

Erstellung einer Software zur Audiovisualisierung mittels Echtzeit-Flüssigkeitssimulation

Bachelorarbeit zur Erlangung des akademischen Grades
BACHELOR OF SCIENCE

Jounes Soheil Jedlaoui
Mat.-Nr: 877601

Berliner Hochschule für Technik, Fakultät VI

Betreuer: Prof. Dr. Tramberend
Gutachter: Prof. Dr. Mixdorff

9. Januar 2023

Zusammenfassung

In dieser Arbeit wurde eine Software erstellt, die eine Echtzeit SPH-Flüssigkeits-Simulation mit detaillierter Audioanalyse verknüpft. Die Simulation orientiert sich an Clavet *et al.* [SCP05] und wird gänzlich auf der GPU berechnet. Die durch die Parallelisierung eingesparte Rechenzeit erlaubt es der Simulation mit einer interaktiven Framerate (min 30 Frames) gerendert zu werden. Die Audioanalyse ist auf ein analoges Saiteninstrument, wie eine Gitarre, ausgelegt. Es wird eine Echtzeit-GPU-Implementierung von Nicht-Negativer Matrix Faktorisierung vorgestellt, die zeitdiskrete Informationen aus einem Gitarrensignal in Informationen über Tonart, Dynamik und Tongeschlecht transformiert und dann in die FS gespeist. Das Resultat ist eine Software, die bei Live-Auftritten von Musikern genutzt werden kann.

Inhaltsverzeichnis

1 Einleitung	3
1.0.1 Abkürzungverzeichnis	4
2 Wissenschaftliche Grundlagen	5
2.1 Flüssigkeitssimulation	5
2.1.1 Navier Stokes Gleichungen	5
2.1.2 Euler Simulation	5
2.1.3 Lagrange Simulation	6
2.1.4 Nachbarsuche	7
2.1.5 GPU-Pipelines und Compute Shader	7
2.2 Audioanalyse	8
2.2.1 Musiktheorie	8
2.2.2 Polyphone Analyse	8
3 Implementierung	14
3.1 Architektur	14
3.2 Pipeline	15
3.3 Flüssigkeits Simulation	16
3.3.1 Datenstrukturen	16
3.3.2 Nachbarsuche	16
3.3.3 Simulationsschritt	17
3.3.4 Viskosität	19
3.3.5 Double Density Relaxation	19
3.3.6 Randbedingungen	23
3.4 Audioanalyse	24
3.4.1 Datenstrukturen	24
3.4.2 Signalvorverarbeitung	24
3.4.3 Polyphone Analyse	25
3.4.4 Interpretierung von H	28
3.5 Visualisierung	29
3.5.1 Farbe	29
3.5.2 Interaktion mit Partikeln	29

4	Ergebnisse	31
4.1	Programm	31
4.2	Nachbarsuche	31
4.3	Simulation	33
4.3.1	Flüssigkeitsverhalten	33
4.3.2	Rendern	36
4.4	Audioanalyse	37
4.4.1	Test	37
4.4.2	Aktivierungen	39
4.4.3	Aktivierungen - Tongeschlecht	39
4.4.4	Interaktionen	40
4.5	Gesamt-Performance	42
5	Analyse	43
5.1	Vergleich	43
5.2	Weiterentwicklungen	44
5.3	Erreichtes	45

Kapitel 1

Einleitung

Über die letzten drei Jahrzehnte wurden eine Vielzahl von Lösungsansätzen für die Echtzeit-Modellierung von Flüssigkeiten entwickelt. Diese lassen sich allgemein in zwei Lager aufteilen: Euler'sche Grids und Lagrange Partikelsysteme.

Euler'sche Grids unterteilen die Darstellungsfläche in viele gleichgroße Zellen. Jede Zelle repräsentiert dabei die Verteilung des Druckes im Raum. Diese sehr intuitive Implementierung der Navier-Stokes Gleichungen ist dazu in der Lage, sehr realistische Strömungen abzubilden.

Lagrange Partikelsysteme diskretisieren die Masse der Flüssigkeit, sodass die einzelnen Komponenten im Raum verfolgt werden können. Diese Methode hat u.a. den Vorteil, dass die Konservierung von Masse automatisch gegeben ist, und sich die Attribute einzelner Partikel leichter manipulieren lassen, um bestimmte Visuelle Effekte zu realisieren. Für diese Arbeit orientiere ich mich an Clavet *et al* [SCP05], die eine effiziente Lösung für das Abbilden von Oberflächenspannung, die sie *Double Density Relaxation* nennen, präsentieren. Partikelsysteme im Allgemeinen haben einen typischen Engpass, den Euler'sche Felder nicht aufweisen; Die Nachbarn eines jeden Partikels müssen in jedem Simulationsschritt identifiziert werden. Hierfür gibt es eine Reihe an Lösungsansätzen, die ich im Folgenden erläutern werde.

Bei den meisten Echtzeit-Implementierungen werden, sowie hier auch, nicht alle physikalischen Eigenschaften von Flüssigkeiten abgebildet. Vielmehr ist das Ziel dieser Arbeit eine interaktive Simulation zu erstellen.

Der zweite Kernaspekt dieser Arbeit ist die Echtzeit-Audioanalyse. Dies ist ebenfalls ein weit erforschtes Gebiet. Der Anwendungsfall, für den dieses Programm ausgelegt ist, ist eine Live-Performance, bei der eine Gitarre über ein Audio-Interface an einen Rechner angeschlossen wird. Das rohe Audiosignal wird dann mittels einer angepassten Variante der in [Len13] und [Har12] vorgestellten Methoden analysiert.

Hier präsentiere ich eine Schnittstelle/Transformationspipeline, um die Audiodaten auf eine intuitive Art und Weise mit der Simulation interagieren zu lassen. Genauere Implementierungsdetails werden später spezifiziert. Kernstück dieser Arbeit ist die Implementierung der Audioanalyse und Flüssigkeitssimulation über Compute-Shader. Der gesamte Prozess wird über eine GPU-Pipeline abgebildet. Das hat zum Vorteil, dass

keine überdurchschnittlich mächtige Hardware benötigt wird, um die Software sinnvoll nutzen zu können und ebenfalls eine Ausweitung auf Mobilgeräte denkbar ist.

1.0.1 Abkürzungverzeichnis

Abkürzung	Name
SPH	Smooth Particle Hydrodynamics
FS	Flüssigkeitssimulation
NMF	Nicht-Negative Matrix Faktorisierung
GPU	Graphics Processsing Unit
CPU	Central Processing Unit
DDR	Double Density Relaxation
STFT	Short Time Fourier Transformation
FFT	Fast Fourier Transformation
API	Application Programming Interface

Kapitel 2

Wissenschaftliche Grundlagen

2.1 Flüssigkeitssimulation

2.1.1 Navier Stokes Gleichungen

Die Navier-Stokes Gleichungen wurden in 1822 von Claude Navier und 1845 von George Stokes formuliert. Sie beschreiben die Konservierung von Masse und die Verteilung von Druck und Beschleunigung in einer Nicht-Newtonischen Flüssigkeit. In der Praxis werden sie verwendet, um das Verhalten von Flüssigkeiten und deren Druck-, Beschleunigungs- und Dichteverteilung zu simulieren. Sie finden jedoch auch Anwendung in der Beschreibung astrophysischer Phänomene, sowie dem Verhalten von Gasen.

- (2.1) Beschreibt die Konservierung der Masse
- (2.2) Beschreibt die Konservierung der Beschleunigung. Mit \mathbf{g} = Externe Kraft und μ = Viskosität. (Vereinfacht nach [MMG03])

$$\frac{\delta \rho}{\delta t} + \Delta \cdot (\rho \mathbf{v}) = 0 \quad (2.1)$$

$$\rho \left(\frac{\delta \mathbf{v}}{\delta t} + \mathbf{v} \cdot \Delta \mathbf{v} \right) = -\Delta p \cdot \rho \mathbf{g} + \mu \Delta^2 \mathbf{v} \quad (2.2)$$

2.1.2 Euler Simulation

Bei einer Euler-Simulation wird der Raum in ein einheitliches Gitter aufgeteilt. Die Felder dieses Gitters repräsentieren v: Beschleunigungsfeld, p: Druckfeld, ρ : Dichtefeld. In jedem Simulationsschritt wird dann verfolgt, wie sich diese Attribute über die Zellen verteilen. Eine zum Einstieg geeignete Implementierung wird in [Sta03] präsentiert. Durch diese räumliche Anordnung sind Berechnungen wie die Nachbarsuche trivial. Einige der, bis jetzt, realistischsten Simulationen wurden mit dem Euler'schen Ansatz erzeugt. [CM11]

Eulersche Grids weisen allerdings auch einige Nachteile auf. So ist zum Beispiel die Simulation von Flüssigkeiten mit hohen Geschwindigkeiten problematisch, da sich die Verteilung der Dichte auf die benachbarten Zellen beschränkt. Außerdem muss in jedem

Simulationsschritt jeder Quadrant berechnet werden. Dadurch wird im Worst Case viel Rechenzeit auf leere Felder verschwendet.

2.1.3 Lagrange Simulation

Eine Lagrange Simulation steht in Opposition zu einem Euler'schen Gitter, indem die Werte der Felder auf einzelne Punkte im Raum diskretisiert werden. Diese Punkte werden als Partikel modelliert und im Raum verfolgt. Der Ansatz hat den Vorteil, dass kein leerer Raum überprüft werden muss, Massekonservierung von sich aus gegeben ist und die Substantielle Ableitung eines Partikels der Ableitung der Beschleunigung entspricht. Dadurch kann Formel 2.1 vernachlässigt werden. Der Term $\frac{\delta v}{\delta t} + v \cdot \Delta v$ in 2.2 wird mit $\frac{Dv}{Dt}$ ersetzt. Siehe [MMG03].

Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) wurden 1977 von J. Monaghan und R. Gingold [GR77] und Lucy Li [L.77] entwickelt, um astrophysikalische Phänomene zu modellieren. Die Methode bietet sich ebenfalls zur Simulation von Flüssigkeiten in einem kleineren Maßstab an. Allgemein werden Attribute des Flüssigkeitsfeldes über eine Vielzahl von Partikeln, die verfolgbare, diskrete Samplingpunkte darstellen, abgebildet. Dabei handelt es sich um Attribute wie Dichte, Druck, Beschleunigung, Position, Viskosität und Temperatur. (letzteres wird in dieser Implementierung nicht berücksichtigt).

In den darauffolgenden Jahren wurde SPH für verschiedene Anwendungsfälle, wie z.B. Videospiele oder Medizintechnik weiterentwickelt. Debrun *et al.* demonstrierten eine der ersten einheitlichen Methoden für Echtzeit Simulationen von verformbaren Körpern. In [MD96] wird insbesondere auf die korrekte Wahl von Glättungskernen und die Umsetzung von Oberflächenspannung eingegangen. Zu Letzterem entwickelten Clavet *et al.* [SCP05] *Double Density Relaxation*; wobei das Bilden einer Oberflächenspannung durch einfache distanzabhängige Kräfte, die in Relation zur lokalen Dichte stehen, erzeugt wird. Zudem simulieren Clavet *et al.* Viskoelastizität und Plastizität mithilfe von temporären 'Federn', die Anziehungskräfte zwischen Partikeln repräsentieren. Viskoelastizität wird häufig als dämpfender Skalar, der auf die Beschleunigung angewendet wird implementiert.

Diese Lösungsansätze sind hier besonders interessant, da sie ein visuell glaubhaftes Verhalten bei interagierbaren Framerates erlauben.

SPH findet jedoch auch Anwendung in der Lösung diverser Probleme. Wie z.B. in [DHY22], wo SPH verwendet wird, um das Schmelzen 2-Dimesionaler Körper und deren Übergang in andere Aggregatzustände zu simulieren. Im Allgemeinen kann das SPH-Paradigma in vielen Anwendungsfällen nützlich sein, da es Entwicklungen und Energieaustausche von einer großen Menge diskreter Informationspunkte abbildet. Dies kann auf viele Arten abstrahiert werden.

2.1.4 Nachbarsuche

Ein klassischer Engpass in SPH ist die Nachbarsuche.

Mit der populärste Ansatz, der besonders für CPU-lastige Implementierungen geeignet ist und in [SCP05] und [MD96] genutzt wird, nennt sich *Spatial Hashing*. Hierbei, wird der Raum in ein einheitliches Gitter aufgeteilt und, basierend auf der Position eines Partikels, berechnet in welcher Zelle es sich befindet. Die zugehörige Zelle dient als Index in einer Hash-Tabelle von Buckets. Somit können Partikel in benachbarten Zellen einfach identifiziert werden. Dieser Ansatz weist jedoch u.a. den Nachteil auf, dass auch für alle leeren Zellen Speicher alloziert werden muss. Ďuríkovič *et al.* adressieren dieses Problem in [O8] mit *Cell Indexing*. Hier wird nach dem Berechnen der zugehörigen Zelle, ein Hashwert, der *cellIndex*, berechnet und das zugehörige Partikel in einer Liste abgelegt. Diese Liste wird daraufhin mit Radix-Sort sortiert. Benachbarte Partikel sind nun ebenfalls in dieser Liste benachbart und die Nachbarsuche kann in $O(nk)$ absolviert werden.

Andere relevante Lösungen implementieren K-Nearest-Neighbors auf der GPU mithilfe von Nvidias CUDA-Bibliothek(siehe [LA]). Diese bietet effiziente Sortier- und Merge-Funktionalitäten, die, mithilfe der Parallelisierung der GPU, die sonst beträchtliche Komplexität ausgleichen kann. Die Metal-API bietet keine solche Bibliothek an, was die Umsetzung eines GPU-Ansatzes erschwert. Wie in [NSG08] gezeigt, ist Radixsort, auch ohne Nutzung der CUDA-Bibliothek, mit einer guten Laufzeit möglich.

2.1.5 GPU-Pipelines und Compute Shader

Compute Shader sind verwandt mit Fragment- und Vertexshadern. Während Letztere spezifische Rückgabewerte aufweisen, die zum Rendern genutzt werden, haben Compute Shader keinen Rückgabewert und können eine Vielzahl von Datentypen bearbeiten. Somit ist es möglich, einen Großteil der Berechnungen auf die GPU auszulagern.

Seit ihrer Konzeption stehen Echtzeit-Partikelsysteme vor der Herausforderung der Berechenbarkeit zu interaktiven Framerates. Eine physikalisch akurate Simulation muss eine ganze Reihe mehr Faktoren berücksichtigen, als eine, die ein rein visuell überzeugendes Ergebnis zum Ziel hat. Die Vielzahl an Berechnungen und voneinander abhängigen Integralen, sowie die vielen verschiedenen Eigenschaften von Flüssigkeiten/Gasen wie Turbulenz, Oberflächenspannung und Viskosität summieren sich zu einem Rechenaufwand pro Simulationsschritt, der die Leistungsfähigkeit durchschnittlicher Maschinen schnell überschreitet. Glücklicherweise ist es, mit der fortschreitenden Entwicklung dedizierter Hardware, nicht mehr notwendig, alle Berechnungen iterativ auf der CPU durchzuführen. Stattdessen kann mit Hilfe von Compute Shadern die Berechnung der Interaktionen der Partikel parallelisiert werden. Die Pipeline-Architektur von GPUs ermöglicht es, einen Simulationsschritt in einzelne Kernel-Berechnungen zu unterteilen und für jedes Partikel in einem eigenen Thread simultan zu berechnen. Somit können deutlich mehr Partikel gleichzeitig abgebildet werden und dann mittels Fragment-Shadern entsprechend visuell angepasst werden. Im fertigen Programm ist es perspektivisch dann möglich die Flüssigkeit in unterschiedlichen Grafikstilen darzustellen.

Durch diese Architektur ist eine effiziente Speicherzugriffsverwaltung essentiell und es muss auf diverse atomare Operationen, sowie voneinander distinkte GPU-Command Encoder, gesetzt werden, um widersprüchliches Schreiben in dieselben Buffer zu vermeiden und die Audioanalyse und Simulation synchron zu halten.

In dieser Implementierung werden Compute Shader für die Nachbarnfindung, sowie Interaktionen zwischen Partikeln und Matrix Berechnungen in der Audioanalyse genutzt.

2.2 Audioanalyse

2.2.1 Musiktheorie

Um spätere Erläuterungen zur Interpretation der Daten aus der Audioanalyse zu vereinfachen, möchte ich hier kurz auf einige Grundlagen der Musiktheorie eingehen. Musik setzt sich aus einzelnen Noten zusammen, die sich durch Attribute wie Tonhöhe, Dauer, Dynamik und Timbre beschreiben lassen.

