebung mit einer geöffneten 3D-Szene 15

5.1. Darstellung der beiden Sichten welche unterschieden werden 18
5.2. Schematisches Kontextdiagramm 19
5.3. MockUp der Unity Interface-Sicht 29

6.1. Übersicht der genutzten Klassen als UML-Klassendiagramm 36
6.2. Sicht auf den AudioController in der Unity IDE 41

7.1. Laufzeiten für die Ausgabe eines Sound-Samples mit ansteigender Kanalzahl . 45
7.2. Speicherbelegung nach 3 Minuten Laufzeit, gemessen mit unterschiedlich eingestellten Puffergrößen 46

A.1. Aufgabenblatt der Requirements-Studie, Blatt 1 52
A.2. Aufgabenblatt der Requirements-Studie, Blatt 2 53
A.3. Fragebogen der Requirements-Studie, Blatt 1 54
A.4. Fragebogen der Requirements-Studie, Blatt 2 55
A.5. Fragebogen der Requirements-Studie, Blatt 3 56

Listings

4.1. Beispielhafter OSC-Sendeaufruf 13

6.1. Pseudocode der implementierten Read-Methode welche von der ASIO-Klasse zyklisch aufgerufen wird 38

1. Einleitung

Die Verbreitung von Virtual-Reality-Anwendungen im Bereich der Unterhaltungsindustrie, aber auch im experimentellen Bereich, hat in den letzten Jahren zugenommen. Insbesondere im visuellen Bereich hat sich die zugrunde liegende Technologie entwickelt und ist mittlerweile auch beim Endverbraucher angekommen. 3D-Brillen und entsprechende virtuelle Umgebungen finden sich in den Wohnzimmern wieder. Zumeist sind es Spiele welche hier zu finden sind. Aber auch 3D-Simulationen und experimentelle Umgebungen sind immer häufiger anzufinden. Oft tritt hier jedoch die auditive Umgebung in den Hintergrund. Zumeist wird der Sound eher oberflächlich behandelt und nach konventionellen Methoden aufbereitet. Jedoch gibt es zu den für Kopfhörer oder konventionelle Mehrkanalsysteme entwickelten Soundumgebungen und Technologien noch Alternativen.

Eine dieser Alternativen ist das Prinzip der Wellenfeldsynthese. Dieses Verfahren ist in [Foh13] genauer beschrieben. Eine Wellenfeldsyntheseanlage, kurz WFS-Anlage ist auch an der HAW Hamburg zu finden. Im Laborumfeld der WFS-Anlage der HAW entstehen viele Projekte sowohl im reinen auditiven Bereich als auch im audio-visuellen.

Viele 3D-Projekte mit einer visuellen Komponente allgemein, aber auch an der HAW, werden mithilfe eines bereits bestehenden Grafik- und Physikgerüsts, einer Engine, umgesetzt. Eine verbreitete Engine, für Spiele, virtuelle Umgebungen, selten auch Animationsfilme, ist die Unity Engine. Auch im Audio-Labor der HAW sind bereits Abschlussarbeiten und Projekte entstanden welche als Basis eben diese Engine nutzen.

Zwei erwähnenswerte Projekte sind die Bachelorarbeit von Felix Baumgartner [Bau17] und das Paper „Detection Thresholds in Audio-Visual Redirected Walking“ von Florian Meyer, Wolfgang Fohl und Malte Nogalski [MNF16], welches auf der Basis von Meyers Bachelorthesis entstand.

Florian Meyer hat das Prinzip des experimentellen redirected Walkings aufgegriffen, wie

es schon von Malte Nogalski in seiner Masterarbeit umgesetzt und erforscht wurde [Nog15]. Im Gegensatz zu einer reinen akustischen Umsetzung wurde den Experimenten in Florian Meyers Arbeit jedoch eine zusätzliche visuelle Komponente hinzugefügt. Die für die Experimente notwendigen Umgebungen wurden in der Unity-Engine entwickelt. Um die Umsetzung eines solchen Projekts im Umfeld der Unity-Umgebung und der Wellenfeldsynthese einfacher zu gestalten und den technischen Overhead zu minimieren, wäre es vorteilhaft, für grundlegende Dinge auf bereits bestehende Lösungen zurückgreifen zu können. Hierzu zählt unter anderem die Audioübertragung an die WFS-Anlage.

Auch Felix Baumgartner hat sich im Zuge seiner Arbeit mit Unity und der WFS-Anlage beschäftigt. Entwickelt wurde ein 3D- bzw. Virtual-Reality-Interface zur Bedienung der Anlage. Das Interface setzte er in Unity um. Jedoch sind hier einige Punkte und Anforderungen offengeblieben. Auch hier ist die Distribution der eigentlichen Audiosignale noch recht umständlich und auf verschiedene Systeme verteilt. Baumgartner nennt den folgenden Punkt:

> *„...Es wäre eine gute Funktionalität, wenn man auch Audiodateien über das Netzwerk übertragen könnte, um diese dann wiedergeben zu können. Dies würde die Erweiterbarkeit der Szenen verbessern, da dann neue Audiodateien leicht hinzugefügt werden könnten. Die Audiodateien müssten dann allerdings über ein anderes Netzwerkprotokoll als OSC übertragen werden, da dies nicht dafür ausgelegt ist, Dateien zu übertragen...“*

Diese aufgeführten Punkte bilden die essentielle Motivation dieser Arbeit.
Ziel der Arbeit ist es, ein Audio-Streaming-Modul zu entwickeln, welches in die Unity-Engine intergriert werden kann. Das Modul soll die Enwicklung von Projekten, wie die zuvor aufgeführten, unterstützen. Außerdem soll auch technisch weniger versierten Nutzern, insbesondere Menschen aus dem künstlerischen und experimentellen Bereich, die Umsetzung von Projekten in solch einem Laborumfeld erleichtert werden.
Generell soll die 3D-Umgebung von Unity mit der WFS-Anlage über ein koppelndes Modul verknüpft werden. Die Auditive Szene in Unity soll unkompliziert und ohne Zwischenschritte an die WFS-Anlage ausgegeben werden können.

2. Theoretische Grundlagen

Dieses Kapitel soll die theoretische Grundlage für die abstrakte Herangehensweise an das Problem behandeln. Erläutert wird das Prinzip von objektbasiertem Audio, das Prinzip welches im 3D-Audio-Labor der HAW zu finden ist, sowie das Prinzip vom konventionellen kanalbasiertem Audio. Insbesondere wird der objektbasierte Ansatz dem des kanalbasiertem Audio gegenübergestellt. Abschließend wird erläutert, warum im Zuge dieser Arbeit die objektbasierte Audio-Ausgabe im Fokus steht und warum die Unity-eigene Audio-Engine im Zielumfeld nicht tauglich ist.

Grundsätzlich lässt sich heute in den Bereichen der Audio-Produktion und Verteilung an den Endnutzer zwischen zwei abstrakten Herangehensweisen unterscheiden: dem kanalbasierten sowie dem objektbasierten Ansatz.

2.1. Der kanalbasierte Ansatz

Beim kanalbasierten Ansatz gibt es die statische Beziehung zwischen einer Schallquelle, einem Audiosignal, und einem dediziertem Lautsprecher. Schon während der Produktion wird festgelegt auf welchen Kanal ein entsprechendes Audiosignal ausgegeben wird. Somit wird hier ein endgültiger Mix für einen vordefinierten Lautsprecheraufbau erzeugt, z.B. ein 5.1-Mix. Dieser Ablauf, von der Produktion bis zur Ausgabe, ist auf Abbildung 2.1 dargestellt.

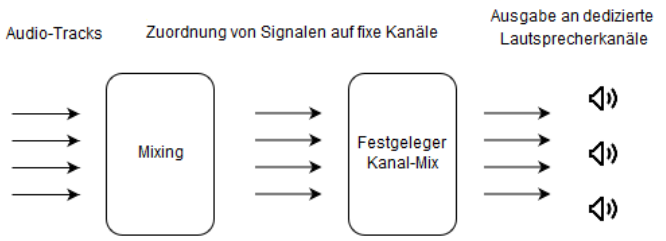


Abbildung 2.1.: Produktion und Distribution beim kanalbasierte Ansatz

Die Immersion einer Klangkulisse wird hier durch Panning erzeugt. Um nun beispielsweise die Hörrichtung eines Objektes mithilfe von Stereo-Panning manipulieren, wird ein Audio-Signal auf zwei, den Hörwinkel abgrenzende, Lautsprecher-Kanäle gespielt. Durch eine Anpassung der jeweiligen Pegel kann so die wahrgenommene Richtung einer Schallquelle angepasst werden. Man spricht hier auch von einer Phantomschallquelle. Um mit einem 2-Kanal Aufbau zum Beispiel ein vorbeifahrendes Auto zu simulieren, würde man das monophone Audiosignal auf zwei Lautsprecher geben und den Pegel über einen gewissen Zeitverlauf entsprechend anpassen.

2.2. Der objektbasierte Ansatz

Die zugrunde liegende Idee hinter dem objektbasiertem Ansatz ist die Auflösung der strikten Kopplung zwischen Schallquellen bzw. Schallereignissen und Kanälen. Schallquellen sind nun als Soundobjekte im Raum zu verstehen und lassen sich durch zwei Dinge definieren [HHKP15]:

- Den eigentlichen Audiosignalen der virtuellen Schallquelle
- Den Koordinaten, welche der virtuellen Schallquelle zugeordnet sind

Während der Produktion und der Aufnahme von Audio-Material werden Audiosignale aufgezeichnet und separat gehalten. Hierzu gehören z.B. der Dialog einer Person, der Ton eines sich bewegenden Autos und dergleichen. Diese Audiosignale bleiben im gesamten Zyklus, wie in Abbildung 2.2 dargestellt, von der Aufnahme bis zum eigentlichen Rendering unterscheidbar und technisch voneinander getrennt. Es existiert somit, anders als beim kanalbasierten Audio, kein statischer Mix.

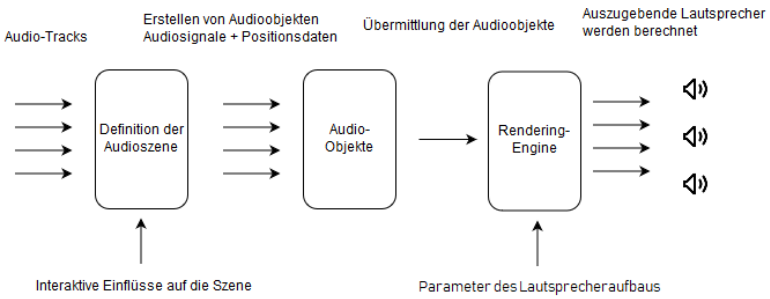


Abbildung 2.2.: Produktion und Distribution beim objektbasierten Ansatz

Die Klangumgebung wird in Echtzeit während der Audioausgabe berechnet. Dies übernimmt

eine Rendering-Engine - in der Regel ein dedizierter Server. Zum Rendering werden die zusätzlichen Informationen zu einem Audiosignal, die Metadaten, benötigt. In der Regel beinhalten diese die Koordinaten, welche sich über die Zeit ändern. Neben den Metadaten der Audiosignale müssen der Rendering-Engine zudem die physikalischen Parameter des spezifischen Lautsprecheraufbaus bekanntgemacht werden. Hierzu zählen unter anderem Parameter wie die exakte Position aller Lautsprecher sowie deren Leistungsdaten.
Eine Gegenüberstellung, wie in [He17] aufgeführt, zeigt unter anderem die folgenden Vorteile des objektbasierten Sounds:

- Die Flexibilität hinsichtlich des Lautsprecheraufbaus. So spielt es keine Rolle, ob eine entsprechende Audio-Szene in einem kleinen Kino mit wenigen Lautsprechern oder einem großen Saal mit vielen abgespielt wird.
- Die Möglichkeit interaktiv mit einer Audio-Szene umzugehen. In Echtzeit kann die Audio-Szene durch den Nutzer bzw. das Publikum beeinflusst und geändert werden.
- Ein sehr realistische auditive Darstellung

Ein signifikanter Nachteil beim objektbasiertem Audio, welcher zu erwähnen ist, ist der erhöhte Ressourcenbedarf bei der Speicherung der Audiodaten sowie deren Übertragung.

Das Prinzip der WFS-Anlage basiert auf dem objektbasiertem Ansatz. So wird die auditive Umgebung durch künstlich erzeugte und berechnete Wellenfelder dargestellt. Für die Grundlage der Berechnung werden Audioobjekte und ihre Koordinaten genutzt [Foh13].

2.2.1. Unity und der Umgang mit Sound

Zwar besitzt Unity selbst eine mächtige Rendering-Engine, die realistische Sound-Umgebungen erzeugen kann und auch die Möglichkeit bietet, über eine offene Schnittstelle eigene Spatialisier einzubinden. Es gibt jedoch keine Möglichkeit, die Signale Objekt-basiert ausgeben zu lassen. Unity selbst gibt hier lediglich einen Mix für verbreitete Lautsprecher-Setups aus. Diese Ausgabe enthält zudem auch keine Positionsdaten oder Metainformationen, die darauf schließen lassen würden. Diese sind jedoch für ein Rendering auf der WFS-Anlage zwingend notwendig.

3. Stand der Forschung

In diesem Kapitel soll der aktuelle Stand der Forschung betrachtet werden. Insbesondere werden einige aktuelle Studien betrachtet. Neben diesen werden auch einige Entwicklungen und Software-Lösungen betrachtet.

3.1. Allgemeine Erkenntnisse

Welche Relevanz Spatialisierung, die Verräumlichung von Audio-Material, im allgemeinen hat, zeigen die Ergebnisse aktueller Studien und Experimente.

Im Zuge einer von Rees-Jones, Brereton und Murphy geführten Studie sollten Probanden ein aktuelles Videospiel spielen und die Wirkung des spatialisierten Sounds bewerten [RJBM15]. Der Sound wurde auf drei verschiedene Arten ausgegeben: monophon, stereophon sowie über einen Surround-Soundaufbau mit einer 8-Kanalausgabe (7.1). Die Teilnehmer der Studie mussten jeweils zwei der drei Einstellungen in einem 12-minütigen Durchlauf erleben. Nach den zwei Durchläufen sollte der entsprechende Höreindruck bewertet werden. Das Ergebnis der Untersuchungen zeigt klar, dass die Probanden die Stereo- und Mehr-Kanal-Systeme im Vergleich zu monophonen Systemen bevorzugten. Wenn auch zwischen Stereo und 7.1 kein allzu großer Unterschied zu erkennen war, wurde der Stereo- und 7.1-Durchlauf im Vergleich zu Mono signifikant besser bewertet.

