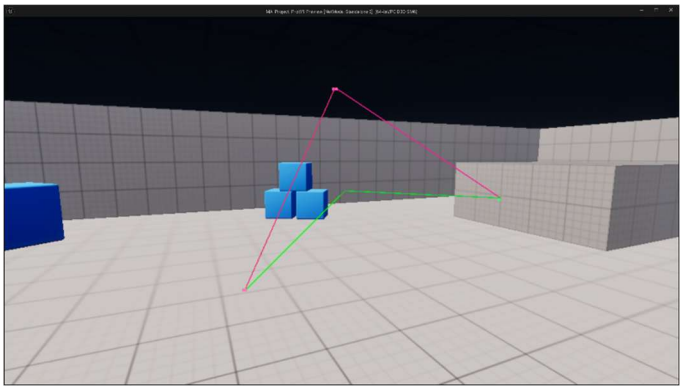
Die Nutzung von Konzepten der Optik in Sound Design für die Simulation von Raumhall in Games

Masterthesis in Sound Design

Cédric Schlegel



Zürcher Hochschule der Künste

Mentoriert von:
Dr. phil. Daniel Hug

Abgabedatum:
18.05.26

Cédric Schlegel
Sonnenbergstrasse 10
9524 Zuzwil
CH Schweiz
cedric.schlegel@bluewin.ch
+41 79 107 26 25

Abstract

Creating a room reverb for use in games is often a time-consuming process. Few attempts have been made to simplify the process and improve the quality. The solutions found so far are either still too time-consuming, too complex, or simply not available on the open market.
This thesis uses ray tracing, a concept from the field of optics, to develop a new workflow for creating a reverb. Ray tracing is typically used to simulate light, shadows and reflections. Here, it has been adapted to simulate an impulse response. The CeSoundTrace plugin developed as part of this thesis is designed to enable sound designers to spend less time on tedious configuration tasks and more time on the creative exploration of the virtual stage. To achieve this, the room reverb is simulated whilst the game is running. A range of parameters is provided for this purpose, allowing for creative intervention in the simulation. This results in a new workflow that is not only resource-efficient but also offers sufficient creative flexibility, rapid iteration and minimal configuration time. As this plugin is due to be released, this technology will become accessible to a wider user base and fills a gap in the market.
Possible applications for such a plugin include procedurally generated levels or dynamically changing geometry. The plugin makes it possible to create large numbers of realistic rooms easily, quickly and without hassle. This is particularly advantageous for large-scale games. Another possibility is to use the plugin for subjectivisation. Through simulation, it allows the essence of the space to be captured and, on this basis, an artistic interpretation to be created.
The plugin was developed for the Wwise audio middleware, which, in conjunction with the Unreal Engine game engine, simulates virtual room reverberation within the game. The objectives are to create an accurate room reverberation simulation and to provide parameters that can be used to manipulate the simulated impulse response. The plugin should also be easy to use, and the simulation should take place in real time wherever possible.
A video record of this project is available in the ZHdK media archive.

Zusammenfassung

Das Erstellen eines Raumhalles zur Verwendung in Games ist oft ein zeitaufwändiges Unterfangen. Es wurden wenige Versuche unternommen die Prozesse zu vereinfachen und die Qualität zu steigern. Die bisher gefundenen Lösungen sind entweder immer noch zu zeitaufwändig, komplex oder auf dem freien Markt schlichtweg nicht verfügbar.
In dieser Arbeit wird Raytracing, ein Konzept aus der Optik, verwendet, um einen neuen Workflow für die Erstellung eines Raumhalles zu entwickeln. Raytracing wird normalerweise eingesetzt, um Licht, Schatten und Reflexionen zu simulieren. Hier wird es für die Simulation einer Impulsantwort adaptiert. Das durch diese Thesis entstandene Plugin *CeSoundTrace* soll es den Sound Designern ermöglichen, sich weniger mit eintönigem Konfigurieren beschäftigen zu müssen und mehr mit der kreativen Auseinandersetzung der virtuellen Bühne. Um dies zu erreichen wird der Raumhall während der Ausführung des Games simuliert. Dazu werden eine Reihe an Parametern zur Verfügung gestellt, die ein kreatives Eingreifen in die Simulation erlauben. Daraus ergibt sich ein neuer Arbeitsablauf, welcher nicht nur Ressourcen schonend ist aber auch genügen kreative Flexibilität, schnelle Iteration und eine geringe Konfigurationszeit bietet. Da dieses Plugin veröffentlicht werden soll, wird diese Technologie einer breiteren Anwenderschaft zugänglich und schliesst eine Marktlücke.
Mögliche Anwendungen für ein solches Plugin sind prozedural generierte Levels oder sich dynamisch verändernde Geometrie. Das Plugin macht es möglich, einfach, schnell und unkompliziert grosse Mengen an realistischen Hallen zu erstellen. Dies ist besonders bei grossen Games von Vorteil. Eine weitere Möglichkeit ist die Anwendung des Plugins für Subjektivierung. Es ermöglicht mittels der Simulation, die Essenz des Raumes zu erhalten und auf dieser Basis eine künstlerische Verarbeitung durchzuführen.
Das Plugin wurde für die Audio-Middleware *Wwise* entwickelt, welche im Zusammenspiel mit der Game Engine *Unreal Engine* den virtuellen Raumhall im Game simuliert. Die Ziele sind die Erstellung einer korrekten Hallsimulation eines Raumes sowie die Bereitstellung von Parametern, welche die simulierte Impulsantwort manipulieren können. Ebenfalls soll das Plugin einfach zu nutzen sein und die Simulation möglichst in Echtzeit erfolgen.
Die Dokumentation dieses Projekts ist in Videoform im Medienarchiv der ZHdK verfügbar.

Danksagung

Ein solches Projekt ist nicht realisierbar ohne die Hilfe von weiteren Personen. Hier möchte ich denjenigen Danke, die mich auf meinem Weg begleitet und unterstützt haben.

Als erstes möchte ich meinem Mentor Dr. phil. Daniel Hug danken, der mich bereits seit meinem Bachelor-Abschlussprojekt begleitet und mentoriert. Dabei hat er mir immer ein Ohr geliehen und mich stets für meine weiteren Schritte beraten. Ich bin auch dankbar für die Unterstützung von Prof. Felix Baumann, der mich über meine gesamte Studienzeit begleitet hat. Durch ihn und sein Engagement ist diese Arbeit überhaupt erst möglich geworden.
Danke an Dr. Kurt Heutschi und Raphaël Jecker, die mich beide tatkräftig bei der Entwicklung mit ihrem Wissen unterstützt und mir alle Fragen mit viel Geduld beantwortet haben. Ohne euch wäre ich nicht so weit gekommen. Ebenfalls vielen Dank an Olav Lervik, welcher mein Interesse an Sound Design, DSP und Game Audio geweckt hat.
Ich danke auch all jenen Dozenten, die mich unterstütz, gefördert, inspiriert und mich durch die Studienzeit begleitet haben, auch wenn ich nicht alle mit Namen auflisten kann.
Ich möchte mich zudem bei meiner Familie bedanken, meinen Eltern und meiner Freundin, ohne deren Einfluss und stetige Unterstützung dieses Projekt nicht zustande gekommen wäre. Ohne Pause habt ihr mich getragen, hattet immer Zeit für meine Probleme und standet stets da, um mich aufzufangen, aufzurichten und in die richtige Richtung zu weisen.

Inhaltsverzeichnis

Notation und Symbole

$\boldsymbol{\alpha}$	Absorptionskoeffizient
$\boldsymbol{d}$	Reflexionskoeffizient
$\boldsymbol{\theta}$	Winkel zwischen Oberflächennormalen und Vektor
$\boldsymbol{\theta_i}/\boldsymbol{\theta_r}$	Einfallswinkel / Ausfallswinkel
$\boldsymbol{\phi}$	Der Azimut eines sphärischen Koordinatensystems
$\boldsymbol{\epsilon}$	Die Höhe eines sphärischen Koordinatensystems
$\boldsymbol{E}$	Energie eines Strahls
$\mathbf{a} \bmod \mathbf{b}$	Rest der Division von **a** durch **b**
$\mathbf{x} \sim \mathcal{U}(\boldsymbol{a} \dots \boldsymbol{b})$	Zufällige Zahl $\mathbf{x}$ im Bereich zwischen $\boldsymbol{a}$ und $\boldsymbol{b}$
$\boldsymbol{h^2}$	Energieimpulsantwort
$\boldsymbol{h}$	Schalldruckimpulsantwort
$\boldsymbol{Env}$	Hüllkurve
$\vec{\boldsymbol{v}}_{\boldsymbol{Norm}}$	Oberflächennormale
$\vec{\boldsymbol{a}}, \vec{\boldsymbol{b}}, \dots \vec{\boldsymbol{v}}$	3D Vektoren

Akronyme

CGI	Computer generated image
CPU	Central processing unit
FDL	Frequency-domain delay line
FDN	Filter delay network
FFT	Fast Fourier transform
FIR	Finite impulse response
GPU	Graphics processing unit
GUPOLS	Generalised uniformly-partitioned Overlap-Save
HRTF	Head-related transfer function
IDE	Integrated developer environment (VSCode, JetBrains CLion, …)
LFO	Low frequency oscillator
NPC	Non player character
NUPOLS	Non-uniformly partitioned Overlap-Save
SDK	software development kit
UE	Unreal Engine
UPOLS	Uniformly partitioned Overlap-Save
VR	Virtual reality
VS	Visual Studio

1 Einleitung

Diese Masterthesis dokumentiert die Erstellung des Plugins *CeSoundTrace* für Sound Design in 3D Games. Es simuliert den Raumhall in einem Game und gewährt dessen Manipulation für eine künstlerische Auseinandersetzung. Es wird beschrieben, wie sich das Plugin zusammensetzt, die einzelnen Systeme ineinandergreifen, begründet die verwendeten Techniken und wie es mit den Konzepten aus der Optik zusammenhängt.
Das Plugin wird für die Game Engine[1] *Unreal Engine*[2] (UE) und für die Audio-Middleware *Wwise*[3] entwickelt.

1.1 Hintergrund und Motivation

Die Faszination von Raytracing begann für mich schon bei der Ankündigung der RTX 20 GPU's von *Nvidia*[4], der ersten Raytracing-fähigen Serie. Die Kombination aus Technik und Games ist für mich eine perfekte Mischung an Themengebieten. Daher unternahm ich immer wieder Versuche, die technischen Verfahren dahinter zu verstehen. Mit dem wachsenden Verständnis folgten einige Überlegungen zur Applikation im Sound Design. Da es sich bei Licht sowie bei Schall um eine Welle handelt und beide ähnliche Prozesse durchlaufen, wenn sie sich in einem Raum ausbreiten, sollte es möglich sein, Raytracing auf die Simulierung einer Finite impulse response (FIR) zu übertragen.
In der Recherche zur Thematik wurde klar, dass Raytracing bei Programmen zur Simulation von Akustik in Gebäuden schon seit längerem zum Einsatz kommt. Allerdings arbeiten diese Programme offline[5]. Meine Absicht ist es, ein Plugin zu entwickeln, das einfach zu benutzen ist, einen hohen Grad an Detail sowie eine Möglichkeit für die Sound Designer bietet, kreativ zu arbeiten. Das Schlüsselelement dabei ist es, Arbeitszeit einzusparen, indem die Konfiguration vereinfacht und Serialisierung des Raumhalls möglich gemacht wird. Dazu muss das Raytracing in Echtzeit geschehen ansonsten wird die gesparte Konfigurationszeit durch Wartezeit ersetzt und damit entfällt eine Arbeitszeitersparnis.
Mit UE und *Wwise* entsteht eine geeignete Ausgangslage für diese Projekt. *Unreal Engine* bietet mit der visuellen Scriptingumgebung *Blueprint* eine Möglichkeit, Prototypen schnell und einfach zu entwickeln. Die starke Verknüpfung mit *Wwise* erlaubt es, die *Wwise SDK* zu

[1] Was eine Game Engine und speziell Unreal Engine ist wird im Kapitel 4: FIR Simulation in Unreal Engine erläutert
[2] Unreal Engine Produktseite (https://www.unrealengine.com/)
[3] Wwise Produktseite (https://www.audiokinetic.com/en/wwise/overview)
[4] Nvidia RTX 20-er Serie (https://www.nvidia.com/de-de/geforce/20-series/)
[5] Offline und Online Rendering wird im Kapitel 2.2.4: Beispiele aus der Architektur erläutert.

nutzten, welche feine Justierungen im DSP-Quellcode des Plugins zulässt und damit die Entwicklung einfacher gestaltet. Zudem erfreuen sich beide Softwares grosser Beliebtheit, was indes mehr potenzielle Nutzer für *CeSoundTrace* bedeutet.

1.2 Übersicht

Die Thesis beginnt mit einer Erläuterung zu der Bedeutung von Hall in Games im Kapitel 2.1: Hall in Games. Anschliessend werden einige Verfahren im Kapitel 2.2: Verfahren zur Hallerzeugung und Kapitel 2.3: Angewendete Verfahren zur Hallerzeugung vorgestellt, und wie diese konkret angewendet werden können. Darauf folgt im Kapitel 3: Überblick über CeSoundTrace eine Prozessmodellierung, welche das Plugin und dessen Ablauf beschreibt und einen Einblick in die Zusammensetzung der einzelnen Systeme zeigt.
Im Kapiteln 4: FIR Simulation in Unreal Engine wird das Tracing in *Unreal Engine* vorgestellt und wie dieses umgesetzt und gelöst wurde. Wie die simulierten FIR gefaltet und umgesetzt wurde, wird im Kapitel 5: Das Faltungshall-Plugin in Wwise vorgestellt.
Abschliessend findet in Kapitel 6: Ergebnis und Nutzungspotenzial ein Fazit statt und in Kapitel 7: Schlussfolgerung und Ausblick ein Ausblick, der meine Vision für die Zukunft dieses Plugins beschreibt.

2 Problemstellung und aktueller Stand der Technik

2.1 Hall in Games

Der Raumhall in einem Game gehört zu den wichtigsten Mitteln der Immersion. Egal ob dieser rein deklarativ, zum Beispiel zur Unterscheidung zwischen Innen- und Aussenräumen, als Subjektivierung, wenn der Spieler beispielsweise einen Schock erleidet, oder als Informationsträger, vorwiegend bei taktischen Shootern[6], eingesetzt wird, der Hall entscheidet über die Qualität dieser Momente. Dieser beschreibt dabei, meist im Unterbewusstsein des Spielendens, die Bühne der aktuellen Szene. Es würde eine Ablenkung entstehen, welche in der Regel nicht erwünscht ist, wenn sich zum Beispiel der Spieler in einer engen Kanalisation befindet, der Raum aber wie eine Kathedrale klingt. Dabei entsteht ein Gegensatz zwischen dem sichtbaren kleinen Raum und dem grossen hörbaren Raum.

Die Wichtigkeit von Hall existiert aber nicht nur in Games. Auch im Film ist der Hall essenziell, erfordert aber einen anderen Umgang. Da der Film linear ist, kann ein einmal angefertigter Hall wieder verwendet werden, sollte sich eine Szene im selben Raum abspielen. Er kann auch als Grundlage für ähnliche Räume verwendet werden. Zudem beschränken sich die Anzahl an verschiedenen Räumen in Filmen stärker.

Ganz anders ist das in Games. Angefangen bei der Quantität der Räume. Oft hat ein Rollenspiel eine geschätzte Spielzeit von 40 bis 60 Stunden, beim Film hingegen beträgt die Dauer im Schnitt nur zweieinhalb Stunden. Daher finden sich in einem Game mehr unterschiedliche Räume, welche es zu verhallen gilt. Dazu kommt, dass sich ein Spieler stetig weiterbewegt und kaum an den gleichen Ort zurückkehren wird. Dabei gibt es auch Ausnahmen, wie zum Beispiel Open World Games[7], welche oft über

Abbildung 1 Ein Beispiel eines Skilltrees aus dem Game "Mittelerde: Mordors Schatten"
Bildschirmaufnahme aus Mittelerde Mordors Schatten

[6] Zu Shootern gehören Games, bei welchen die Hauptmechanik das Schiessen mit einer Waffe ist. Taktik Shooter sind eine Unterkategorie davon, bei denen ein hohes Mass an Taktik zum Einsatz kommt. Beispiele sind: Counter Strike, Valorant, Rainbow Six Siege

[7] Open World Games sind Spiele, die typischerweise eine grosse und offene Spielwelt besitzen. Ein Merkmal dabei ist, dass dem Spieler nicht vorgegeben wird, wann und wo er hingeht, sondern er kann selbständig die Welt erkunden.

Hubs[8] [Abbildung 2] verfügen, zu welchen zurückgekehrt wird. Dafür sind sie um ein Vielfaches grösser als Rollen oder Story Games[9], wobei sich bei diesen die Quantität der Räume nochmals erhöht. Es lässt sich daher sagen, dass der Hauptaufwand für Hall in Games aus der Menge an verschiedenen Orten, Schauplätzen und Räumen stammt. Wenn Qualität und Quantität gefordert werden, gibt es auch kaum eine Möglichkeit, den Aufwand zu reduzieren, ohne an Qualität einzubüssen.