Die *Tonhöhe* ist für das Vorhaben in dieser Arbeit die wichtigste Eigenschaft und repräsentiert die Frequenz der Schwingung, die die Note ausmacht. Über die Tonhöhe können Noten relativ zueinander als höher oder tiefer klassifiziert werden und in 'Notenklassen' (z.B. alle Vorkommen der Note C) unterteilt werden. Es spannen sich ca. 7 Oktaven à 12 Noten über die 88, in westlicher Musik, verwendeten Noten. Und in jeder Oktave findet sich je eine Instanz einer Notenklasse. Hier ist es wichtig zu beachten, dass Instanzen der selben Notenklasse auf unterschiedlichen Oktaven in einem Spekrogramm häufig die selben Harmonien aufweisen (siehe Ergebnisse in 4.4.)

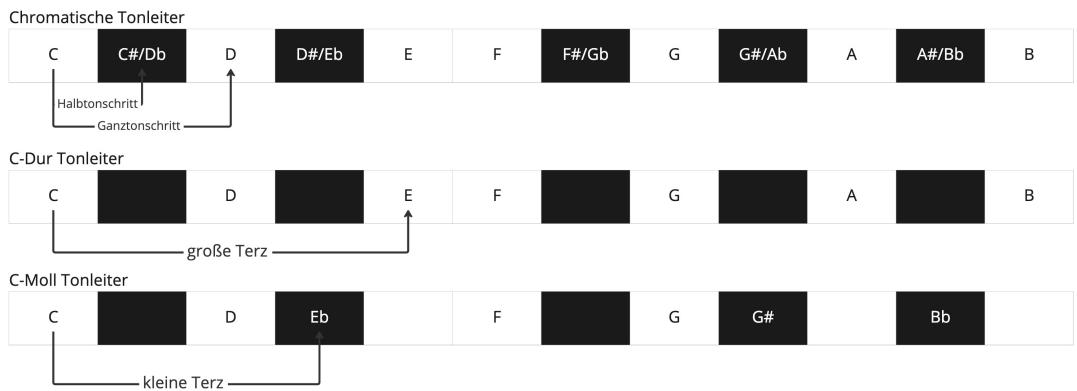
Die *Dauer* und *Dynamik* einer Note beschreiben je, wie lange eine Note in Zählzeiten gespielt wird, und wie laut die Note zu hören ist.

Eine *Tonleiter* beschreibt eine Menge von Noten, die in bestimmten *Intervallen* zueinander stehen. Abb. 2.1 zeigt ein Beispiel einer chromatischen Tonleiter und die C-Dur und C-Moll Tonleitern, die sich aus ihr konstruieren lassen.

Intervalle beschreiben wie viele Noten Abstand zwischen zwei Noten besteht. Die Größe eines Intervalls ist ausschlaggebend in der Bestimmung der Tonart/verwendeten Tonleiter, sowie zur Erkennung eines Akkords. Ein einzelner Notenschritt wird als Halbtonschritt bezeichnet, während ein Schritt zur nächsten 'vollwertigen' Note Ganztonschritt genannt wird. Aus diesen beiden Komponenten lassen sich alle anderen Intervalle ableiten. Eine große Terz vom Grundton aus deutet z.B auf eine Dur-Tonleiter hin. Ein Dur-Akkord besteht aus einem Grundton, seiner großen Terz und seiner Quinte. Im Folgenden wird die Art des Akkords als Tongeschlecht bezeichnet. Grundsätzlich lässt sich sagen, dass Dur als glücklich und Moll als traurig gewertet wird.

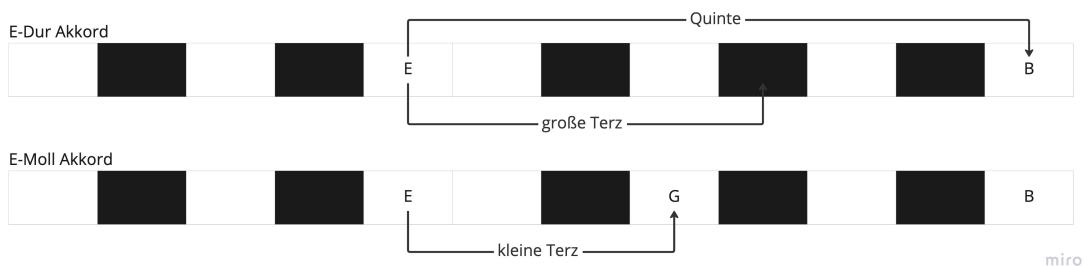
2.2.2 PolypHONE Analyse

Die Analyse von analogen Audiosignalen ist in einer Vielzahl von Anwendungsfällen nützlich. Ziel einer Audioanalyse kann z.B. automatische Transkription, Erkennung der



miro

Abbildung 2.1: Illustrierung Tonleitern und Intervalle



miro

Abbildung 2.2: Beispiel: E-Dur Akkord mit großer Terz und E-Moll Akkord mit kleiner Terz

Tonhöhe zum Stimmen von Saiteninstrumenten oder Beat-Detection für Lichtsysteme die in Liveperformances sein. Jeder dieser Use Cases hat andere Anforderungen an die Analyse. Grundsätzlich ist eine Unterscheidung zwischen mono- und polyphoner Analyse des Frequenzbandes, sowie zeitdiskreter Erkennung von Amplituden-Peaks sinnvoll.

J. Harquist liefert eine aussagekräftige Erklärung der Funktionsweise von Audiosampling in [Har12]. Hier eine übersetzte Zusammenfassung:

Sound setzt sich aus einem kontinuierlichen, aus Schallwellen bestehenden Signal zusammen. Damit solch ein Signal von einem Rechner verarbeitet werden kann, muss es in kleinere Bestandteile heruntergebrochen werden. Diese zeitdiskreten Samples fassen die Informationen des Quellsignals über eine definierte Zeitspanne zusammen. Üblich ist eine Samplingfrequenz von 44100 kHz, was dem CD-Standart entspricht. Um diese Samples zu analysieren, werden üblicherweise eine feste Anzahl an Samples in Zeitfenstern zusammengefasst. Je größer diese Fenster, desto mehr Information über das Audiosignal wird übergeben. Das bedeutet jedoch auch, dass kleinere Änderungen im Signal unbemerkt bleiben können. Deshalb ist es üblich einen Analyseschritt über mehrere kleinere, überlappende Zeitfenster auszuführen.

So können Änderungen im Signal in Relation zum vorherigen Zustand des Signals besser erkannt werden. Besonders wenn Information über die Entwicklung einer Progression erfasst werden soll, ist diese Methodik essentiell. Während Beat-Detection relativ trivial ist, ist das Erkennen unterschiedlicher gespielter Noten deutlich komplizierter.

Der Prozess ist an [Har12] angelehnt. In dem Paper stellt er eine Methode vor, anhand derer feingliedrige Analyse eines analogen Gitarrensignals möglich ist. Seine Methode nutzt Null-Faktor-Matrix-Multiplikation, um einzelne Komponenten eines polyphonen Audiosignals zu erfassen. Somit können Akkorde und die, sie bildenden Noten, erkannt und zueinander in Relation gestellt werden, um Informationen über die Dynamik, den Timbre und Tonleitern des Gespielten zu erfassen. Für einen Musiker, der das hier vorgestellte Programm nutzen möchte, soll es dadurch möglich sein, das visuelle Feedback der Simulation analog zur gespielten Musik intuitiv verstehen zu können.

STFT - Short Time Fourier Transformation

Die Fourier Transformation ist ein essentieller Grundstein der Signalverarbeitung. Sie erlaubt es eine Funktion in ihr Spektrum umzuwandeln. Die Fourier Transformation ist so bedeutend, weil sie sowohl vorwärts, als auch rückwärts funktioniert. Eine jede Funktion kann in $O(n)$ in seine Komponenten aufgeteilt werden.

In der Praxis existieren verschiedene Varianten des Algorithmus, die für unterschiedliche Anwendungsfälle geeignet sind. Da die Fourier Transformation den Frequenzbereich einer Funktion selbst zurückgibt, gibt es ebenfalls Varianten, die mit diskreten Werten arbeiten. Mit niedriger Präzision und oft in der Bildverarbeitung genutzt existiert die

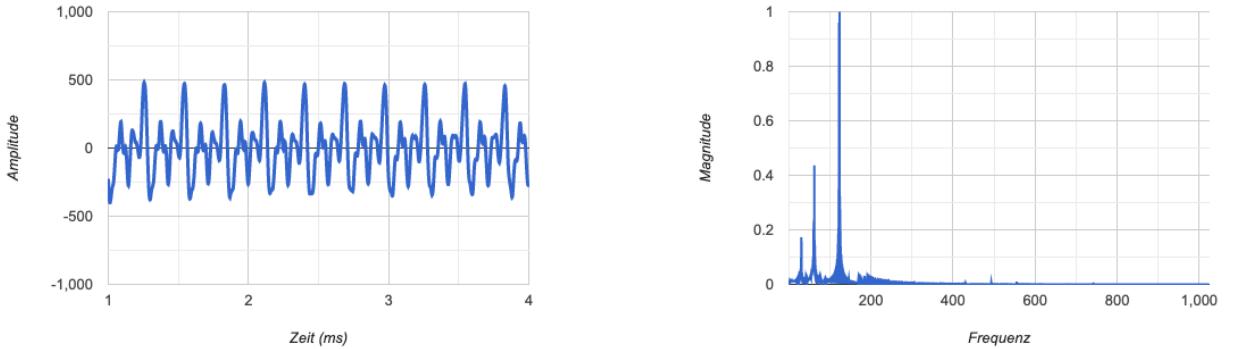


Abbildung 2.3: Zeitdiskrete Schwingung E5

Abbildung 2.4: Frequenzspektrum E5

Diskrete Kosinus Transformation, die einzig reelle Argumente verarbeitet.

Mit einer höheren Präzision zeigt sich die Diskrete Fourier Transformation, die anstatt einer Funktion selbst, das Frequenzspektrum eines diskreten Signals transformiert. Die Fast Fourier Transformation ist eine optimierte Version des DFT Algorithmus, die die Berechnungen in $O(n \log n)$ ausführen kann. In [Hec95] werden die Charakteristika der Fourier Transformation und FFT detailliert dargelegt. Formel 2.3 beschreibt die Diskrete Fourier Transformation:

$$A_k = \sum_{n=0}^{N-1} W_N^{kn} a_n \quad (2.3)$$

STFT ist eine Applikation der FFT die das Frequenzspektrum eines zeitdiskreten Signals über einen Zeitraum berechnet. Anstatt ein langes Signal mit einer normalen Fourier Transformation zu bearbeiten, wird das Signal in kleinere Abschnitte, auf denen je eine FFT durchgeführt wird, unterteilt. STFTs haben u.a. den Vorteil, dass feinere Details des Signals mit größerer Gewichtung in die Analyse eingehen können und sich Informationen über den zeitlichen Verlauf des Frequenzspektrums ableiten lassen. Für einen Echtzeitanwendungsfall kommt in jedem Fall nur eine FFT-Variante mit $O(n \log n)$ in Frage, da herkömmliche FT eine Laufzeit von $O(n)$ aufweist.

Wie in Abbildungen 2.1 und 2.2 zu erkennen ist, wird das zeitdiskrete Signal in ein Frequenzspektrum ohne temporale Informationen umgewandelt.

NMF - Nicht-Negative Matrix Faktorisierung

Nicht-Negative Matrix Faktorisierung wird genutzt um große Datensätze in eine Faktor- und eine Bibliothekskomponente aufzuteilen. Sie wurde zuerst von Lee und Seung in [LS00] vorgestellt und ist stark inspiriert durch Paatero und Tappers vorherige Erkenntnisse zu Positiver Matrix Faktorisierung in [PT94].

Die zugrundeliegende Idee ist die, dass sich komplexere Datenstrukturen als additive

oder multiplikative Relation zwischen einfacher zu verstehenden Komponenten ausdrücken lassen. In [WZ13] wird NMF wie folgt definiert:

Sei eine $M \times N$ -dimensionale, nicht-negative Matrix $V = [v_1, v_2, \dots, v_N] \in \mathbb{R}_{\geq 0}^{M \times N}$ gegeben. Das Ziel von NMF ist es, V in eine nicht-negative $M \times R$ Basismatrix $W = [w_1, w_2, \dots, w_R] \in \mathbb{R}_{\geq 0}^{M \times R}$ und eine nicht-negative $N \times R$ Koeffizientenmatrix $H = [h_1, h_2, \dots, h_N] \in \mathbb{R}_{\geq 0}^{R \times N}$ aufzuteilen, sodass $V \approx WH$. $\mathbb{R}_{\geq 0}^{M \times N}$ sei das Set von $M \times N$ nicht-negativer Matritzen.

Diese allgemeine Form von NMF findet Anwendung [Gil14] in Feldern wie Maschine Learning, Data Mining und Analyse, Bildanalyse und Dekomposition, Wirtschaftswissenschaftlicher Analyse oder Spracherkennung, sowie in der polyphonen Audioanalyse. Für unsere Zwecke ist Letzteres am interessantesten. Während einzeln gespielte Noten recht einfach, durch Bestimmung des globalem Maximums der FFT eines Audiosignals, gefunden werden können, ist eine andere Herangehensweise vonnöten, um die Teilfrequenzen, die das Signal bilden, zeitdiskret zu identifizieren.

Smaragdis und Brown stellen 2003 in [SB03] eine konvolutive NMF-Methode vor, um ganze, polyphone Passagen mittels NMF zu analysieren und eine Transkription zu erstellen. Das zeitdiskrete nicht-negative Magnitudenspektrogramm des Inputsignals wird als Matrix V ausgedrückt und in einer NMF, mit Rang R = Anzahl der zu erkennenden Notensamples, faktorisiert. Die Menge der Magnitudenpektogramme der Notensamples ist die Bibliotheks- oder Basismatrix W und die zeitdiskreten Aktivierungen der Samplesnoten aus W wird durch die Aktivierungs- oder Koeffizientenmatrix H ausgedrückt.

Für eine bildliche Darstellung siehe Abb 2.5. In blau das Spektrogramm eines Inputs. Daneben sind 3 Sample FFTs von einzelnen Gitarrennoten zu sehen. Ziel der NMF ist es eine Kombination aus den Samples zu finden, die das Inputspektrogramm approximieren. Abb 2.6 zeigt, wie die summierten FFTs der einzelnen Sampleschwingungen die Input FFT rekonstruieren können. Auf diesem Konzept baut die Methode in 3.4 auf.

Während Smaragdis' und Browns Methode gute Ergebnisse liefert, ist deren Algorithmus nicht auf eine Echtzeitauswertung eines kontinuierlichen Signals ausgelegt. Hierfür war eine geringfügige Anpassung des Ansatzes notwendig, die in [Har12] vorgestellt wird und auf die ich genauer in 3.4. eingehen werde.