Dass jedoch nicht nur Applikationen entwickelt werden, welche der reinen Unterhaltung dienen, sondern auch einen Anwendungszweck haben, der eher als nützlich bezeichnet werden kann, zeigt die Arbeit von Cavaco, Simões und Silva [CSS15]. Diese entwickelten ein Trainingsprogramm für sehbeeinträchtigte Kinder und Jugendliche einer pädagogischen Institution. Spielerisch sollten die grundlegenden Bewegungen nach vorne, nach hinten sowie seitwärts geübt werden. In der virtuellen Umgebung wurden akustische Hindernisse aufgestellt, denen die Teilnehmer des Programms ausweichen sollten.
Neben Verbesserung innerhalb der Testumgebung zeigten sich jedoch auch hinsichtlich der motorischen Fähigkeiten der Probanden signifikante Verbesserungen. Insbesondere wurde

zum Beispiel eine deutlich erhöhte Sicherheit bei Rotationsbewegungen festgestellt. Neben den Ergebnissen und Erkenntnissen der Arbeit wurden die Durchführenden der Studie von Angestellten der entsprechenden pädagogischen Institution dazu motiviert, weitere Experimente und Trainingsspiele bzw. Applikationen in diesem Bereich zu entwerfen.

Diese beiden Arbeiten verdeutlichen, dass allgemeine und standardisierte Basislösungen im audio-visuellen Bereich durchaus ihre Daseinsberechtigung haben. Dazu zählen Grafik- und Audio-Engines sowie Frameworks.

3.2. Paradigmenwechsel

Dass der Schritt hin zum objektbasierten Audio und ein Entkopplung von klassischen Mustern wie statischen und entgültigen Produktionen für feste Mehrkanalaufbauten ein notwendiger Paradigmenwechsel ist, zeigen viele Überlegungen und Arbeiten im wissenschaftlichen Feld, aber auch Entwicklungen im proprietären und Consumer-Bereich. So entwickeln namenhafte Firmen wie Dolby [1] sowie DTS [2], mittlerweile unter dem Unternehmensnamen Xperi Corporation bekannt, eigene proprietäre Lösungen, hauptsächlich im cinematischen Umfeld.
Für fixe Aufbauten wie Kinos oder andere Vorführräume mag eine Umsetzung des objektbasierten Ansatzes noch leicht zu bewerkstelligen sein, neue Anforderungen zeigen sich jedoch, sobald ein gewisser Grad an Interaktivität hinzukommt.

In ihrer Ausarbeitung beschreiben Oldfield, Shirley und Spille die Schwierigkeiten und Ansätze anhand des Szenarios der verteilten Medienübertragung [OSS14], insbesondere mit Hinblick auf die sich wandelnden Anforderungen hinsichtlich des Konsumverhaltens und der Umgebung, in der dieses stattfindet. Hervorgehoben wird neben der Relevanz des objektbasierten Ansatzes der Bedarf von einem szenenorientierten Herangehen. Insbesondere wird hier auch auf die Thematik der Aufwände und wo diese zu finden sein sollten diskutiert. So sollen diese in Zukunft weniger auf Produktions- und Mastering-Ebene stattfinden, also der statische Prozess des Spatialisierens auf Basis von fixen Kanal-Konstellationen. In dieser Ausarbeitung wird als relevante Technologie das FascinatE Audio System erläutert, welches einen Format-unabhängigen Ansatz verfolgt.
So besteht das System, wie in Abbildung 3.1, aus 2 grundlegenden Komponenten. Eine welche die interaktive und agile Umgebung berücksichtigt und die Audio-Szene entsprechend aufbe-

[1] Dolby Atmos Whitepaper: https://www.dolby.com/us/en/technologies/dolby-atmos/dolby-atmos-next-generation-audio-for-cinema-white-paper.pdf 04.09.2018
[2] DTS:X Website: https://dts.com/dtsx 04.09.2018

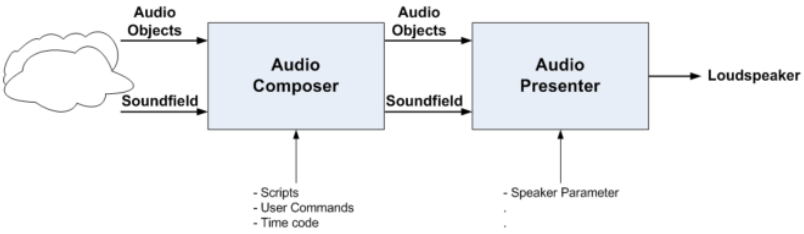


Abbildung 3.1.: Grundlegender Aufbau des FascinatE Audio Systems
Quelle:[OSS14]

reitet und eine zweite welche die objektbasierten Audio-Signale und Klangfelder zugespielt bekommt und entsprechend des spezifischen Lautsprecheraufbaus ausgibt bzw. rendert.

Um in Zukunft die Produktionen im interaktiven und immersiven Umfeld zu vereinheitlichen, bedarf es darüber hinaus gewisser Standards, die für jeden zugänglich sind, aber auch eine Vielzahl an unterschiedlichen Applikationen und Szenarios ermöglichen.

Wie in [FHB+14] beschrieben, soll im Zuge der Entwicklungen des MPEG-H Standards [HHKP15] eine solche Basis etabliert werden. Insbesondere wird die Relevanz der Metadaten hervorgehoben.

3.3. Entwickelte Applikationen und Frameworks

Wenn sich auch viel im auditiven Bereich tut, so ist die Schnittmenge der Standardlösungen, welche die virtuelle Realität, Unity, objektbasiertes Audio sowie das Prinzip der Wellenfeldsynthese einschließen, recht überschaubar.

Ein bereits bestehende Umgebung ist die Pure Data Engine, welche schon in den frühen ’90er Jahren entwickelt wurde. Zielsetzung war bei der Entwicklung dieser Umgebung das interaktive Umfeld von „Computermusik“.

Generell ist Pure Data eine visuelle Programmiersprache, welche auf Objekten beruht, die über Datenströme verlinkt werden können. Ein beispielhafter Anwendungsfall ist auf Abbildung 3.2 dargestellt. Leonard J.Paul beschreibt Ansätze, wie Pure Data in Unity integriert werden kann. Prinzipiell muss im Falle der beschriebenen Implementierung nur ein entsprechendes Package importiert werden. Die Integration von benötigten Ressourcen findet automatisiert statt. Somit beschränkt sich die Umgebung nicht nur auf Informatiker, insbesondere auch wegen der einfach Anwendung der Umgebung, sondern bietet auch Kreativen leichten Zugang. Ein signifikanter Nachteil ist jedoch die hohe Audio-Latenz. Insbesondere mit Hinblick auf die Nutzung von vielen Kanälen ist dies kritisch zu sehen.

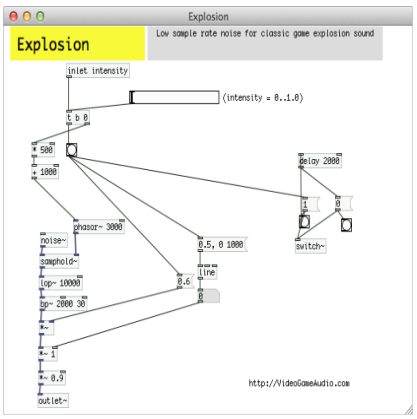


Abbildung 3.2.: Beispielhafte Anwendung von Pure Data

Ein weiterer Ansatz ist in [SBS16] sowie [BSS17] beschrieben. Mit SATIE, dem Spatial Audio Toolkit for Immersive Environments, wurde eine Echtzeit-Rendering-Engine entwickelt, die mit sehr vielen Audioquellen in einer immersiven Umgebung umgehen können soll. So werden hier 3D-Umgebungen wie Unity und Blender allerdings als umliegende steuerende Schicht dargestellt mit dem primären Ziel, eine einfache Gestaltung der Audio-Szene zu ermöglichen. Der Fokus liegt hier somit sehr stark auf der Erstellung der Audio-Szene. Dagegen soll das Ziel dieser Arbeit die Entwicklung eines Moduls zur Unterstützung der audio-visuellen Szenenerstellung sein. Generell können zur Laufzeit der SATIE-Engine Audioquellen hinzugefügt und entfernt werden.
Die Engine lässt sich sehr flexibel in unterschiedlichen Umgebungen verwenden. Jedoch besteht hier, wie es auch beim Unity-eigenen Audio-Renderer der Fall ist, das Problem, dass die Daten gerendert und fix auf Kanäle ausgegeben werden. Grundsätzlich zeigt sich auch hier die Notwendigkeit nach einer Komponente, welche lediglich die Metadaten nach außen hin abbildet.

4. Technisches Umfeld

In diesem Kapitel wird ein kurzer Überblick über die hardwareseitigen und softwareseitigen Technologien, welche dieser Arbeit zu Grunde liegen, gegeben. Von Relevanz sind auf der technischen Ebene die WFS-Anlage, bestehend aus Lautsprecherarrays und Rendering-Engine, die verknüpfenden Netze, Dante sowie OSC, sowie die verschiedenen Workstation-Rechner, auf denen unterschiedliche Audio-Anwendungen, sogenannte DAWs laufen. Des Weiteren wird die Unity-Umgebung entsprechend skizziert. Nach einem zusammenfassenden Überblick über die ASIO-Schnittstelle wird auf die Funktionsweise sowie Funktionalität des im Zuge dieser Arbeit genutzten Audio-Frameworks, NAudio, eingegangen. Auf die Technologien wird soweit eingegangen, wie es für das Verständnis der Arbeit nötig ist.

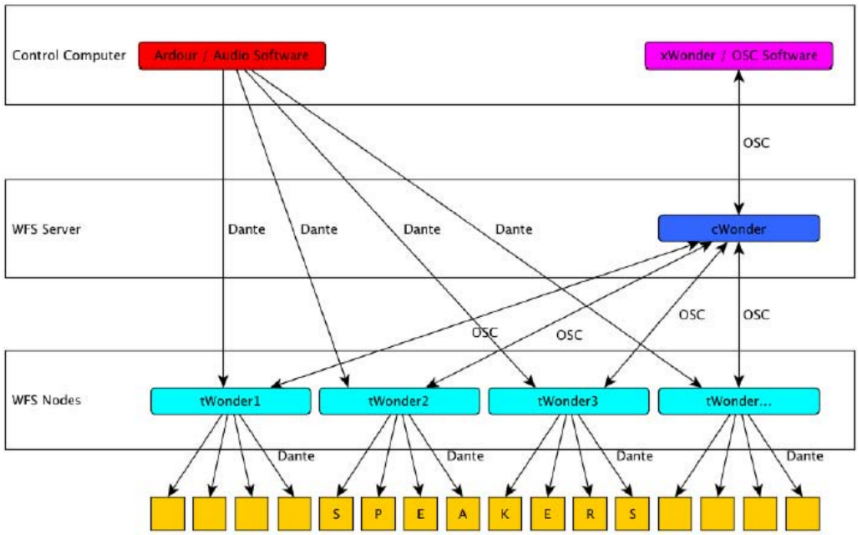


Abbildung 4.1.: Aufbau der WFS Anlage an der HAW Hamburg.
Quelle: Four Audio (2011) Anleitung zur Wellenfeldsynthese-Anlage.

4.1. Die WFS Anlage

Den Kern des Labors für Audiotechnik an der HAW Hamburg bildet die eigentliche WFS-Anlage, bestehend aus einem WFS Server und zwei untergeordneten Rendering-Nodes, welche zuständig für die Berechnung der Wellenfelder sind. Auf den drei Server-Instanzen läuft ein entsprechend angepasstes und echtzeitfähiges Linux-Betriebssystem. Verbunden mit den Nodes sind ingesamt 26 Lautsprechermodule, welche wiederum 8 über diskrete Kanäle angesprochene Lautsprecher enthalten.

Die Softwarebasis bildet die von Marije Baalman entwickelte Software Wonder, eine verteilte Anwendung mit dem Ziel, Enwicklern und Künstlern im auditiven Bereich einen einfachen Zugang zu WFS-Systemen zu bieten [Baa08] (S50). Das von Baalman entwickelte System ist modular aufgebaut, die separaten Module sind stark gekapselt. Die Kommunikation der Module untereinander findet über das OSC-Protokoll statt.
Zentrale Komponente von Wonder ist cWonder, welche als verbindendes Glied zwischen den Rendereinheiten, tWonder, und der Nutzerschnittstelle xWonder agiert. cWonder ist hier für das Verteilen der OSC-Nachrichten zuständig. Über xWonder können, wie in Abbildung 4.2 dargestellt, die momentan in der WFS Anlage aktivierten Quellen inklusive der absoluten Positionen, angezeigt und manipuliert werden. Auch ein Verfolgen der Bewegungen von Quellen ist somit in Echtzeit möglich.

4.2. Die Netzumgebung

Prinzipiell findet jede Art von Kommunikation mit dem technischen System der WFS-Anlage über zwei diskrete und voneinander abgegrenzte Netze statt, der Aufbau ist entsprechend in Abbildung 4.1 dargestellt. Auf der einen Seite gibt es das OSC-Netz, ein reguläres LAN. Die OSC-Nachrichten werden mithilfe des UDP-Protokolls übertragen. Über dieses Netz findet die Befehlssteuerung statt. Des Weiteren gibt es das Dante Netz, über welches die Audiosignale übertragen werden.

Das Dante-Netz, ein Zusammenwirken von proprietärer Audinate-Hardware sowie Software und einem Netzwerkprotokoll, welches konventionelle Netzwerktechnologie (Ethernet) verwendet. Für die Übertragung wird das UDP-Protokoll genutzt [1]. Eine möglichst geringe Latenz und freie Netzkapazität steht hier über der Zusicherung einer vollständigen Übertragung. Ein

[1] Audinate Website: https://www.audinate.com/sites/default/files/PDF/adding-dante-to-your-network-audinate.pdf 05.09.2018

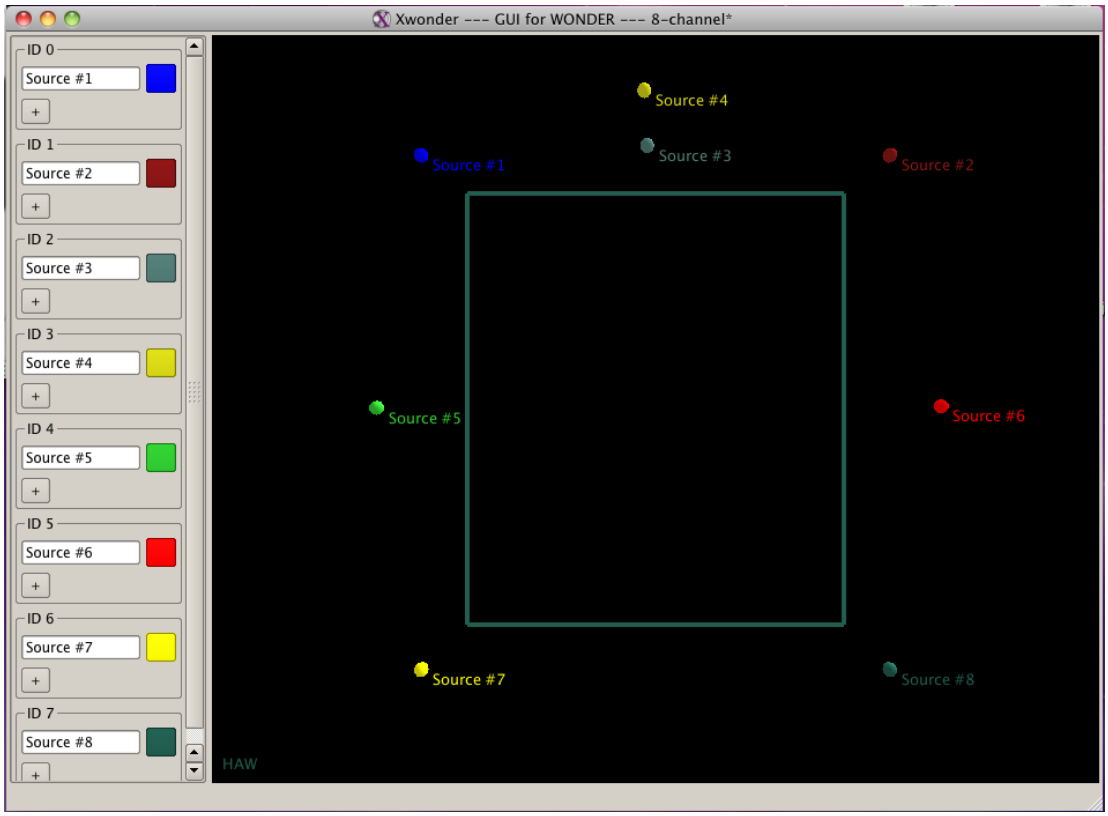


Abbildung 4.2.: Die Grafische Nutzeroberfläche, xWonder mit einigen aktivierten Quellen

positiver Nebeneffekt bei der gebündelten Kanalübertragung über eine Ethernetleitung ist die Einsparung von Kabeln, wie es bei einer Punkt-zu-Punkt-Verbindung der Fall wäre [Foh13]. Dieser Punkt ist aufgrund der hohen Zahl an Lautsprechern innerhalb eines WFS-Systems nicht zu vernachlässigen.