Abbildung 2 Ein Beispiel für einen Hub aus dem Game "Hardspace: Shipbreaker". Der Spieler wertet und repariert hier seine Werkzeuge auf.
Bildschirmaufnahme aus Hardspace Shipbreaker

2.1.1 Presets und ihre Verwendung

Ein Vorgehen, welches aus dem Film stammt, ist die Ver- und Wiederverwendung eines Presets[10] [Abbildung 3] des Halls. Für Games ist dies der konventionelle Arbeitsablauf, da er in seiner Ausführung simpel ist und auf Repetition der immer gleichen Schritte beruht. Da dies schnell zu aufwändig ist, um es innerhalb der Entwicklungszeit zu stemmen, werden Vereinfachungen angewendet. Generell gesagt, handelt es sich dabei immer um eine Verminderung der Qualität. Wie diese Verminderung aussieht, ist sehr situativ. Meist wird aber eine From der Generalisierung verwendet. Dies kann mit dem Wiederverwenden von Presets geschehen, welche nicht ganz zum Raum passen. Man erhöht dabei die Toleranz, mit welcher

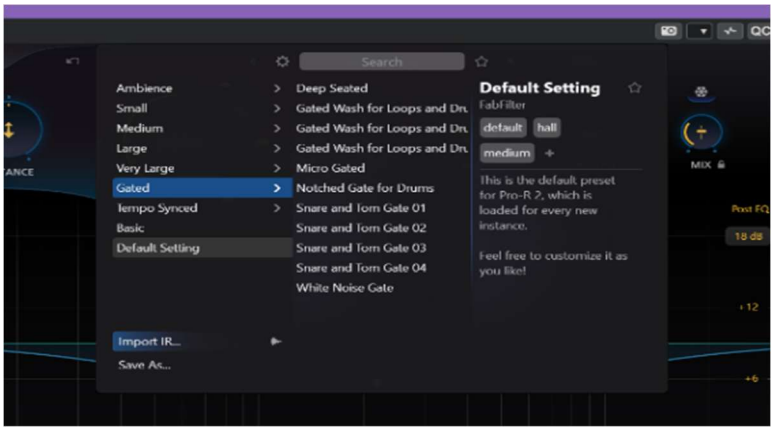


Abbildung 3 Eine Liste von Presets. Diese wurden vom Hersteller erstellt und in Kategorien eingeteilt. Dies dient der Inspiration und der Arbeitsgeschwindigkeit.
Bildschirmaufnahme von Fabfilter Pro-R2

[8] Als Hub wird in Games ein Ort bezeichnet, an den der Spieler immer wieder zurückkehren kann und verschieden Aktivitäten nachgeht. Meist beinhaltet dies die Weiterentwicklung des Charakters oder der Ausrüstung [Abbildung 1]. Diese Orte sind in Spielen oft auch immer friedlich.
[9] Unter Story Games versteht man Spiele, welche den Fokus auf die Narration setzen. Sie sind gerne linear, cineastisch und verfügen über klar definierte Anfänge und Enden. Oft werden solche Games auch in Kapitel oder Akte unterteilt.
[10] Das Preset, ein Schnappschuss aller Einstellungen eines Plugins, welcher gespeichert und wieder abgerufen werden kann.

entschieden wird, wann ein Raum ein eigenes Preset braucht. Es kann auch eine Technik zum Einsatz kommen, welche die Raumgrösse grob ermittelt und damit das Preset erstellt. Dabei werden aber oft Details wie verschieden Materialien und Möblierung ignoriert.
Will man eine hohe Immersion beziehungsweise eine hohe Qualität erreichen, wird der Hall zum Mammutprojekt der Eintönigkeit. Natürlich ist dies eine Frage des Projektrahmens. Doch wenn man davon ausgeht, dass ein qualitativ hochwertiges Produkt entstehen soll, von welchem eine hohe Verkaufszahl erwartet wird, reicht es nicht aus, drei Presets zu erstellen und zwischen diesen hin und her zu schalten. Hier wird von den Spielenden Liebe zum Detail erwartet und diese muss erfüllt werden. Bei einem solchen Projekt benötigt man viele hundert Iterationen von Hall Presets in hunderten von Räumen. Ergo, ein zu hoher Aufwand für zu wenig Entwicklungszeit.

2.1.2 Prozedurale Geometrie und dynamischer Hall in Satisfactory

Ein weiteres Szenario, in dem viel Aufwand für Hall benötigt wird, sind prozedural erstellte Games. In diesen wird in der Regel das Erstellen grosser Spielwelten mittels der prozeduralen Generation erleichtert. Bei der prozeduralen Levelgenerierung wird die Welt in einzelne Bauteile unterteilt. Das können regelmässige Stücke sein, beispielsweise Quadrate oder auch komplexere Geometrien. Diese Bauteile werden dann mit Regeln versehen. Diese Regeln sorgen dafür, dass gewisse Bauteile dann nur mit einem bestimmten anderen verbunden werden dürfen. Für die Erstellung der Spielwelt nimmt sich die Game Engine diese Bauteile und fügt sie so zusammen, dass für alle die jeweiligen Regeln erfüllt sind. Für das Sound Design besteht nun die Herausforderung, dass sich diese Welt jederzeit ändern kann, während der Entwicklung, aber auch wenn das Game gespielt wird, und keine Sounds fest im Level platziert werden können. Noch schwieriger wird nun der Hall. Wie schon beschrieben braucht dieser Prozess ohnehin viel Zeit und erfordert, dass sich die Räume nicht ändern. Bei prozeduralen Games ist dies nicht möglich, daher ergeben sich zwei Möglichkeiten: Erstens, man erstellt einfache Presets und ermittelt grob wie gross ein Raum ist, um zu entscheiden welches Preset verwendet wird. Dies geht in die Richtung von "one size fits all" oder zweitens, man braucht einen dynamischen Hall.
Ein ähnliches Problem haben Spiele, welche über eine feste Welt verfügen, die jedoch vom Spieler verändert werden kann. Ein Beispiel dafür ist *Satisfactory*[11], entwickelt von *Coffee Stain Studios*[12]. In diesem Spiel erstellt man auf einem fremden Planeten grosse Fabriken, um für

[11] Satisfactory Produktseite (https://www.satisfactorygame.com/)
[12] Coffe Stain Studios (https://coffeestain.com/)

seinen fiktiven Arbeitgeber *Ficsit Inc*. Bauteile zu produzieren. Dafür zapft man die lokalen Ressourcen des Planeten an und verarbeitet diese in immer grösser werdenden Produktionsanlagen [Abbildung 4]. Da diese komplett von den Spielern selbst erstellt werden, kann nicht vorhergesagt werden, wie gross oder klein die Fabrikhallen werden. Dies stellt eine Herausforderung für die Sound Designer dar. Zusätzlich zur Grösse kommen noch reflexive Oberflächen in Form der Maschinen und Förderbänder dazu. Abgesehen davon kann sich der Spieler auch frei in der Natur bewegen, welche ebenfalls eine Verhallung braucht.

Abbildung 4 Ein Screenshot aus dem Game "Satisfactory". Zu sehen ist das Innere einer vom Spieler erstellter Fabrik.
Bildschirmaufnahme aus dem Teaservideo zu Satisfactory Update 5

Die Sound Designer lösten das Problem mit einem Hall-Tool, welches über ein dynamisches und ein statisches System verfügt. Das dynamische System benutzt drei Raycasts, welche bestimmen, ob ein Spieler sich draussen oder in einer Halle aufhält. Befindet sich der Spieler in einer Halle, ermitteln diese Raycasts gleich auch ihre Höhe. Gleichzeitig werden einige Raycasts horizontal vom Spieler weggesendet. Diese ermitteln Breite und Länge der Umgebung. Da die Differenzierung von Drinnen und Draussen bereits Übernommen ist, spielt es für die horizontalen Raycast keine Rolle, ob sie auf einen künstlichen Innenraum oder vorgefertigte Levelgeometrie treffen. Damit kann dynamisch eine Grösse für den Raum ermittelt und damit die Nachhallzeit bestimmt werden.

Um den Immersionsgrad zu steigern, wird in Form des statischen Systems Rücksicht auf die Geometrie der Fabriken genommen. Dafür entnehmen die Sound Designer die Position aller Fundamente, Dächer und Wände in der Umgebung des Spielers. Mit diesen Informationen wird in *Wwise* die Geometrie der Fabrik nachgebaut und laufend angepasst. Die Geometrie in *Wwise* dient dazu, die Hallausbreitung zu simulieren, ähnlich wie bei einer Wellensimulation [Kapitel 2.2.2: Wellensimulation]. Doch im Unterschied zur Wellensimulation geschieht dies in einem kleineren Umkreis, anstatt der gesamten Levelgeometrie, und kann laufend angepasst werden (Coffee Stain Studios, 2025).

2.1.3 Parametrische Kontrolle und Subjektivierung einer Hallsimulation

Abbildung 5 Eine Szene aus "Max Payne". In dieser Szene steht der Hauptcharakter unter dem Einfluss einer Droge und nimmt deswegen alles verzerrt war. Ein Beispiel für die Subjektivierung. Bildschirmaufnahme aus Max Payne

Neben den dynamischen Hallsystemen gibt es auch die Möglichkeit, einen Hall zu simulieren. Eine Hallsimulation wird gerne vollständig mit einer Wellensimulation oder annäherungsweise mit Raytracing umgesetzt. Oft entstehen dabei realistische und detailreiche Abbildungen der Räume. Die Vor- und Nachteile dieser Systeme werden im Kapitel 2.3: Angewendete Verfahren zur Hallerzeugung diskutiert, die Erläuterungen der Technik im Kapitel 2.2: Verfahren zur Hallerzeugung. Mit einer Hallsimulation müsste eigentlich die höchstmögliche Immersion erreicht werden.

Doch die realistische Simulation von Hall ist nicht immer zielführend. Dies ist der Fall, wenn ein Hall auf das Spielgeschehen reagieren und dabei unrealistisch werden muss, um nicht die Immersion zu brechen. Dafür werden Parameter benötigt, die es dem Sound Designer ermöglichen, den Hall zu manipulieren. So kann trotz der Simulation eine Abstraktion und oder Subjektivierung stattfinden und dennoch Zeit gespart werden, indem eine Simulation benutzt wird.

Als subjektiver Hall wird ein Hall bezeichnet, der den geistigen Zustand des gespielten Charakters widerspiegelt. Klassische Beispiele aus Games sind Benommenheit oder Drogeneinfluss [Abbildung 5], bei denen das Audio stark verhallt wird, um eine Distanz zur Umgebung zu bekommen. Dies versucht, den realen Zustand, bei dem man die Umgebung und das Geschehen nicht mehr bewusst wahrnimmt, zu rekonstruieren. Der Hall nimmt dabei die Perspektive des Charakters ein und wird vom objektiven, meist Realismus fördernden Mittel, zum Subjektiven (Flueckiger, 2023).

Die Kombination aus Parameterset und Simulation dreht die konventionelle Hallerstellung um. Anders als beim algorithmischen Hall [Kapitel 2.2.1: Algorithmischer Hall], bei dem die Parameter den Hall steuern während dieser berechnet wird, kontrolliert das Parameterset die Simulation bevor diese stattfindet. So gesehen ist das Parameterset eine Art Schablone für die Simulation, in der diese stattfindet und jeder simulierte Hall, je nach Umgebung, ein bisschen anders klingt. Das Preset eines algorithmischen Halls hingegen, lässt weniger Spielraum für Interpretation des Halles und hat damit einen stärkeren Einfluss darauf, wie der Hall klingt.

Ein grosser Mehrwert der parametrisierten Simulation ist die Möglichkeit der Subjektivierung. Eine einfache Simulation eines Raumhalls würde nicht genügen. Sie erlaubt nur eine realistische Simulation oder eine aufwändige Konfiguration der Oberflächen, damit diese kein realistisches Abbild erzeugen. Die Simulation würde damit wieder in die Richtung "one size fits all" gehen.
Das kreative Arbeiten mit Hall sollte seinen Platz bekommen, da er ein grosser Teil der Narration ist. So kann er Gefühlsänderung des Hauptcharakters, beklemmende Situationen oder eine Beeinträchtigung der Sinne übermitteln. Daher muss dem Künster bei der Verwendung von CeSoundTrace eine Freiheit geboten werden, denn nicht immer ist Realismus gewünscht oder verlangt.

2.2 Verfahren zur Hallerzeugung

Die traditionelle und einfachste Methode ist es, mit einem Halleffekt zu arbeiten, wobei dessen Parameter den Gegebenheiten des Spiels angepasst werden. Bei diesem Effekt handelt es sich meist um einen "Random Hall" beziehungsweise einem "Algorithmischen Hall" [Abbildung 6].

Abbildung 6 Das UI eines algorithmischen Halls, hier zu sehen FabFilter Pro-R 2
Bildschirmaufnahme von FabFilter Pro-R2

2.2.1 Algorithmischer Hall

Ein algorithmischer Hall generiert diesen, anders als bei einem Faltungshall, rein mit dem Algorithmus auf der CPU, daher der Name. Dazu kommt meist eine Reihe aus Delays und Filtern zum Einsatz [Abbildung 7]. Der Hall entsteht dabei hauptsächlich aus den Delays. Ist die Delayzeit kürzer als ~30ms, nimmt man den Delay nicht mehr als solchen war, sondern so, als wären das originale und das verzögerte Signal, ein einziges Signal. Dabei kommt es zu einem sogenannten Chorus Effekt. Eine Gefahr

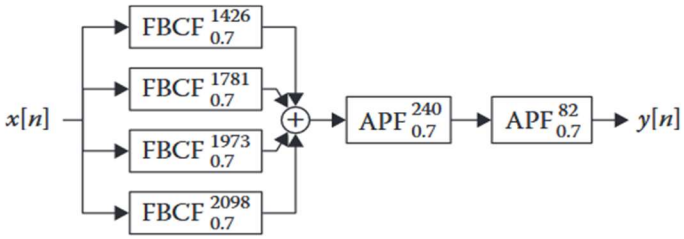


Abbildung 7 Ein Blockdiagramm eines algorithmischen Halls
Diagramm aus Hack Audio, Tarr

dabei ist, dass die Resonanzen in ihrer Lautstärke explodieren. Um dies zu verhindern, wird die Delayzeit mit einem Low frequency oscillator (LFO) leicht moduliert. Wird diese Modulierung aber zu gross, entsteht neben dem Hallsignal ebenfalls ein Chorus Effekt. Um dem Hall eine klangliche Färbung zu geben, kommen zum Beispiel *Allpass* Filter zum Einsatz.
Soll nun die Länge des Halls verändert werden, geschieht dies über den Feedback-Anteil. Je mehr des bereits verzögerten Signals wieder im Delay eingespeist wird, desto länger dauert es, bis dieser Signalteil ausklingt. Dadurch erhöht sich die Länge des Halles. Um die Grösse des Hallraums zu ändern, wird die Delayzeit angepasst. Kürzere Delayzeiten entsprechen einem kleineren Raum, grössere einem Grösseren.
Für die Integration der frühen Reflexionen kann eine *Tapped Delayline*[13] eingesetzt werden. Damit kann ein Hall erzeugt werden, welcher deutlich näher an einer gemessen Impulsantwort und damit realistischer ist. Das ganze System aus *Tapped Delaylines* für die frühen Reflexionen, den parallelen Feedback Delays und dem Allpassfilter ergibt einen algorithmischen Hall. Dieser kann mit der Nutzung eines *Filter Delay Networks* (FDN) [Abbildung 8] vereinfacht werden. Ein FDN besteht aus parallelen *Delaylines*, welche nicht nur ihren Output zurück in sich selbst einspeisen, sondern auch die Möglichkeit haben, diesen zu jeder anderen *Delayline* zu führen. Die Kontrolle darüber geschieht mit einem *Gain* Parameter auf jeder möglichen Verbindung. Dies lässt sich in einer Parametermatrix zusammenfassen. Um nun verschiedene Halle beziehungsweise verschiedene Algorithmen zu erstellen, muss nur noch diese Matrix kontrolliert werden.

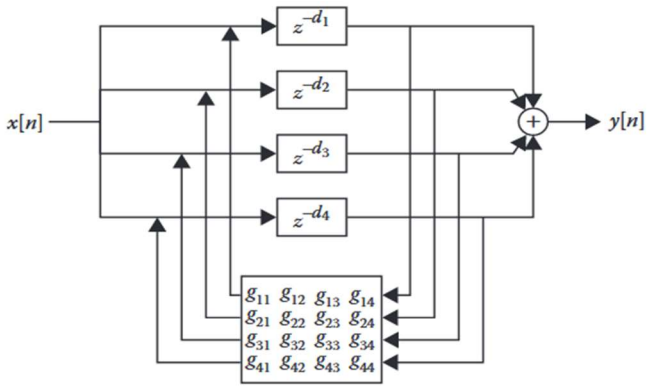


Abbildung 8 Das Blockdiagramm eines FDN. Unten ist die besagte Matrix an Parametern, mit der der Effekt gesteuert wird. Diagramm aus Hack Audio, Tarr

Der grosse Vorteil von algorithmischen Hallen ist die vergleichsweise geringe benötigte Rechenleistung. Daher wurden diese auch in einer Zeit populär, in der ein PC noch zu wenig Leistung für Faltungshalle hatte. Allerdings gibt man dabei auch Details auf und damit auch Realismus (Tarr, 2018).

[13] Als Tapped Delayline bezeichnet man in der Signalverarbeitung eine Verzögerung des Signals mit nur einem Speicher. Der Output wird dabei aus der Summierung verschiedener Taps, was Array Elemente des Speichers sind, in verschiedenen Amplituden berechnet.

Der algorithmische Hall bietet in Games durch seine Parameter grosse Vorteile. Es kann durch die Parameter definiert werden, wie sich der Hall an einer gewissen Stelle zu verhalten hat. Ist man zufrieden mit dem Hall, welcher für einen Raum erstellt wurde, kann man ein Preset der Parameter erstellen. Dies wird dann im Hintergrund in einer Datenbank abgelegt und kann zu jeder Zeit wieder aufgerufen werden. Hier trifft man auf einen weiteren, etwas unscheinbareren Vorteil des algorithmischen Halls. Das Preset kann in einer kleinen Datei gespeichert werden und benötigt nur minimalen Speicherplatz. Wer bereits mit einem Faltungshall gearbeitet hat, kennt die Grösse der Impulsantwort-Datenbank dieser Plugins, wie beispielsweise bei *Altiverb*[14] von *Audio Ease* oder *Chameleon*[15] von *Accentize* [Abbildung 9].

Es gibt aber auch Nachteile bei der Verwendung von algorithmischen Hallen. Ihre Parametrisierung lässt das Klangbild eines Raumes schnell generisch wirken. Dadurch, dass dasselbe Preset denselben Hall erzeugt, entsteht eine gewisse Sterilität. Dies sorgt dafür, dass eine Diskrepanz zwischen Raum und Raumgefühl entsteht. Als Beispiel befindet sich ein Spieler in einer grossen, alten und mehrstöckigen Bibliothek. Ein angenehmer Ort mit Bücherregalen aus Holz, Teppichboden und Kronleuchtern. Mit einem algorithmischen Hall wird es hier schwer sein, diese Wärme und Geborgenheit des Raumes zu vermitteln. Dies geschieht durch diesen künstlichen Nachgeschmack, die Diskrepanz zwischen Raum und Raumgefühl, welcher der Algorithmus hinterlässt. Zugegebenermassen wird es nicht so sein, dass der durchschnittliche Spieler dies kritisieren wird, dennoch wird es positiv auffallen, wenn eine Impulsantwort verwendet wird und nicht ein Algorithmus.

Abbildung 9 Screenshot des Programmes "Chameleon" von Accentize
Bildschirmaufnahme von Chameleon Accentize

[14] Altiverb Produktseite (https://www.audioease.com/altiverb/)
[15] Chameleon Produktseite (https://www.accentize.com/product/chameleon/?v=d88fc6edf21e)

2.2.2 Wellensimulation

Die Wellensimulation nimmt sich die Physik als Vorbild. Das Ziel hierbei ist es, das Verhalten einer Schallwelle möglichst physikalisch korrekt zu simulieren [Abbildung 10]. Der Vorteil dieser Methode ist klar die Qualität. Es ist das momentan genauste Modell, über welches verfügt werden kann. Der grosse Nachteil ist, dass diese Simulation sehr viel Rechenleistung benötigt. Eine Oberfläche kann für eine 125Hz Schallwelle spiegelnd, für eine 1kHz Welle jedoch möglicherweise diffus sein. Daher kann bei der Simulation nicht nur ein Frequenzgang simuliert und die Restlichen berechnet werden, sondern der Vorgang muss für jede simulierte Frequenz wiederholt werden. Dies ist mit einer begrenzten Rechenleistung eine enorme Herausforderung. Dementsprechend wird auch eine gewisse Dichte and Messpunkten benötigt und mit vielen Oberflächen skaliert sich die benötigte Rechenleistung schnell hoch.