*Abb. 2.5 und 2.6 angelehnt an [Har12] Fig. 3.1

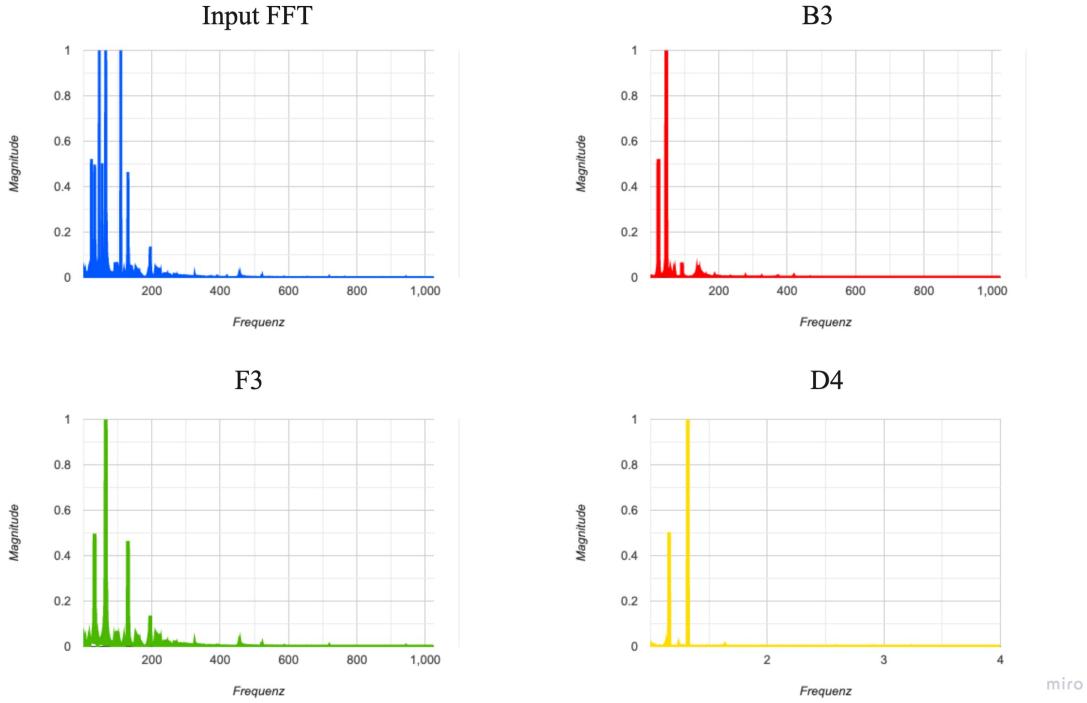


Abbildung 2.5: Input- und Sample-FFTs

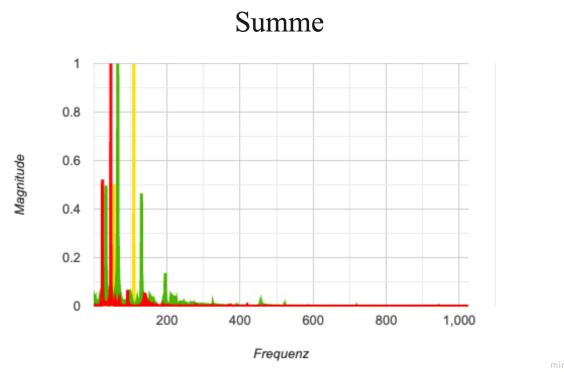


Abbildung 2.6: Die drei FFTs summiert sind gleich der Input-FFT

Kapitel 3

Implementierung

Im Folgenden werde ich auf Implementierungsdetails eingehen und erklären, wie die einzelnen Komponenten funktionieren und miteinander interagieren. Zum besseren Verständnis sind anfolgend alle verwendeten Variablen und deren Datentypen aufgelistet.

Name	Datentyp
<i>particles</i>	Particle MTLBuffer
<i>velocityField</i>	2D-Textur, rgbaInt
<i>neighbors</i>	2D-Textur, Int
<i>particleCount</i>	Partikel in der Simulation, Float
<i>sampleCount</i>	Auflösung des Frequenzbandes, Int
<i>h</i>	Interaktionsradius der Partikel, Float
<i>id</i>	1d-Thread-Id, uint
<i>id2d</i>	2d-Thread-Id, uint2
<i>t</i>	Zeitkonstante, Float
<i>g</i>	Gravitationskonstante, Float
<i>P₀</i>	Restdichte, Float
<i>k</i>	Stauchfaktor, Float
<i>k_{near}</i>	Stauchfaktor für nähere Partikel, Float
<i>D</i>	Verschiebung, 2D-Vector, Float
σ, β	Abhängigkeit zwischen Viskosität und Beschleunigung, Float

3.1 Architektur

Die Software setzt sich aus 3 miteinander kommunizierenden Komponenten zusammen. Sowohl die Audioanalyse als auch die FS nutzen die parallele Architektur der GPU. Die Konstruktion der Datenstrukturen und Modelle für die Analyse und Simulation, sowie die Steuerung der Pipeline werden in Swift mithilfe der Metal API durchgeführt. Auf der CPU werden Buffer alloziert, die Daten aus der Analyse bei musikalischen Events interpretiert und in die FS gespeist. Die Kernelfunktionen, die von der GPU

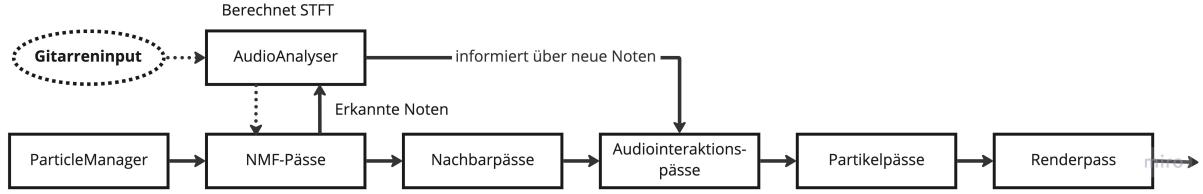


Abbildung 3.1: Aufbau der Pipeline

ausgeführt werden, sind in Metal Shading Language (MSL) formuliert. Diese bietet Zugriff auf Compute Shader um fragment- und vertexunabhängige Berechnungen zu programmieren. Jede Instanz eines Kernels kriegt, neben den zu verarbeitenden Daten, die eigene Thread Id übergeben, anhand derer gezielte Berechnungen ausgeführt werden können.

3.2 Pipeline

Die Berechnungen für die Flüssigkeitssimulation und Audioanalyse sind eng miteinander verbunden. Die Klasse **ParticleManager** verwaltet die gesamte GPU-Pipeline und die Partikelberechnungen. Sie hat außerdem eine **AudioAnalyser**-Instanz, die einen Audiostream zur Inputquelle aufbaut. Dieser Stream läuft unabhängig von den Renderzyklen der Pipeline und bietet der Klasse **NMFSolver** neues Input, sobald es auf der CPU verarbeitet wurde. Die Matrixmultiplikationen der NMF sind der erste Schritt in der GPU-Pipeline. Nachdem eine Iteration des Multiplikativen Updates berechnet wurde, folgen die, durch **AudioAnalyser** informierten, inputabhängigen Pässe, die das Beschleunigungsfeld *velocityField* modifizieren. Erst die Pässe für die Nachbarsuche terminiert haben, werden die *velocityField* Beschleunigungen in der *Double Density Relaxation* auf die Partikel angewendet.

Während die FS mit möglichst vielen Simulationsschritten pro Sekunde ausgeführt werden soll, würde eine vergleichbar hochfrequente Speisung von Audioimpulsen zu einem sehr chaotischen Verhalten der Partikel führen. Deshalb werden solche Impulse nur bei bestimmten Events an die FS übergeben. (genauer beschrieben in 3.4). Dadurch wird Rechenzeit gespart und der Simulation wird Zeit gegeben, um auf Impulse zu reagieren. Aus Abb. 3.1 ist die Hierarchie der Pipeline Pässe zu entnehmen.

Algorithm 1: GPU-Pipeline

- 1: readAudioInput
 - 2: performPolytonalAudioanalysis
 - 3: identifyNeighborParticles
 - 4: applyForces
 - 5: renderParticles
-

3.3 Flüssigkeits Simulation

Die Implementierung der Simulation entspricht dem in Clavet *et al.* [SCP05] dargelegten Algorithmus. Ein Simulationsschritt wird in Alg. 1 dargelegt.

3.3.1 Datenstrukturen

Alle Datentypen werden mithilfe von C-structs abgebildet:

Partikel

```
struct Particle {
    float2 position;
    float2 prevposition;
    float2 velocity;
    float2 dx;
    float p;
    float pnear;
}
```

Der Struct **Particle** beschreibt ein Partikel in der SPH-Simulation. *position*, *prevposition*, *velocity* und *dx* sind 2d-Vektoren, die die aktuelle und letzte Position des Partikels, sowie seine aktuelle Beschleunigung und druckabhängige Transformation speichern. *p* und *pnear* beschreiben Druck- und Dichteverhältnisse zu Nachbarpartikeln. Die Menge an Partikeln wird beim Starten der Simulation in einem regulären Gitter und ohne Beschleunigung initialisiert. Für den daraus resultierenden Array wird dann ein MTLBuffer alloziert, der die Partikelinstanzen auf der GPU zur Verfügung stellt. Metal unterscheidet hier zwischen 'managed' und 'private' Storage um analoge Kopien der Buffer sowohl der GPU, als auch der CPU zur Verfügung zu stellen. Um die Speichernutzung zu minimieren sollten Daten, wenn möglich, nur auf der GPU genutzt werden.

3.3.2 Nachbarsuche

Die parallele Architektur der GPU hat einige Limitationen zur Folge, die die Wahl einer Methode einschränken. Das Allozieren von Buffern und Command Buffern über die CPU ist zeitaufwändig und sollte demnach nicht in jedem Simulationsschritt geschehen. Da sich die Anzahl von Nachbarn eines Partikels in jedem Schritt ändern kann, würde intuitiv ein dynamischer Array von Arrays naheliegen, der vor einem Simulationsschritt mit den Nachbarindexen aller Partikel gefüllt oder aktualisiert wird. Metal unterstützt zum jetzigen Zeitpunkt jedoch nur statische Buffergrößen, was bedeutet, dass Linked-List oder Hash Methoden nicht ohne weiteres in Frage kommen. In der Literatur ist es deshalb üblich eine bestimmte Anzahl von Nachbarn zu identifizieren.

Im Laufe der Entwicklung habe ich zwei verschiedene Nachbar-Algorithmen implementiert, deren unterschiedliche Ergebnisse ich in 4.2.1 vorstelle. Für diese Implementierung

habe ich mich für einen Brute-Force Ansatz entschieden.

In der Pipeline an Stelle 2 steht die Compute Kernel Funktion *findNeighbors()*. Mit einer Laufzeit von $O(n^2)$ wird in einem Pass für jedes Partikelpaar p_i, p_j bestimmt, ob es sich innerhalb des Interaktionsradius befindet. Bei einem Treffer schreibt die Funktion den Nachbarindex in das Pixel mit den Koordinaten $(index(p_i), index(p_j))$.

Algorithm 2: Find Neighbors

```

Data: particles, neighbors, id2d
1 for parallel each Particle  $p_1$  do
2   for parallel each Particle  $p_2$  do
3     if  $p_1 \neq p_2$  then
4        $r = |p_2.pos - p_1|;$ 
5       if  $r \leq H$  then
6         | neighbors.write( $r, id2d$ );
7         end
8       end
9       else
10      | neighbors.write(0, id2d);
11      end
12    end
13  end
14
```

Die Nachbartextur wird anschließend dem Command Encoder übergeben, der die Partikelinteraktionen berechnet.

3.3.3 Simulationsschritt

Die simulierte Flüssigkeit wird durch zeit- und raumdiskrete Partikel abgebildet. In jedem Simulationsschritt werden zuerst fundamentale Kräfte, wie Schwerkraft g , zur Beschleunigung des Partikels addiert. Daraufhin bremst der Viskositätsmechanismus die Beschleunigung zwischen Nachbarpartikeln aus (siehe 3.3.4).

Anschließend wird die Position aus dem letzten Simulationsschritt gespeichert und die aktuelle Position anhand der Beschleunigung berechnet. Der Druckausgleich geschieht anfolgend mittels *Double Density Relaxation* zwischen allen Nachbarpartikeln. In diesem Schritt wird die Position des Partikels ebenfalls manipuliert. Als Letztes wird die vorherige Position verwendet um die nächste Beschleunigung zu berechnen. Wie in [SCP05] erklärt hat ein solches implizites Iterationsschema den Vorteil, dass Beschleunigungen nicht direkt von externen Kräften beeinflusst werden, was zu einer stabileren Simulation führt. Der Algorithmus für einen Simulationsschritt lautet wie folgt:

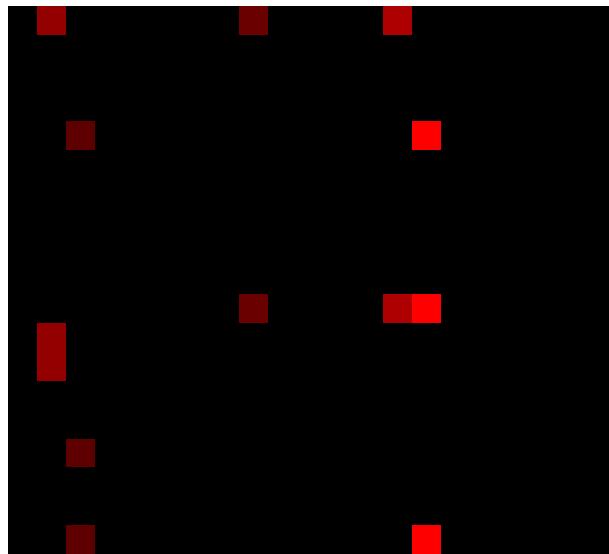


Abbildung 3.2: *neighbors* Textur für 20 Partikel. Der rot-Teil des Pixels indiziert ob es ein Nachbarpartikel ist.

Algorithm 3: Simulationsschritt

```
1: for parallel each Particle p do
2:    $p.velocity \leftarrow p.velocity + \Delta t g$ 
3: end for
   applyViscosity()
4: for parallel each Particle p do
5:    $p.prevposition \leftarrow p.position$ 
6: end for
   doubleDensityRelaxation() {▷ Alg. 5}
7: for parallel each Particle p do
8:    $p.velocity \leftarrow (p.position - p.prevposition)/\Delta t$ 
9: end for
```

3.3.4 Viskosität

Die Berechnung der Viskosität geschieht, analog zur Berechnung der lokalen Dichte, in einem Pass über alle Nachbaren $p_j \in neighbors[i]$ aller Partikel p_i . Wie in [SCP05] vorgestellt, werden paarweise Impulse zwischen Nachbarpartikeln ausgetauscht, die die *velocity* beider Partikel modifizieren.

Algorithm 4: applyViscosity

Data: $particles, neighbors, \Delta t, H, id2d$

```

1 Particle  $p_i = particles[id2d.x]$ ;
2  $p_{jindex} = neighbors[id2d]$ ;
3 if  $p_{jindex} > 0$  then
4   Particle  $p_j = particles[p_{jindex}]$ ;
5    $r_{ij} = |p_j.position - p_i.position|$ ;
6    $q = r_{ij}/H$ ;
7   if  $q < 0$  then
8      $\hat{r}_{ij} = normalize(p_j.position - p_i.position)$ ;
9      $u = (p_i.velocity - p_j.velocity) \cdot \hat{r}_{ij}$ ;
10    if  $length(u) > 0$  then
11       $I = \Delta t \cdot (1 - q) \cdot (\sigma u + \beta u^2)$ ;
12       $p_i.velocity = p_i.velocity - \frac{I}{2}$ ;
13       $p_j.velocity = p_j.velocity + \frac{I}{2}$ ;
14       $particles[id2d.x] = p_i$ ;
15       $particles[p_{jindex}] = p_j$ ;
16    end
17  end
18 end

```

Angelehnt an übliche SPH-Techniken [MD96], wird bestimmt wie schnell sich p_j auf p_i zubewegt, indem die Differenz der Beschleunigung am Normalvektor zwischen beiden Partikeln projiziert wird. Die Stärke des Impulses hängt vom linearen Kernel $(1 - r_{ij}/h)$ ab, während die Faktoren $(\sigma u + \beta u^2)$ die lineare und quadratische Abhängigkeit von der Beschleunigung bestimmen. Eine hohe Viskosität wird mit einem hohen σ erreicht. Laut Clavet *et al.* verhindert der quadratische Term, dass Partikel sich ineinander bewegen indem hohe Beschleunigungen reduziert werden ohne, dass interessantes Verhalten, wie Tropfenbildung, verloren geht.

3.3.5 Double Density Relaxation

Double Density Relaxation ist eine von Clavet *et al.* [SCP05] entwickelte Erweiterung des SPH-Paradigmas. Die Verteilung von Dichte und Druck wird anhand von Wechselwirkungen zwischen Nachbarpartikeln abgebildet. In dieser Implementierung wird davon ausgegangen, dass alle Partikel die selbe Masse haben. Deshalb kann sie aus den

Berechnungen ausgelassen werden.

Ziel der DDR ist es, die Anziehungs- und Abstoßungseffekte zwischen Partikeln zu modellieren.

Dichte und Druck

Verschiebungen zwischen Partikeln röhren größtenteils von lokalem Druckausgleich. Um diesen abzubilden, werden zwei Pseudokräfte P und P_{near} definiert, die die Anziehungs- und Abstoßungskräfte eines jeden Partikels repräsentieren. Diese Kräfte stoßen Nachbarn, die nahe am Partikel sind ab und ziehen Nachbarn, die weiter entfernt sind, an. Allein durch diese beiden Kräfte ist es möglich Tröpfchenbildung und Oberflächenspannung abzubilden.