Jegliches Audio-Material wird roh in Puffer bestimmter Größe geschrieben und somit an die Anlage gestreamed. Die Grundlage einer jeder Applikation innerhalb der WFS-Laborumgebung ist ein Zugang zu Dante über den Treiber einer Netzwerkkarte. Alternativ existiert eine softwareseitige Emulation, die Dante Virtual Soundcard, welche ein Routing über eine handelsübliche Netzwerkkarte abwickelt. Diese wurde im Zuge dieser Arbeit auch auf dem Entwicklungsrechner genutzt.

Das OSC-Protokoll nutzt in der Regel entweder das UDP- oder TCP-Protokoll, ist jedoch nicht

an ein spezfisches Transportprotokoll gebunden [2]. Zudem ist es möglich, hier neben Ethernet auch kabellose Netze zu nutzen. Der Aufbau der Nachrichten ist simpel. Die Adressierung ist dem Aufbau des URL-Schemas nachempfunden (Ressource/Subressource1/Subressource2/etc.). Anhand des folgenden, in Pseudocode gehaltenen Sendeaufrufs, soll der Aufbau kurz erläutert werden:

```
String OSC_IP          = "192.168.2.2";
int OSC_PORT           = 5555;
int source_id          = 0;
float position_x       = 1.0;
float position_y       = 1.0;

SEND_OSCMSG(OSC_IP, OSC_PORT, "/WONDER/source/position",
            source_id, 1.0, 1.);
```

Listing 4.1: Beispielhafter OSC-Sendeaufruf

An einen entsprechenden Empfänger gesendet, mit IP-Adresse und Port, enthält die Nachricht den angesprochenen Parameter. In diesem Fall wird die Position der zu ändernden Schallquelle im WFS-System adressiert. Die zugehörige und eindeutige ID der Quelle wird als Integer übertragen. Gesondert werden die X- sowie Y-Positionen eben jener Quellen als Fließkommawerte übertragen.
Ein weiterer Befehl der übertragen werden kann, ist zum Beispiel die Änderung des Typs einer spezifischen Schallquelle, Optionen sind hier Punkt- oder Planarquelle. Außerdem können Quellen stumm geschaltet oder komplett entfernt werden. Zusätzlich zum reinen Senden von Kontrollbefehlen ist es außerdem möglich, sich an einem der Streams, welche auch unter den Wonder-Komponenten zur Kommunikation untereinander genutzt werden, anzumelden, und Informationen zu Schallquellen zu bekommen.

4.2.1. ASIO

Die Mehrheit der professionellen und semi-professionellen Audiogeräte auf dem Markt nutzt unter Windows das ASIO-Protokoll. Ein proprietärer und von Steinberg entwickelter Schnittstellenstandard, welcher die systemeigene Audio-Verarbeitungskomponente umgeht. Das ASIO-Protokoll bietet somit eine sehr niedrige Latenz sowie die Fähigkeit mit einer theoretisch unbegrenzten Anzahl an Kanälen in hoher Auflösung umzugehen [3]. In der Umgebung

[2] Open Sound Control Specification: http://opensoundcontrol.org/spec-1_0 05.09.2018
[3] Steinberg ASIO Specification Documentation: http://read.pudn.com/downloads119/sourcecode/multimedia/audio/506331/ASIOSDK2/ASIO%20SDK%202.2.pdf

des WFS-Labors können bis zu 64 Kanäle genutzt werden. Ursprünglich für Aufnahmen und Einspeisungen von Audio bei niedriger Verzögerung gedacht, werden ASIO Treiber mittlerweile auch häufig zur reinen Ausgabe genutzt. Generell für Anwendungsentwickler relevante Aspekte von ASIO sind unter anderem:

- Hinsichtlich der Anzahl von Ein- und Ausgabekanälen gibt es keine Limitierungen. Gleiches gilt für die Abtastrate, diese ist lediglich abhängig von der zu Grunde liegenden Hardware. Des Weiteren können beliebige Audio-Auflösungen genutzt werden (16, 24 bit oder 32/64 bit floating point etc.).
- Jedem Kanal (Ein- sowie Ausgabe), ist ein fester Ringpuffer zugeschrieben. Dies erlaubt es, diskrete digitale Signale sehr effizient abzugreifen und zu bearbeiten (Vor-/Nachbearbeitung) und sollte bei der Implementierung von Audio-Anwendungen unbedingt berücksichtigt werden!
- Der Lebenszyklus einer ASIO-Implementation besteht aus 4 Zuständen:
  - Loaded - Im Zustand Loaded hat die entsprechende Audio-Anwendung Zugang zum ASIO-Treibercode.
  - Initialized im Zustand Initialized werden entsprechend Ressourcen für den Treiber allokiert und der entsprechende Treibercode wird instanziiert.
  - Prepared - Um in den Zustand Prepared zu gelangen, müssen die Ein- bzw. Ausgabepuffer allokiert werden. Erst danach wird versucht auf die Hardware zuzugreifen.
  - Running - im Erfolgsfall wird der Treiber in den Zustand Running überführt und das Streaming startet.

4.3. Unity 3D und Mono

Die in der Einleitung erwähnte Umgebung Unity, Homonym für Entwicklungs-, Laufzeit- und 3D-Umgebung, hat sich in den letzten Jahren als häufig genutzte Basis für die unterschiedlichsten 3D- und 2D-Applikationen etabliert. Insbesondere mit Hinsicht auf Virtual-Reality Anwendungen ist Unity mittlerweile die meist genutzte Umgebung, 59% aller VR-Entwickler greifen auf diese zurück [Dea17].
Unity setzt intern auf C++ (Implementierung Physik, Audio, Grafik), als Programmier-Schnittstelle für Entwickler und Künstler setzt Unity auf Mono. Mono ist eine quelloffene Implementierung von Microsofts .NET Plattform, welche erst die Scriptbasis für entsprechende Logik bietet. War es ursprünglich noch möglich die Scripte in Javascript, Boo und C# zu entwickeln, werden die

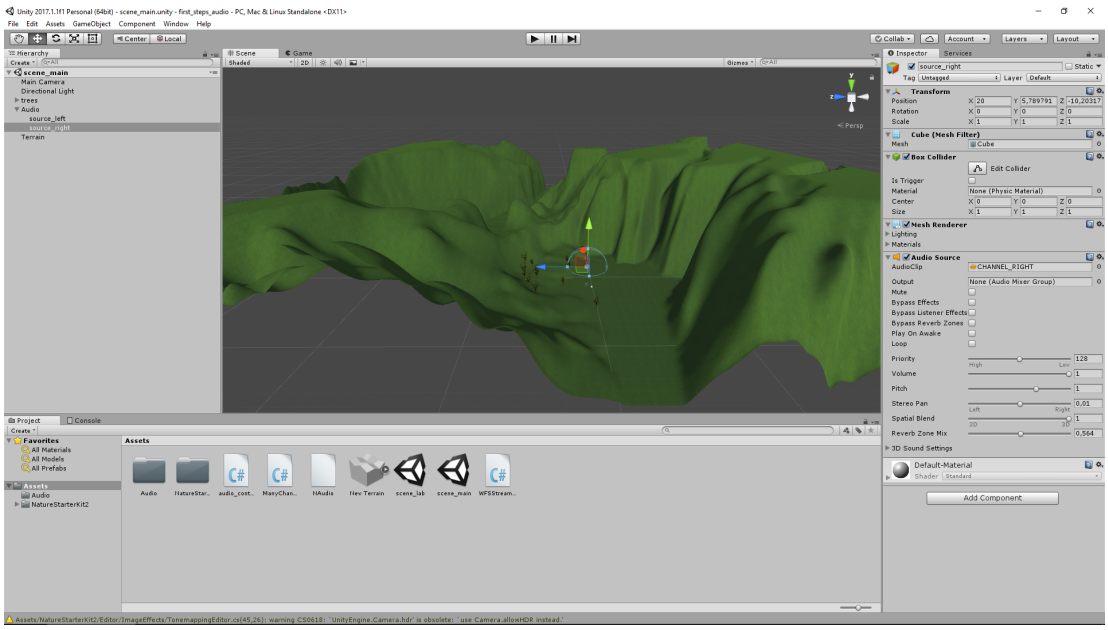


Abbildung 4.3.: Die Unity Entwicklungsumgebung mit einer geöffneten 3D-Szene

ersten beiden fallengelassen und der Fokus auf C# gelegt [4].
Neben seiner vielschichtigen Anwendungsmöglichkeiten (2D/3D Spiele, Animationsfilme, Infotainment Applikationen etc), unterstüzt Unity eine ganze Reihe von Ziel-Plattformen, unter anderem Windows, Mac OS, Android, iOS und diverse Spielekonsolen. Die zentrale Entwicklungsumgebung, in Abbildung 4.3 zu sehen, bietet bereits sehr viele Werkzeuge um virtuelle Umgebungen, sogenannte Szenen zu erstellen. Für die grundlegende Erstellung von 3D-Modellen, muss auf andere Programme zurückgegriffen werden. Zusätzlich wird die Unity-Umgebung noch mit einer angepassten Version von Monodevelop ausgeliefert, einer Entwicklungsumgebung für Sprachen wie C#, F#, VB .NET und Java.

4.4. NAudio und die .NET Umgebung

Im Zuge dieser Arbeit wurde das Audio Framework NAudio genutzt. Das NAudio Framework ist quelloffen, in C# und der .NET-Umgebung entwickelt und bietet direkten Zugang zu den ASIO Treibern. Laut eigener Aussage des Entwicklers M.Heath ist die NAaudio API das einzige Audio Framework, geschrieben in C#, welches nativen ASIO-Zugang bietet [5]. Das Framework bietet viele Klassen mit grundlegender Funktionalität. Zum Beispiel ist das Abspielen von WAV-

[4] Unity Blog: blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset/
[5] NAudio repository: https://github.com/naudio/NAudio

oder MP3-Dateien möglich, das statische Mixen von Kanälen, sowie das Einlesen verschiedener Dateiformate und von den Audioeingangspuffern. Des Weiteren wird durch Quelloffenheit auch die Möglichkeit gegeben, eigene Implementationen zu nutzen und die Funktionalität des Frameworks entsprechend zu erweitern bzw. anzupassen. Vorteilhafterweise unterstützt Unity die Nutzung externer Bibliotheken, die mit der Mono- bzw. .NET-Laufzeitumgebung kompatibel sind.
Je nach genutzter Unity-Version muss in den Build-Einstellungen des Unity-Projekts die experimentelle .NET-Unterstützung aktiviert werden, da der .NET-Standard, der standardmäßig von Unity unterstützt wird, nicht immer aktuell ist.

5. Anforderungsanalyse

Dieses Kapitel beschäftigt sich mit einer detaillierten Analyse des Problems, welches dieser Arbeit zu Grunde liegt und es werden die Anforderungen strukturiert dargestellt. Nach einer Betrachtung der Probleme und Eigenschaften des Umfelds, in welchem das System läuft, werden die technischen Anforderungen erarbeitet. Anschließend werden die nicht-funktionalen Anforderungen hervorgehoben, insbesondere jene, welche die Qualität der Nutzbarkeit und Performanz betreffen. In Hinsicht auf das entwickelte System, wird zwischen der Unity-Script-Schicht und der Interface-Schicht des Audio-Streaming-Moduls unterschieden. Abschließend wird das Ergebnis einer durchgeführten Requirements-Studie diskutiert und evaluiert.

5.1. Allgemeine Anforderungen, die mit Hinsicht auf das Arbeitsumfeld zu beachten sind:

Die WFS-Umgebung, in welcher diese Arbeit entstanden ist, ist mit Hinsicht der diskreten Komponenten ein vergleichsweise sehr komplexes System. Einige grundlegende Umstände sind bei der Implementierung des Systems sowie bei der Integration zu beachten:

- Die Anbindung an das OSC-Netzwerk erfolgt entweder über WLAN oder LAN und eine bestehende Verbindung wird vorausgesetzt. IP-Adresse (IPV4) und Port des Ziels sind dem Nutzer bzw. Aufrufer des Systems bekannt und die Übergabe der entsprechenden Information erfolgt über Argumente die dem entwickelten Modul übergeben werden.
- Die Verbindung mit dem DANTE-Netzwerk und der WFS-Anlage wird als aufgebaut und geprüft angenommen. Das Routing ist entsprechend konfiguriert. Es spielt keine Rolle ob eine native Dante-Karte oder eine von Audinate angebotene virtuelle Soundkarte genutzt wird.
- Eine Unity-Version von 2017 1.0 oder höher wird vorausgesetzt, da erst in dieser der entsprechende .NET-Standard unterstützt wird. Das in dieser Arbeit genutzte Audio-Framework, NAudio setzt diesen voraus. Der erforderliche .NET Standard ist 3.5.

- Ein Rechner, der zumindest die grundlegenden Unity3D Projekte darstellen kann. Messungen zur Perfomanz des entwickelten Systems sowie eine entsprechende Diskussion dieser sind in Kapitel 7 zu finden.

5.2. Abgrenzung der Systemsichten

Bei der Erstellung der Anforderungen wird grundlegend zwischen zwei Systemsichten unterschieden, welche schematisch in Abbildung 5.1 dargestellt sind:

- Die Sicht auf das eigentliche Modul, implementiert und gepackt als .NET-Klassenbibliothek: die eigentliche entwickelte Funktionalität erfolgt per Zugriff über eine API. Funktionen zur Instanziierung und Nutzung werden angeboten.
- Die Unity Script- bzw. Editorsicht: eine möglichst intuitiv zu bedienende Schnittstelle zu den Funktionen des Streamingmoduls. Hierzu dient der AudioController, ein Unityscript welches Funktionen der Klassenbibliothek kapselt und eine abstrahierte Funktionalität realisiert.