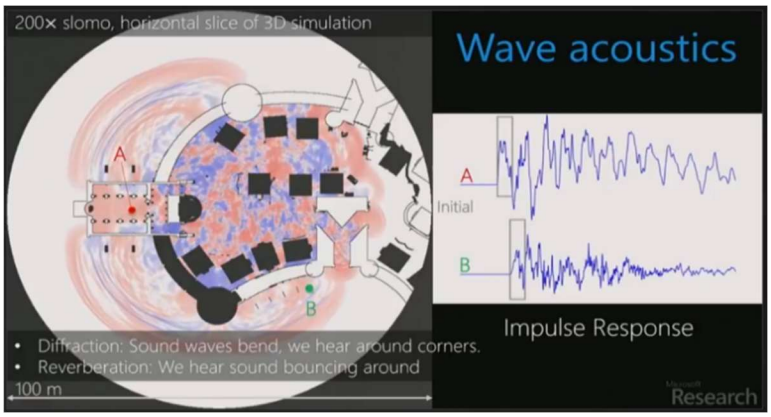


Abbildung 10 Bild einer Wellensimulation von Microsoft Project Triton. Simuliert ein Impuls, welcher durch die Levelgeometrie propagiert.

2.2.3 Raytracing

Raytracing ist ein Verfahren aus der Physik und wurde ursprünglich entwickelt, um eine korrekte Licht-Schatten Simulation zu ermöglichen. Der *Raycast*, ein einzelner Strahl, ist dabei keine Neuheit. Es wird schon länger genutzt, um alle möglichen Abfragen in einem Game zu machen, welches das Verhalten von Oberflächen erfordert. Neu ist die Menge an einzelnen Strahlen mit denen eine pixelgenaue Berechnung von Licht, Schatten und Reflexion möglich ist. Diese Technik hat den Game-Grafik-Markt revolutioniert und das Rennen um den besten Algorithmus und die beste Hardware ist immer noch voll im Gange.

Das Verfahren funktioniert so, dass ein Strahl aus der virtuellen Kamera durch ein Raster geschossen wird, welches so viele Kästchen wie benötigte Pixel enthält. Bei einem Aufprall auf einer Oberfläche wird die Grundfarbe des getroffenen Polygons ermittelt. Um nun herauszufinden wie schattiert die Farbe ist, wird getestet, ob eine direkte Beleuchtung besteht und wenn ja, welche Eigenschaften die Lichtquelle hat. Diese werden dann mit der Grundfarbe verrechnet. Besteht keine direkte Beleuchtung, wird der Strahl nach dem Ermitteln der Grundfarbe auf dem getroffenen Polygon reflektiert. Die Richtung der Reflexion wird anhand der

Rauigkeit der Textur ermittelt. Trifft dieser reflektierte Strahl nun wieder auf eine Oberfläche, wird erneut getestet, ob eine direkte Beleuchtung besteht. Falls dabei festgestellt wird, dass der zweite Strahl eine Oberfläche mit direkter Beleuchtung getroffen hat, so wird die Farbe und Schattierung dieses Polygons ermittelt und mit der ursprünglichen Grundfarbe verrechnet. So entsteht eine indirekte Beleuchtung unter Berücksichtigung der Farben rund um das primäre Polygon [Abbildung 11]. Dieser Prozess wird pro Pixel, welcher gerendert werden muss, mehrmals wiederholt, sodass viele Wege zwischen dem Polygon und den Lichtquellen gefunden werden. Dies ist der Grund warum die Technik manchmal auch als *Path Tracing* bezeichnet wird. Es entsteht damit ein realistisches Abbild der Szene, welche auch auf dynamische Lichtwechsel oder lichtreflektierende Objekte reagieren kann. Allerdings benötigt dieser hohe Detailgrad ebenso viel Rechenleistung (Branch Education, 2024).

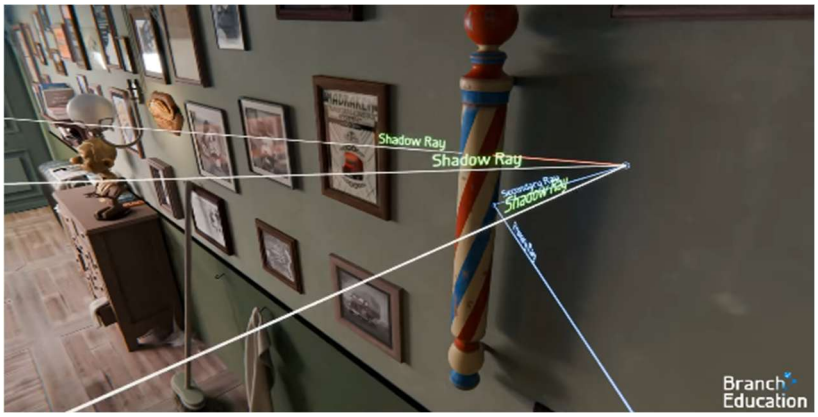


Abbildung 11 Eine Darstellung des Raytracing. Zu sehen ist, wie der initiale Strahl beim Aufschlagpunkt von den Sekundären beeinflusst wird. Bildschirmaufnahme aus dem Video How does Ray Tracing Work in Video Games and Movies? Branch Education

Aus Raytracing können gewisse Aspekte auch fürs Sound Design eingesetzt werden. Einer der grossen Vorteile von Raytracing ist, dass es auf dynamische Lichtänderungen reagieren kann. Dies gilt auch für einen Hall, welcher mit Raytracing erstellt wurde. Dabei reagiert der Hall nicht nur auf die Veränderung der eigenen Objektposition, sondern auch auf die Änderung der Raumgrösse. Dazu kann er auch auf Änderungen anderer Objekte im Raum reagieren, falls sich diese bewegen oder ganz verschwinden. Ähnlich wie optisches Raytracing die Farbe eines Polygons aufgrund von Oberflächenbeschaffenheiten bestimmt, kann es für einen Hall die Absorption- und Reflexionskoeffizienten [Kapitel 4.3.1: Absorption, Kapitel 4.3.2: Reflexion] bestimmen. Damit wird der Hall realistischer und erhöht den Detailgrad einer Raumsituation.

2.2.4 Beispiele aus der Architektur

Games und Filme sind nicht die einzigen Gebiete, in welchen der Hall eines Raumes eine grosse Rolle spielt. In der Architektur werden schon seit Längerem Programme, welche die Akustik eines Raums simulieren können, eingesetzt, um so nahe wie möglich an eine echte FIR zu gelangen. Das Ziel ist es, eine fundierte Beurteilung der Akustik in einem Raum zu erhalten, bevor dieser gebaut oder renoviert wird. Damit wird es möglich, bereits im Vorfeld stehende

Wellen oder lange Hallfahnen zu eliminieren, sollten diese nicht erwünscht sein. Zu dieser Kategorie gehören Programme wie: *Bose Auditioner System*, *CATT Acoustic room simulation*, *Timeware Realwave* und *ADA EASE* (Reilly & McGrath, 1995).
Diese Programme nutzen oft eine Kombination aus Wellensimulation und detailliertem Raytracing. Der grosse Unterschied zu Games ist dabei, dass diese Programme nicht Echtzeitfähig sein müssen und somit eine sehr genaue Berechnung der Akustik machen können.
Dies nennt sich *offline Rendering.* Gemeint ist damit, dass ein Effekt, hier der Hall, nicht in Echtzeit verfügbar sein muss und daher auch keine Synchronität zu einem etwaigen Medium benötigt. Im Gegenteil dazu steht das *online Rendering,* bei welchem Echtzeit und Synchronität verpflichtend sind. Oft gehen diese Begriffe auch mit einem Qualitätsunterschied einher. Wird keine Synchronität benötigt, besteht keine zeitliche Limite, in der die Berechnung erfolgt sein muss. Daher können die aufwändigsten Algorithmen verwendet werden, welche das genauste Resultat liefern. Bei einer Echtzeitberechnung hingegen, werden gerne Algorithmen eingesetzt, welche etwas ungenauer, dafür substanziell schneller, sind. Dies ist nötig, um synchron zu bleiben.
Für CeSoundTrace ist Synchronität wichtig, daher ist offline Rendering nicht erwünscht. Die hier erwähnten Programme sind für diese Thesis aber trotzdem von Bedeutung, da sie das Konzept Raytracing für Hall schon lange nutzen und damit die fundamentale Struktur des CeSoundTrace Konzepts bringen. Die Aufgabe für CeSoundTrace ist es nun, das Konzept dieser offline Programme in ein Onlinefähiges umzuwandeln.

2.3 Angewendete Verfahren zur Hallerzeugung

2.3.1 Rooms and Portals / Audio Volume

Aufbauend auf dem algorithmischen Hall [Kapitel 2.2.1: Algorithmischer Hall], kann natürlich mit mehreren Instanzen dessen gearbeitet werden, doch eine Behebung des Quantität-Problems ist auch damit kaum möglich. So kommen Systeme wie *Rooms and Protals*[16] von *Audiokinetic* oder die *Audio Volumes*[17] von *Unreal Engine* zu tragen. Beide Systeme verfolgen die Simulation von Hall basierend auf dem Raumvolumen. Damit kann die Nachhallzeit eines Raumes geschätzt werden. Allerdings werden dabei die akustischen Parameter der Oberflächen meist aussen vor gelassen, welche für den Charakter eines Raumes ausschlaggebend sind.
Dafür bieten sie die Möglichkeit, die Immersion zu steigern, indem der Hall eines benachbarten

[16] Rooms and Portals (https://www.audiokinetic.com/en/public-library/2025.1.7_9143/?source=SDK&id=using_rooms_and_portals.html)
[17] UE Audio Volumes (https://dev.epicgames.com/documentation/unreal-engine/audio-volumes-in-unreal-engine)

Raumes zu hören ist. Ein Beispiel dafür ist eine Tür. Ist die Tür vollständig geschlossen und der Raum damit zu, gelangt nur der Transmissionsschall von Ausserhalb zum Spieler. Wird diese Tür nun aber geöffnet, verändert sich die Grösse des Raums und damit auch der Hall. Anstatt des Transmissionsschalles, hört der Spieler nun den Direkthall des benachbarten Raumes, auch wenn er sich noch vor dessen Tür befindet. Damit verändert sich auch der Klang von Objekten wie zum Beispiel der Dialog zweier Non player characters (NPC's) oder ein Radio, welches sich in diesem Raum befinden.
Diese Art von dynamischer Veränderung der Raumgrösse stellt ein grosses Problem dar. *Rooms and Portals* versucht dieses zu lösen, indem ein Portal zwischen den Räumen erstellt wird, daher auch der Name des Systems. Diese Portale lassen sich aktivieren, deaktivieren, überblenden und lösen daher dieses Problem. In UE wird es pragmatischer gelöst. Sollte eine Verbindung zwischen zwei Räumen erstellt werden, wird zwischen den *Audio Volumes* der beiden Räume überblendet. Dieser Ansatz ist simpler und damit zeitsparender, jedoch muss dafür auf einen gewissen Detailgrad verzichtet werden.
Rooms and Portals hat hier noch ein Ass im Ärmel. Es verfügt über ein Geometriemodul[18], mit welchem Objekte im Raum vereinfacht dargestellt werden können. Diese Objekte werden dann dem Raummodul übergeben, welches die Geometrie in die Berechnung des Raumhalls einschliesst.
Im Vergleich zum reinen Algorithmischen Hall und seinen Presets sind diese Systeme bereits ein Fortschritt. Zum einen gelingt es damit, die Erstellung der Presets zu automatisieren, da nun das Raumvolumen der ausschlaggebende Parameter ist. Zum anderen wird ein weiterer Schritt in Richtung Realismus gemacht. Dies geschieht dadurch, dass mehr als nur der Raum hörbar ist, in dem sich der Spieler befindet. Beide Systeme simulieren auch den Transmissionsschall und die Verzerrung, welche stattfindet, wenn die Ausbreitung des Halls stark gebogen werden muss. Dies geschieht beispielsweise am Portal zum Raum. Der simulierte Hall wird dadurch einiges besser als ein rein algorithmischer. Trotzdem werden viele Details aussen vorgelassen, welche zu einer besseren Abbildung des Raums führen würden. Die Kontrolle darüber, wie das System die Simulation tätigt, ist ebenfalls limitiert. Dadurch wird es schwieriger, eine Subjektivierung in das System einzubauen.

[18] Wwise Geometry API (https://www.audiokinetic.com/en/public-library/2025.1.7_9143/?source=SDK&id=spatial_audio_apigeometry.html)

2.3.2 Project Acoustics

Das Forschungsprojekt von *Microsoft* mit dem Namen *Project Acoustics*[19] macht sich die Wellensimulation zunutze. Ziel ist es, eine ressourcenschonende Lösung für qualitativ hochwertigen Hall zu schaffen. Dazu wird die Simulation offline berechnet und während der Laufzeit müssen nur Daten in einer Tabelle ausgelesen werden. Umgesetzt wird dies mit einer aufwändigen dafür aber genauen Wellensimulation.

Das Projekt wurde in enger Zusammenarbeit mit Epic Games entwickelt und ist für UE-*Wwise* sowie für UE native Audio verfügbar.

Das Einrichten dieses Plugins wird mit einem *Enginetool*[20] leicht gemacht, welches vier einfache Schritte umfasst. Als erstes müssen alle für die Simulation verwendeten Geometrien mit einem *Tag*[21] versehen werden. Dies kann manuell gemacht oder über das *Enginetool* automatisch selektiert werden. Ebenfalls muss die Simulation wissen, in welchem Bereich sich ein Spieler bewegen kann. Dies kann wieder manuell geschehen oder über ein sogenanntes *NavMap*, ein Datentyp und Objekt von UE, das in der Regel für die Navigation von NPC's verwendet wird. Daraus ermittelt *Project Acoustics* die verwendeten Oberflächenmaterialien, welche in einem nächsten Schritt einem akustischen Material zugewiesen werden, welches die Absorptions- und Reflexionskoeffizienten des Materials festlegt. Mit diesen Daten kann nun *Project Acoustics* das Level vermessen und platziert Messproben. Auch hier gibt es wieder die Möglichkeit einzugreifen und manuelle Korrekturen vorzunehmen. Damit sind die Vorbereitungen abgeschlossen und es kann nun simuliert werden. Dies kann auf drei Arten passieren, mit *Microsoft Azure*, einem privaten *Cluster* oder lokal. Die Berechnung auf *Azure* oder einem *Cluster* ist die schnellste Wahl, da mittels Parallelisierung mehrere Messproben parallel berechnet werden können, während lokal alles in Serie geschieht. Am Ende dieses Prozesses erhält man ein einzelnes *Acoustics-File*, welches alle simulierten Daten enthält, die während der *Ausführung* ausgelesen werden können.

Um ein gewisses Mass an Subjektivierung zuzulassen gibt es noch weitere Parameter, mit welchen man Einfluss auf die Akustik nehmen kann ('Project Triton - Immersive Sound Propagation', n.d.; Unreal Engine, 2022).

Project Acoustics ist dank der Verwendung von Wellensimulation das wahrscheinlich

[19] Project Acoustics GitHub (https://github.com/microsoft/ProjectAcoustics)

[20] Als Engine Tool wird in Unreal Engine ein Tool bezeichnet, welches im Editor mit einem UI verfügbar ist. Es erleichtert das Arbeiten mit Plugins, da es nicht den teilweise etwas unübersichtlichen Outliner von Unreal Engine benötigt um Eigenschaften zu editieren.

[21] Ein Tag ist eine Beschriftung, welche nur innerhalb der Game Engine sichtbar ist. Damit lässt sich ein Objekt oder eine Gruppe von Objekten eindeutig bestimmen.

realistischste der hier vorgestellten Systeme. Dazu gehört auch die Berücksichtigung der verwendeten Materialien, welche in die Simulation miteingebunden werden. Es ist auch hervorzuheben, dass das System nicht komplett autonom arbeitet, sondern auf einigen Ebenen einen künstlerischen Eingriff erlauben. Dies geschieht auch durch eine leicht zu verstehende Bedienoberfläche, welche in UE integriert wurde. Dadurch geschieht ein grosser Teil automatisch, kann aber auf Wunsch entweder verbessert, beziehungsweise näher an den Realismus gebracht, oder verfremdet und subjektiviert werden. Der Nachteile dieses Systems ist klar der hohe Leistungsverbrauch, durch den die Technologie für viele Projekte ausser Reichweite gerät. Ebenfalls negativ ist die Statik dieses Systems. Einmal simuliert, muss der Prozess selbst für kleine Änderungen der Geometrie oder des künstlerischen Ausdrucks wiederholt werden, wodurch ein kreatives Ausprobieren schwierig wird.

2.3.3 Northlight Environmental Audio Tech

Northlight Environmental Audio Tech ist das von *Remedy Entertainment*[22] In-House entwickelte Umgebungsgeräusch-Tool. Es ist aus der Notwendigkeit für einen schnelleren Arbeitsablauf entstanden. Der Vorläufer basierte auf Daten statischer Geometrie, was zur Folge hat, dass bei der Erstellung des Halls nur langsam iteriert werden kann. Ebenfalls wurde, ähnlich wie in vorhergehenden Kapiteln bereits beschrieben, viel Konfigurationszeit benötigt.

Abbildung 12 Screenshot aus Northlight. Zu sehen sind die Parameter, welche durch das Raytracing ermittelt werden und die Verhältnisse von warm/kalt und gross/klein angeben.
Bildschirmaufnahme aus dem Video Wwise Tour 2018 Remedy Entertainment

Northlight Environmental Audio Tech, besteht aus mehreren Modulen. Eines für *Occlusion* und *Obstruction*[23], eines für Hall, eines für objektbasiertes Audio sowie ein Resonanzmodul. Im Rahmen dieser Thesis ist das Hallmodul das Spannendste. Dieses nutzt einen *Raycast* der Physik-Engine von *Northlight,* die Game Engine von *Remedy*, als Basis. Konkret bedeutet das,

[22] Remedy Entertainment Homepage (https://www.remedygames.com/)
[23] Occlusion und Obstruction sind Begriffe für das Verdecken von Klangquellen.

dass das Hallmodul Raytracing einsetzt, um eine ungefähre Grösse des Raumes und die Zusammensetzung des Halls zu ermitteln. Das Modul ermittelt bei Aufschlag auf einer Fläche dessen Oberfläche. Die Oberfläche wird zwischen kalt und warm eingestuft. Als Beispiel: Stein ist kalt und Holz warm. Aus allen Aufschlägen ergibt sich dann ein Verhältnis zwischen warmen und kalten Oberflächen, zu welchen zusätzlich die Raumgrösse einberechnet wird. Es gibt drei Grössen, klein mittel und gross, welche jeweils warm oder kalt sein können. Nun werden die Grössen und die Temperatur verrechnet, woraus eine Mischung von Hallen und Umgebungseffekten entsteht. Zum Beispiel: In einem mittelgrossen Raum, welcher mehrheitlich aus Stein- oder Betonwänden besteht, wobei eine Seite eine Holzverkleidung besitzt, werden folgende Werte ermittelt: warm/mittel 0.15, kalt/mittel 0.72 und kalt/gross 0.09 [Abbildung 12]. Damit weiss das System, dass die Klangschicht für kalte und mittelgrosse Räume am lautesten spielt und dazu die Schicht für warme und mittelgrosse Räume dezent dazu gemischt wird. Die Schicht für kalte und grosse Räume wird kaum hörbar als Akzent hinzugemischt. Daraus entsteht dann der Gesamtklang, eine Mischung aus allen detektierten Grössen und Temperaturen.