Die lokale Dichte ρ_i ist keine echte physikalische Größe, sondern repräsentiert lediglich eine Relation zwischen einem Partikel p_i und seinen Nachbarn $\in N$:

$$\rho_i = \sum_{j \in N[i]}^n (1 - r_{ij}/h)^2 \quad (3.1)$$

mit $r_{ij} = |p_i.position - p_j.position|$, h = Interaktionsradius der Partikel. In Clavet *et al.* [SCP05] hat sich hierfür ein quadratischer Kernel $(1 - r_{ij}/h)^2$ als visuell überzeugend herausgestellt. ρ_i wird in einem Compute Shader Pass über die Nachbarn aller Partikel in *computeDensities* summiert.

Der Druck P in p_i wird ebenfalls als Pseudokraft ausgedrückt. Der Einfachheit halber wurde die übliche normalisierende Konstante durch den Stauchwert k , sowie die Restdichte ρ_0 zusammengefasst. P wird in einem Pass *computePressures* über alle Partikel berechnet.

$$P_i = k(\rho_i - \rho_0) \quad (3.2)$$

Dieser einzelne Druckwert verhindert jedoch kein Clustering der Partikel. Partikel mit vielen Nachbarn können diese so nahe an sich heranziehen, dass die Restdichte schnell erreicht wird. Dies hat zur Folge, dass sich die Flüssigkeit in voneinander unabhängige Cluster aufteilt. Um dieses Verhalten zu vermeiden führen Clavet *et al.* eine weitere Pseudokraft *Near-Dichte* ρ_i^{near} ein. ρ_i^{near} wird mit einem anderen, kubischen Kernel berechnet und dient dazu, nahe Partikel abzustoßen. Somit ist ein Mindestabstand zwischen Partikeln garantiert. Ein Vorteil für die Audiovisualisierung entsteht dadurch, dass es möglich ist das Verhalten der Partikel noch genauer zu beeinflussen.

$$\rho_i^{near} = \sum_{j \in N[i]}^n (1 - r_{ij}/h)^3 \quad (3.3)$$

Der *Near-Druck* P_i^{near} ignoriert die Restdichte und ist definiert als:

$$P_i^{near} = k^{near} \rho_i^{near} \quad (3.4)$$

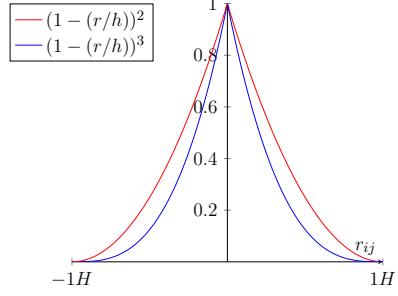


Abbildung 3.3: Smoothing Kernel für P_{near} und P

Um nun die Verschiebung zwischen Nachbarpartikeln zu bestimmen, wird ein Pass über alle Nachbarn j eines Partikels p_i durchgeführt. Die Verschiebung D_{ij} entspricht:

$$D_{ij} = \Delta t^2 (P_i(1 - r_{ij}/h) + P_i^{near}(1 - r_{ij}/h)^2) \hat{r}_{ij} \quad (3.5)$$

Algorithmen

Algorithmus 5 zeigt, wie die CPU die Compute Passes zur *Double Density Relaxation* steuert. Algorithmen 6, 7, 8 und 9 beschreiben parallele Compute Kernel Funktionen. Um einen float im Particle struct zu sparen, werden P und P_{near} in ρ und ρ_{near} in *computePressures* zusammengefasst.

Alle 4 Compute Passes werden auf demselben, global nutzbaren Command Encoder ausgeführt. Dies hat den Vorteil, dass Synchronisation mit dem Fortschritt des Simulationsschrittes über **AudioAnalyser** möglich ist.

Algorithm 5: Double Density Relaxation

Data: $particles, neighbors, P_0, k, k_{near}, id$

```

1 for parallel each Neighbor  $pj$  of Particle  $pi$  do
2   | computeDensities()                                 $\triangleright$  Alg. 6
3 end
4 for parallel each Particle  $pi$  do
5   | computePressures()                                $\triangleright$  Alg. 7
6 end
7 for parallel each Neighbor  $pj$  of Particle  $pi$  do
8   | computeDisplacements()                           $\triangleright$  Alg. 9
9 end
10 for parallel each Particle  $pi$  do
11  | applyDisplacements()                            $\triangleright$  Alg. 8
12 end

```

Algorithm 6: computeDensities

Data: $\text{particles}, \text{neighbors}, H\text{id2d}$

1 Particle $p_i = \text{particles}[\text{id2d}.x]$;
2 $p_{jindex} = \text{neighbors}[\text{id2d}]$
3 **if** $p_{jindex} > 0$ **then**
4 Particle $p_j = \text{particles}[p_{jindex}]$;
5 $r_{ij} = |p_j.\text{position} - p_i.\text{position}|$;
6 $q = r_{ij}/H$;
7 **if** $q < 1$ **then**
8 $p_i.\rho = p_i.\rho + (1 - q)^2$;
9 $p_i.\rho_{near} = p_i.\rho_{near} + (1 - q)^3$;
10 $\text{particles}[\text{id2d}.x] = p_i$;
11 **end**
12 **end**

$\triangleright(1)$

$\triangleright(1)$: Nachbarn haben einen nicht-negativen r-Wert

Algorithm 7: computePressures

Data: $\text{particles}, P_0, k, k_{near}, \text{velocityField}, id$

1 Particle $p_i = \text{particles}[id]$;
2 $modK = \text{velocityField}[p_i.\text{position}]$;
3 **if** $modK \neq 0$ **then**
4 $k = modK$
5 **end**
6 $p_i.\rho = k \cdot (p_i.\rho - P_0)$
7 $p_i.\rho_{near} = k_{near} \cdot p_i.\rho_{near}$;
8 $\text{particles}[id] = p_i$

$\triangleright(1)$

$\triangleright(2)$

$\triangleright(1)$: Modifiziertes k aus der Audioanalyse. Wird nur für dieses Partikel modifiziert.

$\triangleright(2)$: ρ und ρ_{near} speichern lokalen Druck. Dichte wird in diesem Simulationsschritt nicht mehr benötigt.

Algorithm 8: applyDisplacements

Data: $\text{particles}, id$

1 Particle $p_i = \text{particles}[id]$;
2 $p_i.\text{position} = p_i.\text{position} + p_i.dx$;
3 $\text{particles}[id] = p_i$

Algorithm 9: computeDisplacements

Data: $\text{particles}, \text{neighbors}, \Delta t, H, id2d$

```
1 Particle  $p_i = \text{particles}[id2d.x]$ ;  
2  $p_{jindex} = \text{neighbors}[id2d]$ ;  
3 if  $p_{jindex} > 0$  then  
4   Particle  $p_j = \text{particles}[p_{jindex}]$ ;  
5    $r_{ij} = |p_j.\text{position} - p_i.\text{position}|$ ;  
6    $q = r_{ij}/H$ ;  
7   if  $q < 1$  then  
8      $\hat{r}_{ij} = \text{normalize}(p_j.\text{position} - p_i.\text{position})$ ;  
9      $D_{ij} = \Delta t^2(p_i.P \cdot (1 - q) + p_i.P_{\text{near}} \cdot (1 - q)^2) \cdot \hat{r}_{ij}$ ;  
10     $p_j.\text{position} = p_j.\text{position} + \frac{D_{ij}}{2}$ ;  
11     $p_i.dx = p_i.dx - \frac{D_{ij}}{2}$ ;  
12     $\text{particles}[id2d.x] = p_i$ ;  
13     $\text{particles}[p_{jindex}] = p_j$ ;  
14  end  
15 end
```

3.3.6 Randbedingungen

Eine richtige Wahl der Randbedingungen ist essentiell für ein korrektes Flüssigkeitsverhalten. In dieser Implementierung haben zwei Mechanismen Einfluss auf das Randverhalten. Einerseits wird am Ende von *computeNextVelocity()* überprüft, ob sich die aktuelle Position außerhalb des Fensters befindet. Wenn dies der Fall ist, wird die Beschleunigung am betreffenden Bildschirmrand gespiegelt und mit einem konstanten Wert ausgebremst. Somit ist gewährleistet, dass Partikel sich unter keinen Umständen außerhalb des Fensters befinden.

Bei dem zweiten Mechanismus handelt es sich um ein Beschleunigungsfeld *velocityField*, dass auf der GPU als 2D-Textur (Abb. 3.4) von Beschleunigungsvektoren verarbeitet wird. Diese Textur wird beim Programmstart durch einen Compute Shader mit Normalenvektoren, die vom nächsten Rand wegzeigen, initialisiert. Diese Randvektoren sorgen dafür, dass Partikel an der Decke oder den Wänden für eine kurze Zeit festkleben, bevor sie als Tropfen herunterfallen. Während der Laufzeit der Simulation werden die Vektoren in der Textur abhängig von den Ergebnissen der Audioanalyse manipuliert. (siehe 3.5.2.)

3.4 Audioanalyse

3.4.1 Datenstrukturen

Note

```
struct Note {  
    int index;  
    String name;  
    float [] data;  
}
```

Note repräsentiert eine einzelne Note. Dieser Datentyp wird genutzt, um Notensamples in der Bibliothek zu Speichern und Informationen aus der Audioanalyse zwischen **AudioAnalyser** und dem Partikelsystem zu kommunizieren.

index und *name* werden beim Laden der Bibliothek aus dem Dateinamen eines Notensamples geladen. *index* entspricht der Notenreihenfolge in der Bibliothek und *name* dem Namen der Note. (Dateinamenskonvention = 'index_name_.wav', z.B. '1_A3_.wav') Beim Laden eines Samples wird *data* verwendet, um die FFT des Samples zu speichern.

Activations

```
struct Activations {  
    int libCount;  
    int aCount;  
    Note [] activeNotes;  
    float3 intervals;  
    bool isMajor;  
}
```

Activations repräsentiert das Ergebnis einer erfolgreichen NMF. In der Visualisierung wird die Stärke von Effekten auf das Partikelsystem mittels dieser Werte skaliert. *libCount* ist die Anzahl an Notensamples in der Bibliothek. *aCount* ist die Anzahl an erkannten Noten. *activeNotes* beinhaltet die *Note*-Objekte, deren *data* Attribut mit der Magnitude seiner Aktivierung gefüllt ist. *intervals* beschreibt die Intervalldifferenz zwischen den Noten mit der höchsten Magnitude in diesem und dem letzten Aktivierungsvektor.

3.4.2 Signalvorverarbeitung

Damit einkommende Signale mit den Bibliothekssamples verglichen werden können müssen sie erstmal kompatibel gemacht werden. Der folgende Algorithmus beschreibt wie sowohl die Notensamples, als auch die Echtzeit-Audioframes für die NMF vorbereitet werden. *a* ist ein zeitdiskretes rohes Audiosignal und *s* die Länge der zeitdiskreten Samples, die für die STFT genutzt werden. In der Implementierung ist die Funktion

`computeFFT()`, sowie eine Extension zum Anschluss an den Audiostream des Audiointerfaces, der Apple Dokumentation zum Accelerate Framework in je [doca] und [docb] entnommen.

Algorithm 10: Signalvorbereitung

Data: a

```

1  $a = a \cdot hanningwindow;$ 
2  $a = a.split(s)$   $\triangleright(1)$ 
3 for each subarray  $a_i$  do
4   |  $f_i = FFT(a_i)[0...sampleCount]$   $\triangleright(2)$ 
5 end
6  $average(f)$   $\triangleright(3)$ 
7  $f = |f|;$ 
8  $normalize(f)$ 

```

$\triangleright(2)$: Berechne FFT aller Teilarrays und extrahiere nur relevante Frequency Bins

$\triangleright(3)$: Berechne Durchschnitt aller FFTs

Da es sich bei den Samples und dem Gitarreninput um längere Abschnitte handelt, wird der Inputarray in Zeile 2 in Teilarrays mit Länge $= 2^n$ unterteilt. Die FFTs werden dann auf den relevanten Frequenzbereich reduziert, sodass der normalisierte Betrag des Durchschnitts aller FFTs der Teilarrays für die NMF verwendet werden kann. Im Falle des Gitarreninputs wird der Durchschnitt mit der STFT der letzten Audioframes bestimmt.

3.4.3 Polyphone Analyse

Die Audioanalyse ist auf zwei Komponenten verteilt: **AudioAnalyser** nutzt das Accelerate Framework für CPU optimierte FFT-Berechnungen und ist für das Verwalten und Laden der Bibliothek, das Abfangen von Audioinput und dessen Vorverarbeitung, sowie für das Interpretieren der NMF-Ergebnisse und deren Kommunikation an die FS zuständig. **NMFSolver** nutzt parallele GPU-Matrixfunktionen aus dem Metal Performance Shader Framework, um den Aktivierungsvektor für ein einkommendes vorverarbeitetes Audiosignal zu berechnen.

NMF

Wie in [SB03] gezeigt, ist es möglich mittels **Nicht-negativer Matrix Faktorisierung** die einzelnen Notenbestandteile eines Audiosamples mit gewisser Sicherheit zu bestimmen. In [Har12] wird dieses Verfahren auf ein Echtzeitmodell erweitert und für Gitarreninput ausgelegt.

Im Gegensatz zur herkömmlichen Nutzung von NMF, bei der die Inputmatrix V eine $n \times m$ Matrix ist. Werden die Dimensionen von V auf $1 \times m, m = Sampleanzahl$ auf *Frequenzband* reduziert. Jeder neue Frame v des Audiosignals wird mittels einer NMF

mit einer Vergleichsbibliothek W abgeglichen. Der resultierende Vektor H beinhaltet die Vorkommen der einzelnen Noten. Diese Beziehung kann als

$$v = HW \quad (3.6)$$

ausgedrückt werden.

Kostenfunktion

Um die Qualität der Approximierung von H abzuschätzen, wird eine Kostenfunktion genutzt. Durch sie kann das NMF-Problem als Minimierungsfunktion verstanden werden. Ziel der Kostenfunktion ist also mit jeder Iteration der Approximierung kleiner zu werden, bis sie einen Grenzwert *threshold* erreicht oder konvergiert. Lee und Seung stellen in [LS00] zwei Kostenfunktionen vor. Bei Einer handelt es sich um eine Euklidische Distanz, die proportional zu den Matritzen A und B ist. Die andere Funktion, die hier verwendet wird, beschreibt eine asymmetrische Divergenz zwischen A und B und ähnelt der Kullberg-Leibler Divergenz [Har12]. $\|\cdot\|_F$ entspricht der Frobeniusnorm.

$$D(A||B) = \sum_{ij}^n \|A_{ij} \log \frac{A_{ij}}{B_{ij}} - A_{ij} + B_{ij}\|_F \quad (3.7)$$

D kann nur den Wert Null annehmen, wenn $A = B$.

Die Kostenfunktion wird in jeder Iteration des Multiplikativen Updates in Alg. 11 berechnet und mit *threshold* verglichen. A ist hier V und B das Ergebnis der Rekonstruktion durch eine Multiplikation von W und H .

Bibliothek W

Die Vergleichsbibliothek W ist eine $m \times r$ Matrix, mit $r :=$ Anzahl an Notensamples, bei der jede Spalte die FFT eines Notensamples vom Inputinstrument repräsentiert. Diese Bibliothek wird beim Programmstart von **AudioAnalyser** aus dem Library-Ordner geladen und, nachdem FFTs der Samples erstellt wurden (Alg 10), an **NMFSolver** weitergegeben. Somit könnten perspektivisch auch Samples von anderen Instrumenten geladen werden.

Während die Notentemplates verarbeitet werden, werden zwei unterschiedliche Frequenzbandauflösungen verwendet.

Da sich die Magnituden in Frequenz-Bins bei tiefen Noten schnell überschneiden, ist eine hohe Auflösung für das Input der FFTs notwendig. Die zeitdiskreten Samples werden in gleichgroße 2^n lange Teilarrays aufgeteilt, wobei der letzte Teilarray bei Bedarf mit Nullen gefüllt wird, bis er die Länge der nächsten Zweierpotenz erreicht. (Dies sorgt für bessere Laufzeiten des FFT Algorithmus) Ein Durchschnittswert aller Teilarrays wird berechnet. Um nun im Anschluss die Komplexität der Matrixmultiplikationen in der NMF zu reduzieren, werden nur die ersten 1024 Samples des Frequenzbandes verwendet. Diese enthalten alle, für Gitarrennoten relevanten, Frequenz-Bins.