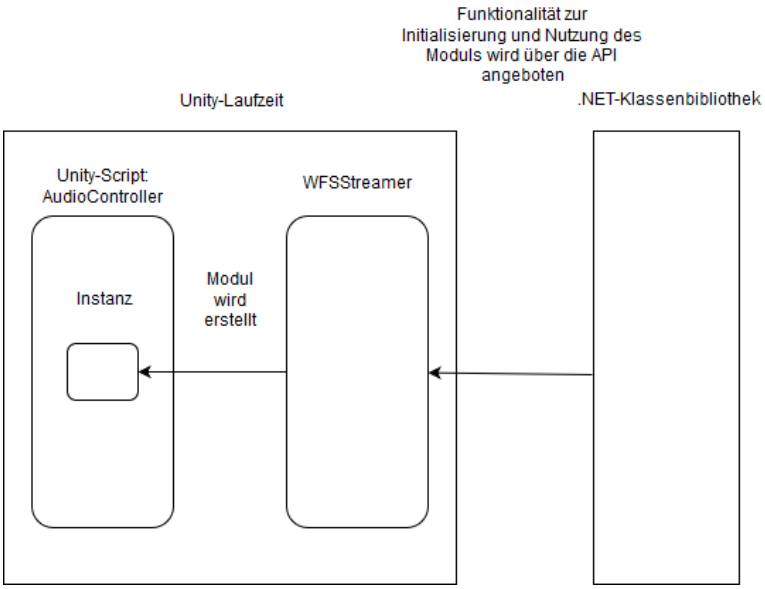


Abbildung 5.1.: Darstellung der beiden Sichten welche unterschieden werden

5.3. Use Cases Klassenbibliothek:

Die im Folgenden ausgeführten Anwendungsfälle wurden herausgearbeitet und zeigen eine Funktionsspezifikation des entwickelten Systems. Des Weiteren ist das Modul als Blackbox über wohldefinierte Schnittstellen abgegrenzt. Das Kontextdiagramm in Abbildung 5.2 soll die Einordnung der Klassenbibliothek verdeutlichen. Der Unity-Entwickler, welcher eine 3D-Szene entwirft, wird entsprechend Verhalten programmieren und über die Unity-Scriptschicht Steuerbefehle und eventuelle Audiodaten an das Streamingmodul reichen. Dieses wiederum, in der Unity-Laufzeit liegend, gibt die objektbasierten Daten und die korrespondierenden Positionsdaten parallel an die WFS-Anlage.

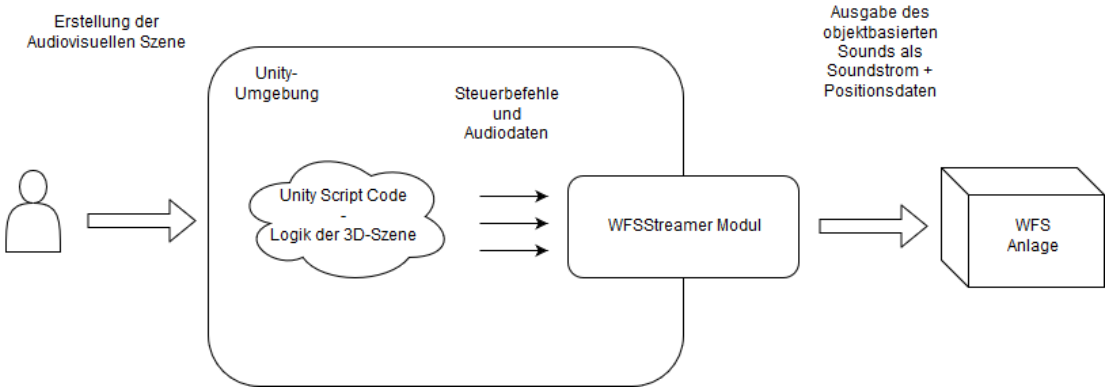


Abbildung 5.2.: Schematisches Kontextdiagramm

5.3.1. Initialisieren und Starten des Moduls

Um das Modul bzw. die Klassenbibliothek nutzen zu können, muss das Modul innerhalb eines Unity-Scripts entsprechend aufgerufen und konfiguriert werden.

Bezeichnung	Modul initialisieren/starten
Ziel	Das Modul läuft innerhalb der Unity-Laufzeit. Virtuelle Schallquellen können erzeugt oder eingespeist werden.
Vorbedingungen	• Die Unity-Szene der Wahl ist soweit vorbereitet. Die Deklaration und Instanziierung des Moduls ist korrekt ausgeführt, z.B in der Start-Methode eines Unity-Scripts. • Entsprechende Pfade zu den Sounddateien sind bekannt und können im Unity Script parametrisiert bzw. Audioclips in der Unity-Umgebung hinzugefügt werden. • Die Laufzeitumgebung ist bzgl. dem .NET-Standard entsprechend konfiguriert. Im Falle von Unity muss mindestens ein .Net-Standard von 3.5 in den Buildeinstellungen gesetzt sein.
Details	• Da dieses Modul wichtige bindende Zugriffe auf Treiber realisiert, darf lediglich eine Instanz genutzt werden. • Ein sauberes Beenden sollte realisiert und auch auf Unity-Script-Ebene genutzt werden. Treiberressourcen müssen unbedingt wieder freigegeben werden.

Mögliche Fehlerfälle	• Ein Zugriff auf den Treiber ist nicht möglich oder ein nicht-existenter Treiberindex wurde angegeben. • Eine unzulässige Anzahl an Ausgabekanälen wurde beim Initialisieren ausgewählt.

5.3.2. Anlegen und Entfernen von Schallquellen

Um eine Instanz einer virtuellen Schallquelle zu erzeugen, welche mit einem Objekt in der virtuellen 3D-Umgebung verknüpft wird, muss das zuvor initialisierte Modul entsprechend angestoßen werden. Es muss außerdem mit Informationen bzgl. der zu nutzenden Audiodaten versorgt werden.

Bezeichnung	Eine neue Schallquelle erzeugen und auch wieder entfernen.
Ziel	Mithilfe von bestehenden Sounddateien von lokalen Speichermedien während der Laufzeit virtuelle Schallquellen erzeugen lassen, so dass ein kontrollierender Zugriff besteht. Außerdem sollen diese Quellen auch wieder entfernt werden können.
Vorbedingungen	• Das Modul wurde wie in Abschnitt 5.3.1 beschrieben erfolgreich erstellt. • Es existieren Audiodaten, auf die zurückgegriffen werden kann.
Details	• Ggf. und abhängig von verfügbaren Systemressourcen, kann es von Vorteil sein, nicht alle Schallquellen bzw. Soundobjekte zur selben Zeit in das Modul zu laden. Ein dynamischer Wechsel schafft hier Abhilfe. • Falls die auszutauschende Sounddatei abgespielt wird, wird sie vor dem Tausch explizit gestoppt. • Neue Quellen werden im Hintergrund hinzugefügt und sind anschließend bereit zum Abspielen

Mögliche Fehlerfälle	• Der Pfad zur entsprechenden Datei ist nicht verfügbar oder fehlerhaft. • Das Modul kann keine weiteren Schallquellen bzw. Soundobjekte mehr verwalten. • Die Audiodaten liegen in einer Auflösung oder Abtastfrequenz vor, die nicht unterstützt wird.

5.3.3. Abspielen und Stoppen von Schallquellen

Zusammen mit dem Anwendungsfall, welcher in Abschnitt 5.3.4 beschrieben ist, bildet dieser die Kernfunktion des Moduls - Das Abspielen und Stoppen von Schallquellen.

Bezeichnung	Spezifische Quelle abspielen/stoppen -(bezieht den Anwendungsfall wie in 5.3.4 mit ein).
Ziel	Schallquellen, die momentan geladen sind, in den abspielenden bzw. pausierten/gestoppten Zustand versetzen.
Vorbedingungen	• Das Modul wurde, wie in Abschnitt 5.3.1 beschrieben, erfolgreich erstellt. • Es existieren Schallquellen, auf welche diese Aktion ausgeführt werden kann.
Details	• Zur Laufzeit wird es erforderlich sein, spezifische Soundquellen zu starten und zu stoppen (bzw. zu pausieren). • Eine Referenz der Soundquelle muss zu jeder Zeit vom umgebenden Unity-Script gehalten werden. • Die Start- bzw. Stopfunktion des Moduls wird entsprechend aufgerufen. • Möglichst zeitnah sollten die entsprechenden Koordinaten des assoziierten Unity-Objekts mit übergeben werden (nur im Fall des Startens, siehe Abschnitt 5.3.4).

Mögliche Fehlerfälle	• Die zugrunde liegende Audiodatei ist fehlerhaft oder beinhaltet keine korrekten Audiodaten. • Es liegt ein unvorhergesehenes Problem bei dem Abspielgerät vor.

5.3.4. Positionsänderung von Schallquellen übermitteln

Zusammen mit dem in 5.3.3 beschriebenen Anwendungsfall ist dieser von höchster Relevanz. Da in einem Umfeld des objektbasierten Audio die Position einer Schallquelle im Raum erst eine korrekte Spatialisierung möglich macht, müssen kontinuierlich die aktuellen Koordinaten übermittelt werden.

Bezeichnung	Positionsänderung einer Schallquelle übermitteln.
Ziel	Das Übermitteln der Position einer spezifischen Schallquelle.
Vorbedingungen	• Das Modul wurde, wie in Abschnitt 5.3.1 beschrieben, erfolgreich erstellt. • Es existieren Schallquellen, auf welche diese Aktion ausgeführt werden kann.
Details	• In der Regel ist jede Schallquelle einem Objekt zugeordnet. Weiterhin gibt es immer relative, abstrahierte Koordinaten in Bezug zur aktuellen Position des Nutzers im Raum. Diese Koordinaten müssen sukzessive errechnet werden. Die Berechnung obliegt dem umgebenden Programm- oder Scriptcode.
Mögliche Fehlerfälle	• Befehle können verloren gehen. Dies kann aufgrund des Protokolls oder eines Problems, welches mit der Verbindung zusammenhängt, passieren. • Irreguläre Positionsdaten werden übergeben. Zum Beispiel liegen die Positionsdaten im falschen Format vor oder befinden sich außerhalb des erlaubten Bereichs.

5.3.5. Anpassen der Lautstärke

Im Labor der HAW ist es nicht möglich die Lautstärke der WFS-Anlage anzupassen. Dies muss in der entsprechenden Audio-Applikation passieren. Des Weiteren kann es als guter Stil aufgefasst werden, dem Nutzer die Möglichkeit zu geben, am Hostrechner Einfluss auf die Lautstärke bzw. den Gain des ausgegeben Audio-Materials zu nehmen.

Titel	Lautstärke einer spezifischen Quelle anpassen (+/-).
Ziel	Die Lautstärke einer Schallquelle ist nach der Anpassung höher oder niedriger.
Vorbedingungen	• Das Modul wurde, wie in Abschnitt 5.3.1 beschrieben, erfolgreich erstellt. • Es existieren Schallquellen, auf welche diese Aktion ausgeführt werden kann.
Details	• Die Schallquelle, welche angepasst werden kann, besitzt eine initiale Lautstärke. Es mag nötig sein, diese anzupassen. • Das Anpassen der Lautstärke kann entweder sukzessive erfolgen oder durch Übergabe eines konkreten Werts.
Mögliche Fehlerfälle	• Aus unbekanntem Grund werden Werte übermittelt, die bei der Anwendung der Lautstärkeanpassung zu korrupten Audiodaten führen.

5.4. Anforderungsprofil der Unity-Sicht:

Die Sicht, die dem Nutzer der Unity-Umgebung auf das entwickelte Modul geboten wird, soll möglichst intuitiv sein. Es soll ein größtmögliches Feld an Nutzern angesprochen werden - auch ein Content-Creator, welcher sich unter Umständen nicht allzu sehr mit einer Programmierschnittstelle beschäftigen möchte, sollte die Funktionalität der WFS-Anlage schnell in sein Projekt integrieren können.
In Abbildung 5.3 ist, farbig hervorgehoben, ein Mock-Up des Moduls innerhalb der Unity-Entwicklungsumgebung zu sehen. Neben zwei Konfigurationseinstellungen, eine für den Treiberindex sowie eine weitere für die Anzahl der Kanäle, ist je nach gewählter Anzahl an zu nutzenden virtuellen Schallquellen, die gesamte Auswahlmaske für die Unity-eigenen Objekte in der Szenerie zu finden. Diese können aus beliebigen Ebenen und Elementen der Hierarchieebene per Drag and Drop bestückt werden. Zusätzlich soll es möglich sein, Audioclips in das Projekt zu laden und diese mit Gameobjekten zu verknüpfen. Auch hier soll der Workflow möglichst unkompliziert sein.

5.5. Nichtfunktionale Anforderungen an das System:

5.5.1. Anforderungen an die Perfomanz

Eines der elementaren Prinzipien einer Echtzeit-Audioanwendung ist das zeitkritische Bereitstellen von Audiodaten. Das Übertragen von Audiodaten einer Anwendung auf einem Hostrechner zu einem Gerät, einem Audio-Interface zum Beispiel, erfolgt in der Regel zyklisch und zu äquidistanten Takten. Dies bedeutet, dass falls die Anwendung den Anforderungen neuer Audiodaten nicht nachkommen kann, Fehler in der Ausgabe der Audiodaten entstehen. Für eine korrekte Wiedergabe von Audiomaterial ist eine möglichst vollständige und gleichmäßige Ausgabe von den abgetasteten Audiodaten vonnöten.

In der Regel werden die Audiodaten in Puffern abgelegt, die seitens des Treibers bereitgestellt werden. Über die Konfiguration, unter anderem die Größe dieser Puffer, hat die Audioanwendung keine Kontrolle! Jedoch sollen die Effekte, die abhängig von der Puffergröße sind, kurz aufgeführt werden:

- Eine Vergrößerung des Puffers führt zu einer Erhöhung der Latenz. Die Daten werden erst gesammelt, bevor sie ausgegeben werden. Befindet man sich im audio-visuellen Umfeld, kann diese Latenz als Versatz zum Bild interpretiert werden. Ab einer gewissen Länge ist diese Latenz als störend wahrzunehmen.