Da das System in Echtzeit arbeiten soll, kontrolliert *Remedy* die benötigte Performance. Dies geschieht über die Anzahl der Strahlen und ihre Aktualisierungsrate [Abbildung 13]. Die Strahlen und ihre Knotenpunkte werden im Uhrzeigersinn aktualisiert. Die Geschwindigkeit dafür kann der Situation angepasst werden (Audiokinetic, 2019).

Abbildung 13 Screenshot aus Northlight. Zu sehen sind die Strahlen des Raytracing Systems
Bildschirmaufnahme aus dem Video Wwise Tour 2018 Remedy Entertainment

Es ist klar, dass dieses System sehr spezifisch für *Remedy* und ihre Projekte entwickelt wurde. Trotzdem können einige Dinge festgestellt werden: Durch die Verwendung von Raytracing, ist das System sehr dynamisch. Um die Balance zwischen Rechenleistung und Qualität zu wahren, hat *Remedy* ein Oberflächensystem entwickelt, wodurch es möglich wird, realistischere Halle in Echtzeit zu simulieren. Insbesondere das warm/kalt- und gross/klein-System ist spannend, da damit ein grosses Spektrum an Hallräumen abgedeckt werden kann. Es ist aber unklar, inwieweit *Remedy* Einfluss auf die simulierten

Parameter nehmen kann. Ich würde jedoch behaupten, dass diese auch manipuliert werden können, wodurch es möglich sein sollte, eine Subjektivierung durchführen zu können.

2.3.4 Steam Audio

Steam Audio[24] ist eine von *Valve*[25] entwickelte Toolbox, welche neben Hall auch Schallausbreitung und HRTF-basiertes Hören ermöglicht. Die Toolbox fokussiert sich dabei in ihrer Anwendung auf VR-Applikationen. Der Hall kann für statische Geometrie vorsimuliert und gespeichert werden. Dies wird gemacht, um möglichst wenige Ressourcen zu nutzen. Man kann mit *Steam Audio* auch Hall in Echtzeit simulieren, damit der Hall auch auf dynamische Änderungen in der Geometrie reagieren kann.
Leider beschreibt *Valve* die Funktionsweise ihrer Technologie nicht sehr detailliert. Anhand der gefundenen Informationen vermute ich, dass es sich um eine Kombination aus Raytracing und Wellensimulation handelt. Die Wellensimulation wird für statische Geometrie verwendet, um möglichst viel Detailgrad zu erlangen und sich das offline Rendering zunutze zu machen und ein leistungsarmes Raytracing wird verwendet, um die dynamischen Änderungen zu inkludieren (Valve, n.d., 2017).

2.3.5 Snowdrop Engine

Das System von *Ubisoft Massive Entertainment*[26] für ihre hauseigene Game Engine *Snowdrop* nutzt Raytracing zur Simulierung von Hallausbreitung [Abbildung 14]. Das System dient zur Ablösung älterer Systeme, bei denen mehr manuelle Arbeit nötig war. Das aktuelle System hat aber jene nicht komplett abgelöst, da es nicht in allen Situationen gleich gut funktioniert. So wird das neue System hybrid eingesetzt. Die grossen Vorteile des neuen Systems sind, dass die Strahlen die Vegetation besser abbilden können und die daraus resultierende höhere Genauigkeit. Auch *Obstruction* und *Occlusion* können mit dem neuen System beinahe automatisch dargestellt werden.

Abbildung 14 Bild aus dem Artikel zum Raytracing in Snowdrop. Gezeigt wird, wie sich der Hall eines Wasserfalls ausbreitet. (Mårtensson, 2024)

[24] Steam Audio GitHub (https://valvesoftware.github.io/steam-audio/)
[25] Valve Homepage (https://www.valvesoftware.com/en/)
[26] Ubisoft Massive Entertainment Homepage (https://www.massive.se/)

Leider ist nicht viel mehr zu dem System bekannt, sodass eine tiefere Analyse unmöglich wird (Mårtensson, 2024).
Abschliessend lässt sich aber sagen, dass dieses System einen grossen Automationsteil verfügt. Mit den Raycasts kann eine nahezu echte FIR simuliert werden und man spart sich viel Zeit beim Konfigurieren.

2.4 Problematik der bisherigen Methoden

- Die in diesem Kapitel vorgestellten Methoden sind trotz der Absicht, die Arbeit zu erleichtern, immer noch zu komplex in der Bedienung oder zu aufwändig in der Anwendung. Vor allem in Anbetracht des Ertrags, kann es nach vielen Stunden des Konfigurierens zu einer Enttäuschung kommen, wenn der Hall nicht den Erwartungen entspricht beziehungsweise das System auch durch ein Einfacheres ersetzt werden hätte können. Zu komplexe Menus oder viele Objekte, welche in der Szene platziert werden müssen, resultieren in viel Arbeit oder grossen Debugging-Sessions weil der Fehler in den unzähligen Einstellungsmöglichkeiten nicht aufzufinden ist. Dieses Problem ist nicht unbekannt. Die meisten der vorgestellten Methoden wurden entwickelt um genau dieses zu lösen und um schneller arbeiten zu können.
- Eine weitere Problematik ist die Qualität des aus dem Verfahren gewonnenen Halls. Grundsätzlich gibt es zwei Ansätze, vorsimulierter Hall aus statischer Geometrie oder ein Raytracing für eine Ermittlung der Koeffizienten. Ein vorsimulierter Hall hat den Vorteil, dass dieser hochwertig sein kann, jedoch ist dafür viel Rechenleistung nötig, worauf viele Entwickler, gerade im Indie-Sektor, keinen Zugriff haben. Daher muss die Qualität beim Einsatz einer solchen Methode erniedrigt werden. Ebenfalls steigt die verlorene Zeit bei der Iteration, da bei jeder kleinsten Änderung an der Szene der ganze Hall neu simuliert werden muss. Die bisherigen Ansätze, welche Raytracing nutzen, verwenden es oft nur, um grob eine Grösse des Raums und ein paar Oberflächen zu ermitteln. Dies mag einigen genügen, doch scheint es so, dass diese Art von Hall nur bei speziellen Situationen funktionieren kann. Also eine sehr situative Lösung, welche als Mittel zum Zweck bezeichnet werden kann. Was fehlt, ist die Mitte zwischen Qualität und Quantität.
- Dazu kommt, dass einige der vorgestellten Methoden nicht für die Öffentlichkeit verfügbar sind. Daher können sie dann, obwohl sie gute, womöglich sogar die aktuell besten Ansätze liefern, nicht von Aussenstehenden genutzt werden. Das ist zum einen verständlich, da die Entwicklungszeit sehr viel Geld kostet, zum anderen ist es ein

Nachteil für alle anderen, vor allem kleineren Indie Studios. Oft ist es so, dass man bei kleineren Studios sehr viel mehr Freiheit fürs Sound Design bekommt, daher wäre das Potenzial für eine gute Hall Simulation gross, da sie das Bindeglied zwischen Kreativität und technischer Arbeit stellt.

- Einige der Methoden wie beispielsweise Project Acoustics wurden leider eingestellt oder werden wie bei Steam Audio, für welches seit 2024 kein Update mehr erschienen ist, nicht mehr aktualisiert. Dies ist aus mehreren Gründen ein Problem, zumal die Technik immer voranschreitet und diese Lösungen stehen bleiben. So wird es über die Zeit immer schwieriger, diese zu nutzen, da sie nicht mehr mit neuen Standards kompatibel sein werden und eine Emulation zu bauen, um sie am Laufen zu halten, wird immer aufwändiger. Ein weiteres Problem ist, dass diese Methoden auch nicht fertig entwickelt sind. Sei es die Steuerung oder das Einrichten, die Algorithmen oder die Effizienz. Sie haben viel Potenzial, welches nicht vollständig ausgeschöpft wird.

2.5 Zielsetzung

Ziel ist es, ein Plugin zu entwickeln welches eine Zeitersparnis beim Umgang mit Hall in Games bringt. Dafür soll es möglichst nahe an eine Echtzeitsimulation kommen. Dies hat die Absicht, dass keine externen Ressourcen wie Serverfarmen oder ähnliches benötigt werden, aber auch nicht zu viel Leistung vom ohnehin knappen Leistungsbudget der lokalen Maschine verwendet wird. Die Spezifizierung mit "möglichst nahe" kommt daher, dass eine tatsächliche Echtzeitsimulation den Rahmen dieser Thesis überschreiten würde. Daher ist dieses Plugin auch als Prototyp und Fundament weiterführender Entwicklung zu verstehen.
Im Rahmen dieser Thesis wurden ebenfalls auf Themen wie, Binauralisierung, Occlusion und Obstruction und die Simulierung von Hall für andere Klangquellen, welche nicht zum Spieler gehören, verzichtet.
Ich beabsichtige, dass durch bereitgestellte Parameter die simulierte FIR zu künstlerischen Zwecken manipuliert werden kann. So soll es möglich sein, dass sich ein Sound Designer kreativ ausleben und das Plugin auch entsprechend der Wünsche der Immersion gestaltet werden kann. Zusätzlich zu den künstlerischen Parametern, sind technische Parameter verfügbar, welche es ermöglichen das Leistungsziel des Plugins anzupassen. So wird es möglich, dass die Balance zwischen Qualität und Quantität gezielt gewählt werden kann.
Die Nutzung von Raytracing macht es möglich, dass das Plugin auf Änderungen in der Levelgeometrie reagiert. Damit wird die Iterationszeit kurzgehalten. Ebenso ermöglicht dies eine einfache Konfigurierung, da nur die Raytracing-Komponente dem Spieler-Charakter hinzugefügt

werden muss. Weitere Konfigurationen sind im *Wwise* Plugin möglich. Diese wurden ähnlich wie bestehende Hall-Plugins erstellt, damit keine Eingewöhnungszeit nötig ist.
Diese technische Grundlagenarbeit soll es ermöglichen, einen neuen Blick auf den Einsatz von Hall in Games zu bekommen. Anstelle von Parametern, welche den Hall direkt steuern, wird hierbei nur eine Parametervorlage erstellt, welche dann auf den "Raum" angewendet wird. Es wird dabei auch möglich, ohne die Manipulation der Parameter einen Raumhall zu erzeugen und führt damit zur gewünschten Zeitersparnis. Die gewonnene Zeit kann dann in die Parametervorlagen investiert werden, um die Immersion zu erhöhen.
CeSoundTrace wird am Ende der Entwicklungszeit über GitHub verfügbar sein.

3 Überblick über CeSoundTrace

CeSoundTrace besteht aus zwei grossen Teilen: einem Faltungshall für *Wwise* und einer Strahlenkanone für *Unreal Engine* [Abbildung 15]. In *Unreal Engine* startet der Ablauf mit dem Aussenden der Strahlen. Dies geschieht automatisch, sobald sich der Spieler eine bestimmte Distanz vom letzten Abschusspunkt entfernt hat. Die Kanone sendet eine festgelegte Anzahl an Strahlen und Reflexionen. Zum Beispiel hundert Strahlen mit einer Reflexion, dann hundert mit zwei Reflexionen, dann mit drei. Dies wiederholt sich, bis die festgelegte Obergrenze an Reflexionen erreicht wird [Kapitel 4.2 Strahlenkanone] und geschieht für jede der acht ausgewählten Frequenzen. Die Frequenzen ergeben sich aus dem Umfang des menschlichen Gehörs. Dessen Umfang kann ungefähr in zehn gleich grosse Oktavbänder unterteilt werden. Die hier verwendeten Frequenzen sind die jeweiligen Mittelfrequenzen dieser Oktavbänder, wobei das tiefste und höchste ignoriert wird, da sie zur Simulation keinen grossen Mehrwert beitragen. Bei jedem Aufschlag eines Strahls wird das *Physical Material* [Kapitel 4.3: Eigenschaften der Oberflächen auslesen] der getroffenen Oberfläche ermittelt. Durch das Material werden der Absorptions- und Reflexionskoeffizient in einer Tabelle ausgelesen. Mit den Koeffizienten wird ermittelt, wieviel Energie dem Strahl entzogen und in welche Richtung der Strahl reflektiert wird [Kapitel 4.3.1: Absorption, Kapitel 4.3.2: Reflexion]. Sinkt dabei die Energie auf null, heisst das, dass der Strahl vollständig von der Oberfläche absorbiert wurde. Der aktuelle Vorgang wird abgebrochen und es wird mit dem nächsten Strahl weitergemacht. Trifft der Strahl aber die Detektorkugel, welche sich am Standort des Spielers befindet, wird ein Eintrag in die Energieimpulsantwort gemacht. Dabei ergibt die Flugzeit des Strahls den Index, an welchem der Eintrag in die Energieimpulsantwort gemacht wird. Die verbleibende Energie wird mit der

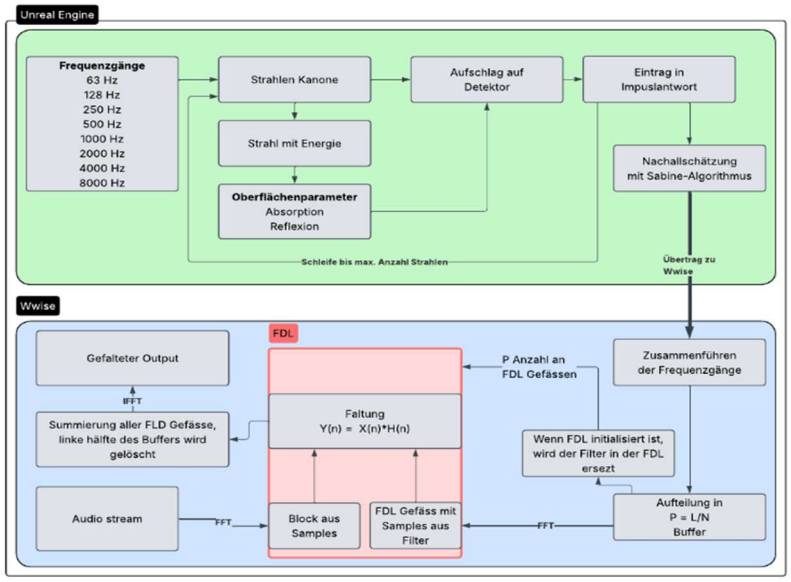


Abbildung 15 Prozessablauf von CeSoundTrace

Dämpfung der Luft verrechnet und am ermittelten Index eingetragen. Wurden alle Strahlen für alle Frequenzen abgehandelt, wird eine Nachhallschätzung mithilfe des Sabine-Algorithmus (‘Nachhallzeit’, 2026) [Kapitel 4.5: RT60 nach Sabine] gemacht. Alle Impulsantworten und die RT60 Zeit werden dann an *Wwise* übergeben [Kapitel 4.6: Datenbrücke zu Wwise].

In *Wwise* werden die FIR’s aller Frequenzen zu einer FIR zusammengefügt und von einer Energieimpulsantwort in eine Schalldruckantwort umgewandelt [Kapitel 5.3.1.1: Zusammenführen der Impulsantworten]. Dann wird die FIR partitioniert und in die Frequency-Domain delay line (FDL) geladen [Kapitel 5.3.1.2: Partitionieren, FDL und Faltung]. Jede Partition ist dabei ein Container der FDL. Befindet sich bereits eine FIR in der FDL so wird die alte FIR einfach mit der neuen ausgetauscht und die Länge der FDL angepasst. Parallel dazu wird der Input in das Effektplugin kumuliert, bis es genügend Samples hat, die der FDL übergeben werden können. Jeder Container mit Inputsamples wird mit einem Container der FDL gefaltet und in einen Platzhalter geschrieben. Anschliessend wird der Container mit den Inputsamples eine Position in der FLD weitergeschoben. Die Platzhalter aller FDL Container werden nun aufsummiert und gekürzt, um Zeit-Aliasing vorzubeugen. Dieser resultierende Container wird dann dem Output übergeben [Kapitel 5.3.2: Signal].
Um die Faltung effizienter zu machen, findet sie im Frequenzspektrum statt. Dafür werden alle Container der FIR und alle Container des Inputs, bevor sie der FDL übergeben werden, mit einem FFT Algorithmus vom Zeit- ins Frequenzspektrum transformiert.

4 FIR Simulation in Unreal Engine

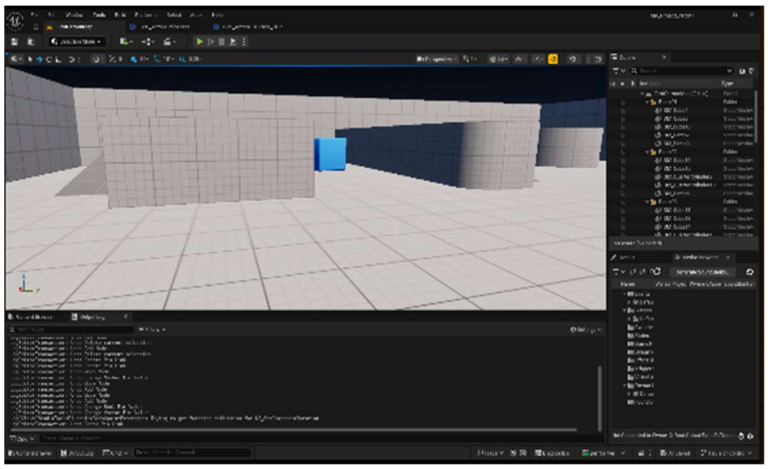


Abbildung 16 Screenshot vom Unreal Engine UI

Die *Unreal Engine* [Abbildung 16] gehört zu den führenden Game Engines und wird von *Epic Games* entwickelt. Die Engine ist für mehr als nur die Erstellung von 2D und 3D Games verwendbar. Sie wird auch in der Film- und der TV-Industrie für CGI bis hin zur Erstellung ganzer Animationsfilme genutzt. Weitere Anwendungsgebiete sind Architektur, Simulation und die Automobilindustrie.

Eine Game Engine hat in der Spieleentwicklung den Auftrag, eine fundamentale Entwicklungsumgebung zu stellen. Dazu gehören verschiedene Editoren für Texturen, Meshes und Animation, Simulationen für Licht, Schatten und Physik, Distributionstools und im Falle von UE, eine visuelle Programmierumgebung. Da diese Tools in fast jeder Game-Entwicklung verwendet werden und sie aufwändig zu konfigurieren sind, eröffnete sich hier bezüglich der Zeitersparnis eine Marktlücke. Heute gehören *Unreal Engine, Unity*[27]*, Cry Engine*[28] und *Godot*[29] zu den prominentesten öffentlich verfügbaren Engines. Im Gegensatz dazu gibt es Studios, oft grössere, welche eigene, der Öffentlichkeit unzugängliche Engines entwickeln und verwenden. Eine Game Engine ist also in ihrem Kern ebenfalls ein Tool, welches die Absicht hat, aufwändige Konfigurationen zu umgehen, Zeit einzusparen und so mehr Zeit für den kreativen Prozess zu ermöglichen.