Das Beispiel in Abb. 3.4 zeigt eine Bibliothek von 5 Notensamples mit einer Frequenzauflösung von 1024.

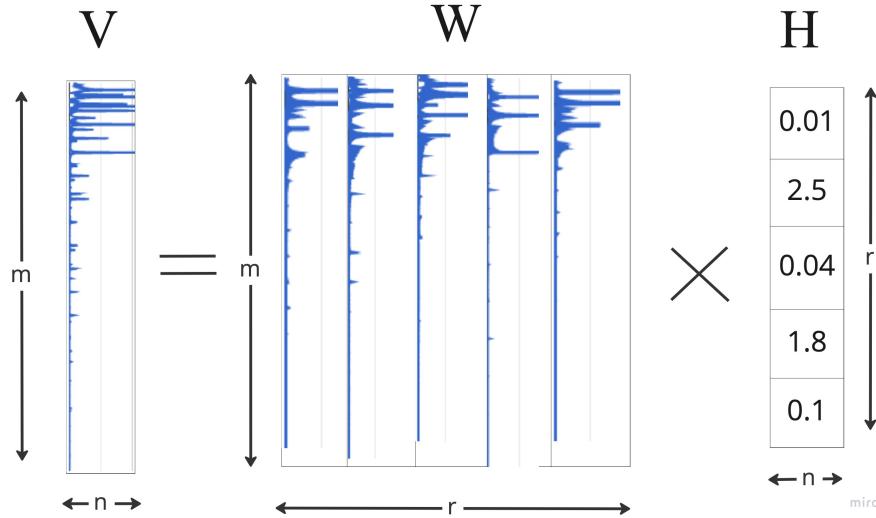


Abbildung 3.4: Dimensionen der NMF-Matrizen. Da es sich bei V um eine FFT handelt, ist $n = 1$, m = Frequenzbereich der FFT, r = Anzahl an Notensamples in W .

In jedem Audioanalyseschritt, in dem ein neuer Peak in der Amplitude festgestellt wird, werden neue Werte für H berechnet. Wie in [SB03] und [Mü15] gezeigt, ist Formel 3.8 ein guter Kompromiss zwischen Schnelligkeit und Einfachheit in der Implementierung. Das Vorkommen aller Noten in W kann wie folgt beschrieben werden.

$$H_{i+1} = H_i \odot (((W_i)^T V \oslash ((W_i)^T W_i H_i)) \quad (3.8)$$

Wobei \odot dem Hadamardprodukt und \oslash einer Hadamarddivision entspricht. Beide lassen sich außerordentlich schnell mittels Compute Shadern berechnen. Diese Multiplikative Update Funktion wird auf H ausgeführt, bis die Kosten-Function *threshold* erreicht oder eine Maximalanzahl an Iterationen durchgeführt wurde.

Die Berechnung der Faktorisierung ist so in die Renderpipeline integriert, dass mit jedem Simulationsschritt eine Iteration des Multiplikativen Updates in 3.8, sowie die Kostenfunktion auf der GPU berechnet wird. Eine Variable *count* verfolgt wie viele NMF-Iterationen schon für das aktuelle Sample berechnet wurden und bricht den Prozess bei Überschreitung einer Maximalanzahl oder Erreichen einer akzeptablen Approximation ab. Neue Input FFTs werden in die NMF gespeist, wenn ein Peak in der Amplitude festgestellt wurde, der auf den Anschlag eines neuen Akkords hinweist. Ein Peak wird erkannt, wenn die Differenz zwischen der aktuellen und der letzten Amplitude einen nutzerdefinierten Schwellwert überschreitet.

Algorithm 11: computeNMF

Data: $V, W, H, count = 0$

```
1 if count < maxIterations then
2    $H = (W^T \cdot V) / (W^T \cdot W \cdot H);$ 
3    $reconstructedWH = W \cdot H;$ 
4    $\epsilon = costFunction(V, reconstructedWH);$ 
5   if  $\epsilon \leq threshhold$  then
6     | return  $H$ 
7   end
8   count++;
9 end
10 else
11   | return -1
12 end
```

▷(1)

▷(1): Informiere **AudioAnalyser**, dass keine Lösung gefunden wurde

Die Matrixmultiplikationen wurden mithilfe von Metal Performance Shadern implementiert [mps]. MPS ist eine Bibliothek mit GPU-optimierten Matrixoperationen wie Multiplikation und Transponieren. Die unüblicheren Hadamard-Operationen wurden mit Compute Shadern umgesetzt.

3.4.4 Interpretierung von H

Das Resultat einer erfolgreichen NMF ist ein Vektor, der für alle Spalten von W einen Faktor enthält. Das bedeutet, dass zwischen gültigen und ungültigen Noten differenziert werden muss. Wie in 2.2.1 bereits erwähnt, enthalten Oktaven einer Note dieselben Harmoniefrequenzen. Aus diesem Grund werden sie oft zusätzlich zur eigentlichen Note erkannt. Ein weiterer Faktor können Störfrequenzen im Signal sein, die z.B. beim Anschlag oder Zupfen der Saiten entstehen.

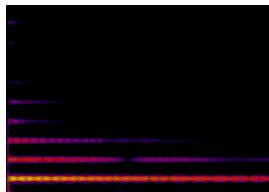


Abbildung 3.5: f5 mit zusätzlich erfassten Harmoniefrequenzen

Um diese falschen Positive herauszufiltern, werden alle Faktoren sortiert und nur die 5 höchsten Werte berücksichtigt. Im Anschluss werden aus den verbliebenen Faktoren all jene entfernt, deren Magnitude deutlich unter dem Maximum liegen. Zur späteren Visualisierung wird die letzte valide Aktivierung ebenfalls von **AudioAnalyser** gespeichert.

Der Struct **Activations** kommuniziert alle wichtigen Informationen an das Partikelsystem und die UI.

3.5 Visualisierung

Wie in 3.3.6 erwähnt, werden die Partikelbeschleunigungen mit einer Textur *velocityField* manipuliert. Diese ist hat die Dimensionen der Bildschirmtextur. Somit kann ein Vektor ohne zusätzliche Umrechnungen einer Partikelposition in Texturkoordinaten angewendet werden.

Zur Visualisierung eines **Activations**-Objektes wurden drei Effekte implementiert:

3.5.1 Farbe

Die 'Grundfarbe' aller Partikel kann während der Simulation mit einer UI-Farbauswahl bestimmt werden. Wenn eine neue Activation erkannt wurde, werden die Indices der aktuellen und der letzten lautesten Note miteinander subtrahiert. Dieser Intervall repräsentiert, wie weit die Noten auseinanderliegen. Der resultierende Vektor wird normalisiert, um mit dem Fragment-Shader Farbbereich übereinzustimmen und dann entweder zur Grundfarbe addiert oder subtrahiert, je nachdem, ob die Summe der Intervalle positiv oder negativ ist. Der Effekt wird sowohl bei einzelnen, als auch mehreren gleichzeitig gespielten Noten angewendet.

3.5.2 Interaktion mit Partikeln

Das Schreiben in *velocityField*, geschieht in einer Vertex-Fragment Pipeline, die vor den Velocityberechnungen der *Double Density Relaxation* ausgeführt wird.

bPush

bPush berechnet eine Zuordnung von erkannten Noten zu Bereichen am unteren Bildschirmrand. Jede Note in der Bibliothek entspricht einer x-Position. Das heißt, dass bei einer Bibliothek von Noten zwischen A3 und A5, die Position für A4 genau in der Mitte des Bildschirms liegt. Für jede aktive Note wird dann ein Punkt-Vertex erstellt, dessen rot und grün Farbkomponenten einen Beschleunigungsvektor repräsentieren. Die rasterisierten Pixel, innerhalb des Durchmessers des Punktes, zeigen von der Vertexposition weg. Ein Faktor, der k in *computePressures* ersetzt, wird in der blau-Komponente abgelegt.

Der daraus resultierende Effekt ist eine notenspezifische Strömung, die angrenzende Partikel um das Punktprimitiv herumfließen lässt. Partikel, die sich innerhalb der aktiven Zone befinden, ziehen nahe Nachbarpartikel zu sich, statt sie abzustoßen.

rPush

rPush ist ein Vertex-Fragment Effekt, der nur bei einem erkannten Akkord auslöst. Nachdem die NMF eine Aktivierung mit mindestens 3 Noten erkannt hat, werden die Intervalle zwischen der tiefsten Note und allen anderen gültigen, aktivierten Noten berechnet. Der Modulo des Oktavintervalls 12 liefert die 'normalisierten' Abstände innerhalb einer Oktave. Nun kann abgeglichen werden, ob die Intervalle dem Muster eines

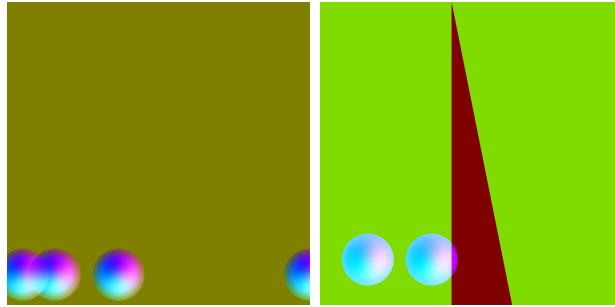


Abbildung 3.6: *velocityField*:

links: *bPush*: Punktprimitive der Noten A3, B3, F3 und E4

rechts: *rPush*: Dreiecksprimitiv eines F-Dur Akkords

Dur- oder Moll Akkords entsprechen. Die Intervalle, die hier entscheidend sind, sind: Dur = [4, 9, 11] und Moll = [3, 8, 10]. Wenn sich nur Intervalle aus einem dieser Templates in der Aktivierung befinden, wird dem **Activations** Objekt ein boolscher Wert übergeben, der indiziert um welche Art von Akkord es sich handelt. Der Grundton des Akkords ist die tiefste Note in *activeNotes* nachdem der Array sortiert wurde.

Von der Notenkoordinate des Grundtons aus wird nun ein rechtwinkliges Dreiecks-Primitiv aufgespannt, dass Partikel, die sich innerhalb der Form befinden in eine, vom Tongeschlecht abhängige, Richtung beschleunigt. Die Beschleunigungsrichtung wird zwischen den Farbwerten, der drei Inputvertices interpoliert. Dies resultiert in größeren Wellen und Drucksammelpunkten, die das Aussehen und Verhalten der gesamten Flüssigkeit beeinflussen.

Kapitel 4

Ergebnisse

Im Folgenden werden die Ergebnisse der Simulation, Audioanalyse und deren Interaktionen in Bezug auf Performance und visuelle Qualität untersucht. Die Tests wurden auf einem 2019 MacBook Pro mit einem 2,4 GHz Quad-Core Intel Core i5 Prozessor und einem Intel Iris Plus Graphics 655 1536 MB Grafikchip durchgeführt. Tests mit einer leistungsstärkeren Grafikkarte wiesen höhere Framerates (4000 Partikel mit 60 FPS) auf. Für die Samplebibliothek, die in der NMF genutzt wird, wurde eine Bibliothek von 20 ca. 1 Sekunden langen Noten von A3 bis E5 einer E-Gitarre verwendet. Ein Steinberg UR22 mk II Audiointerface ermöglicht den Zugriff auf das Echtzeitgitarrensignal.

4.1 Programm

Das Ergebnis dieser Implementierung ist ein macOS-Programm, dass es einem User erlaubt Flüssigkeitssimulationen mit variablen Parametern zu erstellen und diese durch musikalisches Input zu manipulieren.

Ein Overlay-Userinterface kann mit 's' ein- und ausgeblendet werden. 'Enter' startet eine neue Simulation mit den aktuellen Parametern. Alle Werte können während der Simulation verändert werden.

4.2 Nachbarsuche

Neben dem in 3.3.2 vorgestellten Brute-Force Algorithmus habe ich ebenfalls einen KNN-Spatial Hashing Algorithmus implementiert. An [O8] und [THM⁺03] angelehnt wurde der Bildschirm in ein gleichmäßiges, axis-aligned Gitter aufgeteilt. Mit der relativen Position und zugehörigen Bounding Box (Formeln 4.1 und 4.2) kann dann eine zuverlässige räumliche Zuordnung zu einem Hash-Bucket in einer Hashtabelle hergestellt werden. Partikel, die sich im selben, von H abhängigen, Bereich befinden werden so miteinander gruppiert. Das Nachbarset eines Partikels p besteht aus allen Partikeln in demselben Hash-Bucket, die sich innerhalb des Interaktionsradius befinden. Um nur die k relevantesten Partikel zu überprüfen wird das Nachbarset nach der Distanz mittels GPU-Radixsort sortiert. Wie von Teschner *et al.* in Experimenten gezeigt, bringt ei-

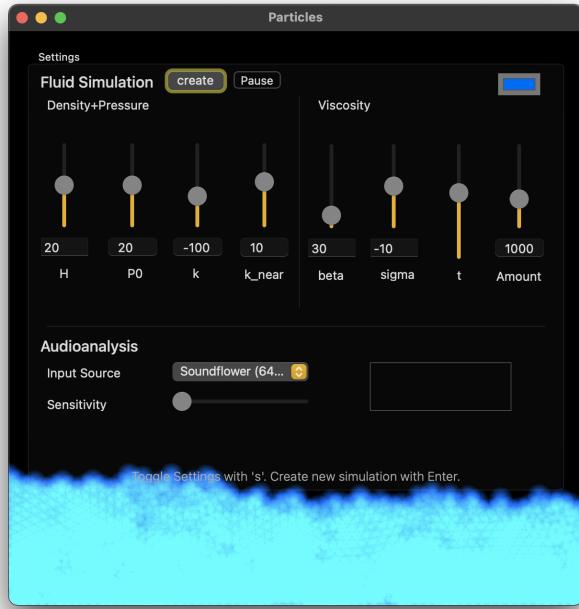


Abbildung 4.1: User Interface mit Simulationseinstellungen

ne Gittergröße von $width/H$ und Hashtabellengröße deutlich größer als $particleCount$ einen guten Kompromiss zwischen Speichernutzung und Schnelligkeit. bb bezeichnet hier die Bounding Box, in der sich das Partikel befindet.

$$relativeCellPosition(x, y) = \left(\frac{x - bb.xmin}{H}, \frac{y - bb.ymin}{H} \right) = (i, j) \quad (4.1)$$

$$hash(i, j) = i \cdot 73856093 \text{ xor } j \cdot 19349663 \quad (4.2)$$

Die in Abb 4.3 dargestellten Messwerte zeigen die GPU-Rechenzeiten, die für die Nachbarsuche verwendet werden, sowie die Framerate und den genutzten RAM-Speicher der beiden Algorithmen. Wider Erwarten, stellte sich der Brute-Force Ansatz als performanter und konsistenter heraus. Obwohl durch Spatial Hashing die Komplexität für alle Flüssigkeitsinteraktionen stark reduziert wurde, ist die Rechenzeit, die für das Füllen und Sortieren des Nachbarbuffers aufgewandt wird, zumindest bei Partikelanzahlen unter 5000, noch zu groß. Wie in Abb 4.2 zu erkennen ist, bleibt die Framerate für Brute-Force zwischen 3000 und 4000 Partikeln konstant bei 30 FPS. Dies könnte u.a. an optimierten Speicherzugriffen auf die Nachbar Textur liegen. Da für die NMF ebenfalls GPU-Rechenzeit benötigt wird und eine hohe und konstante Framerate essentiell ist, habe ich mich für die Brute-Force Methode entschieden. Es soll angemerkt sein, dass Spatial Hashing bei einer größeren Anzahl von Partikeln, deutlich effizienter ist, und hier die Leistungsfähigkeit des Testgeräts, sowie Limitationen der Metal API, ausschlaggebend waren.