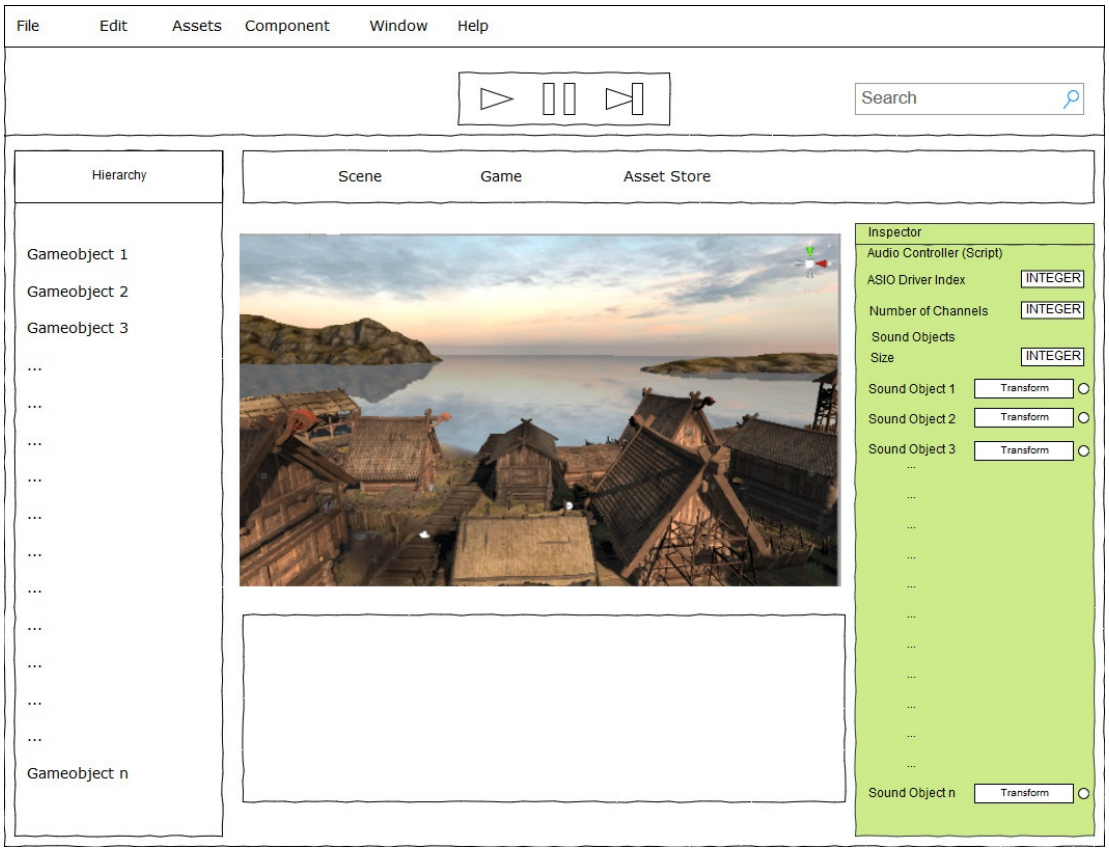


Abbildung 5.3.: MockUp der Unity Interface-Sicht

- Eine Verkleinerung des Puffers führt zwar auch zu einer geringeren Latenz, kann unter Umständen aber auch zu den oben genannten störenden Effekten führen. Dies hängt unter anderem von der Algorithmik der Audioanwendung sowie der Rechengeschwindigkeit des zugrunde liegenden Systems ab. Sollte die Anwendung nicht mit dem Befüllen der Puffer hinterherkommen, werden im Ausgabesignal Lücken sein, sogenannte Glitches.

Neben dem Umstand, dass der Puffer in seiner Größe variieren kann und diese Gegebenheit von dem entwickelten System bzw. der Anwendung nicht weiter beeinflusst werden kann und so hingenommen werden muss, kann trotzdem auf eine möglichst effiziente Umsetzung der Algorithmik geachtet werden.
Zudem sollte das System nur einen kleinen Teil des Speichers einnehmen. In der Regel belegen die 3D-Ressourcen der visuellen Umgebung schon alleine sehr viel Systemressourcen.

5.5.2. Anforderungen an die Erweiterbarkeit, Wartbarkeit und Integration

Der Lebenszyklus des entwickelten Moduls soll auch nach der Fertigstellung dieser Arbeit nicht zu Ende sein. Dementsprechend ist die Konzeption entsprechend modular und erlaubt eine leichte Anpassung der spezifischen Implementationen.

5.6. Requirements-Studie

Um die Sinnhaftigkeit der Requirements zu überprüfen und eventuelle Schwächen sowie wichtige weitere Features zu erschließen, wurde eine Requirements-Studie mit 7 Probanden durchgeführt. Alle 7 Probanden hatten zumindest erste Schritte in der Unity Umgebung gemacht und konnten den Editor gut bedienen.

Die Studie war mithilfe eines Aufgabenzettels geleitet und in 3 Teile aufgeteilt. Diese waren bewusst einfach gehalten und sollten die Nutzung des entwickelten Moduls, mit Klassenbibliothek sowie Unity-Script, in den Vordergrund rücken:

- Das Integrieren des entwickelten Moduls in ein bestehendes Projekt: Gewählt wurde hier das Demo-Projekt „Viking Village “vom Unity-Team selbst. Das Integrieren beinhaltete das Kopieren von Klassenbibliothek sowie des Unity-Scripts, inklusive aller benötigten Dateien. Außerdem mussten gewisse Buildeinstellungen überprüft und ggf. entsprechend angepasst werden.
- Die Nutzung des Unity Scripts über die Gui: Über die grafische Komponente in der Inspectorsicht innerhalb der Unity-Entwicklungsumgebung mussten einige markante Gameobjekte aus dem Projekt mit den korrespondierenden Sounds belegt werden.
- Das Anpassen des Unity-Scripts: Hier wurden die Probanden mithilfe einer Übersicht der Schnittstelle an die Klassenbibliothek herangeführt. Kleine Änderungen am Unity-Script wurden hier eingefordert. Unter anderem das Deaktivieren der Loop-Eigenschaft und das Anpassen der Lautstärke der virtuellen Soundobjekte.

5.6.1. Ergebnisse:

Im allgemeinen wurde das Konzept des Unity-Plugins von allen Probanden adaptiert und es wurden keine blockierenden Probleme festgestellt. Auch das Einbinden des Moduls über das Kopieren der benötigten Dateien wurde von allen Probanden als positiv bewertet. Folgende Dinge wurden jedoch erwähnt:

- Das umständliche Auswählen von Audioquellen über einen Dateipfad. Der Vorschlag war eine Drag and Drop Funktionalität. Zwei User empfahlen auch explizit die Nutzung von den Unity-eigenen Audioclips.
- Das umständliche Selektieren von Gameobjekten sowie der manuellen Auswahl der Userkamera. Fast alle Probanden äußerten auch hier den Wunsch nach einer Drag and

Drop Funktionalität.

- Eine eigenständige Registrierung der Gameobjekte beim entwickelten Controller, das heißt: Gameobjekte sollen eine eigene Schnittstelle zur Audioschicht haben.
- Eine Parametrisierung der virtuellen Soundobjekte, hervorgehoben wurde hier die Anpassung der Lautstärke.
- Ein Level-of-Detail-ähnliches Verhalten, Soundobjekte sollen ab einer gewissen Entfernung ausgeblendet werden, da man sie ja sowieso nicht mehr hört. Dies soll Ressourcen sparen und diese auch wieder für neue Soundobjekte freigeben.
- Die Auslagerung von Konfigurationseinstellungen, wie die Auswahl des Treibers und der zu nutzenden Kanalanzahl, in ein separates Editor-Menü.
- Zusätzlich zum bereits spezifizierten endlosen Wiederholen, ein n-faches Abspielen einer Schallquelle.
- Eine erhöhte Fehlerresistenz gegenüber falschen Eingaben.
- Eine strikte Koppelung von Unity-Gameobjekten an Schallquellen, zum Beispiel über C#-Structs.
- Die Möglichkeit Audiodaten verkettet abspielen zu können.
- Eine Berücksichtigung der Höhenebene in Unity.

5.6.2. Daraus resultierende Anpassungen der Requirements:

Nach einer Evaluation der Ergebnisse wurden die folgenden Punkte mit in die umzusetzenden Requirements übernommen und auch umgesetzt. Nicht übernommene Punkte wurden zum Teil in den Ausblick dieser Arbeit gestellt:

- Eine zusätzliche Schnittstelle in der entwickelten Klassenbibliothek für die Unterstützung von Audiodaten im Float-Format und somit kohärent auch die Unterstützung der Unity-Audioclips.
- Kopplung von Soundobjekt und Gameobjekt durch Arrays von aggregierten Klassen. Diese Klassen sollen eine Referenz auf das Gameobjekt sowie eine auf das virtuelle Soundobjekt halten.
- Eine Möglichkeit, die Lautstärke per Schieberegler in der Inspector- bzw. Editorsicht zu justieren.

6. Konzeption und Implementation des Moduls

Im folgenden Kapitel wird zunächst ein Einblick in die Architektur des genutzten und zugrunde liegenden Audio-Frameworks gegeben. Danach wird der Entwurf des entwickelten Moduls spezifiziert sowie die einzelnen Komponenten und Klassen im Detail erläutert. Des Weiteren wird auf die innerhalb Unity entwickelten Teile dieser Arbeit eingegangen.

6.1. Aufbau des genutzten Audioframeworks

Das als Basis genutzte Audio-Framework, NAudio [1], bietet wie in den technischen Grundlagen erwähnt, viele Standardfunktionen für die Nutzung diverser Use-Cases unter Windows. Wegen der folgenden Funktionen bzw. Eigenschaften wurde NAudio als entsprechendes Basis-Framework ausgewählt:

- Eine vollständige Umsetzung in C#. Zwar können innerhalb der Unity-Laufzeitumgebung auch native Plugins, geschrieben in C, C++ und Objective-C, genutzt werden, jedoch ist die Scriptsprache von Unity C#. Neben einer einfachen Handhabung ist die Integration und Nutzung von Klassen aus dem Framework unproblematisch.
- Eine abstrahierte Umsetzung der ASIO-Ausgabe, welche die benötigte Funktionalität bietet: Die Unterstützung von beliebigen ASIO-fähigen Geräten und die Ausgabe auf beliebig viele Kanäle.

Relevante Interfaces und Klassen, welche in dieser Arbeit genutzt bzw. Interfaces die implementiert wurden, sind im Folgenden erläutert:

6.1.1. IWaveProvider

Innerhalb der NAudio-Umgebung sind abstrahierte Klassen, welche Audiodaten halten, entweder als WaveProvider oder als SampleProvider realisiert. Der Unterschied liegt hier in der

[1] NAudio repository: https://github.com/naudio/NAudio

Interpretation der Audio-Rohdaten. Jede IWaveProvider-realisierende Klasse wird auszugebende Audiodaten in ein Byte-Array schreiben müssen. Jede ISampleProvider-realisierende Klasse wird hier hingegen mit einem Float-Array arbeiten. Für eine Bearbeitung von Audiosignalen kann es durchaus von Vorteil sein mit 32 Bit-IEEE-Float zu arbeiten.
Da die ASIO-Klasse aus dem NAudio-Framework jedoch bei der Initialisierung zwingend einen IWaveProvider fordert, wurde dieser auch genutzt.
Prinzipiell fordert das Interface lediglich zwei Dinge ein:

- Einen Member des Typs WaveFormat
- Eine Readfunktion mit der folgenden Signatur:

```
1       int Read(byte[] buffer, int offset, int count)
```

  Übergeben wird der Zielpuffer. Hier werden die zur direkten Ausgabe bestimmten Audiodaten abgelegt. Der Offset in den Puffer beträgt in der Regel 0. Count gibt die Anzahl an Bytes an, die in den Puffer geschrieben werden müssen.

6.1.2. WaveFormat

WaveFormat beschreibt eine Klasse aus dem Audio-Framework in welcher Informationen über die Eigenschaften der Audiodaten gehalten werden. Unter Anderem die Anzahl der Kanäle, die Abtastrate sowie die Bittiefe der Audiodaten. Die Informationen, die hier gehalten werden, müssen unter allen Umständen korrekt sein, da diese auch dem ausgebenden Audio-Interface mitgeteilt werden. Falsche oder korrupte Informationen können hier zu schwerwiegenden Fehlern bei der Ausgabe führen.

6.1.3. WaveStream

Die WaveStream-Klasse ist eine der Basis-Klassen innerhalb des Frameworks. Diese Klasse ist abstrakt und erbt außerdem von der .NET-eigenen Stream-Klasse. Realisiert wird zudem das Interface IWaveProvider. Die Klasse unterstützt unter anderem die Repositionierung und das Auslesen von Länge und Position des Audio-Stroms. WaveStream ist somit eine abstrakte Klasse für die Komplementierung von Audio-Streams.
Eine unterstützende Eigenschaft jeglicher Stream-Klassen ist, dass ein Leseaufruf automatisch zu einer Aktualisierung der Position führt. Diese Eigenschaft wird auch an die Klasse WaveStream vererbt und gestaltet die Verwaltung von vielen Audioströmen innerhalb der Anwendung sehr einfach.

6.1.4. WaveFileReader

Diese Klasse unterstützt das Lesen von .WAV-Dateien in unterschiedlichen Auflösungen und Abtastraten. Geerbt wird von der WaveStream-Klasse.

6.1.5. RawSourceWaveStream

Ein sehr einfacher Stream, der jedoch einen Konstruktor anbietet, welcher Byte-Arrays entgegennimmt und in Kombination mit einer WaveFormat-Instanz einen Audio-Strom erstellt. Intern wird für die Daten ein MemoryStream genutzt.

6.1.6. AsioOut

Die AsioOut-Klasse abstrahiert die Zugriffe auf den ASIO-Treiber. Intern realisiert das Framework diese Abstraktion über 3 Schichten:

- Die AsioOut-Klasse abstrahiert die Funktionalität soweit, dass lediglich ein WaveProvider übergeben werden muss. Über Play- und Stopmethoden wird die Ausgabe entsprechend gesteuert.
- Die AsioOutExt-Klasse realisiert die Callback-Funktionalität auf die Audiopuffer.
- Die AsioDriver-Klasse stellt die unterste Schicht der ASIO-Umsetzung im Framework dar. Hier wird der Zugriff auf den externen ASIO-Treiber über ein COM-Objekt aus der Windows Registry hergestellt.