Die Gründe, weshalb *Unreal Engine* für dieses Projekt ausgewählt wurde, sind folgende: *Unreal* bietet mit *Blueprint*, der visuellen Programmierumgebung, eine einfache Methode, in kurzer Zeit einen Prototyp eines Systems zu erstellen. Es lässt dabei auch offen, den Prototyp anschliessend von *Blueprint* in natives C++ zu übersetzen. Da C++ eine feinere Kontrolle über den Code ermöglicht und besser optimiert werden kann, bietet UE hier die nötige Flexibilität. Aufbauend darauf ist die Umsetzung eines Raycasts für dieses Projekt wichtig. Mit der LineTraceByChannel Funktion [Abbildung 17] bietet *Unreal Engine* bereits die ideale Basis. In

[27] Unity Produktseite (https://unity.com/de)
[28] Cry Engine Produktseite (https://www.cryengine.com/)
[29] Godot Produktseite (https://godotengine.org/)

Unity gäbe es ebenfalls eine Möglichkeit, mit einem Raycast zu arbeiten, allerdings ist diese sehr viel umständlicher.

4.1 Machbarkeitsnachweis

Um zu evaluieren, ob ein Echtzeit Raytracing System überhaupt umsetzbar ist, wurde eine kurze Machbarkeitsstudie vollzogen. Wie im Kapitel 4: FIR Simulation in Unreal Engine erwähnt, eignet sich dafür *Blueprint* und die LineTraceByChannel Node[30] [Abbildung 17].

Abbildung 17 Die Blueprint Node LineTraceByChannel. Die Node besitzt auf der linken Seite verschieden Eingänge für Variablen, einige davon sind verpflichtend, wie Start und Ende andere dienen nur der Konfiguration, wie die Farbe. Auf der rechten Seite befinden sich die Rückgabe werte, das Struct mit dem blauen Pin und ein boolescher Wert in Rot.

In diesem Nachweis wurde getestet, ob es möglich ist, einen Raycast zu senden und an dessen Aufprallort einen neuen Raycast zu starten. Dazu wurde ein einfaches System erstellt, welches einen Strahl in Blickrichtung der Kamera und aus der Mitte des Viewports nach vorne schiesst. Der Aufschlagpunkt stellt dann den neuen Startpunkt für den nächsten Strahl dar. Mit dieser Methode schlägt der Strahl, vom Spieler ausgehend, viermal auf einer Oberfläche auf und wird davon reflektiert, bevor dieser zurück zum Spieler gesendet wird. Vier Reflexionen wurden für den Nachweis als statische Anzahl an Reflexionen gewählt, um im Verhältnis zwischen Ziel und Komplexität zu bleiben. Die Iteration über eine bestimmte Anzahl an Reflexionen wurde erst in der Weiterentwicklung hinzugefügt. Damit hat der Nachweis das erste Hindernis überwunden. Allerdings wurde hier bereits vermerkt, dass das System umgebaut werden muss, da vier einzelne LineTraceByChannel Nodes genutzt wurden und dies nicht skalierbar ist.

Das grosse Problem des Machbarkeitsnachweises wurde anschliessend angegangen. Zu jenem Zeitpunkt wurde ein einzelner Strahl entlang der Blickrichtung der Kamera gesendet. Für den Machbarkeitsnachweis war dies nicht sonderlich repräsentativ, da es möglich sein musste, ein Trace in alle Richtungen zu senden. Um dies zu testen und nachzuweisen, war das Ziel, vier dieser Strahlen vom Spieler aus gesehen in die vier Himmelsrichtungen vorne, rechts, hinten und links, zu senden. Im damit verbundenen Drehen von dreidimensionalen Koordinaten

[30] Eine Node bezeichnet eine Funktion in Blueprint

versteckte sich die Schwierigkeit. Das bisherige System, welches einen einzelnen Strahl nutzte, funktionierte so, dass aus dem Standort der Kamera die Vorwärtsrichtung ermittelt wurde. Daraus erhielt man einen Einheitsvektor, welcher auf den Punkt zeigte, der in der Vorwärtsrichtung liegt. Dieser Punkt wurde um 10'000 Einheiten[31] nach vorne verlegt. Der daraus resultierende Punkt musste nun um 90°, mal dem Index des Ursprungsstrahls, gedreht werden. Dafür wurde die RotateVectorAroundAxis Node verwendet. Diese Node dreht den Vorwärtsvektor um die angegebene Gradzahl um die Z-Achse.

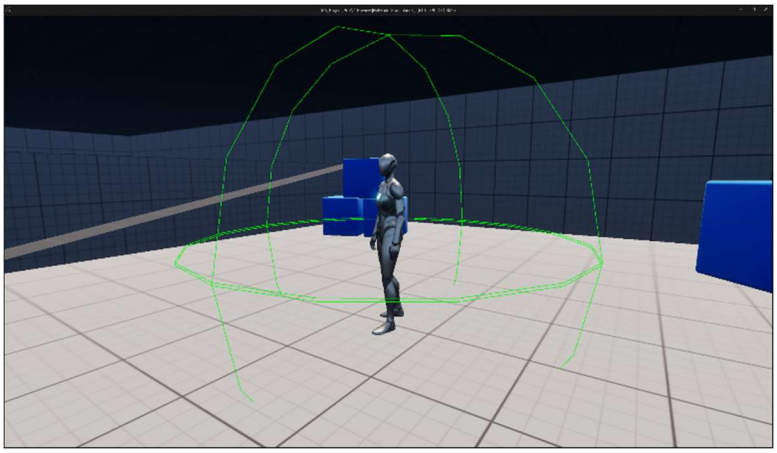


Abbildung 18 Bild einer Kugeltraces, in grün, in Unreal Engine

Das letzte Puzzlestück des Nachweises war die Aktualisierungsrate des Tracings. Die einfachste Methode wäre die Durchführung des Tracings per Intervall. Da dies aber sehr viel Leistung verschwenden würde, insbesondere sobald der Spieler stehen bleibt, musste eine andere Lösung gefunden werden. Die Methode für welche sich entschieden wurde, prüft, wo bereits ein Tracing stattgefunden hat. Dafür hinterlässt das Trace eine Kugel mit einem Tag, welche von einem, im Intervall gefeuerten Kugeltrace[32] [Abbildung 18], einer SphereTraceByChannel Node, detektiert wird. Wenn nichts detektiert wurde, hatte der Spieler genug Abstand zum letzten Trace, sodass ein neuer gemacht werden kann. Dieses System lässt die Möglichkeit offen, für den Fall, dass der Spieler sich erneut am selben Ort befindet, mit den getaggten Kugeln, Traces wiederverwenden zu können,. Dafür müsste das Trace abgespeichert und wieder aufgerufen werden können. Da dies aber für den Nachweis keine Priorität darstellte, sollte die Umsetzung nur im Falle restlich verfügbarer Zeit in Angriff genommen werden.

Der Machbarkeitsnachweis erzeugt keinen Ton. Dies ist für den Nachweis nicht relevant, da die Tonerzeugung selbst aus bereits etablierten Systemen besteht. Für das Tracing hingegen sind kaum Literatur oder Projekte vorhanden, welche als Vorlage dienen könnten. Daher war ein

[31] Einheiten meint hier eine ganzzahlige Veränderung im dreidimensionalen Koordinatensystem von Unreal. Diese Einheit ist üblicherweise äquivalent zu 1cm kann aber vom Entwickler angepasst werden.
[32] Anders als das bisher verwendete Trace, welches eine Linie zwischen zwei Punkten erstellt, erzeugt das Kugeltrace ein sphärisches Netz aus Traces mit einem bestimmten Radius um einen definierten Punkt.

Nachweis für das Tracing nötig, nicht aber für die eigentliche Simulation der FIR und der Faltung.

4.2 Strahlenkanone

Die Überarbeitung des Machbarkeitsnachweis zu einem Prototyp startete mit der Strahlenkanone. Wie erwähnt, bringt diese einige Probleme mit sich. Als erstes wurde die Serialisierung des Strahlensendens in Angriff genommen. In einem ersten Versuch gelang es, die Verwendung von einer Node pro Strahl auf insgesamt zwei zu reduzieren. Die erste für den initialen Strahl und die zweite für alle weiteren. Dies reichte aber nicht ganz aus. Um eine Serialisierung zu erreichen, muss die Kanone auf eine Node reduziert werden können. Das Problem ist, dass *Blueprint* nicht wie konventioneller Code seriell, sondern parallel zu arbeiten scheint. Das heisst, bei der Verwendung einer Node ist für *Blueprint* nicht klar welchen Start- und Endpunkt es für den Strahl nutzen soll. Die Konfusion entsteht daraus, dass diese Punkte nicht klar definiert sind, sondern als Referenz an die LineTraceByChannel Node übergeben werden. *Blueprint* benötigt statische Werte, da aber der Wert der Referenz sich während der Ausführung ändert, entsteht hier ein Problem. Die Lösung ist daher naheliegend. Der Wert muss statisch gehalten werden. Dafür muss die Koordinate, an welcher das Trace das Objekt oder die Oberfläche getroffen hat, in einer Variable gespeichert werden, so dass bei erneutem Aufrufen der LineTraceByChannel Node Start und Endpunkt als statische Referenzen mitgegeben werden können.

Der Startpunkt eines jedes Traces ist die aktuelle Spielerposition. Doch wie bekommt man nach dem ersten Aufschlag den nächsten Zielpunkt? Die Strahlen interagieren mit der Oberfläche, auf welcher sie aufschlagen, und erhalten daraus ihre neue Richtung. Das korrekte Ermitteln der neuen Richtung wird in einem weiteren Schritt gelöst. Zuerst steht die grundlegende Funktionalität im Vordergrund. Es wird dafür angenommen, dass alle Oberflächen für alle Frequenzen spiegelnd sind. Das bedeutet für den Strahl, dass der Aufprallwinkel zur Oberflächennormalen gleich gross ist, wie der ausgehende Reflexionswinkel. Hier zeigt sich wieder der Vorteil von *Blueprint*. Statt dass die Berechnung manuell erfolgt und implementiert werden muss, besteht dafür bereits eine Node und kann verwendet werden. Damit wird der reflektierende Strahl berechnet und, ähnlich wie der initiale Strahl, um 10‘000 Einheiten skaliert. Damit ist es nun möglich eine beliebige Anzahl an Strahlen mit einer variablen Anzahl an Reflexionen zu senden.

Das zweite grosse Problem, welches im Nachweis bereits angesprochen wurde, ist das Senden der initialen Strahlen in die richtige Richtung.

4.2.1 Initiale Richtung der Strahlen

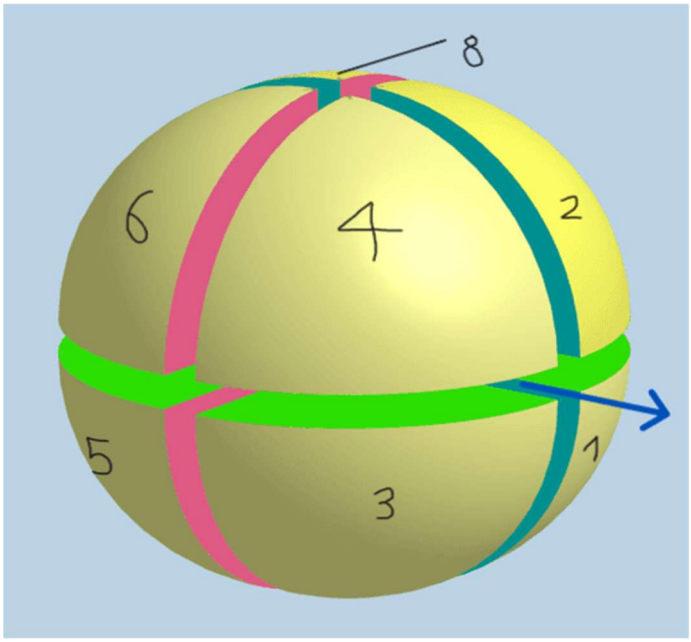


Abbildung 19 Die Gedankenstütze. Eine Kugel aufgeteilt in acht gleich grosse Sektoren. Die Nummern bezeichnen die Reihenfolge der Sektoren, der blaue Pfeil weist in Vorwärtsrichtung.
Bildquelle (https://www.geogebra.org/m/geteeht7)

Da immer noch die Serialisierung im Vordergrund steht, muss ein neues System zur Ausrichtung der initialen Strahlen entwickelt werden. Es reicht nicht mehr aus, Strahlen aufgrund festgelegter Winkel zu drehen. Es wurde eine Node gefunden, welche dieses Problem vereinfacht: GetVectorFromAzimutAndElevation. Diese Node nimmt zwei Fliesskommazahlen-Variablen als Parameter. Die erste ist für den Azimut und die zweite für die Höhe. Diese beiden Werte könnten zufällig generiert werden, jedoch sorgt dies aus statistischen Gründen für eine Einschränkung des Systems. Die Strahlenkanone sollte aber jederzeit abbrechen können und trotzdem ein ausreichendes Ergebnis produzieren. Dafür muss gewährleistet sein, dass die Strahlen den zu simulierenden Raum gleichmässig abgedeckt haben. Dafür wurde eine Gedankenstützte entwickelt. Wenn man sich alle möglichen Winkel vorstellt, in die ein Strahl ausgehend vom Ursprung des Spielers, nach aussen, mit einer Länge von einer Einheit nehmen kann, erhält man eine Kugel mit Radius $\mathbf{r} = \mathbf{1}$. Diese wird nun in acht Sektoren, unterteilt. Damit lässt sich nun zwischen oben und unten in dieser Kugel unterscheiden [Abbildung 19]. Das System beginnt nun beim ersten Sektor und geht dann zum nächsten weiter, bis alle acht abgehandelt wurde. Dann beginnt das System wieder von vorne mit dem ersten Sektor [Abbildung 19]. Um zu bestimmen, in welchen Sektor ein Strahl gehört, wird der ganzzahlige Rest von dessen Index ermittelt. Dies ergibt den Index des Sektors, wobei die Indexierung wieder vorne unten links beginnt, wie bereits in der Gedankenstütze beschrieben. Damit ist es möglich, zu einem beliebigen Zeitpunkt abzubrechen und eine statistisch ausreichende FIR zu erhalten.

Dieses System ist allerdings etwas unelegant und kann vereinfacht werden. Anstatt die Kugel in acht Sektoren zu teilen, nimmt man sich eine der beiden Halbkugeln, und teilt diese in eine beliebige Anzahl Scheiben auf. Die Oberflächen dieser Scheiben sind alle gleich gross. Die Scheibe gibt ein Minimum und Maximum für den Winkel der Höhe vor. Der Azimut $\boldsymbol{\phi}$ wird

zufällig generiert ebenso wie die Höhe ϵ zwischen dem Minimum und Maximum der jeweiligen Scheibe. Daraus ergibt sich für die Winkel:

$$\epsilon \sim \mathcal{U}(0{,}360)$$

$$\phi = \mathrm{a}$$

Mit

- $\mathrm{a} \sim \mathcal{U}(min, max)$
- $min = \begin{cases} b - 1\left(\frac{90}{S_{Anzahl}}\right) - 1, & i \bmod 2 = 0 \\ b - 1\left(\frac{90}{S_{Anzah}}\right), & i \bmod 2 = 1 \end{cases}$
- $max = \begin{cases} b\left(\frac{90}{S_{Anzahl}}\right) - 1, & i \bmod 2 = 0 \\ b\left(\frac{90}{S_{Anzah}}\right), & i \bmod 2 = 1 \end{cases}$

Wobei

- $\mathrm{x} \sim \mathcal{U}(1, S_{Anzahl})$
- $S_{Anzahl} = \text{Anzahl der Halbkugelscheiben}$
- $i = \text{Index des Strahls}$

Dieses System ist durch die beliebige Anzahl an Scheiben der Halbkugel skalierbarer und einfacher zu implementieren als der erste Versuch. Die Eigenschaft, dass das Schiessen der Strahlen zu einem beliebigen Zeitpunkt unterbrochen werden kann und die Impulsantwort statistisch ausreichend ist, bleibt dabei bestehen.

4.3 Eigenschaften der Oberflächen auslesen

Bei einem Aufschlag auf einer Oberfläche füllt das LineTraceByChannel ein Struct[33]. In diesem Struct sind verschiedene Daten vorhanden, wie zum Beispiel die Oberflächennormale, die Weltkoordinaten des Aufschlags sowie Start- und Endpunkt des Trace. Ein weiterer Datentyp dieses Structs ist das *Physical Material*.

Das *Physical Material* ist ein Datencontainer, ähnlich wie ein Struct, welches vorgefertigt, aber nicht initialisiert von *Unreal Engine* bereitgestellt wird. Es wird in der Regel mit Assets der Levelgeometrie verknüpft. *Physical Materials* haben eine Vielzahl an Einsatzmöglichkeiten. Sie dienen zum Beispiel der Charakterbewegung. Der Charakter wird dabei zum Beispiel auf einer

[33] Ein Struct ist eine spezielle C++ Klasse mit der es möglich ist, verschiedene Datentypen in einem Container zu bündeln

Schlammoberfläche abgebremst, beschleunigt beim Laufen auf Asphalt oder rutscht auf Eis aus. Ähnliche System kommen bei Fahrzeugen zum Einsatz, wobei es oft um den Grip eines Fahrzeuges auf verschiedenen Oberflächen geht. Sie können aber auch eingesetzt werden, um zu bestimmen, ob ein Fahrzeug Spuren in der Oberfläche hinterlässt, wie zum Beispiel auf Schnee. Auch für Partikelsysteme kommen sie zum Einsatz, um beispielsweise zu bestimmen welcher Effekt genutzt werden soll. Ebenso können sie determinieren, ob ein Spieler auf einer Schrägen normal laufen, klettern oder gar nicht passieren kann. Auch für Sound Design kommen sie zum Einsatz. Der Oberfläche entsprechende Schrittgeräusche sind ein Klassiker, aber auch Aufschlagsgeräusche von Objekten.

In dieser Thesis wird das *Physical Material* etwas anders eingesetzt, nämlich zur reinen Identifikation von Materialien. Man könnte argumentieren, dass das Material auch vom Namen des Assets bestimmt werden kann. Das setzt aber eine strenge Namenskonvention und dessen Einhaltung voraus. Da, wie im vorherigen Abschnitt bereits erläutert, viele Bereiche der Gameentwicklung ebenfalls *Physical Materials* nutzen, ist es sehr wahrscheinlich, dass es bereits richtig konfiguriert wurde. Damit spart man sich eine Menge an Konfigurationszeit, da sonst für jedes Objekt eine neue Komponente hinzugefügt und in dieser das Material definiert werden müsste. Auf die Richtigkeit der *Physical Materials* kann man sich verlassen, da mehr als nur die Funktionalität von CeSoundTrace davon abhängig ist und damit weitere essenzielle Systeme des Games ihren Dienst einstellen müssten.
Wenn das Struct aus der LineTraceByChannel Node übergeben wird, kann man den Eintrag unter *Physical Material* als Enum[34] nutzten. Dieser liest dann in einer von für CeSoundTrace angelegten Tabelle die gewünschten Daten aus.

[34] Enum ist kurz für Enumerator oder Enumeration zu deutsch. Es ist ein Datentyp, welcher aus einem Namen und dem darunterliegenden Index besteht und in Logikabläufen genutzt wird.