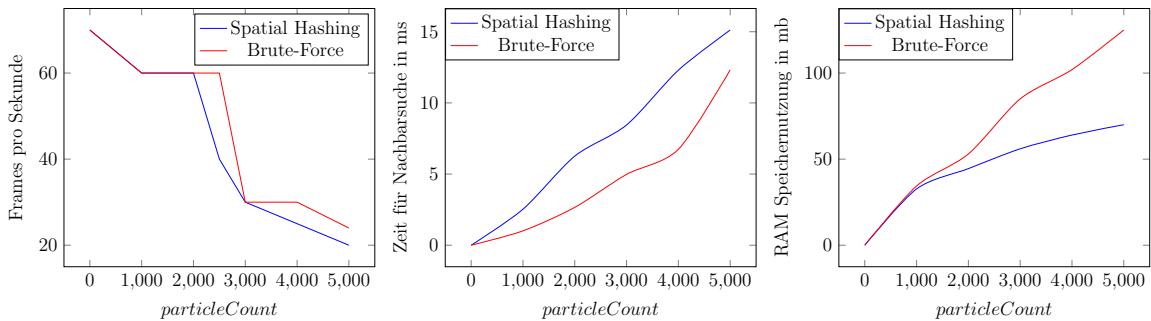


Abbildung 4.2: Performance der Simulation mit Spatial Hashing oder Brute-Force (Ohne aktive NMF)

4.3 Simulation

4.3.1 Flüssigkeitsverhalten

Ein übliches Szenario, um das Verhalten einer simulierten Flüssigkeit zu überprüfen, ist der Dammtest [O8]. Bei einer sich korrekt verhaltenden Flüssigkeit sollten diverse Stadien zu beobachten sein: Zuerst ein initialer Moment, in dem die Partikel durch Schwerkraft miteinander in Kontakt geraten. Darauffolgend ist ein Stauchungsmoment zu erwarten, in dem sich die Flüssigkeit gegen die Kompression wehrt. Daraus resultiert ein Fließen der Partikel in den leeren angrenzenden Raum. Durch Druckausgleich werden die fallenden Partikel gegen den gegenüberliegenden Rand beschleunigt, wo sich eine Welle, durch den Verdräng-Effekt der nachkommenden Partikel, formen sollte. Bei der anschließenden Wechselwirkung zwischen den Wellenpartikeln und den Partikeln aus dem unteren Bereich der Dammtest-Partikel sind leichte Turbulenzen unter der Oberfläche zu erwarten, sowie eine kleine Welle auf der initialen Bildschirmseite. Nachdem die Flüssigkeit die Beschleunigung und den Druck über mehrere Iterationen dieses Prozesses ausgeglichen hat, soll der Ruhezustand erreicht werden. Dieser ist definiert als ein Zustand, in dem sich alle Partikel der Simulation in einer gleichmäßigen Struktur anreihen und nur minimale Interaktionen sichtbar sind. Die Anziehungs- und Abstoßungskräfte gleichen sich aus, und eine deutliche Oberfläche hat sich gebildet.

Die Druckverteilung und Bildung von Wellen sind in Abb. 4.3 gut zu sehen:

Im Ruhezustand reihen die Partikel sich in einem homogenen Muster an und behalten eine konstante Dichte. Hier ist es wichtig, die richtigen Parameter für die Partikel zu wählen. Zu große Zeitdifferenzen oder Druckwerte können dazu führen, dass die Partikel instabil werden. Hier die Simulationsparameter in Abb. 4.2: $H=15$, $P_0=12$, $k=-100$, $k_{near}=10$, $\beta=30$, $\sigma=-10$, $pCount=2000$.

Deutlich sichtbar ist die Anziehungskraft, die zwischen Nachbarn besteht. In Abbildung 4.8 wurden Partikel mit einem rPush in den leeren Raum beschleunigt. Es formen sich sowohl filigrane Strukturen, als auch größere Cluster, die bei einer Kollision mit der Oberfläche eines ruhenden Körpers nahe Partikel verdrängen.

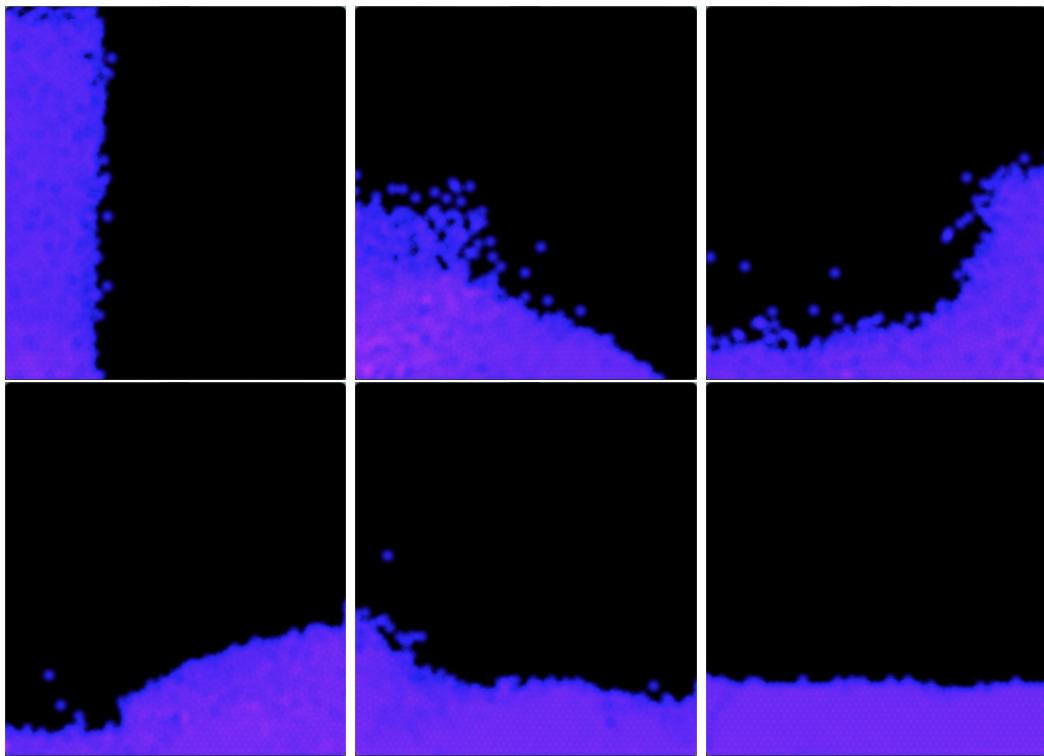


Abbildung 4.3: Von links nach rechts: Stadien des Dammtests

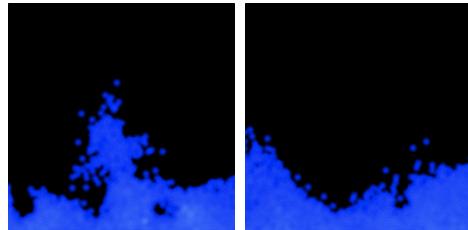


Abbildung 4.4: Tropfen verdrängt angrenzende Flüssigkeit

Verdrängungverhalten

Beim Auftritt eines größeren Clusters von Partikeln auf ein Cluster im Ruhezusand wird ein spezifisches Verdrängungsverhalten erwartet. In Abb. 4.4 sind zwei Zustände der Flüssigkeit zu sehen. Ein Tropfen fällt auf die Oberfläche und die bereits vorhandenen Partikel werden sichtbar verdrängt. Der höhere lokale Druck ist durch die intensivere Färbung der Partikel zu erkennen. Eine kleine Welle breitet sich auf beiden Seiten des Kontaktpunktes aus und kollidiert im Anschluss mit dem Fensterrand. Nach einigen Druckausgleichsimpulsen wird der Ruhezustand wieder erreicht.

Stabilität

Die Stabilität der Simulation hängt stark von den Startparametern ab. Für einen visuell interessanten Start werden Partikel immer im linken Drittel des Bildschirms erstellt, sodass bei jedem Simulationsstart ein Dammtest durchgeführt wird. Die Proportionen

zwischen P_0 und k sind ausschlaggebend für das Verhalten in den ersten Momenten. Eine zu geringe Restdichte sorgt dafür, dass Partikel kaum miteinander interagieren. Das Volumen der Flüssigkeit im Ruhezustand kann sich dadurch so stark reduzieren, dass die Partikel nur noch als komprimierte Linie am unteren Bildschirmrand auszumachen sind.

Ein gegenteiliges Ergebnis ist bei einem sehr niedrigen k , bzw. hohem P_0 zu beobachten. Beim Start der Simulation verdrängen sich Partikel mit einer solch hohen Beschleunigung, dass die Flüssigkeit zu explodieren scheint. Wenn die Simulation in diesem Zustand verlangsamt wird, ist sichtbar, dass die Partikel zwar noch aufeinander reagieren, deren *velocity* jedoch so hoch ist, dass die DDR Verschiebung kaum Einfluss auf die Flugbahn der Partikel hat. Die massive Abstoßung zwischen Partikeln hat somit den Effekt, dass Partikel ungehindert beschleunigt werden. Diesem Effekt wurde durch eine Stauchung der Beschleunigung in *applyVelocity* entgegengewirkt, indem die Beschleunigung in jedem Schritt anteilig verringert wird. Somit soll die Entschleunigung durch Luftwiderstand emuliert sein.

Dies verhindert, dass eine plötzliche 'Explosion' von Partikeln stattfindet, solange die Relation zwischen P_0 , k , k_{near} , *particleCount* und H in etwa den folgenden Kombinationen entspricht:

<i>particleCount</i>	P_0	k	k_{near}	H
500	[10, 20]	[-80,-100]	8	[20, 40]
1000	[10, 20]	-100	10	[13, 30]
2000	20	-100	14	[14, 20]
3000	[16-20]	[-70, -100]	15	[17, 20]
4000	[15, 25]	-100	10	[15-30]

Die Werte von 1000 bis 3000 Partikeln gelten für eine Simulation, die mit der Startfenstergröße ausgeführt wird. Bei mehr als 3000 Partikeln wird der Vollbildmodus empfohlen, da die Partikel so mehr Raum haben, um sich auszubreiten. Die Relation zwischen Restdichte P_0 und H und den Dimensionen des verfügbaren Restraums sollte dabei individuell angepasst werden. Ein zu großes H bei vielen Partikeln und einer geringen Restdichte resultiert, besonders bei einer kleinen Fenstergröße der Applikation, in 'Explosionsverhalten'. Clavet *et al* beugen diesem Verhalten vor, indem bei großen Zeitschritten mehrere Simulationsschritte pro Frame gerendert werden. Dadurch entsteht visuell der Effekt einer sich schnell bewegenden Flüssigkeit. Dies geschieht jedoch auf Kosten der Framerate. Besonders in unserem Fall, würden mehrere Iterationen der Simulation und der NMF zu inakzeptabler Interaktivität führen. Deshalb muss bei einem hohen t ein Ausgleich in P_0 und k vorgenommen werden. Somit wird verhindert, dass die Kräfte der DDR zu intensiv werden und ein Explosionsverhalten verursachen. Dies ist ebenfalls der Grund, weshalb das Audioinput keinen Einfluss auf t hat.

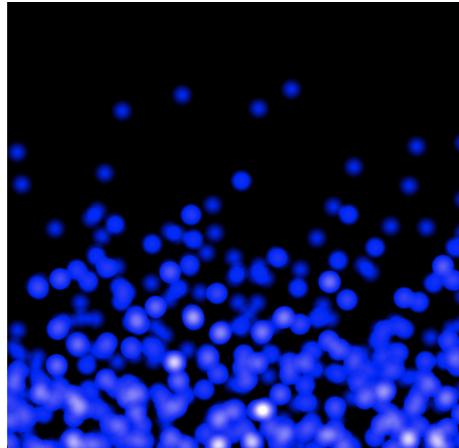


Abbildung 4.5: Invertiertes k , mit $k = 100$

Viskosität

Viskosität ist in den Beschleunigungen der Partikel bei einer Kollision bemerkbar. Eine geringe Viskosität bremst Partikel aus, und reduziert so die Geschwindigkeit und den zugrundeliegenden lokalen Druck, mit denen sich die Welle ausbreitet. Ein Ruhzustand ist deshalb schneller erreicht. In dem Fall einer hohen Viskosität, verlieren Partikel deutlich weniger Beschleunigung. Dies resultiert in einem chaotischeren, und sich schneller ausbreitenden Flüssigkeitsbild. Durch die Wahl einer hohen Viskosität kann dem Explosionverhalten zudem entgegengewirkt werden.

Alternative Nutzung der DDR

Ebenfalls visuell interessant ist was passiert, wenn der Abstoßungsfaktor k umgekehrt wird (Abb. 4.5). Partikel ziehen einander so stark an, dass es zur Formung von vielen radiusabhängigen Dichtemaxima kommt (Durch die weiße Farbe im Innern eines Clusters zu erkennen). Die massive Anziehungskraft innerhalb und außerhalb der Maxima sorgt dafür, dass Dichtecluster einander absorbieren oder bei einer Kollision mit einer Wand in kleinere Cluster aufteilen. Obwohl die Beschleunigungen durch die hohe Anziehungskraft sehr hoch sind, lässt sich dieses Verhalten durch zurücksetzen von k zu einem 'normaleren' Wert (siehe obige Tabelle) in wenigen Frames umkehren. Beschleunigungen über Interaktionseffekte haben trotzdem nicht den gleichen visuellen Effekt wie mit herkömmlichen Parametern. Statt eine klare Beschleunigung in eine Richtung zu sehen, werden die Cluster schnell von der Anziehungskraft angrenzender Partikel ausgebremst. Der Effekt eignet sich demnach vor allem dafür, auf Teilmengen der Partikel angewandt zu werden, anstatt global für alle Partikel zu gelten.

4.3.2 Rendern

Die Partikel werden mit einem Vertex- und Fragment Shader als runde Punkt-Primitive gerendert. Ein additiver Blend-Modus und Smoothstep im Alpha-Kanal sorgen dafür,

dass sich beieinanderliegende Partikeloberflächen überschneiden. Zunächst war zusätzlich eine Visualisierung des lokalen Drucks angedacht. Es stellte sich jedoch heraus, dass durch additive Farbkombination, beinahe der selbe Effekt entsteht.

4.4 Audioanalyse

Um die Qualität der Audioanalyse zu abzuschätzen, wurden Ergebnisse der NMF mit dem tatsächlich Gespielten verglichen. Hier wurde zwischen einzelnen Noten, Mehrtönen und dem Tongeschlecht unterschieden.

Bei den Tests habe ich eine Bibliothek von 20 Notensamples von A3 bis E5 einer Gibson SG E-Gitarre in Standardstimmung (EADGBE) verwendet. Um Störsignale in den Samples vorzubeugen, wurde der Anschlag der Note abgeschnitten, sodass nur die relevanten Schwingungen für die Bibliothek berücksichtigt werden. Die FFTs wurden mit einer Fenstergröße von 8129 erstellt.

4.4.1 Test

Um die Präzision der NMF als Ganzes zu bewerten, habe ich einen an [Har12] angelehnten Test durchgeführt. Für 1 bis 5 gleichzeitig gespielte Noten wurden je 10 verschiedene Gitarreninputs gespielt. Dabei wird für jede Polyphoniestufe **PS** festgehalten, welche Anzahl der Noten korrekt (**KE**) oder nicht erkannt (**NE**) wurde und wie oft die falsche Oktave (**FO**) oder eine falsche Note (**FN**) erkannt wurde. Letztere beinhalten auch zusätzliche Aktivierungen. Um alle, einem Input-Ereignis zuschreibbaren Noten, zu erfassen, wurden alle, durch **AudioAnalyser** gefilterten, validen *H*-Resultate, die während der Dauer des Akkords festgehalten wurden, berücksichtigt. Die Ergebnisse der Genauigkeit **P** und Trefferquote **R** sind der unteren Tabelle zu entnehmen.

PS	Inputs	KE	NE	FO	FN	R	P	AP
1	10	10	0	6	3	1	0.69	0.69
2	20	14	1	3	2	0.7	0.7	0.7
3	30	15	7	6	0	0.5	0.53	0.75
4	40	28	6	6	4	0.7	0.63	0.77
5	50	18	7	20	5	0.36	0.36	0.76

Es fällt auf, dass für eine einzelne Note immer die Richtige erkannt wird. Bei Noten, die als offene Saite spielbar sind, erscheint oft auch zusätzlich eine Oktave. Falsche Noten tauchen in der Form von annähernden Noten auf. z.B. C wird als B oder C# erkannt. Für die Visualisierung sind diese Ergebnisse hinzunehmen, da eine angrenzende Note in einem *bPush* visuell kaum von der Richtigen zu unterscheiden ist. Falsche Oktaven sorgen jedoch dafür, dass zwei an Stelle von einem *bPush* gerendert werden.

Für Polyphoniegrad 2 ist der Anteil an korrekt identifizierten Noten kleiner, jedoch wird immer noch die Mehrzahl der gespielten Töne erkannt. Die Fehlerquote setzt sich ebenfalls aus angrenzenden Noten zusammen.