6.2. Übersicht der umgesetzten Klassen im Detail

In Abbildung 6.1 ist eine Übersicht der implementierten Klassen zu sehen. Die im Zuge dieser Arbeit eigens entwickelten Klassen sind grau unterlegt. Bei der Erstellung der Architektur wurde auf einen möglichst hohen Grad an Modularität geachtet - bei zukünftigen Anpassungen und Erweiterungen kann so unproblematisch Funktionalität und Verhalten eingepflegt oder auch die Implementierung von spezifischen Modulen komplett ersetzt werden. Die diskreten Klassen werden im Folgenden genauer erläutert:

6.2.1. WFSStreamer

Die WFSStreamer-Klasse ist die kapselnde Schicht des Streaming-Moduls und bietet Schnittstellen nach außen hin an. Die WFSStreamer-Klasse ist als Singleton implementiert und nimmt

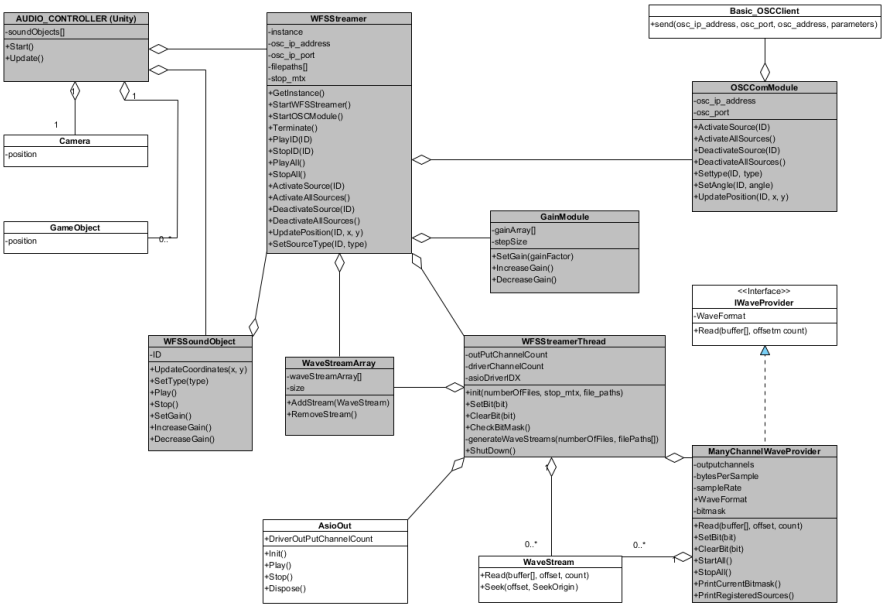


Abbildung 6.1.: Übersicht der genutzten Klassen als UML-Klassendiagramm

als initiale Argumente den Index des zu nutzenden ASIO-Gerätetreibers entgegen, die Anzahl der zu nutzenden Kanäle, sowie die Adressdaten des zu nutzenden OSC Servers. Sollten Werte außerhalb der Spezifikation übergeben werden, wird dies durch entsprechende Exceptions behandelt.

Neben der Singleton-Eigenschaft bietet die WFSStreamer-Klasse außerdem eine Factory-Methode für das Generieren neuer Virtueller Soundobjekte. Grundlegend existieren zwei Überladungen der Factory-Methode:

- Mit der Angabe eines Dateipfads: Dieser muss zwingend zu einer monophonen WAV-Datei mit einer Auflösung von 16 Bit führen und sollte zusätzlich mit einer Abtastfrequenz von 48.000 Hz vorliegen. Der entsprechende Dateipfad wird einem WaveFileReader übergeben. Der WaveFileReader wird dem WaveStreamArray hinzugefügt. Sollte dieses voll sein, wird ein Fehlercode zurückgereicht. Andernfalls wird der interne Index, welcher auch mit der Kanalnummer korrespondiert, zurückgegeben.
- Mit der Übergabe eines Float-Arrays: Diese Überladung wurde zusätzlich implementiert, um die Unity-eigenen Audioclips zu unterstützen. Die Audiodaten, die Unity weiterreicht, liegen lediglich im Float-Format vor. Die Werte werden intern aus dem Float-Bereich in den Integer bzw. Byte-Bereich konvertiert. Anschließend werden die Daten einem

RawSourceWaveStream übergeben, welcher dem WaveStreamArray hinzugefügt wird. Auch hier wird wieder ein Index an das SoundObject gereicht.

Anmerkung zum Kanalindex, welcher an das Soundobjekt weitergereicht wird: Dieser ist für den Nutzer nicht von Relevanz und wird lediglich für die Kommunikation zwischen Soundobjekt und WFSStreamer benötigt. Es können in der aktuellen Version des Systems nur so viele Soundobjekte zur selben Zeit existieren, wie auch Kanäle verfügbar sind. Im Falle eines zurückgereichten Fehlercodes werden die Audiodaten nicht zum WaveStreamArray hinzugefügt.

Generell können innerhalb einer Laufzeitinstanz beide Überladungen simultan genutzt werden. Die Erstellung sowie das Anstarten des Streamerthreads sowie das Instanziieren der OSC-Klasse finden auch hier statt. Neben dieser Funktionalität werden in dieser Klasse auch intern jegliche abstrahierte Steuerbefehle auf virtuelle Schallquellen gesammelt und entsprechend verteilt.

6.2.2. WFSSoundObject

Diese Klasse realisiert und abstrahiert den Zugriff auf die Steuerung der diskreten Soundströme. Nach außen hin wird ein einfach anzusprechendes virtuelles Soundobjekt präsentiert. Der Nutzer hat über die jeweilige Instanz Zugriff auf die folgenden Kontrollfunktionen des Moduls:

- Play: Lässt den zugeordneten Sound abspielen. Ein wiederholter Aufruf auf einen Sound, welcher abgespielt wird, wird ignoriert.
- Stop: Stoppt den zugeordneten Sound und setzt die Position an den Anfang zurück. Ein wiederholter Aufruf wird ignoriert.
- Pause:Stoppt den zugeordneten Sound und speichert die aktuelle Position. Ein erneuter Aufruf wird ignoriert. Ein Aufruf von Play setzt das Abspielen an der pausierten Stelle fort.
- LoopOn/LoopOff: Aktiviert/deaktiviert das endlos wiederholte Abspielen eines Sounds. Initial ist die Wiederholung.
- SetType: Erlaubt ein Umschalten des Quelltypen. 0 steht hier für Planarquellen und 1 für Linearquellen.
- SetTypePoint/SetTypePlane: Wie SetType, jedoch ohne Parameter.

- SetGain: Erlaubt ein Anpassen der Lautstärke für einen spezifizierten Sound. Alternativ gibt es auch eine Master-Gain-Methode. Erlaubte Werte liegen hier zwischen 0 und 1 im Float-Format. Die Verwaltung der Lautstärke läuft über eine weitere entsprechende, interne Klasse.
- IncreaseGain/DecreaseGain: Erlaubt ein sukzessives Erhöhen bzw. Verringern der Lautstärke.
- UpdateCoordinates: Übermittelt die aktuelle virtuelle Position der Schallquelle an das System. Optional auch mit Zeitkomponente.

6.2.3. WFSStreamerThread

Der WFSStreamerThread wird, wie der Name schon schließen lässt, als eigener Thread instanziiert und in einem separaten Thread-Appartment gestartet. Dies ist für die ASIO-Funktionalität nötig. Die entsprechende ASIO-Klasse wird innerhalb dieser Thread-Klasse instanziiert. Außerdem wird die ASIO-Klasse hier auch mit der entsprechenden Instanz initialisiert, welche jegliche Audioströme verwaltet, dem ManyChannelWaveProvider.

6.2.4. WFSManyChannelWaveProvider

Diese Klasse realisiert die Verwaltung der diskreten Audioströme, welche zur Laufzeit auf das entsprechende Audio-Gerät gegeben werden. Der ManyChannelWaveProvider trägt durch die Realisierung des IWaveProvider-Interfaces die Eigenschaft, als WaveStream aufgefasst zu werden und kann somit einer Ausgabekomponente des Frameworks übergeben werden. In diesem Fall der ASIO-Klasse. Generell ist in dieser Klasse die Readfunktion, welche von außerhalb, nämlich vom ASIO-Treiber aus aufgerufen wird, implementiert. Somit werden zyklisch in gewissen zeitlichen Abständen die Audiodaten in die Ausgabepuffer geschrieben. Die zeitlichen Abstände hängen von der Puffergröße ab, welche außerhalb des Moduls eingestellt wurde - in der Regel in den Einstellungen des Audio-Interfaces.

```
1 uint bitmask;
2 WaveStream[] audioData;
3 float[] gain;
4 Read(byte[] buffer, int offset, int count){
5   int frameSize = outputChannels * bytesPerSample;
6   int framesRequired = count / frameSize;
7   int currentPos = offset;
8   int extraBytes = count - (framesReq * frameSize);
```

```
  if (frameSize > count){
    reportError();
  }
  while(framesRequired){
    foreach(entry in audioData){
      if(corresponding playBit is set){
        audioData[i].Read(sample, 0, bytesPerSample);
        if(the end of the audio stream has been reached){
          ResetStream(i);
        if (loop is not enabled){
          ClearPlayBit(i);
          break();
        }
      }
        sample * gain[];
        buffer[currentPos] = sampleLowByte;
        currentPos++;
        buffer[currentPos] = sampleHighByte;
        currentPos++;
      }else{
        Array.Clear(buffer, currentPos, bytesPerSample);
        currentPos += bytesPerSample;
      }
    }
  }
  foreach(emptyChannel){
    Array.Clear(buffer, currentPos, bytesPerSample);
    currentPos += bytesPerSample;
  }
}
```

Listing 6.1: Pseudocode der implementierten Read-Methode welche von der ASIO-Klasse zyklisch aufgerufen wird

Die grundlegende Idee hinter dem Ausgabe-Algorithmus, in Listing 6.1 dargestellt, ist wie folgt:

- Über die gesamte Datenstruktur der Audiosignale wird iteriert.
- Es existiert eine Bitmap, korrespondierend zu den jeweiligen Indices der Audioströme, die in der entsprechenden Datenstruktur abgelegt sind, dem WaveStreamArray.

- Ist das entsprechende Bit gesetzt, soll der relatierte Audiostrom abgespielt werden. Ist das Bit nicht gesetzt, wird der am entsprechenden Index liegende WaveStream oder Audiostream übergangen. Die jeweilige Position bleibt durch die Eigenschaft der Stream-Klasse intern gespeichert.
- Falls das Ende eines WaveStreams erreicht ist, wird der entsprechende Audiostrom zurückgesetzt. Außerdem wird geprüft ob das zugehörige Bit im Loop-Array gesetzt ist. Ist das der Fall, wird der Strom wieder vom Anfang aus angespielt. Ansonsten wird das Abspiel-Bit auf 0 gesetzt, somit wird der Audiostrom von jetzt an übergangen.
- Falls der über den aktuellen Index ausgewählte Stream abgespielt werden soll, wird das abgegriffene Sample mit dem zugehörigen Gainwert (Lautstärke) multipliziert und dann Byte-weise in den Zielpuffer geschrieben.
- Sollte das entsprechende Bit nicht gesetzt sein, werden Nullen in der Ausgabepuffer geschrieben.
- Das Modul kann mit mehr Ausgabekapazitäten initialisiert worden sein als Audioströme geladen sind. In diesem Fall werden alle unbelegten Kanäle mit 0 beschrieben.

Generell kann dieser Algorithmus noch optimiert werden. Nicht zwangsweise muss es nötig sein, für jeden Zyklus und jeden Index Daten in die Ausgabepuffer zu schreiben. Ist ein spezifischer Audiostrom beispielsweise pausiert oder liegen unbelegte Kanäle vor, könnte durch entsprechende Positions-Arithmetik etwas Laufzeit eingespart werden.

6.2.5. OSCComModule

Diese Klasse nutzt das von Stefan Heidtmann entwickelte OSC-Plugin CSharp OSC. Hier werden die Adressdaten des entsprechenden OSC-Servers gehalten und jegliche zu übermittelnde OSC-Befehle gekapselt. Für jeden OSC-Befehl wird eine eigene Methode zur Verfügung gestellt. Bei der Übermittlung von mehreren Nachrichten innerhalb eines kurzen Zeitabschnitts, z.B. das iterative Aktivieren von 32 Quellen, ist zu berücksichtigen, dass Nachrichten verloren gehen können. Dies ist vermutlich auf zu kleine Nachrichtenpuffer und mangelnde Verarbeitungsgeschwindigkeit seitens des OSC-Servers zurückzuführen, wurde jedoch im Zuge dieser Arbeit nicht weiter untersucht.

6.2.6. WFSWaveStreamArray

In dieser Klasse werden die diskreten Audiodaten für jeden Ausgabekanal als WaveStream-Objekte gehalten und verwaltet. Da intern mit fixen Kanalzuordnungen gearbeitet wird - zur

Laufzeit hat ein Audiostrom einen Kanalindex, der sich zur Abspielzeit nicht ändert - wurde hier als Datenstruktur das Array gewählt. Darüber hinaus wird hier auch geprüft, ob für einen hinzufügenden Audiostrom noch hinreichend Kapazitäten verfügbar sind. Falls dem so ist, wird die Referenz auf den Audiostrom in das Array geschrieben und der entsprechend genutzte Index zurück an den Aufrufer gegeben. Im Fehlerfall wird -1 zurückgegeben.

6.3. Entwicklung des Unity-Scripts

Zusätzlich zu der .NET-Klassenbibliothek, aus Unity-Sicht ein managed Plugin, wurde noch ein Unity-Script entwickelt. Dieses kann als Komponente an Elemente in der Unity-Hierarchie hinzugefügt werden und integriert sich somit in die GUI. Ein Ausschnitt der Unity-Nutzeroberfläche in Abbildung 6.2 verdeutlicht dies:

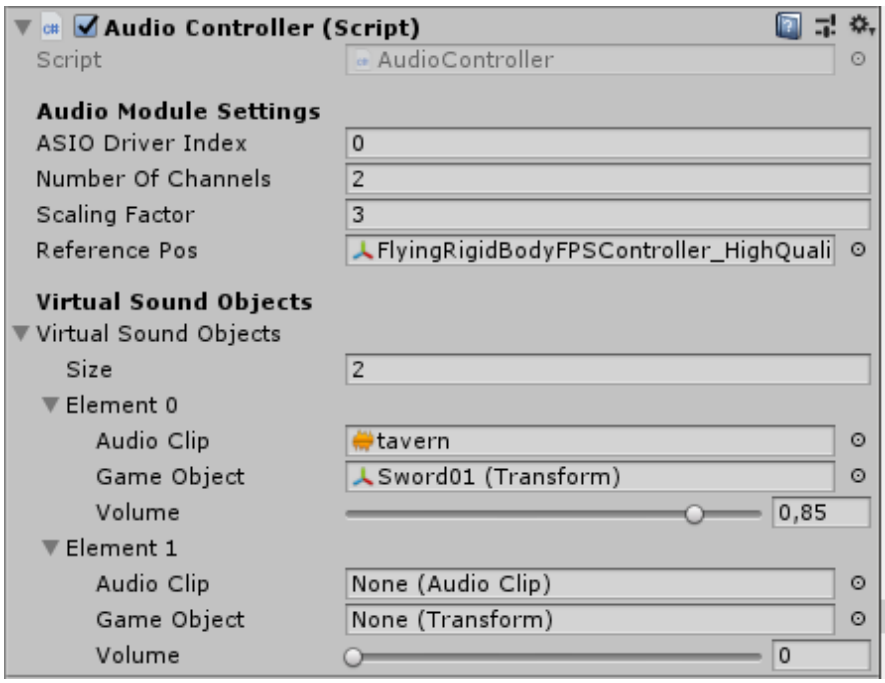


Abbildung 6.2.: Sicht auf den AudioController in der Unity IDE

Im oberen Bereich der AudioController-GUI befinden sich die technischen Einstellungen des Systems. Neben der Eingabemaske für den Treiber, welcher über einen Index angesprochen wird, kann hier auch die Anzahl der zu nutzenden Kanäle ausgewählt werden. Limitiert wird diese Einstellung lediglich durch die zugrunde liegende Hardware. Die eingestellten Werte werden so an die Klassenbibliothek weitergereicht. Des Weiteren lässt sich der Skaling-Faktor

entsprechend regulieren - Der Skaling Faktor staucht bzw. dehnt die Entfernung der virtuellen Schallquelle zur virtuellen Kamera - dem Nutzer. Je nach entwickelter Szene mag es nötig sein, diese Verhältnisse anzupassen.