4.3.1 Absorption

Einer dieser Daten ist der Absorptionsgrad $\boldsymbol{\alpha}$. Dieser Wert beschreibt, wieviel der im Strahl enthaltenen Energie für jede Frequenz der simulierten Schallwelle absorbiert wird. Das heisst konkret, dass es nicht ausreicht, nur die Oberfläche zu kennen. Es muss auch bekannt sein welche Frequenz gerade simuliert wird und wieviel Energie sich im Strahl befindet. Zur Berechnung der verbleibenden Energie nach dem Aufschlag ergibt sich folgende Formel:

$$E_{neu} = E_{alt} \cdot (1 - \alpha)$$ (‘Absorptionsgrad’, 2026)

Ein Spezialfall der Absorption ist die Schallausbreitung im Freien. Da diese sehr klein ist, wird sie nur einmal, bevor die Energie in die Impulsantwort eingetragen wird, berechnet. Dies spart Berechnungszeit. Dafür wird die Flugzeit des Strahls vom Spieler aus bis wieder zurück zur Detektorkugel verfolgt. Für diese Art von Dämpfung existieren bereits Tabellen, die im Internet aufgerufen und in UE übernommen werden können. Das einzige Problem ist, dass die Dämpfung in dB/km angegeben ist. Die Werte müssen daher in Energie/cm konvertiert werden, da in erster Instanz eine Energieimpulsantwort simuliert wird und die Masseinheit von *Unreal Engine* standartgemäss Zentimeter ist.

4.3.2 Reflexion

Der zweite und letzte Wert, der aus einer Oberfläche ausgelesen werden muss, ist der Reflexionskoeffizient $\mathbf{d}$. Dieser gibt an, wie reflexiv eine Oberfläche in Abhängigkeit zur Frequenz des Strahls ist. Daher kommt ein ähnliches Konzept wie schon bei der Absorption zum Tragen.

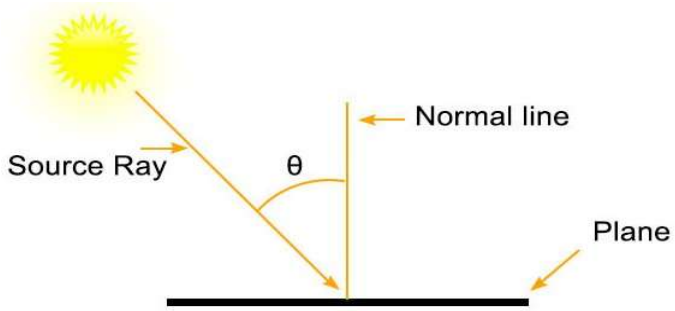


Abbildung 20 Darstellung des Lambert'schen Gesetztes. Hier wird der Einfallswinkel als Theta bezeichnet, in der Thesis ist Theta allerdings der Ausfallswinkel
Bildquelle (https://www.gophotonics.com/community/what-is-lambert-s-cosine-law)

Anders aber als bei der Absorption kann der Reflexionskoeffizient nicht direkt genutzt werden, denn er gibt eine Wahrscheinlichkeit an. Er beschreibt, mit welcher Wahrscheinlichkeit ein Strahl diffus von der Oberfläche reflektiert wird. Dabei kommt das Lambert'sche Gesetzt [Abbildung 20] zum Einsatz (Weik, 2000). Daraus ergibt sich folgendes System: Als erstes wird ermittelt, ob der eintreffende Strahl gespiegelt oder diffus reflektiert wird. Dies geschieht, indem eine zufällige Zahl $\boldsymbol{\rho}$ bestimmt und diese dann mit dem Reflexionskoeffizienten verglichen wird.

$$\rho \sim \mathcal{U}(0,1)$$

Wird der Strahl gespiegelt, bedeutet das, dass der Ausfallswinkel $\boldsymbol{\theta_r}$ dem Einfallswinkel entspricht. Wird er hingegen diffus reflektiert, muss auf Basis des Lambert'schen Gesetztes ein neuer Winkel berechnet werden.

$$u \sim \mathcal{U}(0,1)$$

$$\theta_r = \begin{cases} arcsin(u), & \rho < d \\ \theta_i, & \rho \geq d \end{cases}$$

4.4 Eintrag in die FIR

Wie bereits besprochen, werden viele Informationen ausgelesen, wenn ein Strahl auf einer Oberfläche aufschlägt. Zusätzlich dazu findet bei jedem Aufschlag noch eine weitere Abfrage statt. Es muss ermittelt werden, ob es sich bei der Oberfläche um die Detektorkugel handelt. Dafür besitzt diese einen Tag, mit dem sie identifiziert werden kann. Dafür wird auf dem getroffenen Actor[35] abgefragt, ob dieser einen Tag hat und falls ja, dieser mit demjenigen der Detektorkugel übereinstimmt. Handelt es sich in der Tat um die Detektorkugel, wird eine Reihe von Aktionen ausgelöst. Zuerst wird die zuletzt zurückgelegte Strecke zur totalen Reisestrecke des Strahls addiert. Danach wird der im Strahl verbleibenden Energie noch der Energieverlust durch die Luft abgezogen. Die Reisezeit wird dann in Samples konvertiert. Dies ergibt die Reisezeit in Samples und damit den Index, an welcher die verbleidende Energie des Strahls in die Impulsantwort eingetragen wird. Dann wird der aktuelle Strahl abgebrochen und ein neuer gestartet.

4.5 RT60 nach Sabine

Die RT60 Zeit sagt aus, wie lange ein Impuls braucht, um in einem Raum 60 dB leiser zu werden. In diesem Projekt wird es sich um keine präzise Berechnung dieser Zeit, sondern um eine Schätzung handeln. Dies ist akzeptabel, da diese Zeit nur dann verwendet wird, wenn die FIR zu kurz ist. Dann wird mit einem algorithmischen Hall die Hallfahne verlängert, bis die geschätzte Zeit erreicht wurde. Mit diesem Verfahren kann auch getestet werden, wie gut die Simulation funktioniert. Landet die simulierte Zeit oft ausserhalb der geschätzten Zeit, ist dies ein Zeichen für Fehler oder fehlende Genauigkeit.

Der Algorithmus ergibt sich aus folgender Gleichung:

[35] Der Actor bezeichnet in UE ein Element des Spiels. Dies kann der Spieler selbst, ein interaktives Objekt oder einfach die Geometrie und Objekte des Levels sein.

$$RT60 = 0.161 \cdot \frac{V}{A}$$

Wobei

- $\mathrm{A} = \text{Totale Absorption des Raums in Sabin}$
- $\mathrm{V} = \text{Volumen des Raums in } m^3$

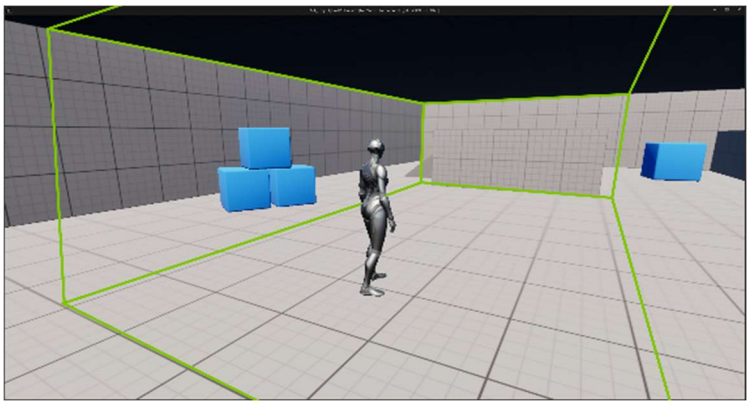


Abbildung 21 Schematisch dargestellt, das geschätzte Raumvolumen innerhalb der grünen Box. Die gesuchten Punkte sind jeweils die Eckpunkte dieser Box

Das Volumen des Raumes ist die leichter zu bestimmende Zahl der beiden. Da durch die Strahlenkanonen bereits viele Punkte[36] im Raum bekannt sind, können diese hier wiederverwendet werden. Dazu wird beim Schiessen der Strahlen jeder Punkt in einen grossen Speicher, einen Vector[37], eingetragen, welcher nach dem Schiessen des letzten Strahls wieder aufgerufen und ausgelesen werden kann. Um nun zu ermitteln, wie gross der Raum ist, werden die Punkte der Grösse nach sortiert. Da es sich sowieso um eine Schätzung handelt, werden kleine Ungenauigkeiten im Volumen des Raums ignoriert. Interessant sind dabei acht Punkte, welche alle relativ zur Spielerposition, links, rechts oben und unten sowie vorne und hinten am weitesten entfernt sind und damit einen Quader um den Spieler formen [Abbildung 21]. Damit lassen sich nun die grösste Länge, Höhe und Breite zwischen den Punkten bestimmen und daraus das ungefähre Volumen des Raumes berechnen.
Schwieriger ist hingegen die totale Absorption des Raums in Sabins.

$$A = S_1\alpha_1 + S_2\alpha_2 + \cdots + S_n\alpha_n = \sum S_i\alpha_i$$

$\boldsymbol{S}$ repräsentiert dabei die Fläche der zu berechnenden Oberfläche und $\boldsymbol{\alpha}$ deren Absorptionskoeffizient. Dazu werden wieder die bereits gespeicherten Aufschlagspunkte zu

[36] Punkte wird hier austauschbar mit dem FVector bzw. dem Vektor benutzt, welcher die drei Koordinaten, x y z, eines Punktes speichert.
[37] Der Vector in C++ ist ein spezieller, dynamischer Listentyp, der bei seiner Initialisierung nicht direkt wissen muss, wie gross er ist. Es besteht hierbei die Verwechslungsgefahr mit dem Vektor, der einen Ort im Raum beschreibt. Deswegen wird in UE von einem FVector gesprochen, der eine 3D Koordinate widerspiegelt. In Unity wäre das Äquivalent dazu ein Vector3 oder ein Vector2 in 2D.

Nutze gemacht und ihnen neue Daten, den Actor und die Oberflächennormale der getroffenen Oberfläche, hinzugefügt. Damit lassen sich die Punkte nach Actor und Oberfläche sortieren. Für jeden Actor wird dafür anhand der Normalen jede Oberfläche ermittelt und alle Punkte dieser Oberfläche gruppiert. Um nun die ungefähre Fläche zu bekommen, wird innerhalb dieser Gruppe bestimmt, welche Punkte am weitesten oben, unten, links und rechts liegen. Die höchsten und tiefsten Punkte werden anhand der z-Koordinate bestimmt. Die am weitesten links und rechts liegenden Punkte sind etwas schwieriger, da diese Begriffe relativ sind. Um diese zu bestimmen wird ein zufälliger Vektor aus dieser Gruppe gewählt. Dieser Vektor wird nun nach rechts gedreht und ausgehend davon berechnet, welche Punkte am weitesten links und rechts davon liegen. Die Distanzen zwischen den Punkten ergeben Breite und Höhe, woraus die Fläche berechnet werden kann. Der Absorptionskoeffizient wird sowieso bestimmt und ist daher bekannt. Damit lässt sich die Formel vervollständigen und die RT60 Zeit bestimmen [Abbildung 22].

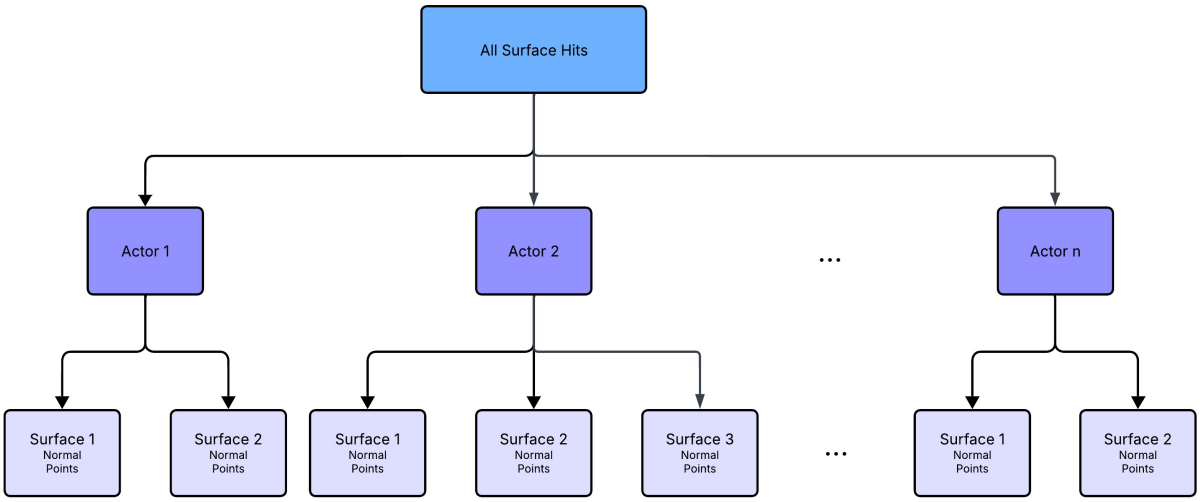


Abbildung 22 Schematische Darstellung der Sortierung für die RT60. Jede Oberfläche verfügt über eine Normale zur Identifikation. Anhand dieser können die Punkte zugeordnet werden. Jede Oberfläche gehört zu einem Actor, alles zusammen ergibt alle Aufschlagpunkte des Traces

4.6 Datenbrücke zu Wwise

Da nun die RT60 Zeit und alle acht Impulsantworten simuliert und berechnet wurden, werden diese Informationen in ein Struct gefüllt, welches den Datencontainer ergibt, der zu *Wwise* gesendet werden muss. Dafür wird die bereits von *Audiokinetic* vorgefertigte Funktion setPluginCustomGameData verwendet. Sie übergibt *Wwise* die Grösse und Speicheradresse des Structs. Die Grösse ist wichtig, da es sich um einen Void-Zeiger[38] handelt. Da dieser Zeiger

[38] Ein Zeiger in C++, oder Pointer in Englisch, enthält die Adresse des Speicherplatzes eines Objektes.

kein Datentyp hat, welcher mit einer Grösse verknüpft ist, muss die Grösse separat mitgegeben werden. Mit diesen Informationen über das Struct, kann *Wwise* dieses auslesen und richtig wieder zusammensetzten.

4.7 Optimierungen

Eine der grössten Herausforderungen beim Raytracing ist der Leistungsverbrauch. Dabei sind die Raycasts, von welchen sehr viele verwendet werden, nicht das grösste Problem, da LineTraceByChannel eine bereits stark optimierte Funktion ist. Vielmehr wird *Blueprint* zum Problem. Nicht nur weil es an Kontrolle über die Ausführung der Nodes mangelt, sondern weil Blueprint nicht auf nativem C++ beruht und daher immer erst umgewandelt werden muss. Diese Umwandlung bringt einige Limitationen und Komplikationen mit sich, daher war es notwendig den Prototyp von *Blueprint* in C++ zu übersetzten.

4.7.1 Blueprint vs. C++

Aus diesem Schritt werden zwei Dinge erwartet. Erstens schnellere Laufzeit und zweitens, mit mehr Kontrolle darüber, wie welche Daten wann verarbeitet werden, einen effizienteren Code. Das Übersetzten von *Blueprint* zu C++ gestaltete sich einfach. Das grössere Problem war es jedoch, die IDE richtig zu konfigurieren. Dies musste mit viel Recherche im Web gelöst werden. Hauptproblem dabei war es, dass das Projekt ursprünglich in der UE Version 5.4 erstellt wurde, es zwischenzeitlichaber zur Version 5.7 aktualisiert wurde. *Epic Games* hat zwischen den Versionen an der *Visual Studio* Integration gearbeitet, weshalb in den Build Scripts von UE, Pfade und Variablen manuell geändert werden mussten, damit VS und UE wieder kompatibel zueinander waren.

In *Blueprint* zu arbeiten ist grundsätzlich sehr schnell und einfach. Doch bei den grossen Datenstrukturen, insbesondere bei der Schätzung der Nachhallzeit und dem Erstellen und Prüfen der Impulsantworten, stösst *Blueprint* an seine Grenzen. Besonders bei den Iterationen durch Listen wie beim Sabine-Algorithmus fehlt es an Kontrolle über die Reihenfolge der Ausführung. Hier schafft C++ Abhilfe, da durch den Code eine strikte Ordnung in der Ausführung von Funktionen erstellt wird. So ist es nicht mehr möglich, dass *Unreal Engine* aufgrund einer Grössenänderung eines Vectors abstürzt. Ebenfalls kann nun kontrolliert werden, welche Daten als Referenz und welche als Kopie bearbeitet werden. Dies ist wichtig für die Effizienz des Codes. In *Blueprint* hat man die Kontrolle über Referenzen und Kopien nur bedingt, daher ist es schwierig zu sagen wie effizient welche Node ist und ab welcher Datengrösse Probleme auftreten. In C++ kann bestimmt werden, was wie übertragen und bearbeitet wird.

Ein weiterer Vorteil von C++ ist, dass der *Blueprint* sowieso konvertiert werden müsste, um die Datenbrücke zu *Wwise* einzubauen.

Das Resultat war erst ernüchternd, weil die Ausführung nur ein wenig schneller wurde. Es zeigte sich aber, dass beim Erhöhen der Strahlenmenge doch von C++ profitiert werden konnte, da sich die Laufzeit kaum veränderte.

5 Das Faltungshall-Plugin in Wwise

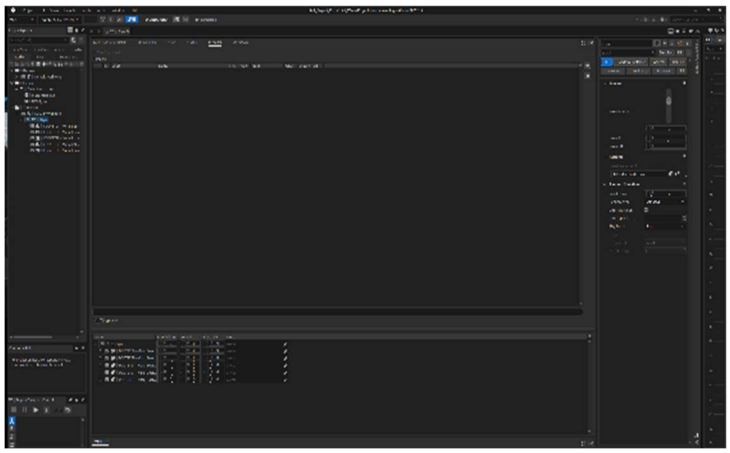


Abbildung 23 Screenshot des UI von Wwise

Was *Unreal Engine* unter den Game Engine ist, dürfte *Wwise* [Abbildung 23] unter den Audio-Middlewares sein. *Wwise* wird von *Audiokinetic* entwickelt und ist seit 2006 verfügbar. Die Middleware erfüllt für Sound Designer einen ähnlichen Zweck wie eine Game Engine für Game Developer. Sie liefert sehr viele Funktionen, die für die Tongestaltung wichtig sind. Dazu gehören Überblendung, Looping, Komprimierung von Assets, Mixing und vieles mehr. Eine Middleware gehört ebenfalls zu den Tools, welche Zeiteinsparnisse und mehr kreative Freiheiten bringen, da nicht immer alles von Grund auf neu entwickelt werden muss. Damit ist *Wwise* die ideale Plattform für ein Projekt in *Unreal Engine*. *Audiokinetic* bietet neben der Software auch eine SDK an, mit der sich eigene Plugins entwickeln lassen. Diese ist nicht nur sehr umfangreich, sondern bietet ebenfalls viel Flexibilität, da sich die Entwicklung nicht gross von der Entwicklung eines DSP-Plugins in C++ unterscheidet.