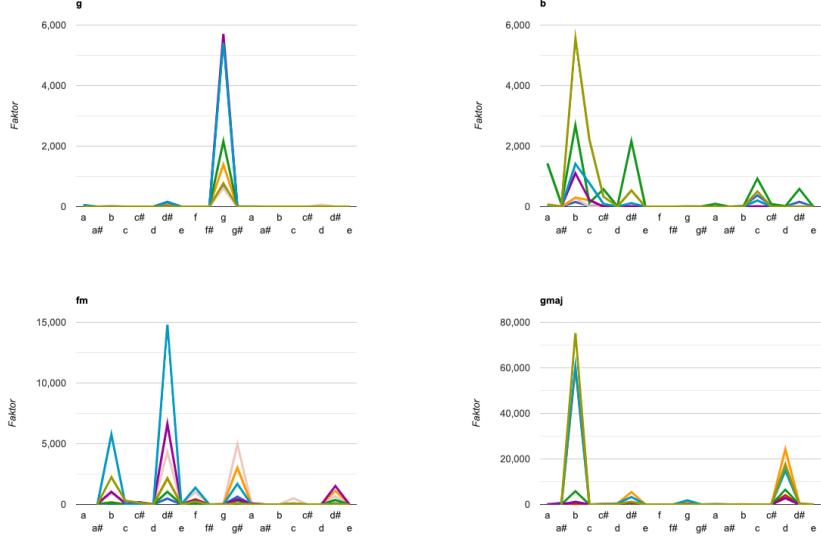


Abbildung 4.6: Ergebnisse von sieben NMFs des gleichen Inputs. Von links nach rechts: G, B, F-Moll, G-Dur

Polyphoniegrad 3 weist die niedrigste Präzision auf. Ab hier war eine klare Diskrepanz zwischen Akkordgriffmustern zu erkennen (Siehe Erläuterung in 4.4.3). Grundnoten auf der E-Saite wurden am häufigsten nicht erkannt. Dafür fällt jedoch auf, dass, im Gegensatz zu **P4** und **P5**, keine falschen Noten identifiziert wurden. Für die Visualierungseffekte ist dies von Vorteil, da ab 3 Noten auf Akkordintervalle untersucht wird. Hierfür, und für alle darauffolgenden Polyphoniegrade gilt, dass falsche Oktaven keinen Einfluss auf das Ergebnis haben, da der Intervall zur Überprüfung immer auf eine Oktave normalisiert wird. Falsche Noten können den Algorithmus jedoch daran hindern, das richtige Tongeschlecht zu erkennen.

Ein angepasster Wert **AP** veranschaulicht den Anteil bei dem alle, für die Tongeschlechtbestimmung relevanten, Informationen korrekt erfasst werden. Das heißt, die Werte für **KE** und **FO** wurden zusammen gezählt.

Es sticht dennoch heraus, dass Genauigkeit und der Trefferquote für 5 gleichzeitig gespielte Noten bemerkenswert niedrig sind. Es werden mehr falsche Oktavierungen erfasst als korrekte Noten. Die falsch erfassten Oktaven sind in der Mehrzahl der Fälle tiefere Oktaven zu denen, die tatsächlich gespielt wurden. z.B. B4 wird zu B3. Des Weiteren fiel auf, dass E2 und E3 nur in Verbindung mit einer anderen Oktave von E erkannt wurden. Stattdessen identifizierte die NMF diese Noten als B3.

Die Nutzung falscher Positiver und Oktaven zu Akkordbestimmung in der Visualisierung deuten darauf hin, dass die Filterkriterien des Aktivierungsvektors in **AudioAnalysen** weiter angepasst werden sollte.

4.4.2 Aktivierungen

Um die Ergebnisse von H zu veranschaulichen wurde jede Note oder Akkord in Abb. 4.6 mehrmals gespielt und die Aktivierungsvektoren in einem Diagramm beieinander platziert. So ist deutlich erkennbar, ob das Ergebnis der NMF konsistent ist.

In den ersten beiden Diagrammen in Abb. 4.6 ist zu erkennen, dass in beiden Fällen, die richtige Note den höchsten Faktor in H hat. Es ist jedoch besonders für B deutlich, dass eine Reihe an falschen Positiven in der Aktivierung auftauchen. Trotzdem wurde die korrekte Note im Durchschnitt in 6 von 7 Fällen als am signifikantesten erkannt. Die Filterung die in **AudioAnalyser** stattfindet, scheint für einzelne Noten funktional zu sein. In den Fällen, in denen zusätzliche Noten erfasst werden, wurden angrenzende Noten oder Oktaven angezeigt. Im Schnitt benötigt die NMF 9 Iterationen von Algorithmus 11, um eine einzelne Note zu erkennen.

4.4.3 Aktivierungen - Tongeschlecht

Zur Erkennung des Tongeschlechts ist es vonnöten, dass mindestens drei Noten korrekt identifiziert wurden. Dann kann das Vorkommen der entsprechenden Intervalle überprüft werden. Hier ist es wichtig zwischen den Ergebnissen der NMF in verschiedenen Stadien des Inputs zu differenzieren. Die Diagramme in Abb. 4.6 zeigen ein H , dass kurz nach dem Anschlag des Akkords/der Saite festgehalten wurde. Über die Dauer einer Gitarrenschwingung sind unterschiedliche Ergebnisse zu beobachten. Zum Zeitpunkt des Anschlags kann es zu einer Häufung von falschen Positiven kommen. Nachdem diese Phase vorüber ist, werden generell die Mehrzahl der Komponentennoten erfasst. Im Ausklang eines Akkords reduzieren sich die erkannten Noten allmählich entweder zum Grundton, oder oft zu der Note, die auf der G-Saite gespielt wurde.

Das Tongeschlecht selbst wird in verschiedenen 'Voicings', – d.h. Griffmuster für Akkorde –, merklich unterschiedlich gut erkannt. Ein C-Shape Akkord, wurde in meinen Tests in 7 von 10 Fällen korrekt als Dur-Akkord interpretiert. Moll-Akkorde sind am zuverlässigsten mittels A-Moll-Voicings in Barré oder Moll-Voicings, die den 10. Intervall beinhalten, zu identifizieren. Offene Saiten hingegen tendieren dazu, das Frequenzband zu dominieren, sodass keine korrekte Lösung gefunden wird.

Im Schnitt erreicht ein polyphones Signal in 17 Iterationen eine Approximierung von H .

Kostenfunktion

Im Laufe der Entwicklung habe ich zwei verschiedene Kostenfunktionen verwendet. Neben der Kullberg-Leibler-Divergenz wurde ebenfalls die Euklidische Distanz genutzt, um die Approximierung zu bewerten. Hier fiel auf, dass die KLD im Schnitt minimal weniger Iterationen brauchte, um einen vergleichbar präzisen Aktivierungsvektor zu produzieren. In Abb. 4.7 ist die Konvergenz beider Funktionen, relativ zur Anzahl an Iterationen zu sehen. Interessant ist jedoch, dass in der euklidischen Distanz ein klarer Unterschied zwischen einzelnen Noten und Akkorden zu erkennen ist, da die Magnituden linear berechnet werden.

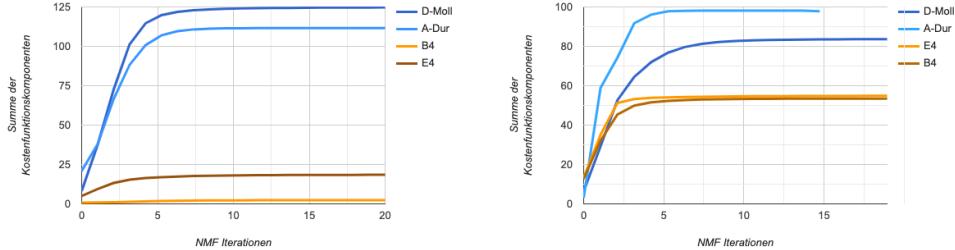


Abbildung 4.7: Links: Euklidische Distanz; Rechts: Kullberg-Leibler-Divergenz

Erkenntnisse

Im Allgemeinen fällt auf, dass die Noten B3 und C#3 überdurchschnittlich oft fälschlicherweise erkannt werden. In angrenzenden Noten wie E oder D kann es vorkommen, dass B3 oder C#3 den mit Abstand höchsten Faktor in H aufweisen. Dies lässt sich vermutlich auf das Überlappen von tieferen Frequenzen und Eigenarten der spezifischen Gitarre zurückführen. In einem Test mit einer Fender Telecaster, die generell ein höherfrequentes Timbre als die Gibson-Gitarre aufweist, wurde stattdessen das hochfrequente E5 häufiger fälschlicherweise als präsent erkannt. Die Qualität der Approximierung von H kann durch einen Hochpass, oder Nutzung der Treble-Tonabnehmer verbessert werden.

Ein weiterer Faktor ist das Erkennen von Oktaven der gespielten Note. Für die Erkennung von Akkorden ist das jedoch nicht von Bedeutung, da die überprüften Intervalle ohnehin auf eine Oktave reduziert werden. Um die Interaktivität zu verbessern, wurde die maximale Anzahl an NMF Iterationen auf 20 limitiert. In [Har12] wird erwähnt, dass ca. 50 Iterationen für eine eindeutige Analyse benötigt werden. Für die Zwecke dieser Applikation sind 1 oder 2 ganze Sekunden, bis die Visualisierung in Kraft tritt, jedoch zu lange, um den gewünschten Effekt zu erzielen. Ein weniger präzises und dafür schwächer gewichtetes Ergebnis wird für eine bessere Interaktivität in Kauf genommen. Abb. 4.6 ist zudem zu entnehmen, dass, selbst wenn sich fehlerhafte Noten in das Ergebnis einschleichen, H zumindest konsistent in den Aktivierungen zu einem bestimmten Input ist.

4.4.4 Interaktionen

In der Praxis manifestieren sich die Effekte *rPush* und *bPush* als Druckwellen und Ströme, denen Partikel in Gruppen folgen.

bPush

Die Druckmanipulation in *bPush* verdrängt Partikel in einer Kreisform und sorgt dafür, dass Partikel um die Vertexfläche herumfließen. Es ist deutlich, dass häufig Oktaven zusätzlich visualisiert werden. Im Render kann das allerdings zu interessanten Artefakten führen. Besonders wenn sowohl *rPush*, als auch *bPush* aktiv sind, befördert die kreisförmige Strömung Partikel in den Bereich eines *rPush*, wo sie dann entweder

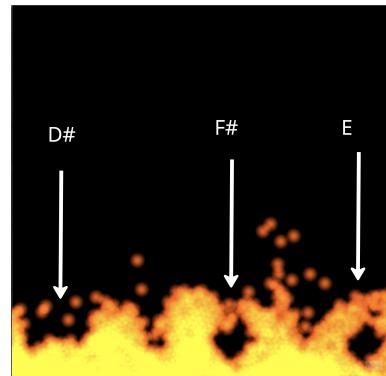


Abbildung 4.8: *bPush* mit den Noten d#, f#, e

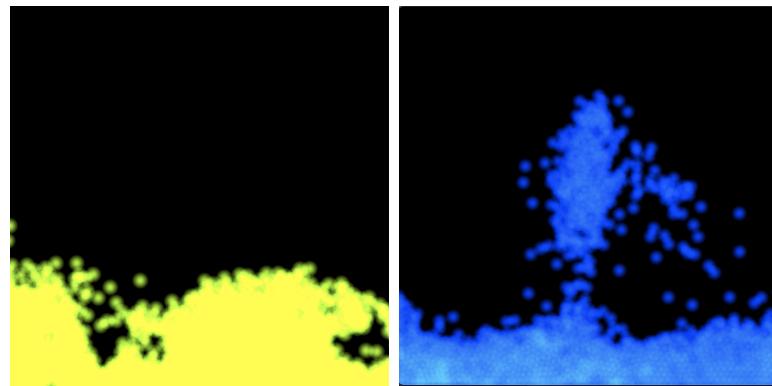


Abbildung 4.9: Akkorde B-Moll und G-Dur visualisiert

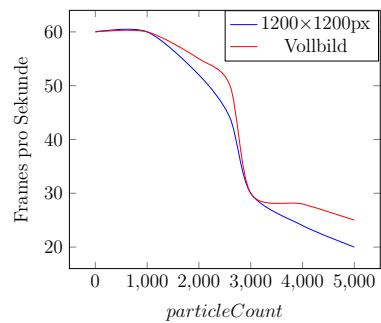


Abbildung 4.10: Performance der Simulation mit Standardfenstergröße und Vollbild

zusammengedrückt oder nach oben beschleunigt werden. Bei kontinuierlichem Spielen entstehen somit komplexe Strukturen, die sich bei Stille wieder in den Ruhezustand ausgleichen.

rPush

In Abb. 4.9 ist die Visualisierung eines D-Moll Akkords zu sehen. Alle Partikel, die sich innerhalb des Bildschirmbereichs der dominantesten Note in der Aktivierung befinden, bewegen sich in Richtung des unteren Bildschirmrandes und verdrängen annähernde Partikel. Das Bild daneben zeigt einen G-Dur Akkord, der Partikel um die G4-Bildschirmposition in den freien Raum beschleunigt. Hier ist die Tropfenbildung gut zu sehen.

Dynamische Farbe

Starke Kontraste in der generellen Tonhöhe werden farblich visualisiert. Der Intervall der Note mit dem höchsten Aktivierungswert im Vergleich zur letzten Aktivierung wird verwendet, um den Hue Wert der Partikelfarbe dauerhaft zu manipulieren. Sprünge zwischen Akkorden oder in einer gespielten Melodie werden so durch eine leichte Änderung der Gesamtfarbe gekennzeichnet. Das Ergebnis ist ein sich dynamisch veränderndes Farbbild, dass uniform auf alle Partikel angewandt wird.

4.5 Gesamt-Performance

Während die Simulation, ohne dass **AudioAnalyser** Daten an **NMFSolver** übergibt, mit einer konstanten Framerate ausgeführt wird (siehe Abb. 4.2), kann es bei gleichzeitiger Ausführung der Simulation, NMF und Visualisierungsfunktionen zu Frameeinbrüchen kommen. Dies ist besonders bei Partikelanzahlen über 2000 Partikeln zu beobachten.

Diese Frameeinbrüche reduzieren die Framerate für den Frame, in dem in *velocityField* geschrieben wird, um ca. 15%. z.B. 2500 Partikel à 60 Frames, reduziert sich für die Dauer der Analyse auf 42-53 Frames. Obwohl diese Frameeinbrüche nur für je einen Frame anhalten, kann kontinuierliches Spielen mit einer niedrigen Peak-Toleranz zu einem ungewünschten Verlangsamen des Renders führen. Grund hierfür ist der zusätzliche Stack an GPU-Befehlen, die durch **NMFSolver** veranlasst werden.

Angemerkt sei, dass das gesamte Programm im Vollbildmodus minimal stabiler läuft, da macOS mehr Ressourcen zum Rendern alloziert. In Abb 4.10 sind zwei Simulationen von 1000 bis 5000 Partikeln geplottet, bei denen dieselbe Akkordfolge gespielt wurde. Einmal in der initialen Fenstergröße von $1200 \times 1200\text{px}$ und einmal im Vollbildmodus.

Video

Der Arbeit sind zwei Videos beigefügt, die die Visualisierung einer A-Dur Tonleiter, sowie einer Akkordfolge demonstrieren.

Kapitel 5

Analyse

Ziel dieser Arbeit war es eine Software zu erstellen, die ein Echtzeit-Gitarrensignal mit einer Flüssigkeitssimulation interagieren lässt. Besonderer Fokus sollte dabei auf Interaktivität bei hohen Framerates, Parallelismus und polyphoner Analyse liegen. Das Gitarrensignal sollte, eine Echtzeit-Nicht-Negative Matrix Faktorisierung nutzend, in seine Schwingungskomponenten aufgeteilt werden, sodass Akkordtyp und Tongeschlecht abgeleitet werden können. Ein nutzerfreundliches User Interface soll es einem Musiker ermöglichen, die Software ohne Vorkenntnisse hinsichtlich Flüssigkeits- oder Partikelsimulationen zu bedienen. Im Folgenden möchte ich die Ergebnisse zusammenfassen und auf Verbesserungsmöglichkeiten und Weiterentwicklungspotential eingehen.

5.1 Vergleich

Um die Ergebnisse dieser Arbeit in einen Kontext zu setzen, werde ich sie hier anderen vergleichbaren SPH-Simulationen und NMF-Anwendungen gegenüberstellen.