Im unteren Segment der GUI findet sich das eigentliche Feld mit abstrakten Objekten, bestehend aus jeweils einem Audioclip und einem an diesen gekoppelten Gameobjekt. Beide Elemente können entweder per Drag-and-Drop in die Felder gezogen werden oder über den Auswahlknopf auf der rechten Seite in der Hierarchie gesucht und hinzugefügt werden. Implementiert sind die gekoppelten Objekte als Klassen.

6.3.1. Übersetzung der Koordinatensystem bzw Umsetzung der Kamera

Um eine korrekte Zuordnung von Schallquellen zu visuellen Objekten zu realisieren, müssen die Positionen zum Bezugspunkt, dem Nutzer, extrahiert werden. Dies sind die eigentlichen Metadaten im Kontext des objektbasierten Audio. Diese Metadaten werden an die Klassenbibliothek weitergereicht und entsprechend an die Rendering-Engine der WFS-Anlage gesendet.

Generell müssen hinsichtlich der Herangehensweise an die Positionsberechnung in der interaktiven 3D-Umgebung zwei Fälle unterschieden werden:

- Der konventionelle Aufbau ohne Virtual-Reality-Headset, wie er im Zuge dieser Arbeit voll implementiert wurde.
- Ein Aufbau mit einem Virtual-Reality-Headset, in dieser Arbeit theoretisch betrachtet, jedoch nicht voll implementiert und getestet.

Im Folgenden sind die beiden Ansätze erläutert:

6.3.1.1. Ohne Virtual-Reality-Brille

Hier ist die visuelle Schnittstelle der Bildschirm, auf dem die 3D-Szene betrachtet wird. Die Sicht, welche der Nutzer auf die 3D-Szene hat, ist gleichbedeutend mit dem Bild, welches durch die virtuelle Kamera in der Unity-Szene „aufgenommen“wird. Dieses virtuelle Kameraobjekt existiert in jeder Szene und wird in der Regel durch die Nutzereingaben über das Koordinatensystem der virtuellen Szene bewegt. Somit lässt sich sagen, dass die Koordinaten, welche die Kamera hat, sich abhängig von den Nutzereingaben verändern.
Zusätzlich hat auch jedes virtuelle Objekt in der Unity-Welt eigene Koordinaten. Somit auch

die in dieser Arbeit eingeführten Schallquellen. Diese sind nun jeweils relativ zum sich bewegenden Kameraobjekt. Diese relative Ausrichtung muss zu jedem berechnetem Bild ebenso neu berechnet werden, damit Bild und Ton auch bei einer Bewegung zueinander passen.
Bei der Berechnung der relativen Koordinaten sind zwei Punkte zu beachten bzw. entsprechend umzusetzen. Generell wird davon ausgegangen, dass der Bezugspunkt in der realen Welt im Zentrum der WFS-Anlage liegt:

- Die starre Bewegung auf der Ebene ohne Drehung. Hier wird die relative Position von virtuellen Schallquellen durch simple Vektorsubtraktion berechnet:

$$\begin{pmatrix} x_{rel} \\ yrel \end{pmatrix} = \begin{pmatrix} xCamera - xSoundObject \\ yCamera - ySoundObject \end{pmatrix} \tag{6.1}$$

- Die Drehbewegung der Kamera. Dies stellt ein Problem dar: Dreht sich der Nutzer auf der eigenen Achse, so verändert sich die Entfernung der virtuellen Schallquelle nicht, jedoch ändert sich der relative Hörwinkel. Eine Schallquelle, die vom Nutzer aus der Blickrichtung wahrgenommen wird, befindet sich nach einer Achsendrehung um 180° hinter ihm. Gelöst wird dieses Problem durch eine Rotationsmatrix. Die Positionsberechnung wird mithilfe der Multiplikation der relativen Positionswerte mit der Matrix durchgeführt. Der Drehwinkel α, um welchen ein Bezugsobjekt um die eigene Achse gedreht ist, wird direkt von der Unity-Engine abgegriffen. Der Initialzustand, nach dem Starten einer virtuellen Szene, ist standardmäßig 0°:

$$\begin{pmatrix} x_{wfs} \\ y_{wfs} \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cdot\cos\alpha \end{pmatrix} \cdot \begin{pmatrix} x_{rel} \\ y_{rel} \end{pmatrix} \tag{6.2}$$

6.3.1.2. Mit Virtual-Reality-Brille

Hier ist die visuelle Schnittstelle zur 3D-Umgebung das Virtual Reality Headset. Ebenso wie im vorherigen Punkt wird hier der Bezug zur virtuellen Umgebung über ein Kameraobjekt hergestellt. Jedoch spielt die Winkelausrichtung des Headsets keine Rolle. Die virtuelle Umgebung ist von der Ausrichtung her und in Bezug zum reellen Raum fixiert.

7. Technische Evaluation des entwickelten Moduls

In diesem Kapitel wird das entwickelte Modul hinsichtlich der Performanz sowie der auf dem Hostsystem verwendeten Ressourcen untersucht. Insbesondere soll verifiziert werden, dass ein Betrieb, parallel zu einer laufenden und ohnehin schon anspruchsvollen visuellen 3D-Plattform problemlos möglich ist.

7.1. Die Testumgebung

Als grundlegende Testumgebung für die Ermittlung des anteiligen Ressourcenverbrauchs wurde Unity in der Version 2018.1 genutzt. Die Vergleichstests hinsichtlich des Ressourcenverbrauchs sowie einiger Zeitmessungen wurden auf Basis des von Unity bereitgestellten Demoprojekts Viking Village ausgeführt. Die Szene im Demo-Projekt beinhaltet viele Assets und aufwendige Texturen sowie Shader und ist mit einer üblichen Szene einer interaktiven Umgebung bzw. einem Level eines Spiels vergleichbar und wird hier als Vergleichsreferenz genommen. Die Laufzeitmessungen des relevanten Codebereichs wurden unter Visual Studio 2017 in der Enterprise-Version genutzt. Die genutzte Hardware war ein Workstation Notebook mit einem Intel I7 Prozessor der 4. Generation, 4 Kernen sowie 2,90 GHz Takt. Alle Messungen und Analysen wurden auf einer SSD und mit 16 GB verfügbarem Arbeitsspeicher durchgeführt. Die Laufzeitmessungen und eine Analyse des Speicherverbrauchs der kritischen Programmteile wurde innerhalb des Leistungs-Profilers von Visual Studio durchgeführt. Außerdem wurde die Debuggingfunktion der IDE genutzt.

7.2. Laufzeitmessungen

Getestet wurde jeweils mit einer unterschiedlichen Anzahl an genutzten Kanälen. Außerdem wurde extern die ASIO-Puffergröße iterativ für jede Testsequenz verringert. Generell ist bei diesem Vorgehen eine lineare Relation von Ergebnissen zu Parametern erwartbar, da der Algorithmus eine Komplexität von O(N) aufweist. Gemessen wurde mithilfe der Stopwatch-Klasse.

Auf dem Testsystem stand ein Timer mit einer Auflösung von 2.825.516 Ticks zur Verfügung. Dies bedeutet, dass in der Theorie Zeitabstände von 353,9 ns gemessen werden können. Dies entspricht 0,000353 ms. Mithilfe des Debuggers wurden über eine Dauer von 30 Sekunden die Zeitstempel genommen. Gemessen wurde hier unmittelbar in der Funktion. Die Ticks wurden jeweils nach einem kompletten Funktionsdurchlauf entnommen.
Gemessen wurde die Laufzeit mit nur einer extern eingestellten Puffergröße. Eine Anpassung dieser wirkt sich lediglich auf die Frequenz aus, mit welcher die Funktion aufgerufen wird. Die Durchlaufzeit der Funktion selbst wird nicht beeinflusst.

In Diagram 7.1 ist die Laufzeit der Readfunktion bei ansteigender Kanalanzahl dargestellt. Zu sehen ist eindeutig das erwartete lineare Verhalten. Als ASIO-Puffergröße waren extern 512 Samples eingestellt. Die Werte wurden entsprechend auf die Bearbeitungsdauer einzelner Samples heruntergerechnet. Anzumerken ist, dass die Länge des Puffers auch gleichzeitig die Latenz bestimmt, mit der Signale ausgegeben werden können - frühestens wenn eine gesamte Pufferlänge beschrieben ist, erfolgt die Ausgabe.

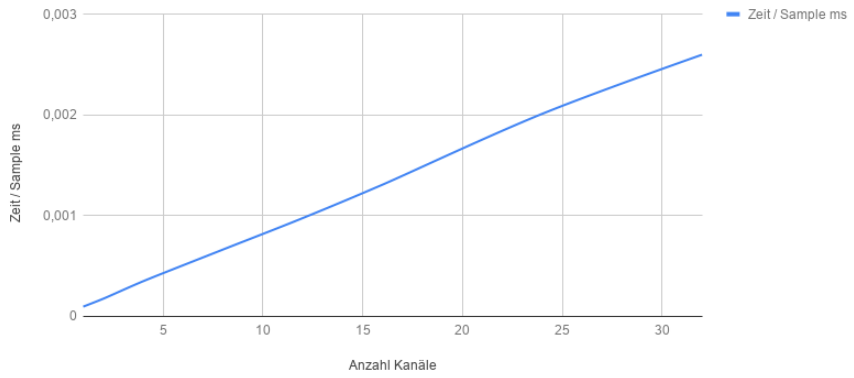
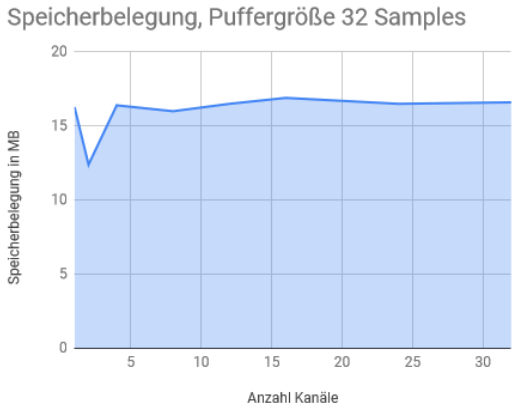


Abbildung 7.1.: Laufzeiten für die Ausgabe eines Sound-Samples mit ansteigender Kanalzahl

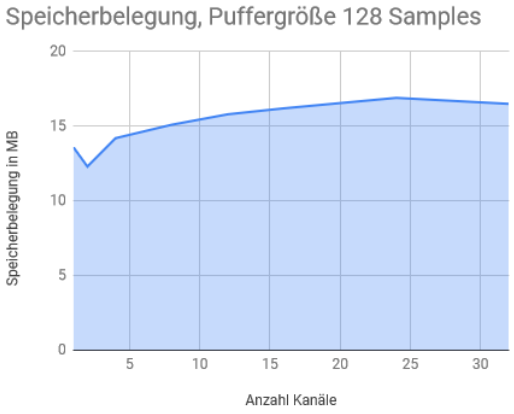
Die WFS-Anlage fordert eine Abtastrate von 48.000 Hz ein. Dies bedeutet, dass alle 0.02083 ms ein Sample zur Ausgabe bereitstehen muss. Selbst bei einer Ausgabe von 32 simultanen Signalen wird die Anlage theoretisch alle 0,0025 ms mit einem Sample beliefert. Aufgrund technischer Limitierungen - die virtuelle Dante-Soundkarte unterstützt maximal 32 Kanäle - konnten jedoch keine Messungen mit höherer Kanalzahl durchgeführt werden. Ausgehend von einem weiterhin linear verlaufenden Verhalten, würde das entwickelte System 256 Signale zeitgleich wiedergeben können.

7.3. Speicherbelegung

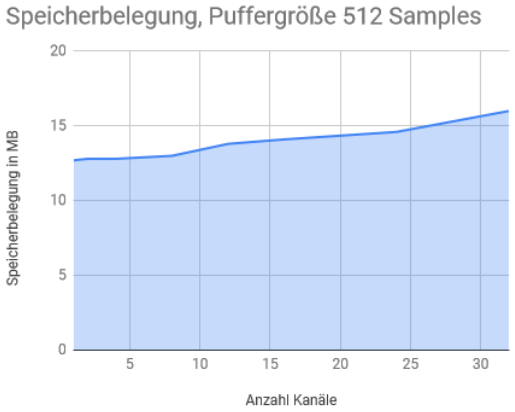
Die Speicherbelegung wurde mithilfe des in Visual Studio integrierten Leistungsprofilers gemessen und über einen Zeitraum von 3 Minuten analysiert. Die Analyse wurde für unterschiedlich extern eingestellte Puffergrößen analysiert. Als Testdatei wurde eine Audiodatei von etwa 5 Minuten Länge gewählt, mit einer Auflösung von 16 Bit. Die Ergebnisse sind in Abbildung 7.2 dargestellt.



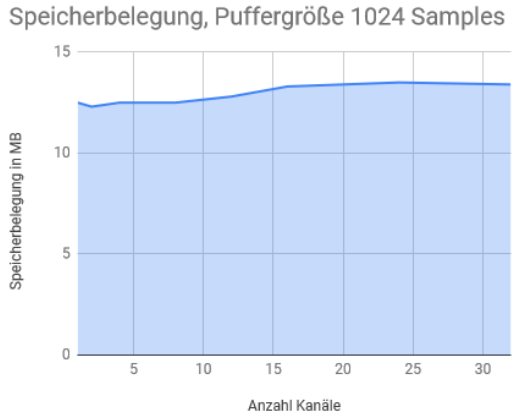
(a) 32 Samples



(b) 128 Samples



(c) 512 Samples



(d) 1024 Samples

Abbildung 7.2.: Speicherbelegung nach 3 Minuten Laufzeit, gemessen mit unterschiedlich eingestellten Puffergrößen

Generell ist erkennbar, dass mit einer Ausgabe von 2 Kanälen der Speicherverbrauch tendenziell am niedrigsten ist. Außerdem erhöht sich der Speicherbedarf mit verringerter Puffergröße, d.h. bei kleinerer Iterationsgröße - die Read-Methode wird häufiger aufgerufen. Insgesamt lässt

sich klar erkennen, dass selbst mit 32 zeitgleich geladenen und auszugebenden Audioströmen kein hoher Speicherverbrauch vorliegt.

7.4. Analyse des Ressourcenverbrauchs der Unity-Umgebung

Zusätzlich zu der Analyse der Laufzeiten und der isolierten Messung des Speicherbedarfs des Moduls, wurden die Werte in den Kontext des in dieser Arbeit genutzten Unity-Beispielprojekts gesetzt. Untersucht wurde ein Build des Projekts außerhalb der Unity-Umgebung. Gemessen wurde eine Speicherbelegung von 230-240 MB. Die CPU-Auslastung betrug zwischen 10% und 14%.