5.1 Neuer Workflow und Parameterset

Durch diese Thesis wird ein neuer Workflow und Umgang mit Hall in Games präsentiert. Wie bereits im Kapitel 2.1.3: Parametrische Kontrolle und Subjektivierung einer Hallsimulation angesprochen entsteht durch die Kombination eines Parametersets und der Simulation einer FIR dieses Umdenken der Hallgestaltung. Anstatt dass Presets streng diktieren, wie ein Algorithmus den Hall zu berechnen hat, gibt das Parameterset lediglich einen Rahmen vor. Für CeSoundTrace bedeutet das, dass es Parameter zur Kontrolle eines Equalizers hat. Dies ist nichts Unkonventionelles, ein Post-Equalizer ist weit verbreitet. Daher ist es wichtig, dass auch die Grundbedürfnisse der Hallbearbeitung abgedeckt sind. Inspiriert von anderen Faltungshallen, weist CeSoundTrace ebenfalls Parameter zur Verlängerung, respektive der Verkürzung der FIR und zur Betonung einzelner Elemente dieser, wie zum Beispiel die Nachhallfahne oder die frühen Reflexionen auf. Diese Parameter ermöglichen nicht nur eine Subjektivierung, sie sind auch der Startpunkt des neuen Workflows. Die Parameter diktieren der Simulation nicht wie genau die FIR zu klingen hat, sie sind lediglich Veränderungen, welche auf

die FIR angewendet werden. Die eigentliche Simulation wird von der Umgebung des Spielers diktiert. So ist es bei der Erstellung des Halls mit CeSoundTrace nicht mehr möglich, genau vorherzusagen, wie der Hall klingen wird. Stattdessen gibt es ein mögliches Spektrum und der Standort des Spielers entscheidet, wie genau der Hall klingt. Aus diesem Grund kann man die Bearbeitung der Parameter als Erstellen einer Schablone bezeichnen. Dies ist auch das, was den Workflow neu und anders macht. Man muss sich stückweit überraschen lassen, was die Schablone und die Simulation gemeinsam ergeben.

5.2 Algorithmen

Das zentrale Element eines jeden Plugins ist dessen Algorithmus. Im Falle eines Faltungshalls sind dies mehrere: die Faltung, die FFT/IFFT und die Partitionierung. Diese Elemente entscheiden über wichtige Parameter, zum Beispiel wieviel Latenz das Plugin im Bus verursacht, wieviel Leistung benötigt wird und wie präzise die Faltung stattfindet.

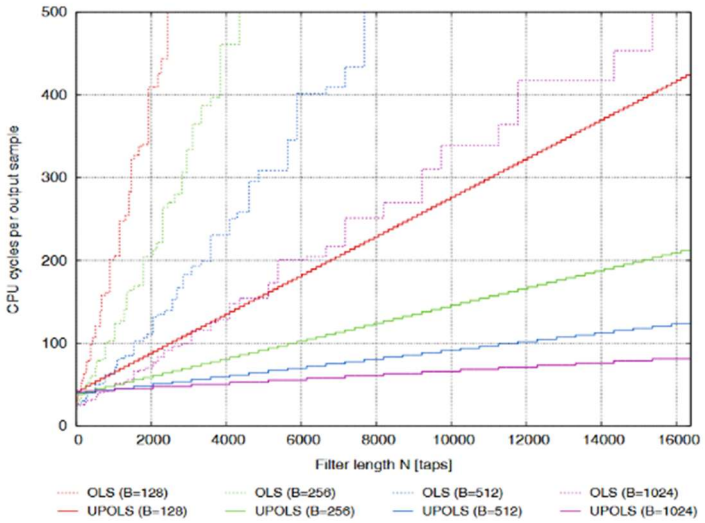


Abbildung 24 Abbildung aus "Partitioned convolution algorithms for real-time auralization". Es zeigt den Unterschied zwischen OLS und UPLS in verbrauchter Leistung (Wefers, 2015)

Der geeignete Faltungsalgorithmus für diese Projekt, GUPOLS [Kapitel 5.2.2: GUPOLS vs. UPOLS vs. NUPOLS] wurde in der Dissertation von Dr. Frank Wefers mit dem Titel „Partitioned convolution algorithms for real-time auralization“ (Wefers, 2015) gefunden.

Da keine FFT und IFFT Funktion von der *Wwise* SDK zur Verfügung gestellt wird, muss eine Bibliothek implementiert werden. Erste Versuche wurden mit der altbewährten *FFTW*[39] Library unternommen, welche später, aus Abhängigkeitsgründen, durch *Pocket FFT*[40] ersetzt wurde.

5.2.1 Warum Partition?

Aus der Dissertation von Dr. Frank Wefers wurde klar, dass der beste Weg über die Partitionierung der Signale führt. Herr Dr. Wefers hat dazu die Effizienz von Faltung mit und ohne Partitionierung direkt verglichen und konnte damit zeigen, dass durch Partitionierung

[39] FFTW Produktseite (https://www.fftw.org/)
[40] Pocket FFT GitHub (https://github.com/mreineck/pocketfft)

signifikante Verringerungen in der verbrauchten Leistung erreicht werden können [Abbildung 24].
Dies kommt daher, dass kleinere Blöcke an Samples weniger Operationen benötigen, um gefaltet zu werden. Als Bonus ist es damit auch möglich, schneller gefaltete Samples in den Output zu schreiben.

5.2.2 GUPOLS vs. UPOLS vs. NUPOLS

Diese drei Abkürzungen stehen fast für dasselbe. Es handelt sich um Bezeichnungen dreier "Overlap and Save" Algorithmen, daher die Endung mit „OLS“. Bei GUPOLS und UPOLS handelt es sich um einheitlich partitionierende Algorithmen, daher das U für "Uniform Partitioned Overlap And Save". NUPOLS hingegen ist uneinheitlich partitioniert, daher "Non Uniform Partitioned Overlap And Save".

Diese drei Algorithmen sind in der Endauswahl gelandet. Der Beste davon ist ganz klar NUPOLS, da er der flexibelste, schnellste und ressourcenschonendste ist. Wenn NUPOLS so gut ist, warum existieren noch zwei weiter in der Auswahl?
Da NUPOLS eine nicht uniforme Partitionierung einsetzt, erfordert dies eine erheblich schwierigere technische Implementierung. Um den Algorithmus zu implementieren, ist die Kontrolle über die Zeitabläufe der Threads notwendig. Da dies den zeitlichen Rahmen dieser Arbeit sprengen würde, mussten Alternativen zu NUPOLS existieren.
Hier kommen GUPOLS und UPOLS ins Spiel. Beide Algorithmen sind uniform partitionierend und unterscheiden sich nur in der Blockgrösse. Der einfache "Uniform Partitioned Overlap And Save" Algorithmus braucht eine feste Blockgrösse, welche zwingend ein Vielfaches von 2 sein muss. Der "Generalised Uniform Overlap And Save" Algorithmus hingegen ist flexibler, aber auch komplexer. Die Flexibilität zeigt sich primär darin, dass die Blockgrösse beliebig gewählt werden kann, da hier mit anderem Zero-Padding und einer anderen FFT-Grösse gearbeitet wird. Der GUPOLS Algorithmus ist effizienter als der UPOLS Algorithmus. Dies kommt daher, dass der GUPOLS Algorithmus für sehr kleine Filter, praktisch wie ein Algorithmus ohne Partition arbeitet und sich damit Zero-Padding und grosse FFT's spart. Für grosse Filter wird die bereits erwähnte Flexibilität der Blockgrösse für die höhere Effizienz von GUPOLS entscheidend.
Aus Sicherheits- und Komplexitätsgründen, wurde der UPOLS Algorithmus zur Implementierung ausgewählt und sollte es die Zeit hergeben, kann GUPOLS immer noch implementiert werden.

5.3 Das Plugin im Detail

Der Signalfluss im Plugin weist zwei Signalwege auf. Derjenige des Filters beziehungsweise der Impulsantwort $\boldsymbol{h}$ und derjenige des Inputs $\boldsymbol{x}$. Beide laufen in der Frequency-domain delay line (FDL) zusammen und das Ergebnis wird in den Output geschrieben.

5.3.1 Filter

5.3.1.1 Zusammenführen der Impulsantworten

Der Filter startet damit, dass die aus UE übermittelte Speicheradresse ausgelesen werden muss. Da es dafür kein Callback gibt, geschieht das Auslesen der Adresse leider bei jedem Aufruf der Hauptfunktion des Plugins, der Execute Funktion. Um Rechenleistung zu sparen, wird eine Versionsnummer benutzt, welche die Verarbeitung der Daten erst auslöst, wenn sich diese Nummer geändert hat.

Die Verarbeitung startet mit dem Zusammenführen der Impulsantworten. Dies wird gemacht, indem die Energieimpulsantworten $\boldsymbol{h_n^2(i)}$, wobei $\boldsymbol{n}$ der Index des Oktavbandes ist, zu einer Breitband-Schalldruckantwort $\boldsymbol{h(i)}$ konvertiert wird. Dazu wird für jeden Eintrag in der Energieimpulsantwort, der nicht Null entspricht, ein Eintrag in der Schalldruckantwort gemacht. Dafür wird für jedes Oktavband ein Filter $\boldsymbol{f_n}$ benötigt. Diese Filter sind Tiefpassfilter, welche ihren Cutoff an der Frequenz der jeweiligen Energieimpulsantwort besitzen. Um nun den Eintrag der Schalldruckantwort zu erhalten, wird eine gewichtete Summe erstellt. Die Gewichte erhält man durch die Wurzel der entsprechenden Energieimpulsantwort. Das Gewicht wird dann mit der Impulsantwort des Oktavfilters multipliziert und mit den anderen Oktaven summiert. Daraus ergibt sich

$$h(i) = \sum_{n=1}^{max(h^2)} \sqrt{h_n^2(i)} \cdot f_n$$

Sobald die Energieimpulsantwort konvertiert worden ist, kann sie weitergereicht werden (Athari, 2023).

5.3.1.2 Partitionieren, FDL und Faltung

Die Breitband-Impulsantwort wird nun in ihre Partitionen aufgeteilt. Die Länge der Impulsantwort bestimmt direkt, wieviel Partitionen $\mathbf{P}$ es gibt und damit ebenfalls wie lange die FDL sein wird. Diese Anzahl ergibt sich aus

$$P = \left\lceil \frac{N}{B} \right\rceil$$

Daraus ergibt sich

$$B \cdot P \geq N$$

Dabei ist $\mathbf{B}$ die Blockgrösse und $\boldsymbol{N}$ die Länge des Filters. Die Partitionen werden dann mit $\mathbf{B}$ Anzahl an Nullen erweitert, sodass jede Partition danach eine Länge von $\mathbf{2B}$ hat. Die erste Hälfte der Einträge sind dabei die Nullen und die zweite die Samples aus dem Filter. Danach erfolgt mit einer $\mathbf{2B}$ langen FFT die Transformierung vom Reellen ins Komplexe. Damit sind die Partitionen abgeschlossen und können in die FDL geladen werden.
Die FDL selbst besitzt immer so viele Container wie der Filter Partitionen. Demnach muss die Länge der FDL nach jeder neuen Impulsantwort angepasst werden. Kommt nun ein Container mit $\boldsymbol{2B}$ Inputsamples, wird dieser mit dem ersten Container der FLD, welcher die erste Partition des Filters enthält, gefaltet. Da dies im Frequenzspektrum geschieht, wird elementweise gerechnet. Dies entspricht einer zirkulären Faltung im Zeitspektrum (Wefers, 2015). Das Ergebnis wird in einen Akkumulator übertragen. Kommt nun der nächste Container mit Inputsamples, wird der erste innerhalb der FDL eine Position weitergeschoben. Der zweite Container mit Inputsamples wir nun mit dem ersten der FDL gefaltet und der erste Container mit Inputsamples wiederum mit dem zweiten der FDL. Beide Ergebnisse werden in den Akkumulator geschrieben und summiert. Dieser Prozess wiederholt sich immer weiter, bis alle verfügbaren Container mit Inputsamples gefaltet sind. Dann tritt der Akkumulator in Kraft. Dieser summiert alle Ergebnisse der FDL zu einem Container mit einer Länge von $\boldsymbol{B} + \mathbf{1}$ Samples auf. Die Summierung kann maximal $\boldsymbol{P} - \mathbf{1}$ Operationen benötigen. Hat der Akkumulator fertig summiert, wird sein Inhalt mit der IFFT vom Komplexen zurück ins Reelle transformiert. Dort wird die erste Hälfte des nun wieder $\mathbf{2B}$ Samples langen Containers gelöscht. Die gelöschten Samples entsprechen dem Zeit-Aliasing. Der resultierende Block von Samples hat nun wieder die Länge $\boldsymbol{B}$ und kann in den Output des Plugins übertragen werden.

5.3.2 Signal

Das Signal startet damit, dass *Wwise* ein Block von Samples an das Plugin übergibt und dessen "Execute" Funktion aufruft. In dieser Funktion finden auch die anderen Vorgänge des Plugins statt, welche im vorhergehenden Kapitel beschrieben wurden.

5.3.2.1 Blockgrösse

Das Spezielle an der Execute Funktion ist, dass die Sampleblöcke, die das Plugin bekommt, keine einheitliche Grösse haben. Dies ist für ein System, das gleich grosse Blöcke erwartet ein Problem. So musste eine Logik implementiert werden, die die Samples in einen separaten Buffer schreibt und diesen and die FDL übergibt, sobald er die gewünschte Grösse erreicht hat. Ähnlich verhält es sich mit dem Output. Da immer die gleiche Anzahl an Samples aus der Faltung kommen, muss aufgepasst werden, dass nicht zu viele oder zu wenige Samples in den Output geschrieben werden. Sollten es zu viele sein, müssen sie zwischengespeichert werden.

5.3.2.2 FDL mit Input Buffern

Wird von der Execute Funktion ein Buffer an Inputsamples an die FDL übergeben, müssen diese erst noch aufbereitet werden. Dieser Vorgang ist fast analog zur Vorbereitung der Partitionen des Filters. Wie bereits beim Filter muss der Buffer mit Inputsamples $\boldsymbol{2B}$ lang sein. Um dies zu erreichen, muss erst abgewartet werden bis Wwise die benötigte Anzahl an Samples der Execute Funktion übergeben hat. Dies gilt für den allerersten Buffer mit Inputsamples. Für alle weiteren werden lediglich $\boldsymbol{B}$ Anzahl Samples benötigt. Um nun den neuen Buffer zu formen, werden aus dem alten Inputbuffer die älteren $\boldsymbol{B}$ Samples gelöscht, die Verbleidenden nachgeschoben und der Freigewordene Platz mit den neue Samples gefüllt. Damit erhält man ein Fenster mit $\boldsymbol{B}$ Länge, welches sich über den Input schiebt. Jeder Buffer mit Inputsamples wird, wie schon beim Filter mit der FFT, vom Reellen ins Komplexe transformiert. Der Inputbuffer muss nun in die erste Position der FDL geschrieben werden. Damit diese Position leer wird, werden alle vorhergehenden Inputbuffer in der FDL eine Position weiter geschoben. Müsste einer dieser Buffer an eine Position geschrieben werden, die grösser ist als die Maximalanzahl an FDL Positionen, so wird dieser gelöscht, da dies Bedeutet, dass er mit der gesamten FIR gefaltet wurde. Damit endet die Abfolge für die Inputsamples, da sie hier mit der FIR gefaltet werden und danach nur noch die gefalteten Samples, welche bereits beschreiben wurden, benötigt werden.

5.4 Filter Wechsel

Sobald eine neue FIR bereitsteht, muss diese die alte ersetzten $\boldsymbol{h_0(n) \rightarrow h_1(n)}$. Dafür gibt es folgende zwei Ansätze, den Austausch im Frequenzspektrum oder im Zeitspektrum zu vollziehen. Aus „Partitioned convolution algorithms for real-time auralization“ (Wefers, 2015) kann man entnehmen, dass sich beide Ansätze nicht gross in ihrem Ressourcenverbrauch unterscheiden. Wenn der Austausch im Frequenzspektrum stattfindet, spart man sich eine IFFT. Hingegen spart man sich im Zeitspektrum die Transformation der Hüllkurven, welche den Filtertausch erst möglich machen. Der Einfachheit halber, wurde sich für den Austausch im Zeitspektrum entschieden.

Der Austausch findet in einem Outputblock statt. Dafür werden zwei Hüllkurven $\boldsymbol{Env_{out}}$ $\boldsymbol{Env_{in}}$ benötigt, um eine Überblendung der beiden FIR zu ermöglichen. Die alte FIR wird dabei ausgeblendet und die neue eingeblendet. Die Hüllkurven müssen zusammen eine konstante Amplitude ergeben, andernfalls entstehen Artefakte.

$$\mathrm{Env}_{out}(n) + \mathrm{Env}_{in}(n) = 1 \qquad (0 \leq n < \mathrm{L})$$

Dafür existieren wiederum zwei mögliche Ansätze: einer mit linearer Hüllkurve und einer mit quadriertem Cosinus.

$$\mathrm{Linear} \qquad \mathrm{Env}_{\mathrm{out}}(\mathrm{n}) = 1 - \frac{\mathrm{n}}{\mathrm{L}} \qquad Env_{\mathrm{in}}(\mathrm{n}) = \frac{\mathrm{n}}{\mathrm{L}}$$

$$QuadrierterCosinusEnv_{out}(n) = cos^2\left(\frac{\pi n}{2L}\right) Env_{in}(n) = sin^2\left(\frac{\pi n}{2L}\right)$$

Wendet man diese beiden Hüllkurven auf die Filter an, so ergibt sich

$$y(n) = y_0(n) \cdot Env_{out}(n) + y_1(n) \cdot Env_{in}(n)$$

Da bereits bei der Wahl des Spektrums auf Simplizität geachtet wurde, bleibt die Wahl bei den Hüllkurven gleich und es wird daher die lineare Variante verwendet (Wefers, 2015).

5.5 Multichannel

Aktuell wird die FIR nur in Mono simuliert. Multichannel ist aber mit *Dolby 5.1*[41] fast ein Standard im Game Audio und mit dem steigenden Bedürfnissen nach *Dolby Atmos*[42] und *DTS X*[43] steigt auch das generelle Bedürfnis nach Multichannel Audio. Daher ist es von Vorteil, wenn die FIR ebenfalls Multichannel-fähig ist.

[41] Dobly Digital 5.1 Produktseite (https://professional.dolby.com/tv/dolby-digital/)
[42] Dolby Atmos Produktseite (https://www.dolby.com/de/technologien/dolby-atmos/)
[43] DTS X Produktseite (https://dts.com/dts-x/)

Die Idee dafür ist, mit mehr als einer Detektorkugel zu arbeiten. Jede dieser Kugeln würde bei einer realen Multichannel-Aufnahme eine Mikrofonkapsel repräsentieren. Das Einzige, was sich ändern würde ist, dass dann pro Frequenz mehrere Impulsantworten gemacht werden. Diese müssten dann nur richtig sortiert werden, denn in *Wwise* iteriert man sowieso über alle verfügbaren Kanäle. Daher müsste nur angepasst werden, dass auch der Kanal der FIR analog zu dem von *Wwise* mititeriert wird.