Am relevantesten ist natürlich das Paper, das die grundlegende Vorgehensweise der Simulation inspiriert hat: Clavet *et al.*'s [[SCP05](#)].

In ihren Ergebnissen wird erläutert, dass mit ihrer Methode eine Echtzeitsimulation mit 1000 Partikeln à 10 Frames die Sekunde möglich ist. Dabei muss berücksichtigt werden, dass sie noch zusätzliche Mechanismen wie Viskoelastizität, Stickiness und Rigid-Body-Kollisionen implementiert haben. Dennoch sollte ein Vergleich oberflächlich als gut bewertet werden, da hier durch die Parallelisierung eine signifikant höhere Framerate bei mehr Partikeln erreicht wurde.

Vergleichbar ist diese Implementierung ebenfalls mit [[MMG03](#)], die in ihren Ergebnissen ein Beispiel mit 1500 à 25 und 3000 Partikeln à 5 Frames pro Sekunde präsentieren. Nicht zu vernachlässigen ist hier natürlich, dass die hiesige Simulation auf 2 Dimensionen beschränkt ist.

Im direkten Vergleich mit vielen State Of The Art SPH-Simulationen, fehlt dieser Implementierung jedoch der Techstack um ähnliche Zahlen aufweisen zu können. So werden in [[Wer15](#)] die Berechnungen zum Sortieren der Nachbarbuffer oder Raymarching zum Rendern einer kontinuierlichen Flüssigkeitsoberfläche auf mehrere GPUs verteilt.

Dadurch und durch die Nutzung der CUDA Bibliothek, sind signifikant höhere Partikelzahlen von bis zu 60 000 Partikeln möglich.

In Anbetracht der Visualisierung werden in State Of The Art Implementierungen meist komplexere Renderansätze verwendet. Diese variieren vom Marching Cube Algorithmus [SCP05], bis hin zu Echtzeit Raytracing auf Highend GPUs.

In vielen SPH-Simulation mit einem größeren Maßstab wird auch die Nachbarsuche mittels CUDA-Sortierfunktionen beschleunigt. Es ist gerade das In-Place Sortieren, dass einen so signifikanten Geschwindigkeitsboost verleiht. Dabei wird oft auf Spatial-Hashing oder KNN-Methoden gesetzt.

Andere kontemporäre Anwendungen nutzen Echtzeit-NMF vor allem zur Spracherkennung in Verbindung mit Machine Learning Systemen. Diese verbessern sich, durch das Deep-Learning, kontinuierlich und weisen so eine deutlich höhere Präzision als der hiesige Algorithmus auf.

Ein relevantes Beispiel in der Transkription von Musik ist die in [DG14] vorgestellte Echtzeit-Transkription eines Standard-Schlagzeugs. Die Umsetzung als VST-Plugin ist in professionellen Musikkreisen sehr nützlich, da es sich in alle möglichen Digital Audio Workstation Softwares ohne Weiteres integrieren lässt. Für den Anwendungsfall dieser Software wäre dies ebenfalls eine sinnhafte Weiterentwicklung, da die angedachte Nutzergruppe u.a. Sound Ingenieure auf Live-Bühnen beinhaltet, die oft VST-Plugins in ihren Live-Stack eingebunden haben.

Zudem liegt ein Vergleich mit [Har12] nahe. In den Tests zu seiner NMF-Implementierung weist er auf eine Präzision von 91% hin. Dies liegt deutlich über den hier festgestellten Messwerten. Besonders die scheinbare Unabhängigkeit von der Anzahl von gespielten Noten fällt auf. Die höhere Präzision mag auf die höhere Anzahl an NMF Iterationen, sowie darauf, dass die Tests mit zuvor aufgenommenen Samples durchgeführt wurden, zurückzuführen sein. Auf Grund der zusätzlichen Rechenlast und der Interaktivitätsanforderung, die auf dieser Implementierung liegt, wird kein Durchschnittswert aus den Aktivierungsframes berechnet. Dies kann jedoch durchaus einen großen Einfluss auf die Präzision des Ergebnisses haben. Wie hier ebenfalls bemerkt wurde, variiert die Anzahl an falschen Positiven, sowie die generelle Anzahl an erkannten Noten über die Dauer eines Gitarrenakkords. Das Berechnen des Durchschnitts über einen Zeitraum könnte konstanten Störsignalen wie dem wiederholten C# vorbeugen.

5.2 Weiterentwicklungen

In der Zukunft sind eine Reihe an Optimierungen und Verbesserungen denkbar.

Die Nachbarsuche birgt hier viel Potential das gesamte Programm zu skalieren. Obwohl es um die Entwicklung von höheren GPU-Bibliotheken für macOS zu Zeit still ist, wäre eine Integration eines optimierten parallelen KNN-Nachbar-Algorithmus sehr interessant. Besonders die Nutzung von Matrixmultiplikation, die in [LA] gezeigt wird, ist denkbar, da sie mit den, aktuell in der Metal Umgebung verfügbaren, Tools zu rea-

lisieren wäre.

Auch für die Flüssigkeitssimulation, gibt es eine Reihe von Eigenschaften, mit denen der jetzige Stand erweitert werden könnte. Viskoelastizität, oder Solid-Body-Interaktionen bieten Potential für eine Reihe an interessanten Visualisierungsszenarios. Und auch eine Erweiterung auf dreidimensionalen Raum ist denkbar. Zudem wäre die Einbindung von kymatischen Formen visuell sehr ansprechend.

Der NMF Algorithmus und besonders die Interpretation der Aktivierungen könnte noch weiterentwickelt werden. Das Filtern von Störsignalen oder eine Optimierung der Kostenfunktion sind beide relevante Punkte, die das Identifizieren der wirklich relevanten Aktivierungen verbessern könnte, da die mangelnde Präzision in dieser Implementierung noch für unerwünschtes Verhalten sorgt. Das Auswerten der Aktivierungen über einen Durchschnittswert oder sinnvoller speisen von FFTs könnten Lösungen für diese Probleme sein.

Eine Optimierungsmethode, die die in 4.4.3 besprochenen Frameeinbrüche minimieren könnte, wäre eine Aufteilung der NMF und der Effekt Shader auf aufeinanderfolgende Renderzyklen. Dadurch könnte vermieden werden, dass der Compute Shader Stack zu groß wird und die Rechenzeit pro Zyklus wird konstanter gehalten. Im jetzigen Zustand, sind die Frameeinbrüche, vor allem wenn bereits eine Systemauslastung besteht, unumgänglich und höchstens, durch Wechseln in den Vollbildmodus, leicht reduzierbar. Das Einbinden von anderen Shadereffekten, sowie Effekten, die die Simulationsparameter in größerem Umfang beeinflussen liegt nahe.

Aus musikalischer Sicht bedarf es außerdem der Integration weiterer Akkordtypen, ähnlich wie in [Len13] (und evtl. Instrumente), sodass z.B 7er Akkorde zu einem bestimmtem Verhalten führen. Die in [VGMA19] vorgestellte Methode zur Visualisierung von Melodie als Richtungs-/Beschleunigungsvektoren, wäre besonders für reine Melodieinstrumente interessant.

Zu guter Letzt, bietet die Apple Entwicklungsumgebung den Vorteil, dass Programme leicht zu mobilen Versionen exportiert werden können. Eine mobile Version der App wäre demnach sinnvoll.

5.3 Erreichtes

Im Laufe der Arbeit ist es gelungen eine parallelisierte Form der in [SCP05] vorgestellten *Double Density Relaxation-* und Viskositäts-Paradigmen mittels Compute Shadern zu realisieren. Mit dieser Formulierung kann diverses Flüssigkeitsverhalten, wie Druckausgleich, Wellen- und Tröpfchenformung, sowie Oberflächenspannung visuell überzeugend dargestellt werden.

Es wurde gezeigt, dass ein Brute-Force Nachbar-Algorithmus, der die massive Parallelisierung der GPU Architektur ausnutzt, eine optimiertere Methode wie Spatial Hashing hinsichtlich Framerate- und GPU-Rechenzeit-Performance, zumindest mit kleineren Partikelanzahlen, übertreffen kann.

Es wurde eine Methode vorgestellt, die Nicht-Negative-Matrixfaktorisierung mittels Metal Performace Shadern umsetzt und in eine SPH-Pipeline integriert. Die Multiplikative Update Funktion ist in der Lage einzelne Noten mit einer Trefferquote von 90% und Teilstufenzen eines Akkords und dessen Tongeschlecht mit ca. 74% Genauigkeit zu bestimmen.

Ein Konzept, dass eine RGB-Textur nutzt, um inputabhängige Modifizierungen der Beschleunigung von Partikeln vorzunehmen, wurde eingeführt und zwei Effekte, die diesen Mechanismus nutzen, wurden implementiert. Diese Modifizierungen können im Schnitt 13 Frames, nachdem das Input auf der Gitarre gespielt wurde, gerendert werden.

Um die Nutzung und Anpassung der Simulation zu vereinfachen, verfügt die Applikation über ein Userinterface, mit dem die Simulationsparameter, sowie die Empfindlichkeit der visuellen Effekte in Echtzeit angepasst werden können.

Auf dem Testgerät konnte eine Audiovisualisierung mit 2500 Partikeln und einer maximalen NMF-Iterationstiefe von 20 Schritten zu 60 Frames pro Sekunde realisiert werden. Das Ziel dieser Arbeit, – die Erstellung eines interaktiven Systems –, konnte somit erreicht werden und die Basis für eine Reihe an Weiterentwicklungen und Optimierungen ist gelegt.

Abbildungsverzeichnis

2.1	Illustrierung Tonleitern und Intervalle	9
2.2	Beispiel: E-Dur Akkord mit großer Terz und E-Moll Akkord mit kleiner Terz	9
2.3	Zeitdiskrete Schwingung E5	11
2.4	Frequenzspektrum E5	11
2.5	Input- und Sample-FFTs	13
2.6	Die drei FFTs summiert sind gleich der Input-FFT	13
3.1	Aufbau der Pipeline	15
3.2	<i>neighbors</i> Textur für 20 Partikel. Der rot-Teil des Pixels indiziert ob es ein Nachbarpartikel ist.	18
3.3	Smoothing Kernel für P_{near} und P	21
3.4	Dimensionen der NMF-Matrizen. Da es sich bei V um eine FFT handelt, ist $n = 1$, $m =$ Frequenzbereich der FFT, $r =$ Anzahl an Notensamples in W.	27
3.5	f5 mit zusätzlich erfassten Harmoniefrequenzen	28
3.6	<i>velocityField</i> : links: <i>bPush</i> : Punktprimitive der Noten A3, B3, F3 und E4 rechts: <i>rPush</i> : Dreiecksprimitiv eines F-Dur Akkords	30
4.1	User Interface mit Simulationseinstellungen	32
4.2	Performance der Simulation mit Spatial Hashing oder Brute-Force (Ohne aktive NMF)	33
4.3	Von links nach rechts: Stadien des Dammtests	34
4.4	Tropfen verdrängt angrenzende Flüssigkeit	34
4.5	Invertiertes k , mit $k = 100$	36
4.6	Ergebnisse von sieben NMFs des gleichen Inputs. Von links nach rechts: G, B, F-Moll, G-Dur	38
4.7	Links: Euklidische Distanz; Rechts: Kullberg-Leibler-Divergenz	40
4.8	<i>bPush</i> mit den Noten d#, f#, e	41
4.9	Akkorde B-Moll und G-Dur visualisiert	41
4.10	Performance der Simulation mit Standardfenstergröße und Vollbild	41

List of Algorithms

1	GPU-Pipeline	15
2	Find Neighbors	17
3	Simulationsschritt	18
4	applyViscosity	19
5	Double Density Relaxation	21
6	computeDensities	22
7	computePressures	22
8	applyDisplacements	22
9	computeDisplacements	23
10	Signalvorbereitung	25
11	computeNMF	28

Literaturverzeichnis

- [CM11] Nuttapong Chentanez and Matthias Müller. Real-time eulerian water simulation using a restricted tall cell grid. *ACM Trans. Graph.*, 30:82, 07 2011.
- [DG14] Christian Dittmar and Daniel Gärtner. Real-time transcription and separation of drum recordings based on nmf decomposition. In *DAFx*, 2014.
- [DHY22] Xiangwei Dong, Guannan Hao, and Ran Yu. Two-dimensional smoothed particle hydrodynamics (sph) simulation of multiphase melting flows and associated interface behavior. *Engineering Applications of Computational Fluid Mechanics*, 16(1):588–629, 2022.
- [doca] vdsp: Finding the component frequencies in a composite sine wave. last visited: 03.12.22. https://developer.apple.com/documentation/accelerate/vdsp/fast_fourier_transforms/finding_the_component_frequencies_in_a_composite_sine_wave.
- [docb] Visualizing sound as an audio spectrogramm. last visited: 03.01.23. https://developer.apple.com/documentation/accelerate/visualizing_sound_as_an_audio_spectrogram.
- [Gil14] Nicolas Gillis. The why and how of nonnegative matrix factorization. *Regularization, Optimization, Kernels, and Support Vector Machines*, 12, 01 2014.
- [GR77] Monaghan J. Gingold R. Smoothed particle hydrodynamics – theory and application to non- spherical stars. *Monthly Notices of the Royal Astronomical Society* 181, 1977.
- [Har12] John Hartquist. Real-time musical analysis of polyphonic guitar audio. 06 2012.
- [Hec95] Paul Heckbert. Fourier transforms and the fast fourier transform algorithm. <https://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/2001/pub/www/notes/fourier/fourier.pdf>, 1995.
- [L.77] Lucy L. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, 82, 1977.

- [LA] Shengren Li and Nina Amenta. Brute-force k-nearest neighbors search on the gpu.
- [Len13] Nathan Lenssen. Applications of fourier analysis to audio signal processing: An investigation of chord detection algorithms. 2013.
- [LS00] Daniel Lee and H. Sebastian Seung. Algorithms for non-negative matrix factorization. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2000.
- [MD96] Marie-Paule Cani Mathieu Desbrun. Smoothed particles: A new paradigm for animating highly deformable bodies. 08 1996.
- [MMG03] David Charypar Matthias Muller and Markus Gross. Particle-based fluid simulation for interactive applications. 2003.
- [mps] Matrix multiplication with metal performance shaders.
last visited: 23.12.22. <https://machinethink.net/blog/mps-matrix-multiplication/>.
- [Mü15] Meinhard Müller. Fundamentals of music processing, chapter 8: Musically informed audio decomposition; last visited 07.01.23. https://www.audiolabs-erlangen.de/content/05-fau/professor/00-mueller/04-bookFMP/02-slides/Mueller_FMP_Chapter8.pdf, 2015.
- [NSG08] M. Harris N. Satish and M. Garland. Designing efficient sorting algorithms for manycore gpus. *NVIDIA Technical Report NVR-2008-001*, 09 2008.
- [O8] Juraj Onderik and Roman Ďuríkovič. Efficient neighbor search for particle-based fluids. *Journal of the Applied Mathematics, Statistics and Informatics* 4, 4:29–43, 01 2008.
- [PT94] Pentti Paatero and Unto Tapper. Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values. *Environmetrics*, 5(2):111–126, 1994.
- [SB03] P. Smaragdis and J.C. Brown. Non-negative matrix factorization for polyphonic music transcription. In *2003 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (IEEE Cat. No.03TH8684)*, pages 177–180, 2003.
- [SCP05] Philippe Beaudoin Simon Clavet and Pierre Poulin. Particle-based Visco-elastic Fluid Simulation (PVFS). 2005.
- [Sta03] Jon Stam. Real-time fluid dynamics for games. 2003.
- [THM⁺03] Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomerańcze, and Markus Gross. Optimized spatial hashing for collision detection of deformable objects. *VMV'03: Proceedings of the Vision, Modeling, Visualization*, 3, 12 2003.

- [VGMA19] Venkata Subramanian Viraraghavan, Rahul Dasharath Gavas, Hema A. Murthy, and Rangarajan Aravind. Visualizing carnatic music as projectile motion in a uniform gravitational field. *Workshop on Speech, Music and Mind (SMM 2019)*, 2019.
- [Wer15] Tim Werner. *Effiziente SPH-basierte Flüssigkeitssimulation mit Visualisierung auf einem GPU-Cluster*. PhD thesis, 09 2015.
- [WZ13] Yu-Xiong Wang and Yu-Jin Zhang. Nonnegative matrix factorization: A comprehensive review. *Knowledge and Data Engineering, IEEE Transactions on*, 25:1336–1353, 06 2013.