7.5. Bewertung

Der gemessene Speicherverbrauch des entwickelten Moduls für eine Nutzung mit bis zu 32 Kanälen befindet sich in einem Bereich von ca. 12 MB bis etwa 16 MB. Die Laufzeit der relevanten Ausgabefunktion ist in einem gemessenen Bereich von 1 bis 32 Kanälen linear. Pro auszugebendem Kanal und mit einer anliegenden ASIO-Puffergröße von 512 Samples ist mit einer CPU-Zeit von etwa 0,05 ms zu rechnen. Berücksichtigt man die Tatsache, dass Speichergrößen von 8 GB bis 16 GB selbst auf Heimanwenderrechnern zu finden sind, so belegen die gemessenen Speicherwerte, dass das Streaming-Modul problemlos in eine Unity-Szene mit vielen Assets integriert werden kann. Aufgrund des Multi-Threading-Verhaltens wird der Unity-eigene Ablauf des Weiteren nicht durch das implementierte Modul unterbrochen. Insbesondere mit Hinsicht auf die Verbreitung von Mehrkern-CPUs stellt auch eine kontinuierliche Ausgabe von vielen Kanälen kein Problem dar.

8. Zusammenfassung

In dieser Arbeit wurde ein Modul entwickelt, welches in Echtzeit objektbasierte Audiodaten von der Unity-Umgebung auf eine WFS-Anlage überträgt. Dies beinhaltet das parallele Übertragen von Audiodaten sowie den zugehörigen Positionsinformationen.

8.1. Fazit

Spezifiziert wurde eine explizite Initialisierung des Moduls sowie das einfache Anlegen von Schallquellen, ausgehend von bestehenden Audiodaten. Dies wurde auf zwei Arten realisiert: Über die direkte Auswahl von den Unity-eigenen Audioclips sowie der Übermittlung von Dateipfaden. Des Weiteren wurde in den Anforderungen eine einfache Kontrolle des Moduls eingefordert. Unter Anderem sollte die Möglichkeit geschaffen werden, Audio-Streams unabhängig voneinander abzuspielen und zu stoppen, die Lautstärke zu regulieren sowie die Positionsänderungen von Schallquellen über OSC-Befehle zu übertragen. Über eine entsprechende API der entwickelten Klassenbibliothek - gepackt als .DLL-Datei - wurden diese Funktionen umgesetzt und angeboten.
Das eigens entwickelte Modul baut auf das bestehende NAudio-Framework auf und nutzt zusätzlich eine externe OSC-Bibliothek für das Übertragen der Metadaten. Neben der entwickelten Klassenbibliothek, welche als dedizierte Datei in ein Unity-Projekt eingebunden werden kann, wurde ein zusätzliches Unity-Script entwickelt. Dieses kapselt die Funktionalität der Klassenbibliothek und automatisiert die Übertragung der Positionsänderungen von Schallquellen. Entwickelt sowie evaluiert und getestet wurde das Modul innerhalb eines von Unity öffentlich bereitgestellten Demo-Projekts. Das entwickelte Modul war voll nutzbar und die gesetzten Anforderungen wurden wie spezifiziert umgesetzt. Auch eine Requirementsstudie zeigte bereits gute Ergebnisse und eröffnete Raum für Anpassungen des Moduls sowie für weitere offene Requirements.

8.2. Probleme

Im Zuge dieser Arbeit haben sich zwei wesentliche Probleme herausgestellt, welche hier kurz erläutert werden sollen:

- Die rotierende Welt:
  Ein Problem, welches auch noch sehr störend bei der Erkundung virtueller Umgebungen ist, ist der Umstand, dass die Welt sich um den Nutzer dreht. Dies betrifft auch die Schallquellen, welche zum Teil in absurder Geschwindigkeit um den Nutzer fliegen, wenn dieser sich mit schnellen Bewegungen durch die Welt bewegt bzw. er sich um die eigene Achse dreht. Dies ist stark abhängig von der Richtung und Geschwindigkeit der Bewegungen innerhalb der 3D-Welt. Der Grund für diese unschönen Effekte ist das physikalisch korrekte Renderingverhalten der WFS-Anlage. So wird bei der Ausgabe in jedem Fall der Dopplereffekt angewandt.

- Die Plattformabhängigkeit:
  In der Unity-Engine entworfene Umgebungen und Projekte sind zwar auf vielen Geräten und Plattformen lauffähig, jedoch ist das genutzte Framework momentan noch sehr stark an die Windows-Umgebung gebunden. Dies liegt unter anderem an dem verwendeten Audio-Framework, welches Betriebssystem-spezifische Aufrufe nutzt.

8.3. Ausblick und offene Requirements

Auch wenn das entwickelte Modul und der in Unity aufgesetzte AudioController bereits voll funktionsfähig sind, gibt es dennoch viele mögliche Erweiterungen und Anpassungen, die vorgenommen werden können:

- Eine noch dynamischere Einbindung in die Unity-Umgebung: So könnten Schallquellen direkt über die Editor- bzw. Inspectorsicht der Unity-Umgebung hinzugefügt werden.

- Eine Ausblendung von Schallquellen, ähnlich dem Level-of-Detail-Prinzip, wie es in der Grafikberechnung zu finden ist: Je nach Entfernung sollen Schallquellen deaktiviert werden und somit wieder Ressourcen für Neue bieten.

- Eine Berücksichtigung der dritten Dimension: Momentan eignet sich das entwickelte Streamingmodul noch nicht gut für 3D-Welten mit vielen Höhenebenen.

Literaturverzeichnis

[Baa08] Marije Alberdina Johanna Baalman. On wave field synthesis and electro-acoustic music, with a particular focus on the reproduction of arbitrarily shaped sound sources. 2008.

[Bau17] Felix Baumgartner. Steuerung einer wellenfeldsynthese-anlage mittels eines virtual-reality-interfaces, 2017.

[BSS17] Nicolas Bouillot, Zack Settel, and Michal Seta. Satie: a live and scalable 3d audio scene rendering environment for large multi-channel loudspeaker configurations. In *New Interfaces for Musical Expression*, 2017.

[CSS15] Sofia Cavaco, Diogo Simões, and Tiago Silva. Spatialized audio in a vision rehabilitation game for training orientation and mobility skills. In *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, 2015. sem pdf conforme despacho.

[Dea17] Dean Takahashi. 59% of vr developers use unity, but devs make more money with unreal. "https://uploadvr.com/vr-developers-unity-unreal/", 2017.

[FHB+14] Simone Fueg, Andreas Hoelzer, Christian Borss, Christian Ertel, Michael Kratschmer, and Jan Plogsties. Design, coding and processing of metadata for object-based interactive audio. In *Audio Engineering Society Convention 137*. Audio Engineering Society, 2014.

[Foh13] Wolfgang Fohl. *The Wave Field Synthesis Lab at the HAW Hamburg*, pages 243–255. Springer International Publishing, Heidelberg, 2013.

[He17] JianJun He. *Spatial audio reproduction with primary ambient extraction*. Springer, 2017.

[HHKP15] Jürgen Herre, Johannes Hilpert, Achim Kuntz, and Jan Plogsties. Mpeg-h 3d audio—the new standard for coding of immersive spatial audio. *IEEE Journal of selected topics in signal processing*, 9(5):770–779, 2015.

[MNF16] Florian Meyer, Malte Nogalski, and Wolfgang Fohl. Detection thresholds in audio-visual redirected walking. In *Proc. 13th Sound and Music Computing Conf.* SMC, September 2016.

[Nog15] Malte Nogalski. Acoustic redirected walking with auditory cues by means of wave field synthesis. Master's thesis, HAW Hamburg, Berliner Tor 5, 20099 Hamburg, 2015.

[OSS14] Robert Oldfield, Ben Shirley, and Jens Spille. An object-based audio system for interactive broadcasting. In *Audio Engineering Society Convention 137*, Oct 2014.

[RJBM15] Joseph David Rees-Jones, Judith Sara Brereton, and Damian Thomas Murphy. Spatial audio quality and user preference of listening systems in video games. pages 223–230, 11 2015.

[SBS16] Zack Settel, Nicolas Bouillot, and Michal Seta. Volumetric approach to sound design and composition using SATIE: a high-density 3D audio scene rendering environment for large multi-channel loudspeaker configurations. In *15th Biennial Symposium on Arts and Technology*, Ammerman Center for Arts and Technology at Connecticut College, New London, February 2016. 8 pages.

A. Anhang zur Requirements-Studie

A.1. Aufgabenblatt der Requirements-Studie

Aufgaben zum WFSStreamer

Die für das Projekt benötigten Dateien, sowie das Projekt Selber findest du auf dem Desktop im Ordner "UnityWFSStreamer", bearbeite die folgenden Aufgabenteile bevor du mit dem Ausfüllen des kurzen Fragebogens fortfährst.

1.) Einbinden des Moduls:

- Starte Unity und öffne das Projekt „RSVikingVillage"
- Binde den WFSStreamer in das Projekt ein. Dazu die Dateien WFSStreamer.dll, NAudio.dll, und AudioController.cs in den Asset-Ordner des Projekts kopieren (die Entwicklungsumgebung wird eventuell einen Moment zum kompilieren brauchen).
- Überprüfe in den Build-Settings des Projekts ob der entsprechende .NET Standard korrekt eingestellt ist (4.x oder höher), andernfalls wird das Modul nicht laufen! File → Build Settings → Player Settings → Configuration
- Gehe in die Ansicht Hierarchy und erstelle eine leeres Gameobjekt „Audio" o.Ä. hier kannst du nun den, in den Asset-Ordner kopierten, AudioController als Komponente hinzufügen. Wähle hierbei „Script" aus.
- In der Scriptansicht (auf der GUI des Editors) kannst du den ASIO Driver Index mit 1 belegen (dies ist die virtuelle Dante-Soundkarte auf dem Labornotebook) und „Number Of Channels" mit 32.
- Reference Pos muss mit der aktiven primären Kamera belegt werden. Im Beispielprojekt finden sich eine Low-Res Kamera und eine High-Res Kamera, nutze hier die High-Res version. Die Kameras sind im Reiter QualityManager zu finden, als Referenzobjekt ist hier der FlyingRigidBody zu nutzen. Das Kameraobjekt dient hier als Referenzpunkt für die Berechnung der Objekte in der 3D-Welt.
  V Sound Objects (Virtuelle Soundobjekte) kann mit beliebigen Unity Objekten belegt werden (drag and drop).

2.) Nutzen des Moduls:

- über den AudioController in der Editoransicht:

Nutze für die folgenden Aufgabenteile die Editoransicht des AudioController-Scripts, öffne diesen dazu in der Inspector-Sicht auf der rechten Seite. Die Pfade zu den Sounddateien können über die Felder Wave Files hinzugefügt werden, beispielsweise:

C:\VikingVillageSounds\fire.wav.

Die Feldgröße von V Sound Objects und Wave Files muss gleich sein. Sobald das Projekt gestartet wurde, wird mit „z" ein Abspielen der Sounds eingeleitet.

Abbildung A.1.: Aufgabenblatt der Requirements-Studie, Blatt 1

- Versehe acht Fackeln mit virtuellen Schallquellen, die Fackelobjekte findest du unter Content → Props → TorechesFires, Gameobjects über die Unity Oberfläche auf die Schallquellen ziehen.
- Versehe eines der großen Häuser mit Tavernensound. Häuser findest du unter Content → Buildings → Buildings.
- Versehe einen der Versammlungssteine mit dem bereitgestellten Musikstück

- über die C# API:

Öffne für den folgenden Aufgabenteil die Datai „AudioController.cs“. Visual Studio sollte sich öffnen. Sobald die Datei gespeichert ist, wird Unity den Scriptcode neu kompilieren.

- Schaue nun einmal in den Code des Audiocontrollers und versuche die folgenden Dinge anzupassen:
  - Reduziere die Lautstärke aller Quellen auf die Hälfte (0.5f)
  - Stelle das wiederholte Abspielen einer Quelle aus (Loop)
  - Implementiere ein Stoppen auf Tastendruck. Play ist bereits implemtiert, du kannst dich gerne daran orientieren

Viel Erfolg!

Abbildung A.2.: Aufgabenblatt der Requirements-Studie, Blatt 2

A.2. Fragebogen der Requirements-Studie

Fragebogen zur Requirements-Studie

Allgemeine Fragen:

Wie erfahren würdest du einschätzen, im Umgang mit	Einsteiger	Fortgeschrittener	Profi
- der Unity-Umgebung	□	□	□
- der Entwicklung von C#-Code	□	□	□

In Welche Fachrichtung würdest du dich einordnen:	Informatik	Design/Kunst	Andere
	□	□	□

Falls andere: _____

Fragen zur Integration des WFSStreamers:

Wie verständlich fandest du das Einbinden des Moduls in die Unity-Umgebung mithilfe der Anleitung?	Verständlich □	Weiß nicht □	Unverständlich □
Findest du, dass das Modul zu viele Dateien enthält die eingebunden werden müssen?	Ja □	Weiß nicht □	Nein □
Findest du, dass zu viele Schritte durchlaufen werden müssen um das Modul einzubinden?	Ja □	Weiß nicht □	Nein □
Würdest du die Nutzung einer zusätzlichen Konfigurationsdatei bevorzugen (Für technische Einstellungen wie IP-Adresse, Port, Treiber-Index etc…)	Ja □	Weiß nicht □	Nein □

Abbildung A.3.: Fragebogen der Requirements-Studie, Blatt 1

Was könnte man deiner Meinung nach bei der Einbindung des Moduls noch verbessern? Bzw. was würdest du dir wünschen:

_____

_____

_____

Fragen zur Nutzung des WFSStreamers über die Editoransicht:

Wie intuitiv findest du die Nutzung des Moduls über das Script im Unity Editor?	Intuitiv ☐	Weiß nicht ☐	Umständlich ☐
Hattest du Schwierigkeiten damit, die Unity-Objekte mit Schallquellen zu belegen?	Ja ☐	Weiß nicht ☐	Nein ☐

Was würdest du dir für den AudioController noch wünschen? Fallen dir noch essentielle Funktionen ein, welche du für deinen grundlegenden Workflow noch brauchen könntest?

_____

_____

_____

Fragen zur Nutzung des WFSStreamers über die C#-API:

Hast du bereits Ideen, wie du den WFSStreamer in deinem eigenen Scriptcode nutzen könntest, nachdem du die Aufgaben bearbeitet hast?	Ja ☐	Weiß nicht ☐	Nein ☐
Findest du die Steuerung, das Abspielen/Stoppen von Sounds, das übermitteln von Positionsdaten etc. über die Sound-Objekte (WFSSoundObject) leicht zugänglich und auch verständlich?	Ja ☐	Weiß nicht ☐	Nein ☐

Abbildung A.4.: Fragebogen der Requirements-Studie, Blatt 2

Was würdest du dir für die C#-API noch wünschen? Fallen dir noch essentielle Funktionen ein, welche du für deinen grundlegenden Workflow noch brauchen könntest?

Weitere Anmerkungen:

Danke für deine Zeit!

Abbildung A.5.: Fragebogen der Requirements-Studie, Blatt 3

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

Hamburg, 20. September 2018 Darjush Bahreini