6 Ergebnis und Nutzungspotenzial

Das Ergebnis dieser Arbeit ist ein Prototyp. Es gibt daher einige Probleme, die etwas rudimentär und nicht mit Finesse gelöst wurden. Dazu gehören oberflächlichere Dinge wie das *User Interface* aber auch tiefer versteckte, wie die Nachhallzeit und Effizienz der Strahlenkanone. Gerade Letzteres ist ein Knackpunkt. Als die Idee für dieses Projekt entstand, war es mein Ziel, dass das Plugin ganz in Echtzeit arbeiten kann. Aktuell ist es davon noch ein Stück entfernt. Dies wäre auch für ein fertiges Produkt essenziell, da es das Plugin einzigartig macht und sich daher gut verbreiten lassen sollte. Ein weiteres grosses Thema, mit dem ich nicht glücklich bin, ist die Implementierung des Sabine-Algorithmus. Diese ist grob und bedarf definitiv weiterer Überarbeitung. Eine Idee wäre beispielsweise, die Punkte für das Erstellen einer konvexen Hülle zu verwenden, anstatt damit Höhen und Breiten zu berechnen. Dazu würde etwas wie der Chan's Algorithmus genutzt, welcher eine genauere Berechnung von Oberflächen und Volumen zulässt ('Chan's Algorithm', 2025). Es stellt sich aber grundsätzliche die Frage, inwieweit die Nachhallzeit wirklich benötigt wird oder ob es reicht, mit genügender Qualität des Raytracings zu arbeiten. Daraus ergibt sich die Frage, wann ein Raytracing genügend ist. Dazu müsste eine Befragung gemacht werden, bei dem Personen Hallfahnen von diesem Plugin und algorithmischen Hallen einem Bild eines Raumes zuordnen müssen. Gleichzeitig könnte damit die Qualität und die Nützlichkeit des Plugins getestet werden. Im Rahmen dieser Arbeit ist dies aber nicht möglich, da es dem Plugin zu diesem Zeitpunkt noch an Distributionsfähigkeit mangelt. Ohne diese ist der logistische Aufwand zu hoch, was sich negativ auf die Befragung auswirken würde und wodurch dessen Daten nichtig würden.
Ein weiterer Punkt, welcher in dieser Thesis nicht detailliert beschrieben ist, ist der Datensatz der Oberflächen. Dieser ist aktuell von mir erstellt und auf die Demonstrationsumgebungen angepasst. Es stellt sich daher die Frage, inwieweit ein solcher Datensatz wirklich von Nutzen ist, ob dieser Datensatz mit dem Plugin versendet werden soll oder ob jeder Sound Designer seinen eigenen Datensatz erstellen muss. Dies würde aber bedeuten dass jeder Benützer seine eigene Tabelle an Materialien, inklusive den Absorptions- und Reflexionskoeffizienten jeder Simulationsfrequenz erstellen müsste. Bezüglich der Datensätze stellt sich auch die Frage, ob die Schnittstelle des Plugins geöffnet werden sollte, sodass dem *Wwise* Plugin ein beliebiger Datensatz übergeben werden kann. So würde sich die kreative Nutzung des Plugins um ein Vielfaches erweitern.
Die Plattformen auf der das Plugin aktuell operiert sind ebenfalls bescheiden. Aktuell getestet ist es nur für UE 5.7 mit *Wwise* 2025 auf *Windows* 11.
Diese Punkte zeigen auf, dass es noch viel Arbeit gibt, bis das Plugin Marktreife erlangt. Dies

involviert das Bilden eines kleines Teams und ein bis zwei Jahre weitere Entwicklungszeit. Daher ist der Nutzen aktuell recht eingeschränkt. Dennoch zeigen dieses und das folgende Kapitel das enorme Potential, welches hinter diesem Projekt steckt. Inwieweit aber das Plugin seinen Nutzen erfüllen wird, wird sich daher erst noch zeigen müssen.

7 Schlussreflexion und Ausblick

Der Weg zu diesem Prototyp war lange und hat mich teilweise an den Rand des Wahnsinns gebracht. Dafür ist mein Wissen um einiges grösser geworden, gerade was Coding und das Verständnis von DSP angeht. Dazu konnte ich *Unreal Engine* vertieft kennenlernen und mich mit *Wwise* und dessen SDK auseinandersetzten. Natürlich wäre es schön gewesen, mit einem fertigen Produkt abzuschliessen, doch dies war in diesem Rahmen leider nicht möglich. Ich nehme sehr viel Wissen aus dieser Arbeit mit, welches ich hoffentlich in meiner zukünftigen Arbeit einsetzten kann.

7.1 Optimierung und Qualität

Wie bereits angesprochen sind noch einige Optimierungen nötig. Zum einen muss der Code aufgeräumt werden, zum anderen könnte es sich lohnen, einen eigenen LineTraceByChannel zu schreiben, da ich viele Funktionen davon gar nicht nutzte. Bei grösseren Operationen mit den Vectors der FIR gäbe es bessere Wege. Die FIR mit einer Art Kompression zu speichern wäre einer davon. Alle genannten Punkte tragen dazu bei, dass der Code schneller läuft und damit wiederum die Anzahl Strahlen und Reflexionen erhöht werden kann. Damit würde es möglich, die Qualität weiter zu steigern.
Weiter Punkte, welche die Effizienz steigern würden, wären: einen anderen FFT Algorithmus zu benutzen, GUPOLS oder NUPOLS implementieren und der Ablauf der Partition ändern.

7.2 GPU Audio

Ein weiteres Vorhaben ist das Einbinden der *GPU Audio*[44] API. Diese ermöglicht es, Berechnungen von der CPU auf die GPU zu verschieben. Da die GPU sowohl über Raytracing-Kerne sowohl als auch über Tensor-Kerne verfügt, macht es sie ideal für das Arbeiten mit Faltung und Raytracing. Da gerade bei vielen Indie Spielen eine moderne Grafikkarte überhaupt nicht ausgelastet ist, bleibt viel Rechenleistung ungenutzt. Selbst bei AAA-Games wird langsam ein Overhead erkennbar, da der AI-Boom für eine hohe Anzahl an Tensor-Kernen sorgt, welche nicht gleich wichtig für Gaming sind. Es wäre daher ein grosser Mehrwert für dieses Projekt, einen Teil der Rechenlast and die GPU zu übergeben, zumal so eine höhere oder gleich hohe Qualität mit weniger CPU Nutzung gewährleistet werden kann.
Ein ähnliches Projekt wurde bereits von VSL realisiert. Da wird *GPU Audio* benutzt, um ihr Hallsystem MIR Pro 3D[45], welches detaillierte Hallmodelle von bekannten Konzertsälen als Hall

[44] GPU Audio Produktseite (https://www.gpu.audio/)
[45] MIR 3D Produktseite (https://www.vsl.co.at/products/software/vienna-mir-pro-3d)

verfügbar macht, schneller zu machen. Dieses System ist in seiner vollen Ausführung auf einem einfachen Computer ansonsten fast nicht denkbar, da parallel dazu meist noch grosse VSTi Bibliotheken geladen sind.

7.3 User Interface und Visualisierung

Wie bereits angesprochen, gehört das UI zu den Punkten, welche aktuell nur rudimentär gelöst sind. In meiner Philosophie der Softwareentwicklung gehört ein schönes, übersichtliches und einfach zu bedienendes UI aber zum Produkt. Das UI fördert die Interaktion zwischen Benützer und Software. Gerade in den kreativen Bereichen ist dies ein wichtiger Bestandteil, da sich nicht alle gerne mit komplizierten technischen Begriffen herumschlagen. Es braucht daher ein übersichtliches UI, das möglichst intuitiv funktioniert und Fachbegriffe meidet. Dabei ist es hilfreich sich an bestehenden Faltungshallen zu orientieren. Daraus erschliesst sich, dass eine Art Wasserfalldiagramm nützlich zur Beurteilung des Halls ist. Zusätzlich könnte es für Multichannel-Funktionalität nützlich sein ein Energie-Visualisierer einzubauen, ähnlich wie der von IEM[46]. Die weiteren Parameter könnten mit konventionellem UI realisiert werden. Die einzige Ausnahme davon wäre der EQ. Dieser benötigt etwas intuitiveres für die Kontrolle der Bänder, wahrscheinlich eine klassische Kombination eines grafischen EQ's.
Ist es aufgrund des UI's frustrierend ein Plugin zu benutzen, wird es in die Ecke geworfen. Daher ist eine gute Umsetzung wichtig, ansonsten wird *CeSoundTrace* leider keine Erfolg haben.

7.4 Distribution

Genauso lästig wie ein unübersichtliches UI sind komplizierte Installationsanleitungen. Ich bin ein grosser Verfechter der Executables. Einmal doppelklicken und der Rest geschieht von allein. Wie schon beim UI bin ich der Meinung, dass eine komplexe Anleitung für das Installieren einer Software bei vielen potenziellen Nutzern abschreckend wirkt. Für dieses Projekt gibt es viele Distributionsmöglichkeiten. Wie schon angesprochen, könnte man es in einer ausführbaren Datei verpacken. *Wwise* und UE besitzen aber beide eigene Marktplätze für Plugins. Es wäre also möglich das Plugin auch darüber zu versenden, allerdings wird dafür eine Anleitung von Nöten sein, da man beide Teile des Plugins braucht und diese an zwei verschieden Orten installiert werden müssten. Eine weitere Möglichkeit wäre es, den Code online zu stellen, zum Beispiel auf *GitHub,* und dann mit dem Readme eine Anleitung zu schreiben, wie das Plugin installiert werden muss oder direkt eine ausführbare Datei anbieten.

[46] IEM Energy Visualizer Produktseite (https://plugins.iem.at/#tab-EnergyVisualizer)

7.5 Environmental Sound Suite – Die Vision und Zukunft für CeSoundTrace

Die Vision für dieses Projekt geht so weit, dass nicht nur die fehlende Occlusion und Obstruction eingebaut werden sollen, sondern weitere Elemente der auditiven Umgebungsgestaltung dazukommen sollen. Auch Sounds, welche nicht vom Spieler oder von Objekten in unmittelbarer Nähe entstehen, sollen einen Hall bekommen. Damit können alle Objekte mit Ton im Spiel mit *CeSoundTrace* verhallt werden. Dies würde auch zu einer weiteren Vereinfachung der Konfiguration führen und die Benutzung um ein vielfaches vereinfachen. Erst damit hätte CeSoundTrace alle Funktionen bekommen und wäre ein vollständiges Produkt. Darauf aufbauend könnte man in weitere Branchen expandieren, wie zum Beispiel in die *Virtual Production.* Dazu würden Export- und Timeline-Funktionen kommen, die aus dem reinen Interaktiven-Plugin ein Lineares machen. Damit wäre es dann möglich einen Hall zu simulieren und diesen als Datei zu exportieren. Oder es könnte auch in einer DAW genutzt werden, fast wie ein konventioneller Faltungshall.

CeSoundTrace müsste aber nicht das einzige Plugin dieser Art sein. Raytracing und prozedurale Generation können auch für mehr als nur Hall benutzt werden. Dies bildet die Basis für eine Reihe an Plugins, welche als Gesamtes die *Environmental Sound Suite* ergeben. Beispiele dafür wären: Ein prozedurales Umgebungsgeräusche-Plugin. Es nutzt Raytracing, um die Umgebung, die verwendeten Materialien und Objekte zu analysieren und platziert basierend darauf Geräusch-Emitter. Um die *Wwise*-UE Kombination auch in diesem Plugin zu integrieren, könnte der Pool and Geräusch-Emitter, welche dem Plugin zur Verfügung stehen, aus in Wwise erstellten Events sein. Des Weiteren könnte mit einem ähnlichen System ein Regensimulator-Plug erstellt werden. Wieder wird Raytracing für eine Umgebungsanalyse gebraucht. Dabei wird sich diese Plugin auf die nach oben gerichteten Oberflächen konzentrieren und deren Materialien und Position nutzen um die korrekten Regengeräusche im Level zu platzieren. Dabei könnte mit Parametern die Grösse der Tropfen, die Intensität und die Beeinflussung durch Wind gesteuert werden. Anstatt Regen zu generieren könnte dasselbe System auch genutzt werden um Tiergeräusche, wie zum Beispiel Vogelgesang, in der Welt zu platzieren. Dabei würde das Plugin, verschiedene Vegetation erkennen und darin Vogelgesang-Emitter platzieren. Auch hier gäbe es wieder Parameter zur Kontrolle von Spieldauer, Häufigkeit, Grösse des Vogelschwarms und wann die Vögle verstummen und wegfliegen. Ergänzend zu diesen Plugins könnte auch ein prozeduraler Feuersimulator und ein Motorensimulator kommen.

Alle gennannten Ideen für die *Environmental Sound Suite* verfolgen ähnliche Konzepte wie schon *CeSoundTrace*. Damit kann diese Arbeit als Grundlage für eine grosse Vision verstanden

werden. *CeSoundTrace* gliedert sich in eine Reihe an Plugins, welche alle das Ziel haben, kreatives Arbeiten zu fördern und dabei Mühseligkeit einzusparen. Meine Arbeit an *CeSoundTrace* endet nicht mit der Abgabe dieser These. Ich will das Plugin weiterentwickeln bis es zu einem Vollständigen Produkt wird und vertrieben werden kann.

8 Appendix

Deklaration zur Verwendung von AI

Allgemein wurde die kostenlose Version von ChatGPT eingesetzt, um einen Überblick über grundlegende Fragen zu erhalten. So habe ich beispielsweise ChatGPT angewiesen, eine Liste von Faltungsalgorithmen zu erstellen. Ausgehend davon habe ich eine traditionelle Recherche fortgesetzt.

In der Thesis

In der Thesis wurde keine AI eingesetzt.

Im Projekt

Im Projekt wurde keine AI eingesetzt, um Code zu generieren. Es wurde aber die kostenlose Version von ChatGPT eingesetzt, um Build, Compiler und Linker Errors der IDE zusammenzufassen. Ebenfalls in diese Kategorie fallen Zusammenfassungen der Crashreports und Logdateien von UE. Ebenfalls wurde ChatGPT für das Debugging eingesetzt, wobei auch hier der Transformer nie selbst Änderungen am Code vorgenommen hat. Alle Änderungen wurden von mir geprüft und editiert.

Link zum Medienarchiv

Medienarchiv (https://medienarchiv.zhdk.ch/sets/de025397-ea38-427f-89aa-3526520ece32)

9 Quellverzeichnis

Athari, S. (2023). *A hybrid acoustics simulation of 3D urban street canyon design scenarios with multiple reflections used for auralising a moving sound source* (p. 88 p.) [ETH Zurich; Application/pdf]. https://doi.org/10.3929/ETHZ-B-000607512

Audiokinetic (Director). (2019, February 7). *Wwise Tour 2018—Remedy Entertainment (2 of 3) Northlight Environmental Audio Tech* [Video recording]. https://www.youtube.com/watch?v=WsXHcDyMhWM

Branch Education. (2024, August 17). *How does Ray Tracing Work in Video Games and Movies?* [Video recording]. https://www.youtube.com/watch?v=iOlehM5kNSk

Chan's algorithm. (2025). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Chan%27s_algorithm&oldid=1323561551

Coffee Stain Studios. (2025, November 11). *Why does Satisfactory sound like that?* [Video recording]. https://www.youtube.com/watch?v=mYB5dR7Pl_s

Mårtensson, P. (2024, April 4). *Ray Tracing Audio in Snowdrop: Creating a Living Pandora*. Massive Entertainment. https://www.massive.se/blog/news/ray-tracing-audio-in-snowdrop-creating-a-living-pandora/

Middle-Earth: Shadow of Mordor. (n.d.). WB Games. Retrieved 15 May 2026, from http://warnerbrosgames.com/game/middle-earth-shadow-of-mordor

Nachhallzeit. (2026). In *Wikipedia*. https://de.wikipedia.org/w/index.php?title=Nachhallzeit&oldid=265855726

Project Triton—Immersive sound propagation. (n.d.). *Microsoft Research*. Retrieved 23 March 2026, from https://www.microsoft.com/en-us/research/project/project-triton/

Reilly, A., & McGrath, D. (1995). *Convolution Processing for Realistic Reverberation*. https://aes2.org/publications/elibrary-page/

Tarr, E. (2018). *Hack Audio: An Introduction to Computer Programming and Digital Signal Processing in MATLAB*® (1st edn). Routledge. https://doi.org/10.4324/9781351018463

Unreal Engine. (2022, December 7). *Microsoft Project Acoustics in Unreal Engine 5 | GameSoundCon 2022* [Video recording]. https://www.youtube.com/watch?v=MAMz9dSHU04

Valve. (n.d.). *Steam Audio*. Retrieved 22 May 2024, from https://valvesoftware.github.io/steam-audio/

Valve. (2017, February 23). *Steam: Steam Audio :: Jetzt neu: Steam Audio*. https://store.steampowered.com/news/app/596420/view/1589135317218111209

Wefers, F. (2015). *Partitioned convolution algorithms for real-time auralization*. Logos Verlag Berlin GmbH.

Weik, M. H. (2000). Lambert's cosine law. In *Computer Science and Communications Dictionary* (pp. 868–868). Springer, Boston, MA. https://doi.org/10.1007/1-4020-0613-6_9901

Selbständigkeitserklärung Master-Arbeit

Hiermit bestätige ich, dass ich die vorliegende Master-Arbeit mit dem Titel "Die Nutzung von Konzepten der Optik in Sound Design für die Simulation von Raumhall in Games" eigenständig und ohne fremde Hilfe verfasst habe.

Jegliche Inhalte, die aus anderen Quellen wie Texten, Bildern, Audio, Grafiken, Software usw. übernommen wurden, sei es wörtlich oder sinngemäss, sind unter vollständiger Nennung der Urheberschaft und Quelle korrekt zitiert. Darüber hinaus sind sämtliche Passagen, die mithilfe KI-gestützter Programme erstellt wurden, eindeutig gekennzeichnet und mit genauer Bezeichnung des verwendeten Programms und des angewendeten Prompts versehen. Es ist deklariert, wie KI-Tools für die Übersetzung des eigenen Textes, Ideenfindung, Brainstorming oder ähnliches genutzt wurden.

Des Weiteren versichere ich, dass die Arbeit bisher nicht veröffentlicht wurde und in keiner identischen oder ähnlichen Form als Prüfungs- oder Abschlussleistung an einer anderen Hochschule, Ausbildungseinrichtung oder in einem anderen Studiengang eingereicht wurde.

Ich nehme zur Kenntnis, dass eine Verletzung dieser Vorgaben rechtliche und disziplinarische Konsequenzen gemäss § 26 Rahmenordnung für Bachelor- und Masterstudiengänge der Zürcher Hochschule der Künste i.V.m. §§ 8 ff. Verordnung zum Fachhochschulgesetz nach sich ziehen.

Mit meiner Unterschrift bestätige ich die Richtigkeit dieser Angaben:

Vorname: Cédric

Nachname: Schlegel

Schweizerische Matrikelnummer: 19-970-177

Datum: 18.05.2026

Unterschrift: